(19) Deutsches Patent- und Markenamt

(10) **DE 10 2022 124 603 A1** 2023.03.30

(12) # Offenlegungsschrift

(21) Aktenzeichen: **10 2022 124 603.5**
(22) Anmeldetag: **26.09.2022**
(43) Offenlegungstag: **30.03.2023**

(51) Int Cl.: ***G06T 1/20*** (2006.01)
G06T 15/06 (2011.01)
G06F 9/455 (2006.01)

(30) Unionspriorität:
17/485,399 25.09.2021 US

(71) Anmelder:
Intel Corporation, Santa Clara, CA, US

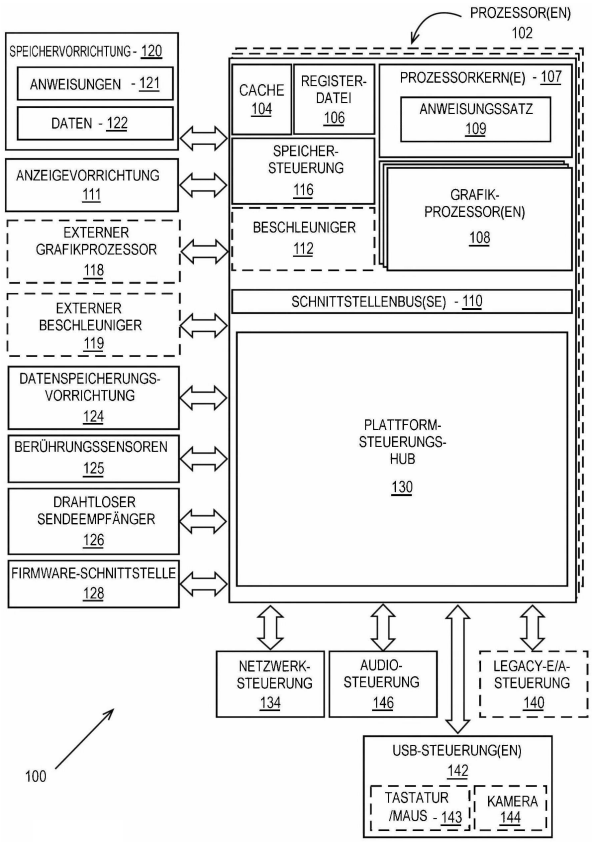
(74) Vertreter:
BOEHMERT & BOEHMERT Anwaltspartnerschaft mbB - Patentanwälte Rechtsanwälte, 28209 Bremen, DE

(72) Erfinder:
Volkov, Stanislav, Santa Clara, CA, US; Pillow, Scott, Santa Clara, CA, US; Barczak, Joshua, Santa Clara, CA, US; Levit-Gurevich, Konstantin, Santa Clara, CA, US; Surmin, Igor, Santa Clara, CA, US

Die folgenden Angaben sind den vom Anmelder eingereichten Unterlagen entnommen.

(54) Bezeichnung: **EINRICHTUNG UND VERFAHREN FÜR RAY-TRACING MIT SHADER-AUFRUF-GRAPHENMANALYSE**

(57) Zusammenfassung: Eine Einrichtung und ein Verfahren zur Verbesserung der Raytracing-Effizienz. Zum Beispiel umfasst eine Ausführungsform einer Einrichtung Folgendes: Eine Einrichtung, die umfasst: Eine Binärinstrumentierungs-Engine zum Durchführen einer binären Instrumentierung von Raytracing-Shadern und zum Verfolgen der Ausführung der Raytracing-Shader, um Ausführungsmetriken zu erzeugen; Aufruf-Graphen-Konstruktionslogik zum Konstruieren eines Shader-Aufruf-Graphen basierend auf den Ausführungsmetriken; Shader-Quellabbildungslogik zum Abbilden des Shader-Aufruf-Graphen auf Shader-Quellcode, um eine Quellcodeabbildung zu erzeugen; Effizienzanalyselogik zum Bestimmen von Ineffizienzen bei der Raytracing-Shader-Ausführung basierend auf der Quellcodeabbildung; und Optimierungslogik zum Identifizieren von Optimierungshandlungen basierend auf den Ineffizienzen.



Beschreibung

HINTERGRUND

Gebiet der Erfindung

[0001] Die vorliegende Erfindung bezieht sich allgemein auf das Gebiet der Grafikprozessoren. Insbesondere betrifft die Erfindung eine Einrichtung und ein Verfahren zum Raytracing mit Shader-Aufruf-Graphenanalyse.

Beschreibung der verwandten Technik

[0002] Raytracing (Strahlverfolgung) ist eine Technik, bei der ein Lichttransport durch physisch basiertes Rendern simuliert wird. Während es beim kinematischen Rendern vielfältige Anwendung findet, wurde es bis vor wenigen Jahren als zu ressourcenintensiv zur Echtzeit-Performance angesehen. Eine der Schlüsseloperationen beim Raytracing ist das Verarbeiten einer Sichtbarkeitsanfrage für als „Strahltraversierung“ bekannte Strahl-Szenen-Überschneidungen, wobei Strahl-Szenen-Überschneidungen durch Traversieren und Überschneiden von Knoten in einer Hüllkörperhierarchie (BVH: Bounding Volume Hierarchy) berechnet werden.

[0003] Rasterisierung ist eine Technik, bei der Bildschirmobjekte aus 3D-Modellen von Objekten, die aus einem Mesh aus Dreiecken erzeugt werden, erzeugt werden. Die Vertices jedes Dreiecks schneiden die Eckpunkte anderer Dreiecke unterschiedlicher Form und Größe. Jeder Vertex weist eine Position im Raum sowie Informationen über Farbe, Textur und seine Normale auf, die verwendet werden, um zu bestimmen, wie die Oberfläche eines Objekts zeigt. Eine Rasterisierungseinheit wandelt die Dreiecke der 3D-Modelle in Pixel in einem 2D-Bildschirmraum um, wobei jedem Pixel ein Anfangsfarbwert basierend auf den Vertexdaten zugeordnet werden kann.

Figurenliste

[0004] Ein besseres Verständnis der vorliegenden Erfindung kann aus der folgenden ausführlichen Beschreibung in Verbindung mit den folgenden Zeichnungen erhalten werden, wobei gilt:

Fig. 1 ist ein Blockdiagramm einer Ausführungsform eines Computersystems mit einem Prozessor, der einen oder mehrere Prozessorkerne und Grafikprozessoren aufweist;

Fig. 2A-D veranschaulichen Rechensysteme und Grafikprozessoren, die durch Ausführungsformen der Erfindung bereitgestellt werden;

Fig. 3A-C veranschaulichen Blockdiagramme zusätzlicher Grafikprozessor- und Rechenbeschleunigerarchitekturen;

Fig. 4 ist ein Blockdiagramm einer Ausführungsform einer Grafikverarbeitungs-Engine für einen Grafikprozessor;

Fig. 5A-B veranschaulichen eine Thread-Ausführungslogik, die ein Array von Verarbeitungselementen beinhaltet;

Fig. 6 ist ein Blockdiagramm einer Thread-Ausführungslogik, die ein Array von Verarbeitungselementen beinhaltet;

Fig. 7 veranschaulicht ein Grafikprozessor-Ausführungseinheit-Anweisungsformat gemäß einer Ausführungsform;

Fig. 8 ist ein Blockdiagramm einer anderen Ausführungsform eines Grafikprozessors, der eine Grafik-Pipeline, eine Medien-Pipeline, eine Anzeige-Engine, Thread-Ausführungslogik und eine Renderausgabe-Pipeline beinhaltet;

Fig. 9A ist ein Blockdiagramm, das ein Grafikprozessor-Befehlsformat veranschaulicht, gemäß einer Ausführungsform;

Fig. 9B ist ein Blockdiagramm, das eine Grafikprozessor-Befehlssequenz veranschaulicht, gemäß einer Ausführungsform;

Fig. 10 veranschaulicht eine beispielhafte Grafiksoftwarearchitektur für ein Datenverarbeitungssystem gemäß einer Ausführungsform;

Fig. 11A veranschaulicht beispielhafte IP-Kern-Entwicklungssysteme, die verwendet werden können, um eine integrierte Schaltung zum Durchführen von Operationen herzustellen, gemäß einer Ausführungsform;

Fig. 11B-D veranschaulichen beispielhafte Packaging-Anordnungen einschließlich Chiplets und Interposer-Substraten;

Fig. 12 veranschaulicht eine beispielhafte integrierte System-on-Chip-Schaltung, die unter Verwendung eines oder mehrerer IP-Kerne gefertigt werden kann, gemäß einer Ausführungsform;

Fig. 13 veranschaulicht einen beispielhaften Grafikprozessor einer integrierten System-on-Chip-Schaltung, die unter Verwendung eines oder mehrerer IP-Kerne gefertigt werden kann;

Fig. 14 veranschaulicht einen zusätzlichen beispielhaften Grafikprozessor einer integrierten System-on-Chip-Schaltung, die unter Verwendung eines oder mehrerer IP-Kerne gefertigt werden kann;

Fig. 15 veranschaulicht eine Architektur zum Durchführen eines anfänglichen Trainings einer Maschinenlernarchitektur;

Fig. 16 veranschaulicht, wie eine Maschinenlern-Engine während der Laufzeit kontinuierlich trainiert und aktualisiert wird;

Fig. 17 veranschaulicht, wie eine Maschinenlern-Engine während der Laufzeit kontinuierlich trainiert und aktualisiert wird;

Fig. 18A-B veranschaulichen, wie Maschinenlerndaten in einem Netzwerk gemeinsam genutzt werden; und

Fig. 19 veranschaulicht ein Verfahren zum Trainieren einer Maschinenlern-Engine;

Fig. 20 veranschaulicht, wie Knoten Ghost-Bereichsdaten austauschen, um verteilte Entrauschungsoperationen durchzuführen;

Fig. 21 veranschaulicht eine Architektur, in der Bildrender- und Entrauschungsoperationen über mehrere Knoten hinweg verteilt sind;

Fig. 22 veranschaulicht zusätzliche Einzelheiten einer Architektur zum verteilten Rendern und Entrauschen;

Fig. 23 veranschaulicht ein Verfahren zum Durchführen eines verteilten Renderns und Entrauschens;

Fig. 24 veranschaulicht ein Maschinenlernverfahren;

Fig. 25 veranschaulicht mehrere miteinander verbundene Allzweck-Grafikprozessoren;

Fig. 26 veranschaulicht einen Satz von Faltungsschichten und vollständig verbundenen Schichten für eine Maschinenlernimplementierung;

Fig. 27 veranschaulicht ein Beispiel für eine Faltungsschicht;

Fig. 28 veranschaulicht ein Beispiel für einen Satz miteinander verbundener Knoten in einer Maschinenlernimplementierung;

Fig. 29 veranschaulicht ein Trainingsframework, innerhalb dessen ein neuronales Netzwerk unter Verwendung eines Trainingsdatensatzes lernt;

Fig. 30A veranschaulicht Beispiele für Modellparallelität und Datenparallelität;

Fig. 30B veranschaulicht ein System-on-Chip (SoC);

Fig. 31 veranschaulicht eine Verarbeitungsarchitektur, die Raytracing-Kerne und Tensorkerne beinhaltet;

Fig. 32 veranschaulicht ein Beispiel eines Strahlenbündels;

Fig. 33 veranschaulicht eine Einrichtung zum Durchführen von Strahlenbündel-Tracing;

Fig. 34 veranschaulicht ein Beispiel einer Strahlenbündelhierarchie;

Fig. 35 veranschaulicht ein Verfahren zum Durchführen von Strahlenbündel-Tracing;

Fig. 36 veranschaulicht ein Beispiel für eine verteilte Raytracing-Engine;

Fig. 37-38 veranschaulichen eine Komprimierung, die in einem Raytracing-System durchgeführt wird;

Fig. 39 veranschaulicht ein Verfahren, das auf einer Raytracing-Architektur implementiert wird;

Fig. 40 veranschaulicht eine beispielhafte hybride Raytracing-Einrichtung;

Fig. 41 veranschaulicht Stapel, die für Raytracing-Operationen verwendet werden;

Fig. 42 veranschaulicht zusätzliche Einzelheiten für eine hybride Raytracing-Einrichtung;

Fig. 43 veranschaulicht eine Hüllkörperhierarchie;

Fig. 44 veranschaulicht einen Aufrufstapel und eine Traversierungszustandsspeicherung;

Fig. 45 veranschaulicht ein Verfahren zur Traversierung und Überschneidung;

Fig. 46A-B veranschaulichen, wie mehrere Dispatch-Zyklen erforderlich sind, um gewisse Shader auszuführen;

Fig. 47 veranschaulicht, wie ein einziger Dispatch-Zyklus mehrere Shader ausführt;

Fig. 48 veranschaulicht, wie ein einziger Dispatch-Zyklus mehrere Shader ausführt;

Fig. 49 veranschaulicht eine Architektur zum Ausführen von Raytracing-Anweisungen;

Fig. 50 veranschaulicht ein Verfahren zum Ausführen von Raytracing-Anweisungen innerhalb eines Threads;

Fig. 51 veranschaulicht eine Ausführungsform einer Architektur für asynchrones Raytracing;

Fig. 52A veranschaulicht eine Ausführungsform einer Strahltraversierungsschaltung;

Fig. 52B veranschaulicht Prozesse, die in einer Ausführungsform ausgeführt werden, um Strahlspeicherungsbänke zu verwalten;

Fig. 53 veranschaulicht eine Ausführungsform einer Prioritätsauswahl-Schaltungsanordnung/-logik;

Fig. 54 und **Fig. 55A-B** veranschaulichen verschiedene Arten von Raytracing-Daten, darunter Flags, Ausnahmen und Culling-Daten, die in einer Ausführungsform der Erfindung verwendet werden;

Fig. 56 veranschaulicht eine Ausführungsform zur Frühbestimmung aus der Raytracing-Pipeline;

Fig. 57 veranschaulicht eine Ausführungsform einer Prioritätsauswahl-Schaltungsanordnung/-logik;

Fig. 58 veranschaulicht eine beispielhafte Hüllkörperhierarchie (BVH), die für Strahltraversierungsoperationen verwendet wird;

Fig. 59A-B veranschaulichen zusätzliche Traversieru ngsoperationen;

Fig. 60 veranschaulicht eine Ausführungsform einer Stapelverwaltungsschaltungsanordnung zum Verwalten eines BVH-Stapels;

Fig. 61A-B veranschaulichen beispielhafte Datenstrukturen, Teilstrukturen und Operationen, die für Strahlen, Treffer und Stapel durchgeführt werden;

Fig. 62 veranschaulicht eine Ausführungsform eines Detaillierungsgradselektors mit einer N-Bit-Vergleichsoperationsmaske;

Fig. 63 veranschaulicht eine Beschleunigungsdatenstruktur gemäß einer Ausführungsform der Erfindung;

Fig. 64 veranschaulicht eine Ausführungsform eines Komprimierungsblocks, der Residuumwerte und Metadaten beinhaltet;

Fig. 65 veranschaulicht ein Verfahren gemäß einer Ausführungsform der Erfindung;

Fig. 66 veranschaulicht eine Ausführungsform eines Blockoffsetindex-Komprimierungblocks;

Fig. 67A veranschaulicht eine hierarchische Bitvektorindexierung (HBI) gemäß einer Ausführungsform der Erfindung;

Fig. 67B veranschaulicht einen Indexkomprimierungsblock gemäß einer Ausführungsform der Erfindung; und

Fig. 68 veranschaulicht eine beispielhafte Architektur einschließlich einer BVH-Komprimierungsschaltungsanordnung/-logik und -Dekomprimierungsschaltungsanordnung/-logik.

Fig. 69A veranschaulicht eine Verschiebungsfunktion, die auf ein Mesh angewendet wird;

Fig. 69B veranschaulicht eine Ausführungsform einer Komprimierungsschaltungsanordnung zum Komprimieren eines Mesh oder Meshlets;

Fig. 70A veranschaulicht eine Verschiebungsabbildung auf einer Basisunterteilungsfläche;

Fig. 70B-C veranschaulichen Differenzvektoren relativ zu einem groben Basis-Mesh;

Fig. 71 veranschaulicht ein Verfahren gemäß einer Ausführungsform der Erfindung;

Fig. 72-74 veranschaulichen ein Mesh, das mehrere miteinander verbundene Vertices umfasst;

Fig. 75 veranschaulicht eine Ausführungsform eines Tessellators zum Erzeugen eines Mesh;

Fig. 76-77 veranschaulichen eine Ausführungsform, bei der Hüllkörper basierend auf einem Mesh gebildet werden;

Fig. 78 veranschaulicht eine Ausführungsform eines Mesh, das überlappende Vertices gemeinsam nutzt;

Fig. 79 veranschaulicht ein Mesh mit gemeinsam genutzten Kanten zwischen Dreiecken;

Fig. 80 veranschaulicht eine Raytracing-Engine gemäß einer Ausführungsform;

Fig. 81 veranschaulicht einen BVH-Komprimierer gemäß einer Ausführungsform;

Fig. 82A-C veranschaulichen beispielhafte Datenformate für ein 64-Bit-Register;

Fig. 83A-B veranschaulichen eine Ausführungsform eines Indexes für einen Ringpuffer;

Fig. 84A-B veranschaulichen beispielhafte Ringpuffer-Atomics für Hersteller und Verbraucher;

Fig. 85A veranschaulicht eine Ausführungsform einer gekachelten Ressource;

Fig. 85B veranschaulicht ein Verfahren gemäß einer Ausführungsform der Erfindung;

Fig. 86A veranschaulicht eine Ausführungsform einer BVH-Verarbeitungslogik einschließlich eines On-Demand-Builders;

Fig. 86B veranschaulicht eine Ausführungsform eines On-Demand-Builders für eine Beschleunigungsstruktur;

Fig. 86C veranschaulicht eine Ausführungsform einer sichtbaren Beschleunigungsstruktur-Abbildung unterster Ebene;

Fig. 86D veranschaulicht verschiedene Arten von Instanzen und Traversierungsentscheidungen;

Fig. 87 veranschaulicht eine Ausführungsform einer materialbasierten Cull-Maske;

Fig. 88 veranschaulicht eine Ausführungsform, bei der eine Quadtree-Struktur über einem Geometrie-Mesh gebildet wird;

Fig. 89A veranschaulicht eine Ausführungsform einer Raytracing-Architektur;

Fig. 89B veranschaulicht eine Ausführungsform, die eine Meshlet-Komprimierung beinhaltet;

Fig. 90 veranschaulicht mehrere Threads einschließlich synchroner Threads, divergierender Spawn-Threads, regulärer Spawn-Threads und konvergierender Spawn-Threads;

Fig. 91 veranschaulicht eine Ausführungsform einer Raytracing-Architektur mit einem bindungslosen Thread-Dispatcher;

Fig. 92 veranschaulicht ein Raytracing-Cluster gemäß einer Ausführungsform;

Fig. 93-100 veranschaulichen Ausführungsformen der Verwendung von Proxy-Daten in einer Mehrfachknoten-Raytracing-Implementierung;

Fig. 101 veranschaulicht ein Verfahren gemäß einer Ausführungsform der Erfindung;

Fig. 102 veranschaulicht einen beispielhaften Satz von Shader-Aufzeichnungen, die mit einem ersten und zweiten Sub-Slice assoziiert sind;

Fig. 103 veranschaulicht ein Verfahren gemäß einer Ausführungsform der Erfindung;

Fig. 104 veranschaulicht eine Einrichtung gemäß einer Ausführungsform der Erfindung;

Fig. 105 veranschaulicht einen Instrumentierungscode, der in Raytracing-Shader-Chunks eingefügt ist;

Fig. 106 veranschaulicht eine Ausführungsform, in der gewisse Tracing-Daten basierend auf einem spezifizierten Timing gesammelt werden; und

Fig. 107 veranschaulicht mehrere Shader-Aufzeichnungen, die mit Primär- und Sekundärstrahlen assoziiert sind.

AUSFÜHRLICHE BESCHREIBUNG

[0005] In der folgenden Beschreibung werden zum Zweck der Erläuterung zahlreiche spezifische Einzelheiten dargelegt, um ein umfassendes Verständnis der unten beschriebenen Ausführungsformen der Erfindung bereitzustellen. Für den Fachmann auf dem Gebiet ist jedoch offensichtlich, dass die Ausführungsformen der Erfindung ohne manche dieser speziellen Details umgesetzt werden können. In anderen Fällen sind wohlbekannte Strukturen und Vorrichtungen in Blockdiagrammform gezeigt, um zu vermeiden, dass die zugrunde liegenden Prinzipien der Ausführungsformen der Erfindung unklar gemacht werden.

BEISPIELHAFTE GRAFIKPROZESSORARCHITEKTUREN UND DATENTYPEN

Systemübersicht

[0006] Fig. 1 ist ein Blockdiagramm eines Verarbeitungssystems 100 gemäß einer Ausführungsform. Das System 100 kann in einem Einzelprozessor-Desktop-System, einem Multiprozessor-Workstation-System oder einem Serversystem mit einer großen Anzahl von Prozessoren 102 oder Prozessorkernen 107 verwendet werden. In einer Ausführungsform ist das System 100 eine Verarbeitungsplattform, die innerhalb einer integrierten System-on-Chip(SoC)-Schaltung zur Verwendung in mobilen, handgehaltenen oder eingebetteten Vorrichtungen, wie etwa innerhalb von Internet-der-Dinge(IoT)-Vorrichtungen mit drahtgebundener oder drahtloser Konnektivität zu einem Lokal- oder Weitbereichsnetzwerk, integriert ist.

[0007] In einer Ausführungsform kann das System 100 eine serverbasierte Gaming-Plattform; eine Spielkonsole, einschließlich einer Spiel- und Medienkonsole; eine Mobile-Gaming-Konsole, eine handgehaltene Spielkonsole oder eine Onlinespielkonsole beinhalten, mit dieser gekoppelt oder darin integriert sein. In einigen Ausführungsformen ist das System 100 Teil eines Mobiltelefons, eines Smartphones, einer Tablet-Rechenvorrichtung oder einer mobilen internetverbundenen Vorrichtung, wie etwa eines Laptops mit geringer interner Speicherkapazität. Das Verarbeitungssystem 100 kann auch Folgendes beinhalten, damit gekoppelt oder darin integriert sein: eine Wearable-Vorrichtung, wie etwa eine Smartwatch-Wearable-Vorrichtung; eine Smart-Brille oder Smart-Kleidung, erweitert mit Augmented-Reality(AR)- oder Virtual-Reality(VR)-Merkmalen, um visuelle, auditive oder taktile Ausgaben bereitzustellen, um visuelle, auditive oder taktile reelle Erfahrungen zu ergänzen oder anderweitig Text, Audio, Grafiken, Video, holografische Bilder oder Video oder taktiles Feedback bereitzustellen; eine andere Augmented-Reality(AR)-Vorrichtung; oder eine andere Virtual-Reality (VR)-Vorrichtung. In einigen Ausführungsformen beinhaltet das Verarbeitungssystem 100 eine Fernseher- oder Set-Top-Box-Vorrichtung oder ist Teil davon. In einer Ausführungsform kann das System 100 ein selbstfahrendes Fahrzeug, wie etwa einen Bus, einen Zugmaschinenanhänger, ein Auto, ein Motorrad oder ein Elektrofahrrad, ein Flugzeug oder einen Gleiter (oder eine beliebige Kombination davon), aufweisen, mit diesem gekoppelt oder in diesem integriert sein. Das selbstfahrende Fahrzeug kann das System 100 zur Verarbeitung der um das Fahrzeug herum erfassten Umgebung verwenden.

[0008] In einigen Ausführungsformen beinhalten der eine oder die mehreren Prozessoren 102 jeweils einen oder mehrere Prozessorkerne 107, um Anweisungen zu verarbeiten, die dann, wenn sie ausgeführt werden, Operationen für System- und Anwender-Software durchführen. In manchen Ausführungsformen ist mindestens einer des einen oder der mehreren Prozessorkerne 107 dazu konfiguriert, einen speziellen Anweisungssatz 109 zu verarbeiten. In einigen Ausführungsformen kann der Anweisungssatz 109 Berechnungen mit komplexem Anweisungssatz (CISC), Berechnung mit reduziertem Anweisungssatz (RISC) oder Berechnungen über ein sehr langes Befehlswort (VLIW) unterstützen. Ein oder mehrere Prozessorkerne 107 können einen unterschiedlichen Anweisungssatz 109 verarbeiten, der Anweisungen enthalten kann, um die Emulation anderer Anweisungssätze zu unterstützen. Der Prozessorkern 107 kann auch andere Verarbeitungsvorrichtungen beinhalten, wie etwa einen Digitalsignalprozessor (DSP).

[0009] In manchen Ausführungsformen beinhaltet der Prozessor 102 einen Cachespeicher 104. In Abhängigkeit von der Architektur kann der Prozessor 102 einen einzigen internen Cache oder mehrere Ebenen von internem Cache aufweisen. In manchen Ausführungsformen wird der Cachespeicher durch verschiedene

Komponenten des Prozessors 102 gemeinsam genutzt. In einigen Ausführungsformen verwendet der Prozessor 102 auch einen externen Cache (z. B. einen Level-3(L3)-Cache oder einen Cache der letzten Ebene (LLC)) (nicht gezeigt), der durch Prozessorkerne 107 unter Verwendung bekannter Cachekohärenztechniken gemeinsam genutzt wird. Eine Registerdatei 106 kann zusätzlich in dem Prozessor 102 enthalten sein und kann verschiedene Arten von Registern zum Speichern verschiedener Arten von Daten (z. B. Ganzzahlregister, Gleitkommaregister, Statusregister und ein Anweisungszeigerregister) beinhalten. Manche Register können Allzweckregister sein, während andere Register für die Gestaltung des Prozessors 102 spezifisch sein können.

[0010] In einigen Ausführungsformen sind der eine oder die mehreren Prozessoren 102 mit einem oder mehreren Schnittstellenbussen 110 zum Übertragen von Kommunikationssignalen, wie etwa Adress-, Daten- oder Steuersignalen, zwischen dem Prozessor 102 und anderen Komponenten in dem System 100 gekoppelt. Der Schnittstellenbus 110 kann in einer Ausführungsform ein Prozessorbus sein, wie etwa eine Version des DMI-Busses (DMI: Direct Media Interface). Prozessorbusse sind jedoch nicht auf den DMI-Bus beschränkt und können einen oder mehrere Peripheral Component Interconnect Busse (z. B. PCI, PCI-Express), Speicherbusse oder andere Arten von Schnittstellenbussen umfassen. In einer Ausführungsform beinhalten der eine oder die mehreren Prozessoren 102 eine integrierte Speichersteuerung 116 und einen Plattformsteuerungshub 130. Die Speichersteuerung 116 ermöglicht eine Kommunikation zwischen einer Speichervorrichtung und anderen Komponenten des Systems 100, während der Plattformsteuerungshub (PCH: Plattform Controller Hub) 130 Verbindungen zu E/A-Vorrichtungen über einen lokalen E/A-Bus bereitstellt.

[0011] Die Speichervorrichtung 120 kann eine Dynamischer-Direktzugriffsspeicher(DRAM)-Vorrichtung, eine Statischer-Direktzugriffsspeicher(SRAM)-Vorrichtung, eine Flash-Speichervorrichtung, eine Phasenwechselspeichervorrichtung oder eine andere Speichervorrichtung sein, die eine geeignete Performanz aufweist, um als ein Prozessspeicher zu dienen. In einer Ausführungsform kann die Speichervorrichtung 120 als Systemspeicher für das System 100 arbeiten, um Daten 122 und Anweisungen 121 zu speichern, die verwendet werden, wenn der eine oder die mehreren Prozessoren 102 eine Anwendung oder einen Prozess ausführt. Die Speichersteuerung 116 ist auch mit einem optionalen externen Grafikprozessor 118 gekoppelt, der mit dem einen oder den mehreren Grafikprozessoren 108 in den Prozessoren 102 kommunizieren kann, um Grafik- und Medienoperationen durchzuführen. In einigen Ausführungsformen können Grafik-, Medien- und/oder Rechenoperationen durch einen Beschleuniger 112 unterstützt werden, der ein Coprozessor ist, der dazu konfiguriert sein kann, einen spezialisierten Satz von Grafik-, Medien- oder Rechenoperationen durchzuführen. In einer Ausführungsform ist der Beschleuniger 112 zum Beispiel ein Matrixmultiplikationsbeschleuniger, der verwendet wird, um Maschinenlern- oder Rechenoperationen zu optimieren. In einer Ausführungsform ist der Beschleuniger 112 ein Raytracing-Beschleuniger, der verwendet werden kann, um Raytracing-Operationen in Abstimmung mit dem Grafikprozessor 108 durchzuführen. In einer Ausführungsform kann ein externer Beschleuniger 119 anstelle des Beschleunigers 112 oder zusammen mit diesem verwendet werden.

[0012] In einigen Ausführungsformen kann eine Anzeigevorrichtung 111 mit dem einen oder den mehreren Prozessoren 102 verbunden sein. Die Anzeigevorrichtung 111 kann eine interne Anzeigevorrichtung, wie in einer mobilen elektronischen Vorrichtung oder einer Laptopvorrichtung, und/oder eine externe Anzeigevorrichtung sein, die über eine Anzeigeschnittstelle (z. B. DisplayPort usw.) angeschlossen ist. In einer Ausführungsform kann die Anzeigevorrichtung 111 eine am Kopf angebrachte Anzeige (HMD: Head Mounted Display) sein, wie etwa eine stereoskopische Anzeigevorrichtung zur Verwendung bei Virtual-Reality(VR)-Anwendungen oder Augmented-Reality(AR)-Anwendungen.

[0013] In einigen Ausführungsformen ermöglicht der Plattformsteuerungshub 130, dass Peripheriegeräte über einen Hochgeschwindigkeits-E/A-Bus mit der Speichervorrichtung 120 und dem Prozessor 102 verbunden sind. Die E/A-Peripheriegeräte schließen unter anderem eine Audiosteuerung 146, eine Netzwerksteuerung 134, eine Firmware-Schnittstelle 128, einen drahtlosen Sendeempfänger 126, Berührungssensoren 125, eine Datenspeicherungsvorrichtung 124 (z. B. nichtflüchtigen Speicher, flüchtigen Speicher, Festplattenlaufwerk, Flash-Speicher, NAND, 3D-NAND, 3D-XPoint usw.) ein. Die Datenspeicherungsvorrichtung 124 kann über eine Speicherungsschnittstelle (z. B. SATA) oder über einen Peripheriebus, wie etwa einen Peripheral-Component-Interconnect-Bus (z. B. PCI, PCI Express), verbunden sein. Die Berührungssensoren 125 können Touchscreen-Sensoren, Drucksensoren oder Fingerabdrucksensoren einschließen. Der drahtlose Sendeempfänger 126 kann ein WiFi-Sendeempfänger, ein Bluetooth-Sendeempfänger oder ein Mobilnetz-Sendeempfänger sein, wie etwa ein 3G-, 4G-, 5G- oder Long-Term-Evolution(LTE)-Sendeempfänger. Die Firmware-Schnittstelle 128 ermöglicht die Kommunikation mit System-Firmware und kann beispielsweise eine vereinheitlichte erweiterbare Firmware-Schnittstelle (UEFI: Unified Extensible Firmware Interface) sein. Die Netzwerksteuerung 134 kann eine Netzwerkverbindung zu einem drahtgebundenen Netzwerk ermögli-

chen. In einigen Ausführungsformen ist eine Hochleistungsnetzwerksteuerung (nicht gezeigt) mit dem Schnittstellenbus 110 gekoppelt. Die Audiosteuerung 146 ist in einer Ausführungsform eine Mehrkanal-High-Definition-Audiosteuerung. In einer Ausführungsform weist das System 100 eine optionale Legacy-E/A-Steuerung 140 zum Koppeln von Legacy(z. B. Personal System 2 (PS/2))-Vorrichtungen mit dem System auf. Der Plattformsteuerungshub 130 kann auch mit einem oder mehreren USB(Universal Serial Bus)-Steuerungen 142 verbunden sein, welche Eingabevorrichtungen, wie etwa Kombinationen aus Tastatur und Maus 143, eine Kamera 144 oder andere USB-Eingabevorrichtungen, verbinden.

[0014] Es versteht sich, dass das gezeigte System 100 beispielhaft und nicht beschränkend ist, da andere Arten von Datenverarbeitungssystemen, die anders konfiguriert sind, ebenfalls verwendet werden können. Zum Beispiel kann eine Instanz der Speichersteuerung 116 und des Plattformsteuerungshubs 130 in einen diskreten externen Grafikprozessor, wie etwa den externen Grafikprozessor 118, integriert sein. In einer Ausführungsform können der Plattformsteuerungshub 130 und/oder die Speichersteuerung 116 extern zu dem einen oder den mehreren Prozessoren 102 sein. Zum Beispiel kann das System 100 eine externe Speichersteuerung 116 und einen Plattformsteuerungshub 130 beinhalten, die als ein Speichersteuerungshub und ein Peripheriesteuerungshub innerhalb eines Systemchipsatzes konfiguriert sein können, der mit dem einen oder den mehreren Prozessoren 102 in Kommunikation steht.

[0015] Zum Beispiel können Leiterplatten („Schlitten“) verwendet werden, auf denen Komponenten, wie etwa CPUs, Speicher und andere Komponenten, platziert sind, die für eine erhöhte thermische Leistungsfähigkeit gestaltet sind. Bei manchen Beispielen befinden sich Verarbeitungskomponenten, wie etwa die Prozessoren, auf einer Oberseite eines Schlittens, während sich naher Speicher, wie etwa DIMMs, auf einer Unterseite des Schlittens befinden. Infolge des verbesserten Luftstroms, der durch diese Gestaltung bereitgestellt wird, können die Komponenten bei höheren Frequenzen und Leistungspegeln als in typischen Systemen arbeiten, wodurch die Leistungsfähigkeit erhöht wird. Des Weiteren sind die Schlitten dazu konfiguriert, blind mit Strom- und Datenkommunikationskabeln in einem Rack zusammenzupassen, wodurch ihre Fähigkeit verbessert wird, schnell entfernt, aufgerüstet, wieder installiert und/oder ersetzt zu werden. Gleichermaßen sind individuelle auf den Schlitten befindliche Komponenten wie Prozessoren, Beschleuniger, Speicher und Datenspeicherungslaufwerke so konfiguriert, dass sie sich dank ihrer größeren Beabstandung zueinander leicht aufrüsten lassen. In der veranschaulichenden Ausführungsform beinhalten die Komponenten zusätzlich Hardwareattestierungsmerkmale, um ihre Authentizität nachzuweisen.

[0016] Ein Datenzentrum kann eine einzelne Netzwerkarchitektur („Fabric“) nutzen, die mehrere andere Netzwerkarchitekturen unterstützt, darunter Ethernet und Omni-Path. Die Schlitten können über optische Fasern mit Switches gekoppelt sein, die eine höhere Bandbreite und eine niedrigere Latenz als eine typische Twisted-Pair-Verkabelung (z. B. Kategorie 5, Kategorie 5e, Kategorie 6 usw.) bereitstellen. Aufgrund der hochbandbreitigen, latenzarmen Verbindungen und Netzwerkarchitektur kann das Datenzentrum im Gebrauch Ressourcen, wie etwa Speicher, Beschleuniger (z. B. GPUs, Grafikbeschleuniger, FPGAs, ASICs, Neuronalnetzwerk- und/oder Künstliche-Intelligenz-Beschleuniger usw.) und Datenspeicherungslaufwerke, die physisch getrennt sind, zusammenschließen und sie Rechenressourcen (z. B. Prozessoren) nach Bedarf bereitstellen, wodurch ermöglicht wird, dass die Rechenressourcen auf die gepoolten Ressourcen zugreifen, als wären sie lokal.

[0017] Eine Leistungsversorgung oder -quelle kann Spannung und/oder Strom an das System 100 oder eine beliebige Komponente oder ein beliebiges System, die/das hierin beschrieben wird, liefern. In einem Beispiel beinhaltet die Leistungsversorgung einen AC/DC(Wechselstrom-zu-Gleichstrom)-Adapter zum Einstecken in eine Wandsteckdose. Eine solche AC-Leistungsquelle kann eine erneuerbare Energiequelle (z. B. Solarstrom) sein. In einem Beispiel beinhaltet die Leistungsquelle eine DC-Leistungsquelle, wie etwa einen externen AC/DC-Wandler. In einem Beispiel schließt die Leistungsquelle oder Leistungsversorgung drahtlose Lade-Hardware zum Laden über die Nähe zu einem Ladefeld ein. In einem Beispiel kann die Leistungsquelle eine interne Batterie, eine Wechselstromversorgung, eine bewegungsbasierte Leistungsversorgung, eine Solarstromversorgung oder eine Brennstoffzellenquelle beinhalten.

[0018] **Fig. 2A-2D** veranschaulichen Rechensysteme und Grafikprozessoren, die durch hierin beschriebene Ausführungsformen bereitgestellt werden. Die Elemente der **Fig. 2A-2D** mit den gleichen Bezugszeichen (oder Bezeichnungen) wie die Elemente jeder anderen Figur hierin können auf irgendeine Weise ähnlich der an anderer Stelle hierin beschriebenen arbeiten oder funktionieren, sind aber nicht darauf beschränkt.

[0019] **Fig. 2A** ist ein Blockdiagramm einer Ausführungsform eines Prozessors 200, der einen oder mehrere Prozessorkerne 202A-202N, eine integrierte Speichersteuerung 214 und einen integrierten Grafikprozessor

208 aufweist. Der Prozessor 200 kann zusätzliche Kerne bis zu und einschließlich des zusätzlichen Kerns 202N beinhalten, die durch die gestrichelten Kästchen dargestellt sind. Jeder der Prozessorkerne 202A-202N beinhaltet eine oder mehrere interne Cacheeinheiten 204A-204N. In einigen Ausführungsformen hat jeder Prozessorkern auch Zugriff auf eine oder mehrere gemeinsam genutzte Cacheeinheiten 206. Die internen Cacheeinheiten 204A-204N und die gemeinsam genutzten Cacheeinheiten 206 repräsentieren eine Cachespeicherhierarchie innerhalb des Prozessors 200. Die Cachespeicherhierarchie kann mindestens eine Ebene („Level“) von Anweisungs- und Datencache innerhalb jedes Prozessorkerns und eine oder mehrere Ebenen von gemeinsam genutztem Mid-Level-Cache, wie etwa Level 2 (L2), Level 3 (L3), Level 4 (L4) oder andere Cacheebenen beinhalten, wobei die höchste Cacheebene vor dem externen Speicher als LLC klassifiziert wird. In einigen Ausführungsformen bewahrt eine Cachekohärenzlogik die Kohärenz zwischen den verschiedenen Cacheeinheiten 206 und 204A-204N.

[0020] In manchen Ausführungsformen kann der Prozessor 200 auch einen Satz von einer oder mehreren Bussteuerungseinheiten 216 und einen Systemagentenkern 210 beinhalten. Die eine oder die mehreren Bussteuerungseinheiten 216 verwalten einen Satz von Peripheriebussen, wie etwa einen oder mehrere PCI- oder PCI-Express-Busse. Der Systemagentenkern 210 bietet Verwaltungsfunktionalität für die verschiedenen Prozessorkomponenten. In einigen Ausführungsformen beinhaltet der Systemagentenkern 210 eine oder mehrere integrierte Speichersteuerungen 214 zum Verwalten des Zugriffs auf verschiedene externe Speichervorrichtungen (nicht gezeigt).

[0021] In einigen Ausführungsformen beinhalten einer oder mehrere der Prozessorkerne 202A-202N eine Unterstützung für simultanes Multithreading. In einer solchen Ausführungsform beinhaltet der Systemagentenkern 210 Komponenten zum Koordinieren und Betreiben der Kerne 202A-202N während einer Multithread-Verarbeitung. Der Systemagentenkern 210 kann zusätzlich eine Leistungssteuereinheit (PCU) beinhalten, die Logik und Komponenten beinhaltet, um den Leistungszustand der Prozessorkerne 202A - 202N und des Grafikprozessors 208 zu regulieren.

[0022] In einigen Ausführungsformen enthält der Prozessor 200 zusätzlich einen Grafikprozessor 208 zum Ausführen von Grafikverarbeitungsoperationen. In manchen Ausführungsformen ist der Grafikprozessor 208 mit dem Satz gemeinsam genutzter Cacheeinheiten 206 und dem Systemagentenkern 210 einschließlich der einen oder der mehreren integrierten Speichersteuerungen 214 gekoppelt. In einigen Ausführungsformen beinhaltet der Systemagentenkern 210 zudem eine Anzeigesteuerung 211 zum Steuern der Ausgabe des Grafikprozessors an eine oder mehrere gekoppelte Anzeigen. In einigen Ausführungsformen kann es sich bei der Anzeigesteuerung 211 auch um ein separates Modul handeln, das über mindestens ein Interconnect mit dem Grafikprozessor gekoppelt ist, oder sie kann in dem Grafikprozessor 208 integriert sein.

[0023] In einigen Ausführungsformen wird eine ringbasierte Interconnect-Einheit 212 verwendet, um die internen Komponenten des Prozessors 200 zu koppeln. Es kann jedoch auch eine alternative Interconnect-Einheit verwendet werden, wie etwa ein Punkt-zu-Punkt-Interconnect, ein geschaltetes Interconnect oder andere Techniken, einschließlich im Stand der Technik bekannter Techniken. In einigen Ausführungsformen ist der Grafikprozessor 208 über einen E/A-Link 213 mit dem Ring-Interconnect 212 gekoppelt.

[0024] Der beispielhafte E/A-Link 213 repräsentiert mindestens einen von mehreren Arten von E/A-Interconnects, einschließlich eines On-Package-E/A-Interconnect, das die Kommunikation zwischen verschiedenen Prozessorkomponenten und einem eingebetteten Hochleistungsspeichermodul 218 wie etwa einem eDRAM-Modul ermöglicht. In einigen Ausführungsformen können jeder der Prozessorkerne 202A-202N und der Grafikprozessor 208 eingebettete Speichermodule 218 als gemeinsam genutzten Last-Level-Cache verwenden.

[0025] In einigen Ausführungsformen sind die Prozessorkerne 202A-202N homogene Kerne, die die gleiche Anweisungssatzarchitektur ausführen. In einer anderen Ausführungsform sind die Prozessorkerne 202A-202N hinsichtlich der Anweisungssatzarchitektur (ISA) heterogen, wobei ein oder mehrere der Prozessorkerne 202A-202N einen ersten Anweisungssatz ausführen, während mindestens einer der anderen Kerne einen Teilsatz des ersten Anweisungssatzes oder einen anderen Anweisungssatz ausführt. In einer Ausführungsform sind die Prozessorkerne 202A-202N hinsichtlich der Mikroarchitektur heterogen, wobei ein oder mehrere Kerne mit einem relativ höheren Leistungsverbrauch mit einem oder mehreren Leistungskernen mit einem geringeren Leistungsverbrauch gekoppelt sind. In einer Ausführungsform sind die Prozessorkerne 202A-202N hinsichtlich der Rechenleistung heterogen. Außerdem kann der Prozessor 200 auf einem oder mehreren Chips oder als eine integrierte SoC-Schaltung mit den veranschaulichten Komponenten zusätzlich zu anderen Komponenten implementiert werden.

[0026] **Fig. 2B** ist ein Blockdiagramm einer Hardwarelogik eines Grafikprozessorkerns 219 gemäß manchen hierin beschriebenen Ausführungsformen. Elemente von **Fig. 2B** mit den gleichen Bezugszeichen (oder Bezeichnungen) wie die Elemente einer beliebigen anderen Figur hierin können auf beliebige Weise ähnlich der an anderer Stelle hierin beschriebenen arbeiten oder funktionieren, sind aber nicht darauf beschränkt. Der Grafikprozessorkern 219, der manchmal als ein Kern-Slice bezeichnet wird, kann ein oder mehrere Grafikkerne innerhalb eines modularen Grafikprozessors sein. Der Grafikprozessorkern 219 ist beispielhaft für ein Grafikkern-Slice, und ein Grafikprozessor, wie hierin beschrieben, kann mehrere Grafikkern-Slices auf der Grundlage von Zielleistungs- und Leistungsverhaltenshüllkurven beinhalten. Jeder Grafikprozessorkern 219 kann einen Festfunktionsblock 230 beinhalten, der mit mehreren Unterkernen 221A-221F, auch als Unter-Slices bezeichnet, die modulare Blöcke von Allzweck- und Festfunktionslogik beinhalten, gekoppelt ist.

[0027] In einigen Ausführungsformen beinhaltet der Festfunktionsblock 230 eine Geometrie-/Festfunktions-Pipeline 231, die durch alle Unterkerne in dem Grafikprozessorkern 219 gemeinsam genutzt werden kann, zum Beispiel bei Grafikprozessorimplementierungen mit geringerer Performanz und/oder geringerer Leistung. In verschiedenen Ausführungsformen beinhaltet die Geometrie-/Festfunktions-Pipeline 231 eine 3D-Festfunktions-Pipeline (z. B. 3D-Pipeline 312, wie nachstehend in **Fig. 3** und **Fig. 4** beschrieben), eine Video-Frontend-Einheit, einen Thread-Spawner und Thread-Dispatcher und einen Vereinheitlichter-Rückgabepuffer-Manager, der vereinheitlichte Rückgabepuffer (z. B. den vereinheitlichten Rückgabepuffer 418 in **Fig. 4** wie unten beschrieben) verwaltet.

[0028] In einer Ausführungsform beinhaltet der Festfunktionsblock 230 auch eine Grafik-SoC-Schnittstelle 232, einen Grafik-Mikrocontroller 233 und eine Medien-Pipeline 234. Die Grafik-SoC-Schnittstelle 232 stellt eine Schnittstelle zwischen dem Grafikprozessorkern 219 und anderen Prozessorkernen innerhalb einer integrierten System-on-Chip-Schaltung bereit. Der Grafik-Mikrocontroller 233 ist ein programmierbarer Subprozessor, der dazu konfigurierbar ist, verschiedene Funktionen des Grafikprozessorkerns 219 zu verwalten, einschließlich Thread-Dispatch, Scheduling und Präemption. Die Medien-Pipeline 234 (z. B. die Medien-Pipeline 316 von **Fig. 3** und **Fig. 4**) beinhaltet eine Logik, um das Decodieren, Codieren, Vorverarbeiten und/oder Nachverarbeiten von Multimediadaten, einschließlich Bild- und Videodaten, zu erleichtern. Die Medien-Pipeline 234 implementiert Medienoperationen über Anfragen zum Berechnen oder Sampeln einer Logik innerhalb der Unterkerne 221-221F.

[0029] In einer Ausführungsform ermöglicht die SoC-Schnittstelle 232 dem Grafikprozessorkern 219 die Kommunikation mit Prozessorkernen für Allzweckanwendungen (z. B. CPUs) und/oder anderen Komponenten innerhalb eines SoC, darunter Speicherhierarchieelemente wie etwa ein gemeinsam genutzter Last-Level-Cachespeicher, der System-RAM und/oder eingebetteter On-Chip- oder On-Package-DRAM. Die SoC-Schnittstelle 232 kann auch die Kommunikation mit Festfunktionseinrichtungen innerhalb des SoC ermöglichen, wie etwa Kamerabildgebungs-Pipelines, und ermöglicht die Verwendung von und/oder implementiert globale Speicher-Atome, die zwischen dem Grafikprozessorkern 219 und CPUs innerhalb des SoC gemeinsam genutzt werden können. Die SoC-Schnittstelle 232 kann zudem Leistungsverwaltungssteuerungen für den Grafikprozessorkern 219 implementieren und eine Schnittstelle zwischen einem Taktbereich des Grafikkerns 219 und anderen Taktbereichen innerhalb des SoC ermöglichen. In einer Ausführungsform ermöglicht die SoC-Schnittstelle 232 den Empfang von Befehlspuffern von einem Befehls-Streamer und einem globalen Thread-Dispatcher, die dazu konfiguriert sind, Befehle und Anweisungen an jeden von einem oder mehreren Grafikkernen innerhalb eines Grafikprozessors bereitzustellen. Die Befehle und Anweisungen können an die Medien-Pipeline 234 versendet werden, wenn Medienoperationen durchgeführt werden sollen, oder an eine Geometrie- und Festfunktions-Pipeline (z. B. Geometrie- und Festfunktions-Pipeline 231, Geometrie- und Festfunktions-Pipeline 237), wenn Grafikverarbeitungsoperationen durchgeführt werden sollen.

[0030] Der Grafik-Mikrocontroller 233 kann dafür konfiguriert sein, verschiedene Scheduling- und Verwaltungsaufgaben für den Grafikprozessorkern 219 durchzuführen. In einer Ausführungsform kann der Grafik-Mikrocontroller 233 Grafik- und/oder Rechenlast-Scheduling auf den verschiedenen parallelen Grafik-Engines innerhalb der Arrays 222A-222F, 224A-224F der Ausführungseinheiten (EU: Execution Unit) innerhalb der Unterkerne 221A-221F durchführen. In diesem Scheduling-Modell kann Host-Software, die auf einem CPU-Kern eines den Grafikprozessorkern 219 beinhaltenden SoC ausgeführt wird, Arbeitslasten eine von mehreren Grafikprozessor-Doorbells liefern, was eine Scheduling-Operation auf der entsprechenden Grafik-Engine aktiviert. Zu Scheduling-Operationen zählen Bestimmen, welche Arbeitslast als nächstes ausgeführt werden soll, Liefern einer Arbeitslast an einen Befehls-Streamer, Präemptieren bestehender Arbeitslasten, die auf einer Engine ausgeführt werden, Überwachen des Fortschritts einer Arbeitslast und Benachrichtigen einer Host-Software, wenn eine Arbeitslast abgeschlossen ist. In einer Ausführungsform kann der Grafik-Mikrocontroller 233 auch Niedrigleistungs- oder Ruhezustände für den Grafikprozessorkern 219 ermöglichen,

wodurch dem Grafikprozessorkern 219 die Fähigkeit gegeben wird, Register innerhalb des Grafikprozessorkerns 219 über Niedrigleistungszustandsübergänge hinweg unabhängig von dem Betriebssystem und/oder der Grafiktreibersoftware auf dem System zu speichern und wiederherzustellen.

[0031] Der Grafikprozessorkern 219 kann mehr als oder weniger als die veranschaulichten Unterkerne 221A-221F aufweisen, bis zu N modulare Unterkerne. Für jeden Satz von N Unterkernen kann der Grafikprozessorkern 219 auch Logik 235 mit gemeinsam genutzter Funktion, gemeinsam genutzten Speicher und/oder Cachespeicher 236, eine Geometrie-/Festfunktions-Pipeline 237 sowie zusätzliche Festfunktionslogik 238 zum Beschleunigen verschiedener Grafik- und Rechenverarbeitungsoperationen beinhalten. Die Logik 235 mit gemeinsam genutzter Funktion kann Logikeinheiten beinhalten, die mit der Logik 420 mit gemeinsam genutzter Funktion von **Fig. 4** assoziiert sind (z. B. Sampler-, Mathe- und/oder Inter-Thread-Kommunikationslogik), die durch jeden der N Unterkerne innerhalb des Grafikprozessorkerns 219 gemeinsam genutzt werden können. Der gemeinsam genutzte Speicher und/oder Cachespeicher 236 kann ein Last-Level-Cache für den Satz von N Unterkernen 221A-221F innerhalb des Grafikprozessorkerns 219 sein und kann auch als gemeinsam genutzter Speicher dienen, auf den mehrere Unterkerne zugreifen können. Die Geometrie-/Festfunktions-Pipeline 237 kann anstelle der Geometrie-/Festfunktions-Pipeline 231 innerhalb des Festfunktionsblocks 230 enthalten sein und dieselben oder ähnliche Logikeinheiten aufweisen.

[0032] In einer Ausführungsform beinhaltet der Grafikprozessorkern 219 zusätzliche Festfunktionslogik 238, die verschiedene Festfunktionsbeschleunigungslogik zur Verwendung durch den Grafikprozessorkern 219 beinhalten kann. In einer Ausführungsform beinhaltet die zusätzliche Festfunktionslogik 238 eine zusätzliche Geometrie-Pipeline zur Verwendung beim Nur-Positions-Shading. Beim Nur-Positions-Shading existieren zwei Geometrie-Pipelines, die vollständige Geometrie-Pipeline innerhalb der Geometrie-/Festfunktions-Pipeline 238, 231 und eine Cull-Pipeline, die eine zusätzliche Geometrie-Pipeline ist, die in der zusätzlichen Festfunktionslogik 238 enthalten sein kann. In einer Ausführungsform ist die Cull-Pipeline eine abgespeckte Version der vollständigen Geometrie-Pipeline. Die vollständige Pipeline und die Cull-Pipeline können verschiedene Instanzen derselben Anwendung ausführen, wobei jede Instanz einen separaten Kontext aufweist. Das Nur-Positions-Shading kann lange Aussortierdurchläufe von verworfenen Dreiecken verbergen, wodurch das Shading in manchen Fällen früher abgeschlossen werden kann. Zum Beispiel kann in einer Ausführungsform die Cull-Pipeline-Logik innerhalb der zusätzlichen Festfunktionslogik 238 Positions-Shader parallel zur Hauptanwendung ausführen und generiert im Allgemeinen kritische Ergebnisse schneller als die vollständige Pipeline, da die Cull-Pipeline nur das Positionsattribut der Vertices abruft und an diesen Shading durchführt, ohne Rasterisierung und Rendering der Pixel zum Framepuffer durchzuführen. Die Cull-Pipeline kann die erzeugten kritischen Ergebnisse verwenden, um Sichtbarkeitsinformationen für alle Dreiecke zu berechnen, ohne Rücksicht darauf, ob diese Dreiecke aussortiert werden. Die vollständige Pipeline (die in diesem Fall als Wiedergabe-Pipeline bezeichnet werden kann) kann die Sichtbarkeitsinformationen verbrauchen, um die aussortierten Dreiecke zu überspringen, um nur an den sichtbaren Dreiecken Shading durchzuführen, die schließlich an die Rasterisierungsphase übergeben werden.

[0033] In einer Ausführungsform kann die zusätzliche Festfunktionslogik 238 auch eine Maschinenlernbeschleunigungslogik, wie etwa eine Festfunktionsmatrixmultiplikationslogik, für Implementierungen einschließlich Optimierungen für Maschinenlerntraining oder Inferenzieren beinhalten.

[0034] Innerhalb jedes Grafikunterkerns 221A-221F ist ein Satz von Ausführungsressourcen enthalten, die verwendet werden können, um Grafik-, Medien- und Rechenoperationen als Reaktion auf Anforderungen durch eine Grafik-Pipeline, Medien-Pipeline oder Shader-Programme durchzuführen. Die Grafikunterkerne 221A-221F beinhalten mehrere EU-Arrays 222A-222F, 224A-224F, Thread-Dispatch- und Inter-Thread-Kommunikations(TD/IC)-Logik 223A-223F, einen 3D-Sampler (z. B. Textur-Sampler) 225A-225F, einen Medien-Sampler 206A-206F, einen Shader-Prozessor 227A-227F und einen gemeinsam genutzten lokalen Speicher (SLM: Shared Local Memory) 228A-228F. Die EU-Arrays 222A-222F, 224A-224F beinhalten jeweils mehrere Ausführungseinheiten, die Allzweck-Grafikverarbeitungseinheiten sind, die Gleitkomma- und Ganzzahl-/Festkommalogikoperationen im Dienst einer Grafik-, Medien- oder Rechenoperation durchführen können, einschließlich Grafik-, Medien- oder Berechnungs-Shader-Programmen. Die TD/IC-Logik 223A-223F führt lokale Thread-Dispatch- und Thread-Steueroperationen für die Ausführungseinheiten innerhalb eines Unterkerns aus und ermöglicht die Kommunikation zwischen Threads, die auf den Ausführungseinheiten des Unterkerns ausgeführt werden. Der 3D-Sampler 225A-225F kann Textur oder andere 3D-Grafik-bezogene Daten in den Speicher lesen. Der 3D-Sampler kann Texturdaten basierend auf einem konfigurierten Sample-Status und dem Texturformat, das mit einer gegebenen Textur assoziiert ist, unterschiedlich lesen. Der Medien-Sampler 206A-206F kann ähnliche Leseoperationen basierend auf der Art und dem Format durchführen, die mit den Mediendaten assoziiert sind. In einer Ausführungsform kann jeder Grafikunterkern 221A-221F abwechselnd

einen vereinheitlichten 3D- und Medien-Sampler beinhalten. Threads, die auf den Ausführungseinheiten in jedem der Unterkerne 221A-221F ausgeführt werden, können den gemeinsam genutzten lokalen Speicher 228A-228F in jedem Unterkern verwenden, um zu ermöglichen, dass Threads, die innerhalb einer Thread-Gruppe ausgeführt werden, unter Verwendung eines gemeinsamen On-Chip-Speicher-Pools ausgeführt werden.

[0035] **Fig. 2C** veranschaulicht eine Grafikverarbeitungseinheit (GPU) 239, die dedizierte Sätze von Grafikverarbeitungsressourcen beinhaltet, die in Mehrkerngruppen 240A-240N angeordnet sind. Während die Einzelheiten lediglich einer einzelnen Mehrkerngruppe 240A bereitgestellt sind, versteht es sich, dass die anderen Mehrkerngruppen 240B-240N mit denselben oder ähnlichen Sätzen von Grafikverarbeitungsressourcen ausgerüstet sein können.

[0036] Wie veranschaulicht, kann eine Mehrkerngruppe 240 A einen Satz von Grafikkernen 243, einen Satz von Tensorkernen 244 und einen Satz von Raytracing-Kernen 245 beinhalten. Ein Scheduler/Dispatcher 241 plant und sendet die Grafik-Threads zur Ausführung auf den verschiedenen Kernen 243, 244, 245 aus. Ein Satz von Registerdateien 242 speichert Operandenwerte, die von den Kernen 243, 244, 245 verwendet werden, wenn die Grafik-Threads ausgeführt werden. Diese können zum Beispiel Ganzzahlregister zum Speichern von Ganzzahlwerten, Gleitkommaregister zum Speichern von Gleitkommawerten, Vektorregister zum Speichern von gepackten Datenelementen (Ganzzahl- und/oder Gleitkommadatenelementen) und Kachelregister zum Speichern von Tensor-/Matrixwerten beinhalten. In einer Ausführungsform sind die Kachelregister als kombinierte Sätze von Vektorregistern implementiert.

[0037] Ein oder mehrere kombinierte Level-1(L1)-Caches und gemeinsam genutzte Speichereinheiten 247 speichern Grafikdaten, wie Texturdaten, Vertexdaten, Pixeldaten, Strahlendaten, Begrenzungsvolumendaten usw., lokal innerhalb jeder Mehrkerngruppe 240A. Eine oder mehrere Textureinheiten 247 können auch verwendet werden, um Texturierungsoperationen, wie etwa Texturabbildung und -Sampling, durchzuführen. Ein Level-2(L2)-Cache 253, der durch alle oder eine Teilmenge der Mehrkerngruppen 240A-240N gemeinsam genutzt wird, speichert Grafikdaten und/oder Anweisungen für mehrere gleichzeitige Grafik-Threads. Wie veranschaulicht, kann der L2-Cache 253 über mehrere Mehrkerngruppen 240A-240N hinweg gemeinsam genutzt werden. Eine oder mehrere Speichersteuerungen 248 koppeln die GPU 239 mit einem Speicher 249, der ein Systemspeicher (z. B. DRAM) und/oder ein dedizierter Grafikspeicher (z. B. GDDR6-Speicher) sein kann.

[0038] Eine Eingabe/Ausgabe(E/A)-Schaltungsanordnung 250 koppelt die GPU 239 mit einer oder mehreren E/A-Vorrichtungen 252, wie etwa Digitalsignalprozessoren (DSPs), Netzwerksteuerungen oder Benutzereingabevorrichtungen. Ein On-Chip-Interconnect kann verwendet werden, um die E/A-Vorrichtungen 252 mit der GPU 239 und dem Speicher 249 zu koppeln. Eine oder mehrere E/A-Speicherverwaltungseinheiten (IOMMUs) 251 der E/A-Schaltungsanordnung 250 koppeln die E/A-Vorrichtungen 252 direkt mit dem Systemspeicher 249. In einer Ausführungsform verwaltet die IOMMU 251 mehrere Sätze von Seitentabellen, um virtuelle Adressen auf physische Adressen im Systemspeicher 249 abzubilden. In dieser Ausführungsform können sich die E/A-Vorrichtungen 252, die CPU(s) 246 und die GPU(s) 239 denselben virtuellen Adressraum teilen.

[0039] In einer Implementierung unterstützt die IOMMU 251 Virtualisierung. In diesem Fall kann sie einen ersten Satz von Seitentabellen dahingehend verwalten, virtuelle Gast-/Grafikadressen auf physische Gast-/Grafikadressen abzubilden, und einen zweiten Satz von Seitentabellen dahingehend verwalten, die physischen Gast-/Grafikadressen auf physische System-/Hostadressen (z. B. innerhalb des Systemspeichers 249) abzubilden. Die Basisadressen sowohl des ersten als auch des zweiten Satzes von Seitentabellen können in Steuerregistern gespeichert werden und bei einem Kontextwechsel ausgelagert werden (z. B. sodass der neue Kontext Zugriff auf den relevanten Satz von Seitentabellen erhält). Obgleich dies in **Fig. 2C** nicht veranschaulicht ist, kann jeder der Kerne 243, 244, 245 und/oder die Mehrkerngruppen 240A-240N Übersetzungspuffer (TLBs: Translation Lookaside Buffers) beinhalten, um Virtuell-Gast-zu-Physisch-Gast-Übersetzungen, Physisch-Gast-zu-Physisch-Host-Übersetzungen und Virtuell-Gast-zu-Physisch-Host-Übersetzungen zu cachen.

[0040] In einer Ausführungsform sind die CPUs 246, GPUs 239 und E/A-Vorrichtungen 252 auf einem einzelnen Halbleiterchip und/oder Chip-Package integriert. Der veranschaulichte Speicher 249 kann auf demselben Chip integriert sein oder kann über eine chipexterne Schnittstelle mit den Speichersteuerungen 248 gekoppelt sein. Bei einer Implementierung umfasst der Speicher 249 einen GDDR6-Speicher, der denselben

virtuellen Adressraum wie andere physische Systemebenenspeicher nutzt, obgleich die zugrundeliegenden Prinzipien der Erfindung nicht auf diese spezielle Implementierung beschränkt sind.

[0041] In einer Ausführungsform weisen die Tensorkerne 244 mehrere Ausführungseinheiten auf, die speziell zum Durchführen von Matrixoperationen, die die grundlegende Rechenoperation sind, die zum Durchführen von Deep-Learning-Operationen verwendet wird, ausgebildet sind. Zum Beispiel können simultane Matrixmultiplikationsoperationen für neuronales Netzwerktraining und Inferenz verwendet werden. Die Tensorkerne 244 können eine Matrixverarbeitung unter Verwendung einer Vielzahl von Operandenpräzisionen durchführen, darunter Gleitkomma mit einfacher Präzision (z. B. 32 Bit), Gleitkomma mit halber Präzision (z. B. 16 Bit), Ganzzahlwörter (16 Bit), Bytes (8 Bit) und Halbbytes (4 Bits). In einer Ausführungsform extrahiert eine Neuronalnetzwerk-Implementierung Merkmale jeder gerenderten Szene, wobei potenziell Details aus mehreren Frames kombiniert werden, um ein qualitativ hochwertiges Endbild zu konstruieren.

[0042] Bei Deep-Learning-Implementierungen kann Parallelmatrix-Multiplikationsarbeit zur Ausführung auf den Tensorkernen 244 geplant werden. Insbesondere erfordert das Training neuronaler Netzwerke eine signifikante Anzahl von Matrix-Skalarprodukt-Operationen. Um eine Innenproduktformulierung einer N × N × N Matrixmultiplikation zu verarbeiten, können die Tensorkerne 244 mindestens N Skalarprodukt-Verarbeitungselemente beinhalten. Bevor die Matrixmultiplikation beginnt, wird eine ganze Matrix in Kachelregister geladen und wird pro Zyklus für N Zyklen mindestens eine Spalte einer zweiten Matrix geladen. In jedem Zyklus gibt es N Skalarprodukte, die verarbeitet werden.

[0043] Matrixelemente können in Abhängigkeit von der speziellen Implementierung mit unterschiedlichen Präzisionen gespeichert werden, darunter 16-Bit-Wörter, 8-Bit-Bytes (z. B. INT8) und 4-Bit-Halbbytes (z. B. INT4). Modi mit unterschiedlichen Präzisionen können für die Tensorkerne 244 spezifiziert werden, um sicherzustellen, dass die effizienteste Präzision für unterschiedliche Arbeitslasten verwendet wird (z. B. wie etwa Inferenzieren von Arbeitslasten, die Quantisierung zu Bytes und Halbbytes tolerieren können).

[0044] In einer Ausführungsform beschleunigen die Raytracing-Kerne 245 Raytracing-Operationen sowohl für Echtzeit-Raytracing- als auch für Nicht-Echtzeit-Raytracing-Implementierungen. Insbesondere beinhalten die Raytracing-Kerne 245 eine Strahltraversierungs-/-überschneidungsschaltungsanordnung zum Durchführen von Strahltraversierung unter Verwendung von Hüllkörperhierarchien (BVHs: Bounding Volume Hierarchies) und Identifizieren von Überschneidungen zwischen Strahlen und Primitiven, die in den BVH-Volumina enthalten sind. Die Raytracing-Kerne 245 können auch Schaltungsanordnungen zum Durchführen von Tiefenprüfung und -Culling (z. B. unter Verwendung eines Z-Puffers oder einer ähnlichen Anordnung) beinhalten. Bei einer Implementierung führen die Raytracing-Kerne 245 Traversierungs- und Überschneidungsoperationen in Übereinstimmung mit den hierin beschriebenen Bildentrauschungstechniken durch, von denen zumindest ein Teil auf den Tensorkernen 244 ausgeführt werden kann. In einer Ausführungsform implementieren die Tensorkerne 244 zum Beispiel ein neuronales Deep-Learning-Netzwerk zum Durchführen, umfassend einen lokalen Speicher 9010 (und/oder Systemspeicher),von Entrauschen der durch die Raytracing-Kerne 245 erzeugten Frames. Die CPU(s) 246, Grafikkerne 243 und/oder Raytracing-Kerne 245 können jedoch auch alle oder einen Teil der Entrauschungs- und/oder Deep-Learning-Algorithmen implementieren.

[0045] Zudem kann, wie oben beschrieben, ein verteilter Ansatz zur Rauschentfernung eingesetzt werden, bei dem sich die GPU 239 in einer Rechenvorrichtung befindet, die über ein Netzwerk oder ein Hochgeschwindigkeits-Interconnect mit anderen Rechenvorrichtungen gekoppelt ist. Bei dieser Ausführungsform teilen sich die miteinander verbundenen Rechenvorrichtungen neuronale Netzwerklern-/Trainingsdaten, um die Geschwindigkeit zu verbessern, mit der das Gesamtsystem lernt, eine Entrauschung für unterschiedliche Typen von Bildframes und/oder unterschiedliche Grafikanwendungen durchzuführen.

[0046] In einer Ausführungsform verarbeiten die Raytracing-Kerne 245 alle BVH-Traversierungen und Strahl-Primitiv-Überschneidungen, wodurch verhindert wird, dass die Grafikkerne 243 mit tausenden Anweisungen pro Strahl überlastet werden. In einer Ausführungsform weist jeder Raytracing-Kern 245 einen ersten Satz spezialisierter Schaltungsanordnungen zum Durchführen von Begrenzungsrahmentests (z. B. für Traversierungsoperationen) und einen zweiten Satz spezialisierter Schaltungsanordnungen zum Durchführen der Strahl-Dreieck-Überschneidungstests (z. B. kreuzende Strahlen, die durchlaufen wurden) auf. Somit kann in einer Ausführungsform die Mehrkerngruppe 240A einfach eine Strahlsonde aussenden, und die Raytracing-Kerne 245 führen unabhängig Strahltraversierung und -überschneidung durch und geben Trefferdaten (z. B. ein Treffer, kein Treffer, mehrere Treffer usw.) an den Thread-Kontext zurück. Die anderen Kerne 243, 244 werden freigegeben, um andere Grafik- oder Rechenarbeit durchzuführen, während die Raytracing-Kerne 245 die Traversierungs- und Überschneidungsoperationen durchführen.

[0047] In einer Ausführungsform beinhaltet jeder Raytracing-Kern 245 eine Traversierungseinheit zum Durchführen von BVH-Prüfungsoperationen und eine Überschneidungseinheit, die Strahl-Primitiv-Überschneidungsprüfungen durchführt. Die Überschneidungseinheit erzeugt eine „Treffer"-, „Kein-Treffer"- oder „Mehrfachtreffer"-Antwort, die sie dem geeigneten Thread bereitstellt. Während der Traversierungs- und Überschneidungsoperationen werden die Ausführungsressourcen der anderen Kerne (z. B. Grafikkerne 243 und Tensorkerne 244) freigegeben, um andere Arten von Grafikarbeit durchzuführen.

[0048] In einer speziellen Ausführungsform, die unten beschrieben ist, wird ein hybrider Rasterisierung/Raytracing-Ansatz verwendet, bei dem Arbeit zwischen den Grafikkernen 243 und Raytracing-Kernen 245 verteilt wird.

[0049] In einer Ausführungsform weisen die Raytracing-Kerne 245 (und/oder andere Kerne 243, 244) Hardwareunterstützung für einen Raytracing-Anweisungssatz, wie etwa DirectX Ray Tracing (DXR) von Microsoft, der einen DispatchRays-Befehl aufweist, sowie Strahlerzeugung, Nächstgelegener-Treffer-, Beliebiger-Treffer- und Fehltreffer-Shader, die die Zuweisung eindeutiger Sätze von Shadern und Texturen für jedes Objekt ermöglichen, auf. Eine andere Raytracing-Plattform, die durch die Raytracing-Kerne 245, Grafikkerne 243 und Tensorkerne 244 unterstützt werden kann, ist Vulkan 1.1.85. Es sei jedoch angemerkt, dass die zugrundeliegenden Prinzipien der Erfindung nicht auf irgendeine spezielle Raytracing-ISA beschränkt sind.

[0050] Im Allgemeinen können die verschiedenen Kerne 245, 244, 243 einen Raytracing-Anweisungssatz unterstützen, der Anweisungen/Funktionen für Strahlerzeugung, Nächstgelegener-Treffer, Beliebiger-Treffer, Strahl-Primitiv-Überschneidung, Primitiv-weise und hierarchische Begrenzungsrahmenkonstruktion, Fehltreffer, Visit und Ausnahmen aufweist. Genauer gesagt beinhaltet eine Ausführungsform Raytracing-Anweisungen zum Durchführen der folgenden Funktionen:

Strahlenerzeugung - Strahlenerzeugungsanweisungen können für jedes Pixel, jedes Sample oder jede andere benutzerdefinierte Arbeitszuweisung ausgeführt werden.

Nächstgelegener-Treffer - Eine Nächstgelegener-Treffer-Anweisung kann ausgeführt werden, um die nächstgelegene Überschneidung eines Strahls mit Primitiven innerhalb einer Szene zu lokalisieren.

Beliebiger-Treffer - Eine Beliebiger-Treffer-Anweisung identifiziert mehrere Überschneidungen zwischen einem Strahl und Primitiven innerhalb einer Szene, um potenziell eine neue nächstgelegene Überschneidung zu identifizieren.

Überschneidung - Eine Überschneidungsanweisung führt eine Strahl-Primitiven-Überschneidungsprüfung durch und gibt ein Ergebnis aus.

Primitiv-weise Begrenzungsrahmenkonstruktion - Diese Anweisung erstellt einen Begrenzungsrahmen um ein gegebenes Primitiv oder eine gegebene Gruppe von Primitiven herum (z. B. wenn eine neue BVH- oder andere Beschleunigungsdatenstruktur erstellt wird).

Fehltreffer - gibt an, dass ein Strahl alle Geometrie innerhalb einer Szene oder ein spezifiziertes Gebiet einer Szene verfehlt.

Visit - gibt die Nachfolgevolumina an, die ein Strahl durchlaufen wird.

Ausnahmen - beinhalten verschiedene Typen von Ausnahme-Handler (z. B. für verschiedene Fehlerbedingungen aufgerufen).

[0051] Fig. 2D ist ein Blockdiagramm einer Allzweck-Grafikverarbeitungseinheit (GPGPU) 270, die als ein Grafikprozessor und/oder Rechenbeschleuniger konfiguriert sein kann, gemäß hierin beschriebenen Ausführungsformen. Die GPGPU 270 kann über einen oder mehrere System- und/oder Speicherbusse mit Hostprozessoren (z. B. einer oder mehreren CPUs 246) und Speicher 271, 272 verbunden sein. In einer Ausführungsform ist der Speicher 271 Systemspeicher, der mit der einen oder den mehreren CPUs 246 gemeinsam genutzt werden kann, während der Speicher 272 Vorrichtungsspeicher ist, der der GPGPU 270 dediziert ist. In einer Ausführungsform können Komponenten innerhalb der GPGPU 270 und des Vorrichtungsspeichers 272 in Speicheradressen abgebildet werden, die für die eine oder die mehreren CPUs 246 zugänglich sind. Der Zugriff auf die Speicher 271 und 272 kann über eine Speichersteuerung 268 ermöglicht werden. In einer Ausführungsform beinhaltet die Speichersteuerung 268 eine interne Direktspeicherzugriff(DMA)-Steuerung 269 oder kann Logik zum Durchführen von Operationen beinhalten, die ansonsten durch eine DMA-Steuerung durchgeführt würden.

[0052] Die GPGPU 270 enthält mehrere Cachespeicher, einschließlich eines L2-Cache 253, L1-Cache 254, eines Anweisungscache 255 und eines gemeinsam genutzten Speichers 256, von dem zumindest ein Teil auch als Cachespeicher partitioniert werden kann. Die GPGPU 270 enthält außerdem mehrere Berechnungseinheiten 260A-260N. Jede Berechnungseinheit 260A-260N beinhaltet einen Satz von Vektorregistern 261, Skalarregister 262, Vektorlogikeinheiten 263 und Skalarlogikeinheiten 264. Die Berechnungseinheiten 260A-260N können auch einen lokalen gemeinsam genutzten Speicher 265 und einen Programmzähler 266 beinhalten. Die Berechnungseinheiten 260A-260N können mit einem konstanten Cache 267 gekoppelt werden, der zum Speichern von konstanten Daten verwendet werden kann, die Daten sind, die sich während der Ausführung eines Kernel- oder Shader-Programms, das auf der GPGPU 270 ausgeführt wird, nicht ändern. In einer Ausführungsform ist der konstante Cache 267 ein Skalardaten-Cache und gecachte Daten können direkt in die Skalarregister 262 abgerufen werden.

[0053] Während des Betriebs können die eine oder die mehreren CPUs 246 Befehle in Register oder Speicher in der GPGPU 270 schreiben, die in einen zugänglichen Adressraum abgebildet wurde. Die Befehlsprozessoren 257 können die Befehle aus Registern oder dem Speicher lesen und bestimmen, wie diese Befehle innerhalb der GPGPU 270 verarbeitet werden. Ein Thread-Dispatcher 258 kann dann verwendet werden, um Threads an die Berechnungseinheiten 260A-260N zu senden, um diese Befehle auszuführen. Jede Berechnungseinheit 260A-260N kann Threads unabhängig von den anderen Berechnungseinheiten ausführen. Zusätzlich dazu kann jede Berechnungseinheit 260A-260N unabhängig für eine bedingte Berechnung konfiguriert sein und kann die Ergebnisse der Berechnung bedingt an den Speicher ausgeben. Die Befehlsprozessoren 257 können die eine oder die mehreren CPU 246 unterbrechen, wenn die übermittelten Befehle abgeschlossen sind.

[0054] **Fig. 3A-3C** veranschaulichen Blockdiagramme zusätzlicher Grafikprozessor- und Berechnungsbeschleunigerarchitekturen, die durch hierin beschriebene Ausführungsformen bereitgestellt werden. Die Elemente der **Fig. 3A-3C** mit den gleichen Bezugszeichen (oder Bezeichnungen) wie die Elemente jeder anderen Figur hierin können auf irgendeine Weise ähnlich der an anderer Stelle hierin beschriebenen arbeiten oder funktionieren, sind aber nicht darauf beschränkt.

[0055] **Fig. 3A** ist ein Blockdiagramm eines Grafikprozessors 300, der eine diskrete Grafikverarbeitungseinheit sein kann oder ein Grafikprozessor sein kann, der mit mehreren Verarbeitungskernen integriert ist, oder anderen Halbleitervorrichtungen, wie etwa unter anderem Speichervorrichtungen oder Netzwerkschnittstellen. In einigen Ausführungsformen kommuniziert der Grafikprozessor über eine speicherabgebildete E/A-Schnittstelle mit Registern auf dem Grafikprozessor und mit Befehlen, die in den Prozessorspeicher platziert werden. In einigen Ausführungsformen beinhaltet der Grafikprozessor 300 eine Speicherschnittstelle 314, um auf Speicher zuzugreifen. Die Speicherschnittstelle 314 kann eine Schnittstelle zu einem lokalen Speicher, einem oder mehreren internen Caches, einem oder mehreren gemeinsam genutzten externen Caches und/oder einem Systemspeicher sein.

[0056] In manchen Ausführungsformen beinhaltet der Grafikprozessor 300 auch eine Anzeigesteuerung 302, um Anzeigeausgabedaten zu einer Anzeigevorrichtung 318 anzusteuern. Die Anzeigesteuerung 302 beinhaltet Hardware für eine oder mehrere Überlagerungsebenen für die Anzeige und Zusammensetzung mehrerer Schichten von Video- oder Benutzeroberflächenelementen. Die Anzeigevorrichtung 318 kann eine interne oder externe Anzeigevorrichtung sein. In einer Ausführungsform ist die Anzeigevorrichtung 318 eine am Kopf angebrachte Anzeigevorrichtung, wie etwa eine VR(Virtual Reality)-Anzeigevorrichtung oder eine AR(Augmented Reality)-Anzeigevorrichtung. In einigen Ausführungsformen beinhaltet der Grafikprozessor 300 eine Video-Codec-Engine 306 zum Codieren, Decodieren oder Transcodieren von Medien zu, von oder zwischen einem oder mehreren Mediencodierformaten, darunter unter anderem MPEG-Formate (MPEG: Moving Picture Experts Group) wie MPEG-2, AVC-Formate (AVC: Advanced Video Coding) wie H.264/MPEG-4 AVC, H.265/HEVC, Alliance for Open Media (AOMedia) VP8, VP9 sowie die SMPTE-421M/VC-1- (SMPTE: Society of Motion Picture & Television Engineers) und JPEG-Formate (JPEG: Joint Photographic Experts Group) wie JPEG- und Motion-JPEG(MJPEG)-Formate.

[0057] In manchen Ausführungsformen beinhaltet der Grafikprozessor 300 eine Blockbildtransfer(BLIT: Block Image Transfer)-Engine 304, um zweidimensionale (2D) Rasterisierer-Operationen, einschließlich zum Beispiel Bitgrenzenblocktransfers, durchzuführen. In einer Ausführungsform werden jedoch 2D-Grafikoperationen unter Verwendung einer oder mehrerer Komponenten der Grafikverarbeitungs-Engine (GPE) 310 ausgeführt. In manchen Ausführungsformen ist die GPE 310 eine Berechnungs-Engine zum Durchführen von Grafikoperationen, einschließlich dreidimensionaler (3D) Grafikoperationen und Medienoperationen.

[0058] In manchen Ausführungsformen beinhaltet die GPE 310 eine 3D-Pipeline 312 zum Durchführen von 3D-Operationen, wie etwa Rendering von dreidimensionalen Bildern und Szenen unter Verwendung von Verarbeitungsfunktionen, die auf 3D-Primitivformen (z. B. Rechteck, Dreieck usw.) wirken. Die 3D-Pipeline 312 beinhaltet programmierbare und feste Funktionselemente, die verschiedene Aufgaben innerhalb des Elements durchführen und/oder Ausführungs-Threads zu einem 3D/Medien-Subsystem 315 erzeugen. Während die 3D-Pipeline 312 zum Durchführen von Medienoperationen verwendet werden kann, beinhaltet eine Ausführungsform der GPE 310 auch eine Medien-Pipeline 316, die speziell zum Durchführen von Medienoperationen, wie etwa Videonachbearbeitung und Bildverbesserung, verwendet wird.

[0059] In manchen Ausführungsformen beinhaltet die Medien-Pipeline 316 Logikeinheiten, die eine feste Funktion aufweisen oder programmierbar sind, um eine oder mehrere spezialisierte Medienoperationen, wie etwa eine Videodecodierungsbeschleunigung, Videoentschachtelung und Videocodierungsbeschleunigung, anstelle von oder im Auftrag der Video-Codec-Engine 306 durchzuführen. In einigen Ausführungsformen beinhaltet die Medien-Pipeline 316 zusätzlich eine Thread-Spawning-Einheit auf, um Threads zur Ausführung auf dem 3D-/Medien-Subsystem 315 zu spawnen. Die gespawnten Threads führen Berechnungen für die Medienoperationen auf einer oder mehreren Grafikausführungseinheiten durch, die in dem 3D/Medien-Subsystem 315 enthalten sind.

[0060] In einigen Ausführungsformen weist das 3D-Medien-Subsystem 315 Logik zum Ausführen von Threads auf, die durch die 3D-Pipeline 312 und die Medien-Pipeline 316 gespawnt werden. In einer Ausführungsform senden die Pipelines Thread-Ausführungsanforderungen an das 3D-/Medien-Subsystem 315, das Thread-Dispatch-Logik zum Arbitrieren und Versenden der verschiedenen Anforderungen an verfügbare Thread-Ausführungsressourcen beinhaltet. Die Ausführungsressourcen beinhalten ein Array von Grafikausführungseinheiten zum Verarbeiten der 3D- und Medien-Threads. In einigen Ausführungsformen weist das 3D-/Media-Subsystem 315 einen oder mehrere interne Caches für Thread-Anweisungen und Daten auf. In einigen Ausführungsformen enthält das Subsystem auch gemeinsam genutzten Speicher, einschließlich Registern und adressierbarem Speicher, um Daten zwischen Threads gemeinsam zu nutzen und Ausgabedaten zu speichern.

[0061] **Fig. 3B** veranschaulicht einen Grafikprozessor 320 mit einer gekachelten Architektur gemäß hierin beschriebenen Ausführungsformen. In einer Ausführungsform beinhaltet der Grafikprozessor 320 einen Grafikverarbeitungs-Engine-Cluster 322, der mehrere Instanzen der Grafikverarbeitungs-Engine 310 von **Fig. 3A** innerhalb einer Grafik-Engine-Kachel 310A-310D aufweist. Jede Grafik-Engine-Kachel 310A-310D kann über einen Satz von Kachel-Interconnects 323A-323F miteinander verbunden sein. Jede Grafik-Engine-Kachel 310A-310D kann auch über Speicher-Interconnects 325A-325D mit einem Speichermodul oder einer Speichervorrichtung 326A-326D verbunden sein. Die Speichervorrichtungen 326A-326D können eine beliebige Grafikspeichertechnologie verwenden. Zum Beispiel können die Speichervorrichtungen 326A - 326D ein Grafikspeicher mit doppelter Datenrate (GDDR) sein. Die Speichervorrichtungen 326A-326D sind in einer Ausführungsform Speichermodule mit hoher Bandbreite (HBM-Module), die sich auf einem Die mit ihrer jeweiligen Grafik-Engine-Kachel 310A-310D befinden können. In einer Ausführungsform sind die Speichervorrichtungen 326A-326D gestapelte Speichervorrichtungen, die auf ihrer jeweiligen Grafik-Engine-Kachel 310A-310D gestapelt werden können. In einer Ausführungsform befinden sich jede Grafik-Engine-Kachel 310A-310D und der assoziierte Speicher 326A-326D auf separaten Chiplets, die an einen Basis-Die oder ein Basissubstrat gebondet sind, wie ausführlicher in den **Fig. 11B-11D** beschrieben ist.

[0062] Der Grafikverarbeitungs-Engine-Cluster 322 kann mit einem On-Chip- oder On-Package-Fabric-Interconnect 324 verbunden sein. Das Fabric-Interconnect 324 kann die Kommunikation zwischen den Grafik-Engine-Kacheln 310A-310D und Komponenten wie dem Video-Codec 306 und einer oder mehreren Kopier-Engines 304 ermöglichen. Die Kopier-Engines 304 können verwendet werden, um Daten aus den, in die und zwischen den Speichervorrichtungen 326A - 326D und einem Speicher extern zu dem Grafikprozessor 320 (z. B. Systemspeicher) zu verschieben. Das Fabric-Interconnect 324 kann auch verwendet werden, um die Grafik-Engine-Kacheln 310A-310D miteinander zu verbinden. Der Grafikprozessor 320 kann optional eine Anzeigesteuerung 302 beinhalten, um eine Verbindung mit einer externen Anzeigevorrichtung 318 zu ermöglichen. Der Grafikprozessor kann auch als Grafik- oder Berechnungsbeschleuniger konfiguriert sein. In der Beschleunigerkonfiguration können die Anzeigesteuerung 302 und die Anzeigevorrichtung 318 weggelassen werden.

[0063] Der Grafikprozessor 320 kann über eine Hostschnittstelle 328 mit einem Hostsystem verbunden sein. Die Hostschnittstelle 328 kann eine Kommunikation zwischen dem Grafikprozessor 320, dem Systemspei-

cher und/oder anderen Systemkomponenten ermöglichen. Die Hostschnittstelle 328 kann zum Beispiel ein PCI-Express-Bus oder eine andere Art von Hostsystemschnittstelle sein.

[0064] Fig. 3C veranschaulicht einen Berechnungsbeschleuniger 330 gemäß hierin beschriebenen Ausführungsformen. Der Berechnungsbeschleuniger 330 kann architektonische Ähnlichkeiten mit dem Grafikprozessor 320 von **Fig. 3B** aufweisen und ist für die Berechnungsbeschleunigung optimiert. Ein Berechnungs-Engine-Cluster 332 kann einen Satz von Berechnungs-Engine-Kacheln 340A - 340D beinhalten, die eine Ausführungslogik beinhalten, die für parallele oder vektorbasierte Allzweck-Rechenoperationen optimiert ist. In einigen Ausführungsformen beinhalten die Berechnungs-Engine-Kacheln 340A-340D keine Grafikverarbeitungslogik mit fester Funktion, obwohl in einer Ausführungsform eine oder mehrere der Berechnungs-Engine-Kacheln 340A-340D Logik zum Durchführen einer Medienbeschleunigung beinhalten können. Die Berechnungs-Engine-Kacheln 340A-340D können über Speicher-Interconnects 325A-325D mit dem Speicher 326A-326D verbunden werden. Der Speicher 326A - 326D und die Speicher-Interconnects 325A - 325D können eine ähnliche Technologie wie im Grafikprozessor 320 sein oder können anders sein. Die Grafikberechnungs-Engine-Kacheln 340A-340D können auch über einen Satz von Kachel-Interconnects 323A-323F miteinander verbunden sein und können mit einem Fabric-Interconnect 324 verbunden und/oder durch dieses miteinander verbunden sein. In einer Ausführungsform beinhaltet der Berechnungsbeschleuniger 330 einen großen L3-Cache 336, der als ein vorrichtungsumfassender Cache konfiguriert werden kann. Der Berechnungsbeschleuniger 330 kann auch über eine Hostschnittstelle 328 in ähnlicher Weise wie der Grafikprozessor 320 von **Fig. 3B** mit einem Hostprozessor und Speicher verbunden sein.

Grafikverarbeitungs-Engine

[0065] Fig. 4 ist ein Blockdiagramm einer Grafikverarbeitungs-Engine 410 eines Grafikprozessors gemäß manchen Ausführungsformen. In einer Ausführungsform ist die Grafikverarbeitungs-Engine (GPE) 410 eine Version der in **Fig. 3A** gezeigten GPE 310 und kann auch eine Grafik-Engine-Kachel 310A-310D von **Fig. 3B** repräsentieren. Elemente von **Fig. 4** mit den gleichen Bezugszeichen (oder Bezeichnungen) wie die Elemente einer beliebigen anderen Figur hierin können auf beliebige Weise ähnlich der an anderer Stelle hierin beschriebenen arbeiten oder funktionieren, sind aber nicht darauf beschränkt. Zum Beispiel sind die 3D-Pipeline 312 und die Medien-Pipeline 316 von **Fig. 3A** veranschaulicht. Die Medien-Pipeline 316 ist in einigen Ausführungsformen der GPE 410 optional und ist möglicherweise nicht explizit in der GPE 410 enthalten. Zum Beispiel und in mindestens einer Ausführungsform ist ein separater Medien- und/oder Bildprozessor mit der GPE 410 gekoppelt.

[0066] In einigen Ausführungsformen ist die GPE 410 mit einem Befehls-Streamer 403 gekoppelt oder beinhaltet diesen, der der 3D-Pipeline 312 und/oder den Medien-Pipelines 316 einen Befehlsstrom bereitstellt. In manchen Ausführungsformen ist der Befehls-Streamer 403 mit einem Speicher, der ein Systemspeicher sein kann, oder einem internen Cachespeicher und/oder einem gemeinsam genutzten Cachespeicher gekoppelt. In manchen Ausführungsformen empfängt der Befehls-Streamer 403 Befehle von dem Speicher und sendet die Befehle an die 3D-Pipeline 312 und/oder die Medien-Pipeline 316. Bei den Befehlen handelt es sich um Direktiven, die aus einem Ringpuffer abgerufen werden, der Befehle für die 3D-Pipeline 312 und die Medien-Pipeline 316 speichert. In einer Ausführungsform kann der Ringpuffer zusätzlich Batch-Befehlspuffer aufweisen, die Batches aus mehreren Befehlen speichern. Die Befehle für die 3D-Pipeline 312 können auch Verweise auf Daten beinhalten, die im Speicher gespeichert sind, wie etwa unter anderem Vertex- und Geometriedaten für die 3D-Pipeline 312 und/oder Bilddaten und Speicherobjekte für die Medien-Pipeline 316. Die 3D-Pipeline 312 und die Medien-Pipeline 316 verarbeiten die Befehle und Daten durch Durchführen von Operationen über Logik innerhalb der jeweiligen Pipelines oder durch Versenden eines oder mehrerer Ausführungs-Threads an ein Grafikkernarray 414. In einer Ausführungsform beinhaltet das Grafikkernarray 414 einen oder mehrere Blöcke von Grafikkernen (z. B. einen oder mehrere Grafikkerne 415A, einen oder mehrere Grafikkerne 415B), wobei jeder Block einen oder mehrere Grafikkerne beinhaltet. Jeder Grafikkern beinhaltet einen Satz von Grafikausführungsressourcen, der eine Allzweck- und grafikspezifische Ausführungslogik zum Ausführen von Grafik- und Rechenoperationen sowie Festfunktionstexturverarbeitung und/oder Maschinenlernen und eine Künstliche-Intelligenz-Beschleunigungslogik beinhaltet.

[0067] In verschiedenen Ausführungsformen kann die 3D-Pipeline 312 Festfunktions- und programmierbare Logik beinhalten, um ein oder mehrere Shader-Programme, wie etwa Vertex-Shader, Geometrie-Shader, Pixel-Shader, Fragment-Shader, Berechnungs-Shader oder andere Shader-Programme zu verarbeiten, indem die Anweisungen verarbeitet und Ausführungs-Threads an das Grafikkern-Array 414 versendet werden. Das Grafikkernarray 414 stellt einen vereinheitlichten Block von Ausführungsressourcen zur Verwendung bei der Verarbeitung dieser Shader-Programme bereit. Eine Allzweckausführungslogik (z. B. Ausfüh-

rungseinheiten) innerhalb des einen oder der mehreren Grafikkerne 415A-414B des Grafikkernarrays 414 unterstützt verschiedene 3D-API-Shader-Sprachen und kann mehrere simultane Ausführungs-Threads ausführen, die mit mehreren Shadern assoziiert sind.

[0068] In einigen Ausführungsformen beinhaltet das Grafikkernarray 414 Ausführungslogik zum Durchführen von Medienfunktionen, wie etwa Video- und/oder Bildverarbeitung. In einer Ausführungsform beinhalten die Ausführungseinheiten Allzwecklogik, die programmierbar ist, parallele Allzweckberechnungsoperationen zusätzlich zu Grafikverarbeitungsoperationen durchzuführen. Die Allzwecklogik kann Verarbeitungsoperationen parallel oder in Verbindung mit der Allzwecklogik innerhalb des einen bzw. der mehreren Prozessorkerne 107 von **Fig. 1** oder des Kerns 202A-202N wie in **Fig. 2A** durchführen.

[0069] Ausgabedaten, die durch Threads erzeugt werden, die auf dem Grafikkernarray 414 ausgeführt werden, können Daten an einen Speicher in einem vereinheitlichten Rückgabepuffer (URB) 418 ausgeben. Der URB 418 kann Daten für mehrere Threads speichern. In einigen Ausführungsformen kann der URB 418 verwendet werden, um Daten zwischen verschiedenen Threads zu senden, die auf dem Grafikkernarray 414 ausgeführt werden. In manchen Ausführungsformen kann der URB 418 zusätzlich zur Synchronisation zwischen Threads auf dem Grafikkernarray und einer Festfunktionslogik innerhalb der Logik 420 mit gemeinsam genutzter Funktion verwendet werden.

[0070] In manchen Ausführungsformen ist das Grafikkernarray 414 skalierbar, sodass das Array eine variable Anzahl von Grafikkernen mit jeweils einer variablen Anzahl an Ausführungseinheiten, die auf der Zielleistung und der Performanzstufe der GPE 410 basiert, beinhaltet. In einer Ausführungsform sind die Ausführungsressourcen dynamisch skalierbar, sodass Ausführungsressourcen nach Bedarf aktiviert oder deaktiviert werden können.

[0071] Das Grafikkernarray 414 ist mit Logik 420 mit gemeinsam genutzter Funktion gekoppelt, die mehrere Ressourcen beinhaltet, die unter den Grafikkernen in dem Grafikkernarray gemeinsam genutzt werden. Die gemeinsam genutzten Funktionen innerhalb der Logik 420 mit gemeinsam genutzter Funktion sind Hardware-Logikeinheiten, die dem Grafikkernarray 414 eine spezialisierte Zusatzfunktionalität bereitstellen. In verschiedenen Ausführungsformen beinhaltet die Logik 420 mit gemeinsam genutzter Funktion, ohne darauf beschränkt zu sein, eine Sampler- 421, eine Mathe- 422 und eine Inter-Thread-Kommunikations(ITC: Inter-Thread Communication)-Logik 423. Zusätzlich implementieren manche Ausführungsformen einen oder mehrere Caches 425 innerhalb der Logik 420 mit gemeinsam genutzter Funktion.

[0072] Eine gemeinsam genutzte Funktion wird zumindest in einem Fall implementiert, in dem die Anforderung für eine gegebene spezialisierte Funktion für die Aufnahme in das Grafikkernarray 414 unzureichend ist. Stattdessen wird eine einzelne Instanziierung dieser spezialisierten Funktion als eine eigenständige Entität in der Logik 420 mit gemeinsam genutzter Funktion implementiert und unter den Ausführungsressourcen innerhalb des Grafikkernarrays 414 gemeinsam genutzt. Der genaue Satz von Funktionen, die von dem Grafikkernarray 414 gemeinsam genutzt werden und im Grafikkernarray 414 enthalten sind, variiert zwischen den Ausführungsformen. In einigen Ausführungsformen können spezifische gemeinsam genutzte Funktionen innerhalb der Logik 420 mit gemeinsam genutzter Funktion, die umfangreich von dem Grafikkernarray 414 verwendet werden, innerhalb der Logik 416 mit gemeinsam genutzter Funktion innerhalb des Grafikkernarrays 414 enthalten sein. In verschiedenen Ausführungsformen kann die Logik 416 mit gemeinsam genutzter Funktion innerhalb des Grafikkernarrays 414 einen Teil der oder die gesamte Logik innerhalb der Logik 420 mit gemeinsam genutzter Funktion beinhalten. In einer Ausführungsform können alle Logikelemente innerhalb der Logik 420 mit gemeinsam genutzter Funktion innerhalb der Logik 416 mit gemeinsam genutzter Funktion des Grafikkernarrays 414 dupliziert werden. In einer Ausführungsform ist die Logik 420 mit gemeinsam genutzter Funktion zugunsten der Logik 416 mit gemeinsam genutzter Funktion innerhalb des Grafikkernarrays 414 ausgeschlossen.

Ausführungseinheiten

[0073] **Fig. 5A-5B** veranschaulichen Thread-Ausführungslogik 500 einschließlich eines Arrays von Verarbeitungselementen, die in einem Grafikprozessorkern eingesetzt werden, gemäß hierin beschriebenen Ausführungsformen. Elemente der **Fig. 5A-5B** mit den gleichen Bezugszeichen (oder Bezeichnungen) wie die Elemente einer beliebigen anderen Figur hierin können auf beliebige Weise ähnlich der an anderer Stelle hierin beschriebenen arbeiten oder funktionieren, sind aber nicht darauf beschränkt. Die **Fig. 5A-5B** veranschaulichen eine Übersicht über die Thread-Ausführungslogik 500, die für eine Hardwarelogik repräsentativ ist, die mit jedem Unterkern 221A-221F von **Fig. 2B** veranschaulicht ist. **Fig. 5A** ist repräsentativ für eine Ausfüh-

rungseinheit innerhalb eines Allzweck-Grafikprozessors, während **Fig. 5B** für eine Ausführungseinheit repräsentativ ist, die in einem Berechnungsbeschleuniger verwendet werden kann.

[0074] Wie in **Fig. 5A** veranschaulicht, beinhaltet die Thread-Ausführungslogik 500 in einigen Ausführungsformen einen Shader-Prozessor 502, einen Thread-Dispatcher 504, einen Anweisungscache 506, ein skalierbares Ausführungseinheitenarray mit mehreren Ausführungseinheiten 508A-508N, einen Sampler 510, gemeinsam genutzten lokalen Speicher 511, einen Datencache 512 und einen Datenport 514. In einer Ausführungsform kann das skalierbare Ausführungseinheitenarray dynamisch skalieren, indem eine oder mehrere Ausführungseinheiten (z. B. eine beliebige der Ausführungseinheiten 508A, 508B, 508C, 508D bis 508N-1 und 508N) basierend auf den Berechnungsanforderungen einer Arbeitslast aktiviert oder deaktiviert werden. In einer Ausführungsform sind die enthaltenen Komponenten über ein Interconnect-Fabric, das mit jeder der Komponenten verknüpft ist, miteinander verbunden. In einigen Ausführungsformen beinhaltet die Thread-Ausführungslogik 500 eine oder mehrere Verbindungen zum Speicher wie etwa zum Systemspeicher oder Cachespeicher über den Anweisungscache 506 und/oder den Datenport 514 und/oder den Sampler 510 und/oder die Ausführungseinheiten 508A-508N. In manchen Ausführungsformen ist jede Ausführungseinheit (z. B. 508A) eine selbständige programmierbare Allzweck-Recheneinheit, die in der Lage ist, mehrere simultane Hardware-Threads auszuführen, während sie mehrere Datenelemente parallel für jeden Thread verarbeitet. In verschiedenen Ausführungsformen kann das Array aus Ausführungseinheiten 508A-508N skalierbar sein, sodass es eine beliebige Anzahl einzelner Ausführungseinheiten enthält.

[0075] In manchen Ausführungsformen werden die Ausführungseinheiten 508A-508N hauptsächlich zum Ausführen von Shader-Programmen verwendet. Ein Shader-Prozessor 502 kann die verschiedenen Shader-Programme verarbeiten und Ausführungs-Threads, die mit den Shader-Programmen assoziiert sind, über einen Thread-Dispatcher 504 versenden. In einer Ausführungsform beinhaltet der Thread-Dispatcher Logik zum Arbitrieren von Thread-Initiierungsanforderungen von den Grafik- und Medien-Pipelines und Instanziieren der angeforderten Threads auf einer oder mehreren Ausführungseinheiten in den Ausführungseinheiten 508A-508N. Zum Beispiel kann eine Geometrie-Pipeline Vertex-, Tessellations- oder Geometrie-Shaders an die Thread-Ausführungslogik zur Verarbeitung versenden. In einigen Ausführungsformen kann der Thread-Dispatcher 504 auch Laufzeit-Thread-Spawning-Anforderungen von den ausgeführten Shader-Programmen verarbeiten.

[0076] In einigen Ausführungsformen unterstützen die Ausführungseinheiten 508A-508N einen Anweisungssatz, der native Unterstützung für viele Standard-3D-Grafik-Shader-Anweisungen beinhaltet, sodass Shader-Programme aus Grafikbibliotheken (z. B. Direct 3D und OpenGL) mit einer minimalen Übersetzung ausgeführt werden. Die Ausführungseinheiten unterstützen Vertex- und Geometrieverarbeitung (z. B. Vertex-Programme, Geometrieprogramme, Vertex-Shader), Pixelverarbeitung (z. B. Pixel-Shader, Fragment-Shader) und Allzweckverarbeitung (z. B. Berechnungs- und Medien-Shader). Jede der Ausführungseinheiten 508A-508N ist zu einer SIMD-Ausführung (SIMD: Single Instruction Multiple Data - einzelne Anweisung, mehrere Daten) fähig und eine Multithread-Operation ermöglicht eine effiziente Ausführungsumgebung angesichts von Speicherzugriffen mit höherer Latenz. Jeder Hardware-Thread innerhalb jeder Ausführungseinheit weist eine dedizierte Registerdatei mit hoher Bandbreite und einen assoziierten unabhängigen Thread-Zustand auf. Die Ausführung ist eine Mehrfach-Erteilung pro Takt an Pipelines mit Gleitkomma-Operationen mit ganzzahliger, einfacher und doppelter Genauigkeit, SIMD-Verzweigungskapazität, logischen Operationen, transzendenten Operationen und anderen sonstigen Operationen. Während auf Daten aus dem Speicher oder einer der gemeinsam genutzten Funktionen gewartet wird, bewirkt die Abhängigkeitslogik innerhalb der Ausführungseinheiten 508A-508N, dass ein wartender Thread in den Schlafzustand geht, bis die angeforderten Daten zurückgegeben worden sind. Während der wartende Thread im Schlafzustand ist, können sich Hardwareressourcen der Verarbeitung der anderen Threads widmen. Zum Beispiel kann während einer Verzögerung, die mit einer Vertex-Shader-Operation assoziiert ist, eine Ausführungseinheit Operationen für einen Pixel-Shader, Fragment-Shader oder einen anderen Typ von Shader-Programm durchführen, einschließlich eines anderen Vertex-Shaders. Verschiedene Ausführungsformen können für die Verwendung einer Ausführung unter Verwendung von SIMT (Single Instruction Multiple Thread - einzelne Anweisung, mehrere Threads) als Alternative zur Verwendung von SIMD oder zusätzlich zur Verwendung von SIMD gelten. Eine Bezugnahme auf einen SIMD-Kern oder eine SIMD-Operation kann auch für SIMT gelten oder für SIMD in Kombination mit SIMT gelten.

[0077] Jede Ausführungseinheit in den Ausführungseinheiten 508A-508N arbeitet an Arrays von Datenelementen. Die Anzahl der Datenelemente ist die „Ausführungsgröße“ oder die Anzahl von Kanälen für die Anweisung. Ein Ausführungskanal ist eine logische Ausführungseinheit für Datenelementzugriff, Maskierung und Ablaufsteuerung innerhalb von Anweisungen. Die Anzahl von Kanälen kann unabhängig von der Anzahl

von physischen arithmetischen Logikeinheiten (ALUs) oder Gleitkommaeinheiten (FPUs) für einen bestimmten Grafikprozessor sein. In einigen Ausführungsformen unterstützen die Ausführungseinheiten 508A-508N Ganzzahl- und Gleitkomma-Datentypen.

[0078] Der Ausführungseinheit-Anweisungssatz beinhaltet SIMD-Anweisungen. Die verschiedenen Datenelemente können als ein gepackter Datentyp in einem Register gespeichert werden und die Ausführungseinheit wird die verschiedenen Elemente basierend auf der Datengröße der Elemente verarbeiten. Wenn zum Beispiel mit einem 256 Bit breiten Vektor gearbeitet wird, werden die 256 Bits des Vektors in einem Register gespeichert, wobei die Ausführungseinheit am Vektor als vier separate gepackte 54-Bit-Datenelemente (Datenelemente in Vierfachwort(QW)-Größe), acht separate gepackte 32-Bit-Datenelemente (Datenelemente in Doppelwort(DW)-Größe), sechzehn separate gepackte 16-Bit-Datenelemente (Datenelemente in Wort(W)-Größe oder zweiunddreißig separate gepackte 8-Bit-Datenelemente (Datenelemente in Byte(B)-Größe) arbeitet. Es sind jedoch unterschiedliche Vektorbreiten und Registergrößen möglich.

[0079] In einer Ausführungsform können eine oder mehrere Ausführungseinheiten zu einer fusionierten Ausführungseinheit 509A-509N mit einer Thread-Steuerlogik (507A-507N) kombiniert werden, die den fusionierten EUs gemein ist. Mehrere EUs können zu einer EU-Gruppe fusioniert werden. Jede EU in der fusionierten EU-Gruppe kann derart konfiguriert sein, dass sie einen separaten SIMD-Hardware-Thread ausführt. Die Anzahl der EUs in einer fusionierten EU-Gruppe kann je nach Ausführungsformen variieren. Außerdem können verschiedene SIMD-Breiten pro EU durchgeführt werden, einschließlich, jedoch nicht beschränkt auf, SIMD8, SIMD16 und SIMD32. Jede fusionierte Grafikausführungseinheit 509A-509N beinhaltet mindestens zwei Ausführungseinheiten. Zum Beispiel beinhaltet die fusionierte Ausführungseinheit 509A eine erste EU 508A, eine zweite EU 508B und eine Thread-Steuerlogik 507A, die der ersten EU 508A und der zweiten EU 508B gemein ist. Die Thread-Steuerlogik 507A steuert Threads, die auf der fusionierten Grafikausführungseinheit 509A ausgeführt werden, wodurch jeder EU innerhalb der fusionierten Ausführungseinheiten 509A-509N ermöglicht wird, unter Verwendung eines gemeinsamen Anweisungszeigerregisters ausgeführt zu werden.

[0080] Ein oder mehrere interne Anweisungscaches (z. B. 506) sind in der Thread-Ausführungslogik 500 enthalten, um Thread-Anweisungen für die Ausführungseinheiten zu cachen. In manchen Ausführungsformen sind ein oder mehrere Datencaches (z. B. 512) enthalten, um Thread-Daten während der Thread-Ausführung zu cachen. Threads, die auf der Ausführungslogik 500 ausgeführt werden, können auch explizit verwaltete Daten im gemeinsam genutzten lokalen Speicher 511 speichern. In manchen Ausführungsformen ist ein Sampler 510 enthalten, um Textur-Sampling für 3D-Operationen und Medien-Sampling für Medienoperationen bereitzustellen. In manchen Ausführungsformen beinhaltet der Sampler 510 spezialisierte Textur- oder Medien-Sampling-Funktionalität, um Textur- oder Mediendaten während des Sampling-Prozesses zu verarbeiten, bevor die gesampelten Daten einer Ausführungseinheit bereitgestellt werden.

[0081] Während der Ausführung senden die Grafik- und Medien-Pipelines Thread-Initiierungsanforderungen an die Thread-Ausführungslogik 500 über die Thread-Spawning- und -Dispatch-Logik. Sobald eine Gruppe geometrischer Objekte verarbeitet und zu Pixeldaten rasterisiert wurde, wird Pixelprozessorlogik (z. B. Pixel-Shader-Logik, Fragment-Shader-Logik usw.) innerhalb des Shader-Prozessors 502 aufgerufen, um Ausgabeinformationen weiter zu berechnen und zu bewirken, dass Ergebnisse auf Ausgabeoberflächen (z. B. Farbpuffer, Tiefenpuffer, Schablonenpuffer usw.) geschrieben werden. In manchen Ausführungsformen berechnet ein Pixel-Shader oder Fragment-Shader die Werte der verschiedenen Vertexattribute, die über das rasterisierte Objekt interpoliert werden sollen. In einigen Ausführungsformen führt die Pixelprozessorlogik innerhalb des Shader-Prozessors 502 dann ein von einer Anwendungsprogrammierungsschnittstelle (API) geliefertes Pixel- oder Fragment-Shader-Programm aus. Um das Shader-Programm auszuführen, versendet der Shader-Prozessor 502 Threads über den Thread-Dispatcher 504 an eine Ausführungseinheit (z. B. 508A). In einigen Ausführungsformen verwendet der Shader-Prozessor 502 Textur-Sampling-Logik im Sampler 510, um auf Texturdaten in Texturabbildungen zuzugreifen, die im Speicher gespeichert sind. Arithmetische Operationen an den Texturdaten und den Eingabegeometriedaten berechnen Pixelfarbdaten für jedes geometrische Fragment oder verwerfen ein oder mehrere Pixel von der weiteren Verarbeitung.

[0082] In einigen Ausführungsformen stellt der Datenport 514 einen Speicherzugriffsmechanismus für die Thread-Ausführungslogik 500 bereit, um verarbeitete Daten an den Speicher zur weiteren Verarbeitung auf einer Grafikprozessorausgabe-Pipeline auszugeben. In einigen Ausführungsformen beinhaltet der Datenport 514 einen oder mehrere Cachespeicher (z. B. Datencache 512) oder ist mit diesen gekoppelt, um Daten für einen Speicherzugriff über den Datenport zu cachen.

[0083] In einer Ausführungsform kann die Ausführungslogik 500 auch einen Strahlverfolger (Raytracer) 505 beinhalten, der eine Raytracing-Beschleunigungsfunktionalität bereitstellen kann. Der Strahlverfolger 505 kann einen Raytracing-Anweisungssatz unterstützen, der Anweisungen/Funktionen zur Strahlerzeugung beinhaltet. Der Raytracing-Anweisungssatz kann dem Raytracing-Anweisungssatz, der von den Raytracing-Kernen 245 in **Fig. 2C** unterstützt wird, ähneln oder von diesem verschieden sein.

[0084] **Fig. 5B** veranschaulicht beispielhafte interne Einzelheiten einer Ausführungseinheit 508 gemäß Ausführungsformen. Eine Grafikausführungseinheit 508 kann eine Anweisungsabrufeinheit 537, ein Allgemeinregisterdatei-Array (GRF) 524, ein Architekturregisterdatei-Array (ARF) 526, einen Thread-Arbiter 522, eine Sendeeinheit 530, eine Verzweigungseinheit 532, einen Satz von SIMD-Gleitkommaeinheiten (FPUs) 534 und in einer Ausführungsform einen Satz von dedizierten Ganzzahl-SIMD-ALUs 535 beinhalten. Das GRF 524 und das ARF 526 beinhalten den Satz von Allgemeinregisterdateien und Architekturregisterdateien, die mit jedem simultanen Hardware-Thread assoziiert sind, der in der Grafikausführungseinheit 508 aktiv sein kann. In einer Ausführungsform wird ein Architekturzustand pro Thread in dem ARF 526 beibehalten, während Daten, die während der Thread-Ausführung verwendet werden, in dem GRF 524 gespeichert werden. Der Ausführungszustand jedes Threads, einschließlich der Anweisungszeiger für jeden Thread, kann in Threadspezifischen Registern im ARF 526 gehalten werden.

[0085] In einer Ausführungsform weist die Grafikausführungseinheit 508 eine Architektur auf, die eine Kombination von simultanem Multi-Threading (SMT) und feinkörnigem verschachteltem Multi-Threading (IMT) ist. Die Architektur weist eine modulare Konfiguration auf, die zur Gestaltungszeit basierend auf einer Zielanzahl von gleichzeitigen Threads und einer Anzahl von Registern pro Ausführungseinheit fein abgestimmt werden kann, wobei Ressourcen der Ausführungseinheit über die Logik aufgeteilt werden, die zum Ausführen mehrerer gleichzeitiger Threads verwendet wird. Die Anzahl von logischen Threads, die durch die Grafikausführungseinheit 508 ausgeführt werden können, ist nicht auf die Anzahl von Hardware-Threads beschränkt, und jedem Hardware-Thread können mehrere logische Threads zugewiesen werden.

[0086] In einer Ausführungsform kann die Grafikausführungseinheit 508 mehrere Anweisungen gemeinsam ausgeben, die jeweils unterschiedliche Anweisungen sein können. Der Thread-Arbiter 522 des Grafikausführungseinheit-Threads 508 kann die Anweisungen an entweder die Sendeeinheit 530, die Verzweigungseinheit 532 oder die SIMD-FPU(s) 534 zur Ausführung versenden. Jeder Ausführungs-Thread kann auf 128 Allzweckregister innerhalb des GRF 524 zugreifen, wobei jedes Register 32 Byte speichern kann, die als ein SIMD-8-Element-Vektor von 32-Bit-Datenelementen zugänglich sind. In einer Ausführungsform hat jeder Ausführungseinheit-Thread Zugriff auf 4 kByte innerhalb des GRF 524, obwohl die Ausführungsformen nicht darauf beschränkt sind und mehr oder weniger Registerressourcen in anderen Ausführungsformen bereitgestellt werden können. In einer Ausführungsform ist die Grafikausführungseinheit 508 in sieben Hardware-Threads partitioniert, die unabhängig Rechenoperationen durchführen können, obwohl die Anzahl von Threads pro Ausführungseinheit auch gemäß Ausführungsformen variieren kann. Beispielsweise werden in einer Ausführungsform bis zu 16 Hardware-Threads unterstützt. In einer Ausführungsform, bei der sieben Threads auf 4 kByte zugreifen können, kann das GRF 524 insgesamt 28 kByte speichern. Wenn 16 Threads auf 4 kBytes zugreifen können, kann das GRF 524 insgesamt 64 kBytes speichern. Flexible Adressierungsmodi können ermöglichen, dass Register zusammen adressiert werden, um effektiv breitere Register zu bilden oder um streifenförmige rechteckige Blockdatenstrukturen zu repräsentieren.

[0087] In einer Ausführungsform werden Speicheroperationen, Sampler-Operationen und andere Systemkommunikationen mit längerer Latenz über „Send (Senden)“-Anweisungen versendet, die durch die Nachrichtenweiterleitungssendeeinheit 530 ausgeführt werden. In einer Ausführungsform werden Verzweigungsanweisungen an eine dedizierte Verzweigungseinheit 532 versendet, um SIMD-Divergenz und eventuelle Konvergenz zu ermöglichen.

[0088] In einer Ausführungsform beinhaltet die Grafikausführungseinheit 508 eine oder mehrere SIMD-Gleitkommaeinheiten (FPU(s)) 534, um Gleitkommaoperationen durchzuführen. In einer Ausführungsform unterstützen die eine oder die mehreren FPUs 534 auch eine Ganzzahlberechnung. In einer Ausführungsform können die FPU(s) 534 bis zu M 32-Bit-Gleitkomma(oder Ganzzahl-)-Operationen SIMD-ausführen, oder bis zu 2M 16-Bit Ganzzahl- oder 16-Bit-Gleitkommaoperationen SIMD-ausführen. In einer Ausführungsform stellt mindestens eine der einen oder der mehreren FPUs eine erweiterte Mathefähigkeit bereit, um transzendentische Mathefunktionen mit hohem Durchsatz und 54-Bit-Gleitkomma mit doppelter Präzision zu unterstützen. In einigen Ausführungsformen ist auch ein Satz von 8-Bit-Ganzzahl-SIMD-ALUs 535 vorhanden und kann spezifisch optimiert werden, um Operationen durchzuführen, die mit Maschinenlernberechnungen assoziiert sind.

[0089] In einer Ausführungsform können Arrays von mehreren Instanzen der Grafikausführungseinheit 508 in einer Grafik-Unterkerngruppierung (z. B. einem Sub-Slice) instanziiert werden. Zur Skalierbarkeit können Produktarchitekten die genaue Anzahl an Ausführungseinheiten pro Unterkerngruppierung auswählen. In einer Ausführungsform kann die Ausführungseinheit 508 Anweisungen über mehrere Ausführungskanäle ausführen. In einer weiteren Ausführungsform wird jeder Thread, der auf der Grafikausführungseinheit 508 ausgeführt wird, auf einem anderen Kanal ausgeführt.

[0090] **Fig. 6** veranschaulicht eine zusätzliche Ausführungseinheit 600 gemäß einer Ausführungsform. Die Ausführungseinheit 600 kann eine rechenoptimierte Ausführungseinheit zur Verwendung beispielsweise in einer Berechnungs-Engine-Kachel 340A-340D wie in **Fig. 3C** sein, ist jedoch nicht darauf beschränkt. Varianten der Ausführungseinheit 600 können auch in einer Grafik-Engine-Kachel 310A-310D wie in **Fig. 3B** verwendet werden. In einer Ausführungsform weist die Ausführungseinheit 600 eine Thread-Steuereinheit 601, eine Thread-Zustandseinheit 602, eine Anweisungsabruf-/-vorabrufeinheit 603 und eine Anweisungsdecodiereinheit 604 auf. Die Ausführungseinheit 600 weist zusätzlich eine Registerdatei 606 auf, die Register speichert, die Hardware-Threads innerhalb der Ausführungseinheit zugeordnet werden können. Die Ausführungseinheit 600 weist zusätzlich eine Sendeeinheit 607 und eine Verzweigungseinheit 608 auf. In einer Ausführungsform können die Sendeeinheit 607 und die Verzweigungseinheit 608 ähnlich wie die Sendeeinheit 530 und eine Verzweigungseinheit 532 der Grafikausführungseinheit 508 von **Fig. 5B** arbeiten.

[0091] Die Ausführungseinheit 600 beinhaltet auch eine Berechnungseinheit 610, die mehrere unterschiedliche Typen von Funktionseinheiten beinhaltet. In einer Ausführungsform beinhaltet die Berechnungseinheit 610 eine ALU-Einheit 611, die ein Array von arithmetischen Logikeinheiten beinhaltet. Die ALU-Einheit 611 kann dazu konfiguriert sein, 64-Bit-, 32-Bit- und 16-Bit-Ganzzahl- und Gleitkommaoperationen durchzuführen. Ganzzahl- und Gleitkommaoperationen können simultan durchgeführt werden. Die Berechnungseinheit 610 kann auch ein systolisches Array 612 und eine Mathe-Einheit 613 beinhalten. Das systolische Array 612 beinhaltet ein W-breites und D-tiefes Netzwerk von Datenverarbeitungseinheiten, die verwendet werden können, um Vektor- oder andere datenparallele Operationen auf eine systolische Weise durchzuführen. In einer Ausführungsform kann das systolische Array 612 dazu konfiguriert sein, Matrixoperationen, wie etwa Matrix-Skalarproduktoperationen, durchzuführen. In einer Ausführungsform unterstützt das systolische Array 612 16-Bit-Gleitkommaoperationen sowie 8-Bit- und 4-Bit-Ganzzahloperationen. In einer Ausführungsform kann das systolische Array 612 dazu konfiguriert sein, Maschinenlernoperationen zu beschleunigen. In solchen Ausführungsformen kann das systolische Array 612 mit Unterstützung für das bfloat 16-Bit-Gleitkommaformat konfiguriert werden. In einer Ausführungsform kann eine Mathe-Einheit 613 enthalten sein, um eine spezifische Teilmenge mathematischer Operationen auf effiziente und leistungsschwächere Weise als die ALU-Einheit 611 durchzuführen. Die Mathe-Einheit 613 kann eine Variante einer Mathe-Logik beinhalten, die in einer Logik mit gemeinsam genutzter Funktion einer Grafikverarbeitungs-Engine gefunden werden kann, die durch andere Ausführungsformen bereitgestellt wird (z. B. Mathe-Logik 422 der Logik 420 mit gemeinsam genutzter Funktion von **Fig. 4**). In einer Ausführungsform kann die Mathe-Einheit 613 dazu konfiguriert sein, 32-Bit- und 64-Bit-Gleitkommaoperationen durchzuführen.

[0092] Die Thread-Steuereinheit 601 enthält Logik zum Steuern der Ausführung von Threads innerhalb der Ausführungseinheit. Die Thread-Steuereinheit 601 kann Thread-Arbitrierungslogik beinhalten, um die Ausführung von Threads innerhalb der Ausführungseinheit 600 zu starten, zu stoppen und zu präemptieren. Die Thread-Zustandseinheit 602 kann verwendet werden, um den Thread-Zustand für Threads zu speichern, die zur Ausführung auf der Ausführungseinheit 600 zugewiesen sind. Das Speichern des Thread-Zustands in der Ausführungseinheit 600 ermöglicht die schnelle Präemption von Threads, wenn diese Threads blockiert oder inaktiv werden. Die Anweisungsabruf-/-vorabrufeinheit 603 kann Anweisungen aus einem Anweisungscache der Ausführungslogik höherer Ebene (z. B. Anweisungscache 506 wie in **Fig. 5A**) abrufen. Die Anweisungsabruf-/-vorabrufeinheit 603 kann auch Vorabrufanforderungen für Anweisungen ausgeben, die in den Anweisungscache geladen werden sollen, basierend auf einer Analyse von gegenwärtig ausgeführten Threads. Die Anweisungsdecodiereinheit 604 kann verwendet werden, um von den Berechnungseinheiten auszuführende Anweisungen zu decodieren. In einer Ausführungsform kann die Anweisungsdecodiereinheit 604 als ein Sekundärdecodierer verwendet werden, um komplexe Anweisungen in konstituierende Mikrooperationen zu decodieren.

[0093] Die Ausführungseinheit 600 beinhaltet zusätzlich eine Registerdatei 606, die durch Hardware-Threads verwendet werden kann, die auf der Ausführungseinheit 600 ausgeführt werden. Register in der Registerdatei 606 können über die Logik aufgeteilt werden, die verwendet wird, um mehrere simultane Threads innerhalb der Berechnungseinheit 610 der Ausführungseinheit 600 auszuführen. Die Anzahl von logischen Threads, die durch die Grafikausführungseinheit 600 ausgeführt werden können, ist nicht auf die

Anzahl von Hardware-Threads beschränkt, und jedem Hardware-Thread können mehrere logische Threads zugewiesen werden. Die Größe der Registerdatei 606 kann zwischen Ausführungsformen basierend auf der Anzahl unterstützter Hardware-Threads variieren. In einer Ausführungsform kann Registerumbenennung verwendet werden, um Register dynamisch zu Hardware-Threads zuzuordnen.

[0094] **Fig. 7** ist ein Blockdiagramm, das ein Grafikprozessor-Anweisungsformat 700 gemäß einigen Ausführungsformen veranschaulicht. In einer oder mehreren Ausführungsformen unterstützen die Grafikprozessorausführungseinheiten einen Anweisungssatz mit Anweisungen in mehreren Formaten. Die Kästchen mit durchgezogenen Linien veranschaulichen die Komponenten, die allgemein in einer Ausführungseinheitenanweisung enthalten sind, während die gestrichelten Linien Komponenten enthalten, die optional sind oder die nur in einer Teilmenge der Anweisungen enthalten sind. In einigen Ausführungsformen handelt es sich bei dem beschriebenen und veranschaulichten Anweisungsformat 700 um Makroanweisungen, insofern es sich um Anweisungen handelt, die der Ausführungseinheit zugeführt werden, im Gegensatz zu Mikrooperationen, die sich aus der Anweisungsdecodierung ergeben, sobald die Anweisung verarbeitet wird.

[0095] In manchen Ausführungsformen unterstützen die Grafikprozessorausführungseinheiten Anweisungen nativ in einem 128-Bit-Anweisungsformat 710. Ein verdichtetes 64-Bit-Anweisungsformat 730 ist für manche Anweisungen basierend auf der ausgewählten Anweisung, den Anweisungsoptionen und der Anzahl der Operanden verfügbar. Das native 128-Bit-Anweisungsformat 710 bietet Zugriff auf alle Anweisungsoptionen, während manche Optionen und Operationen im 64-Bit-Format 730 beschränkt sind. Die nativen Anweisungen, die im 64-Bit-Format 730 verfügbar sind, variieren je nach Ausführungsform. In einigen Ausführungsformen wird die Anweisung teilweise unter Verwendung eines Satzes von Indexwerten in einem Indexfeld 713 verdichtet. Die Ausführungseinheit-Hardware referenziert einen Satz von Verdichtungstabellen basierend auf den Indexwerten und verwendet die Verdichtungstabellenausgaben, um eine native Anweisung im 128-Bit-Anweisungsformat 710 zu rekonstruieren. Andere Anweisungsgrößen und -formate können verwendet werden.

[0096] Für jedes Format definiert der Anweisungs-Opcode 712 die Operation, die die Ausführungseinheit durchführen soll. Die Ausführungseinheiten führen jede Anweisung parallel über die mehreren Datenelemente jedes Operanden aus. Zum Beispiel führt die Ausführungseinheit als Reaktion auf eine Addieranweisung eine simultane Addieroperation über jeden Farbkanal aus, der ein Texturelement oder Bildelement repräsentiert. Standardmäßig führt die Ausführungseinheit jede Anweisung über alle Datenkanäle der Operanden aus. In einigen Ausführungsformen ermöglicht das Anweisungssteuerungs-Feld 714 die Steuerung über gewisse Ausführungsoptionen, wie etwa Kanalauswahl (z. B. Prädikation) und Datenkanalreihenfolge (z. B. Swizzle). Für Anweisungen im 128-Bit-Anweisungsformat 710 begrenzt ein Ausführungsgröße-Feld 716 die Anzahl an Datenkanälen, die parallel ausgeführt werden. In einigen Ausführungsformen steht das Ausführungsgröße-Feld 716 nicht zur Verwendung in dem kompakten 64-Bit-Anweisungsformat 730 zur Verfügung.

[0097] Einige Ausführungseinheitsanweisungen haben bis zu drei Operanden einschließlich zwei Quelloperanden, src0 720, src1 722, und ein Ziel 718. In manchen Ausführungsformen unterstützen die Ausführungseinheiten duale Zielanweisungen, wobei eines der Ziele impliziert ist. Datenmanipulationsanweisungen können einen dritten Quelloperanden aufweisen (z. B. SRC2 724), wobei der Anweisungs-Opcode 712 die Anzahl der Quelloperanden bestimmt. Der letzte Quelloperand einer Anweisung kann ein Immediate-Wert (z. B. festcodierter Wert) sein, der mit der Anweisung übergeben wird.

[0098] In einigen Ausführungsformen beinhaltet das 128-Bit-Anweisungsformat 710 ein Zugriffs-/Adressmodus-Feld 726, das zum Beispiel spezifiziert, ob ein direkter Registeradressierungsmodus oder ein indirekter Registeradressierungsmodus verwendet wird. Wenn der direkte Registeradressierungsmodus verwendet wird, wird die Registeradresse eines oder mehrerer Operanden direkt durch Bits in der Anweisung bereitgestellt.

[0099] In einigen Ausführungsformen beinhaltet das 128-Bit-Anweisungsformat 710 ein Zugriffs-/Adressmodus-Feld 726, das einen Adressmodus und/oder einen Zugriffsmodus für die Anweisung spezifiziert. In einer Ausführungsform wird der Zugriffsmodus verwendet, um eine Datenzugriffsausrichtung für die Anweisung zu definieren. Manche Ausführungsformen unterstützen Zugriffsmodi einschließlich eines 16-Byte-ausgerichteten Zugriffsmodus und eines 1-Byte-ausgerichteten Zugriffsmodus, wobei die Byteausrichtung des Zugriffsmodus die Zugriffsausrichtung der Anweisungsoperanden bestimmt. Zum Beispiel kann die Anweisung, wenn sie sich in einem ersten Modus befindet, eine Byteausgerichtete Adressierung für Quell- und Zielope-

randen verwenden, und, wenn sie sich in einem zweiten Modus befindet, kann die Anweisung eine 16-Byte-ausgerichtete Adressierung für alle Quell- und Zieloperanden verwenden.

[0100] In einer Ausführungsform bestimmt der Adressmodusabschnitt des Zugriffs-/Adressmodus-Feldes 726, ob die Anweisung eine direkte oder indirekte Adressierung verwenden soll. Wenn der direkte Registeradressierungsmodus verwendet wird, stellen die Bits in der Anweisung direkt die Registeradresse eines oder mehrerer Operanden bereit. Wenn der indirekte Registeradressierungsmodus verwendet wird, kann die Registeradresse eines oder mehrerer Operanden basierend auf einem Adressregisterwert und einem Adress-Immediate-Feld in der Anweisung berechnet werden.

[0101] In einigen Ausführungsformen werden Anweisungen basierend auf Bitfeldern des Opcodes 712 gruppiert, um die Opcode-Decodierung 740 zu vereinfachen. Bei einem 8-Bit-Opcode ermöglichen die Bits 4, 5 und 6 der Ausführungseinheit, den Opcode-Typ zu bestimmen. Die gezeigte präzise Opcode-Gruppierung ist lediglich ein Beispiel. In manchen Ausführungsformen beinhaltet eine Verschiebungs- und Logik-Opcode-Gruppe 742 Datenverschiebungs- und Logikanweisungen (z. B. move (mov) (Verschieben), compare (cmp) (Vergleichen)). In manchen Ausführungsformen nutzt die Verschiebungs- und Logikgruppe 742 die fünf höchstwertigen Bits (MSB) gemeinsam, wobei Verschiebungs(mov)-Anweisungen die Form 0000xxxxb haben und Logikanweisungen die Form 0001xxxxb haben. Eine Ablaufsteuerungsanweisungsgruppe 744 (z. B. call (Aufrufen), jump (jmp) (Springen)) beinhaltet Anweisungen in Form von 0010xxxxb (z. B. 0x20). Eine gemischte Anweisungsgruppe 746 beinhaltet eine Mischung von Anweisungen einschließlich Synchronisationsanweisungen (z. B. wait, send) in der Form 0011xxxxb (z. B. 0x30). Eine Parallel-Mathe-Anweisungsgruppe 748 beinhaltet komponentenweise arithmetische Anweisungen (z. B. add (Addieren), multiply (mul) (Multiplizieren)) in der Form 0100xxxxb (z. B. 0x40). Die Parallel-Mathe-Gruppe 748 führt die arithmetischen Operationen parallel über Datenkanäle aus. Die Vektor-Mathe-Gruppe 750 beinhaltet arithmetische Anweisungen (z. B. dp4) in der Form 0101xxxxb (z. B. 0x50). Die Vektor-Mathe-Gruppe führt Arithmetik wie z. B. Skalarproduktberechnungen an Vektoroperanden aus. Die veranschaulichte Opcode-Decodierung 740 kann in einer Ausführungsform verwendet werden, um zu bestimmen, welcher Abschnitt einer Ausführungseinheit verwendet wird, um eine decodierte Anweisung auszuführen. Zum Beispiel können einige Anweisungen als systolische Anweisungen bezeichnet werden, die durch ein systolisches Array ausgeführt werden. Andere Anweisungen, wie etwa Raytracing-Anweisungen (nicht gezeigt) können zu einem Raytracing-Kern oder einer Raytracing-Logik innerhalb eines Slice oder einer Partitionierung der Ausführungslogik geleitet werden.

Grafik-Pipeline

[0102] **Fig. 8** ist ein Blockdiagramm einer anderen Ausführungsform eines Grafikprozessors 800. Elemente von **Fig. 8** mit den gleichen Bezugszeichen (oder Bezeichnungen) wie die Elemente einer beliebigen anderen Figur hierin können auf beliebige Weise ähnlich der an anderer Stelle hierin beschriebenen arbeiten oder funktionieren, sind aber nicht darauf beschränkt.

[0103] In einigen Ausführungsformen beinhaltet der Grafikprozessor 800 eine Geometrie-Pipeline 820, eine Medien-Pipeline 830, eine Anzeige-Engine 840, Thread-Ausführungslogik 850 und eine Render-Ausgabe-Pipeline 870. In manchen Ausführungsformen ist der Grafikprozessor 800 ein Grafikprozessor in einem Mehrkern-Verarbeitungssystem, das einen oder mehrere Allzweck-Verarbeitungskerne beinhaltet. Der Grafikprozessor wird durch Registerschreibvorgänge in ein oder mehrere Steuerregister (nicht gezeigt) oder über Befehle gesteuert, die über ein Ring-Interconnect 802 an den Grafikprozessor 800 ausgegeben werden. In manchen Ausführungsformen koppelt das Ring-Interconnect 802 den Grafikprozessor 800 mit anderen Verarbeitungskomponenten, wie etwa anderen Grafikprozessoren oder Allzweckprozessoren. Befehle vom Ring-Interconnect 802 werden von einem Befehls-Streamer 803 interpretiert, der Anweisungen an einzelne Komponenten der Geometrie-Pipeline 820 oder der Medien-Pipeline 830 liefert.

[0104] In einigen Ausführungsformen leitet der Befehls-Streamer 803 den Betrieb eines Vertex-Abrufers 805 an, der Vertex-Daten aus dem Speicher liest und Vertex-Verarbeitungsbefehle ausführt, die durch den Befehls-Streamer 803 bereitgestellt werden. In manchen Ausführungsformen stellt der Vertex-Abrufer 805 Vertex-Daten an einen Vertex-Shader 807 bereit, der Koordinatenraumtransformations- und Beleuchtungsoperationen an jedem Vertex durchführt. In einigen Ausführungsformen führen der Vertex-Abrufer 805 und der Vertex-Shader 807 Vertex-Verarbeitungsanweisungen aus, indem sie Ausführungs-Threads über einen Thread-Dispatcher 831 an die Ausführungseinheiten 852A-852B versenden.

[0105] In einigen Ausführungsformen sind die Ausführungseinheiten 852A-852B ein Array von Vektorprozessoren mit einem Anweisungssatz zum Ausführen von Grafik- und Medienoperationen. In einigen Ausfüh-

rungsformen weisen die Ausführungseinheiten 852A-852B einen angehängten L1-Cache 851 auf, der für jedes Array spezifisch ist oder von den Arrays gemeinsam genutzt wird. Der Cache kann als Datencache, als Anweisungscache oder als Einzelcache konfiguriert sein, der derart partitioniert ist, dass er Daten und Anweisungen in verschiedenen Partitionierungen enthält.

[0106] In manchen Ausführungsformen beinhaltet die Geometrie-Pipeline 820 Tessellationskomponenten zum Durchführen einer hardwarebeschleunigten Tessellation von 3D-Objekten. In manchen Ausführungsformen konfiguriert ein programmierbarer Hüllen-Shader 811 die Tessellationsoperationen. Ein programmierbarer Domänen-Shader 817 stellt eine Backend-Auswertung der Tessellationsausgabe bereit. Ein Tessellator 813 arbeitet in Richtung des Hüllen-Shaders 811 und enthält eine Spezialzwecklogik zum Erzeugen eines Satzes detaillierter geometrischer Objekte basierend auf einem groben geometrischen Modell, das als Eingabe in die Geometrie-Pipeline 820 bereitgestellt wird. In manchen Ausführungsformen können, falls keine Tessellation verwendet wird, Tessellationskomponenten (z. B. Hüllen-Shader 811, Tessellator 813 und Domänen-Shader 817) umgangen werden.

[0107] In einigen Ausführungsformen können vollständige geometrische Objekte durch einen Geometrie-Shader 819 über einen oder mehrere Threads verarbeitet werden, die an die Ausführungseinheiten 852A-852B versendet werden, oder können direkt zu dem Clipper 829 weitergehen. In manchen Ausführungsformen arbeitet der Geometrie-Shader an ganzen geometrischen Objekten anstatt an Vertices oder Patches von Vertices wie in vorherigen Stufen der Grafik-Pipeline. Falls die Tessellation deaktiviert ist, empfängt der Geometrie-Shader 819 eine Eingabe vom Vertex-Shader 807. In manchen Ausführungsformen ist der Geometrie-Shader 819 durch ein Geometrie-Shader-Programm dafür programmierbar, eine Geometrietessellation durchzuführen, falls die Tessellationseinheiten deaktiviert sind.

[0108] Vor der Rasterisierung verarbeitet ein Clipper 829 Vertexdaten. Der Clipper 829 kann ein Clipper mit fester Funktion oder ein programmierbarer Clipper mit Clipping- und Geometrie-Shader-Funktionen sein. In einigen Ausführungsformen versendet eine Rasterisierer-und-Tiefenprüfung-Komponente 873 in der Rendering-Ausgabe-Pipeline 870 Pixel-Shader, um die geometrischen Objekte in Pro-Pixel-Repräsentationen umzuwandeln. In manchen Ausführungsformen ist eine Pixel-Shader-Logik in der Thread-Ausführungslogik 850 enthalten. In einigen Ausführungsformen kann eine Anwendung die Rasterisierer-und-Tiefenprüfung-Komponente 873 umgehen und über eine Stream-Out-Einheit 823 auf nicht rasterisierte Vertex-Daten zugreifen.

[0109] Der Grafikprozessor 800 weist einen Interconnect-Bus, ein Interconnect-Fabric oder einen anderen Interconnect-Mechanismus auf, der ein Weitergeben von Daten und Nachrichten zwischen den Hauptkomponenten des Prozessors ermöglicht. Bei manchen Ausführungsformen sind die Ausführungseinheiten 852A-852B und die assoziierten Logikeinheiten (z. B. L1-Cache 851, Sampler 854, Textur-Cache 858 usw.) über einen Datenport 856 miteinander verbunden, um einen Speicherzugriff durchzuführen und mit Rendering-Ausgabe-Pipeline-Komponenten des Prozessors zu kommunizieren. Bei einigen Ausführungsformen weisen der Sampler 854, die Caches 851, 858 und die Ausführungseinheiten 852A-852B jeweils getrennte Speicherzugriffspfade auf. In einer Ausführungsform kann der Texturcache 858 auch als Sampler-Cache konfiguriert sein.

[0110] In manchen Ausführungsformen enthält die Rendering-Ausgabe-Pipeline 870 eine Rasterisierer- und Tiefenprüfungskomponente 873, die vertexbasierte Objekte in eine assoziierte pixelbasierte Repräsentation umwandelt. In manchen Ausführungsformen beinhaltet die Rasterisiererlogik eine Fenstererstellungs-/Maskiereinheit zum Durchführen einer Dreiecks- und Linienrasterisierung mit fester Funktion. Ein assoziierter Rendering-Cache 878 und Tiefencache 879 sind bei manchen Ausführungsformen ebenfalls verfügbar. Eine Pixeloperationskomponente 877 führt pixelbasierte Operationen an den Daten aus, wenngleich in einigen Fällen Pixeloperationen, die mit 2D-Operationen assoziiert sind (z. B. Bitblockbildtransfers mit Blending), von der 2D-Engine 841 ausgeführt oder zur Anzeigezeit von der Anzeigesteuerung 843 unter Verwendung von Overlay-Anzeigeebenen ersetzt werden. In manchen Ausführungsformen ist ein gemeinsam genutzter L3-Cache 875 für alle Grafikkomponenten verfügbar, was die gemeinsame Nutzung von Daten ohne die Verwendung von Hauptsystemspeicher ermöglicht.

[0111] In manchen Ausführungsformen beinhaltet die Grafikprozessor-Medien-Pipeline 830 eine Medien-Engine 837 und ein Video-Frontend 834. In manchen Ausführungsformen empfängt das Video-Frontend 834 Pipeline-Befehle von dem Befehls-Streamer 803. In einigen Ausführungsformen beinhaltet die Medien-Pipeline 830 einen separaten Befehls-Streamer. In manchen Ausführungsformen verarbeitet das Video-Frontend 834 Medienbefehle, bevor der Befehl an die Medien-Engine 837 gesendet wird. In einigen Ausführungsfor-

men weist die Medien-Engine 837 eine Thread-Spawning-Funktionalität auf, um Threads zum Versenden an die Thread-Ausführungslogik 850 über den Thread-Dispatcher 831 zu spawnen.

[0112] In einigen Ausführungsformen weist der Grafikprozessor 800 eine Anzeige-Engine 840 auf. In manchen Ausführungsformen befindet sich die Anzeige-Engine 840 außerhalb des Prozessors 800 und ist mit dem Grafikprozessor über das Ring-Interconnect 802 oder einen anderen Interconnect-Bus oder ein anderes Interconnect-Fabric gekoppelt. In manchen Ausführungsformen weist die Anzeige-Engine 840 eine 2D-Engine 841 und eine Anzeigesteuerung 843 auf. In manchen Ausführungsformen enthält die Anzeige-Engine 840 Speziallogik, die in der Lage ist, unabhängig von der 3D-Pipeline zu arbeiten. In manchen Ausführungsformen ist die Anzeigesteuerung 843 mit einer Anzeigevorrichtung (nicht gezeigt) gekoppelt, die eine systemintegrierte Anzeigevorrichtung, wie in einem Laptop-Computer, oder eine externe Anzeigevorrichtung, die über einen Anzeigevorrichtungsverbinder angebracht ist, sein kann.

[0113] In einigen Ausführungsformen sind die Geometrie-Pipeline 820 und die Medien-Pipeline 830 konfigurierbar, Operationen basierend auf mehreren Grafik- und Medienprogrammierschnittstellen durchzuführen, und sind nicht spezifisch für irgendeine Anwendungsprogrammierschnittstelle (API). In einigen Ausführungsformen übersetzt Treibersoftware für den Grafikprozessor API-Aufrufe, die für eine bestimmte Grafik- oder Medienbibliothek spezifisch sind, in Befehle, die durch den Grafikprozessor verarbeitet werden können. In manchen Ausführungsformen wird eine Unterstützung für Open Graphics Library (OpenGL), Open Computing Language (OpenCL) und/oder Vulkan Graphics und Rechen-API bereitgestellt, die alle von der Khronos Group stammen. In manchen Ausführungsformen kann Unterstützung auch für die Direct3D-Bibliothek von der Microsoft Corporation bereitgestellt werden. In einigen Ausführungsformen kann eine Kombination dieser Bibliotheken unterstützt werden. Eine Unterstützung kann auch für die Open Source Computer Vision Library (OpenCV) bereitgestellt werden. Eine zukünftige API mit einer kompatiblen 3D-Pipeline wird ebenfalls unterstützt, falls eine Abbildung von der Pipeline der zukünftigen API auf die Pipeline des Grafikprozessors vorgenommen werden kann.

Grafik-Pipeline-Programmierung

[0114] **Fig. 9A** ist ein Blockdiagramm, das ein Grafikprozessor-Befehlsformat 900 veranschaulicht, gemäß manchen Ausführungsformen. **Fig. 9B** ist ein Blockdiagramm, das eine Grafikprozessor-Befehlssequenz 910 gemäß einer Ausführungsform veranschaulicht. Die durchgezogenen Kästen in **Fig. 9A** veranschaulichen die Komponenten, die im Allgemeinen in einem Grafikbefehl enthalten sind, während die gestrichelten Linien Komponenten enthalten, die optional sind oder die nur in einer Teilmenge der Grafikbefehle enthalten sind. Das beispielhafte Grafikprozessor-Befehlsformat 900 von **Fig. 9A** enthält Datenfelder, um einen Client 902, einen Befehlsoperationscode (Opcode) 904 und Daten 906 für den Befehl zu identifizieren. Ein Sub-Opcode 905 und eine Befehlsgröße 908 sind auch in einigen Befehlen enthalten.

[0115] In einigen Ausführungsformen spezifiziert der Client 902 die Client-Einheit der Grafikvorrichtung, die die Befehlsdaten verarbeitet. In manchen Ausführungsformen untersucht ein Grafikprozessor-Befehls-Parser das Client-Feld jedes Befehls, um die weitere Verarbeitung des Befehls zu konditionieren und die Befehlsdaten an die geeignete Client-Einheit zu leiten. In manchen Ausführungsformen beinhalten die Grafikprozessor-Client-Einheiten eine Speicherschnittstelleneinheit, eine Rendering-Einheit, eine 2D-Einheit, eine 3D-Einheit und eine Medieneinheit. Jede Client-Einheit weist eine entsprechende Verarbeitungs-Pipeline auf, die die Befehle verarbeitet. Sobald der Befehl durch die Client-Einheit empfangen wird, liest die Client-Einheit den Opcode 904 und, falls vorhanden, den Sub-Opcode 905, um die durchzuführende Operation zu bestimmen. Die Client-Einheit führt den Befehl unter Verwendung von Informationen in dem Datenfeld 906 aus. Für einige Befehle wird eine explizite Befehlsgröße 908 erwartet, um die Größe des Befehls anzugeben. In einigen Ausführungsformen bestimmt der Befehls-Parser automatisch die Größe von zumindest einigen der Befehle basierend auf dem Befehls-Opcode. In manchen Ausführungsformen werden Befehle über Vielfache eines Doppelwortes ausgerichtet. Andere Befehlsformate können verwendet werden.

[0116] Das Flussdiagramm in **Fig. 9B** veranschaulicht eine beispielhafte Grafikprozessor-Befehlssequenz 910. In manchen Ausführungsformen verwendet Software oder Firmware eines Datenverarbeitungssystems, das eine Ausführungsform eines Grafikprozessors aufweist, eine Version der gezeigten Befehlssequenz, um einen Satz von Grafikoperationen einzurichten, auszuführen und zu beenden. Eine beispielhafte Befehlssequenz wird nur zu Beispielszwecken gezeigt und beschrieben, da Ausführungsformen nicht auf diese spezifischen Befehle oder auf diese Befehlsfolge beschränkt sind. Darüber hinaus können die Befehle als Batch von Befehlen in einer Befehlssequenz ausgegeben werden, sodass der Grafikprozessor die Befehlssequenz zumindest teilweise gleichzeitig verarbeiten wird.

[0117] In einigen Ausführungsformen kann die Grafikprozessor-Befehlssequenz 910 mit einem Pipeline-Flush-Befehl 912 beginnen, um zu bewirken, dass jede aktive Grafik-Pipeline die gegenwärtig anstehenden Befehle für die Pipeline abschließt. In manchen Ausführungsformen arbeiten die 3D-Pipeline 922 und die Medien-Pipeline 924 nicht gleichzeitig. Der Pipeline-Flush wird durchgeführt, um die aktive Grafik-Pipeline zu veranlassen, alle ausstehenden Befehle abzuschließen. Als Reaktion auf einen Pipeline-Flush pausiert der Befehls-Parser für den Grafikprozessor die Befehlsverarbeitung, bis die aktiven Zeichnung-Engines ausstehende Operationen abschließen und die relevanten Lese-Caches ungültig gemacht werden. Optional können beliebige Daten in dem Rendering-Cache, die als „verschmutzt" markiert sind, in den Speicher geflusht werden. In einigen Ausführungsformen kann der Pipeline-Flush-Befehl 912 für die Pipeline-Synchronisation oder vor dem Versetzen des Grafikprozessors in einen Niedrigleistungszustand verwendet werden.

[0118] In einigen Ausführungsformen wird ein Pipeline-Auswahlbefehl 913 verwendet, wenn eine Befehlssequenz erfordert, dass der Grafikprozessor explizit zwischen Pipelines umschaltet. In manchen Ausführungsformen wird ein Pipeline-Auswahlbefehl 913 nur einmal in einem Ausführungskontext vor dem Ausgeben von Pipeline-Befehlen benötigt, es sei denn, der Kontext gibt Befehle für beide Pipelines aus. In manchen Ausführungsformen ist ein Pipeline-Flush-Befehl 912 unmittelbar vor einem Pipeline-Umschalten über den Pipeline-Auswahlbefehl 913 erforderlich.

[0119] In manchen Ausführungsformen konfiguriert ein Pipeline-Steuerbefehl 914 eine Grafik-Pipeline für den Betrieb und wird verwendet, um die 3D-Pipeline 922 und die Medien-Pipeline 924 zu programmieren. In manchen Ausführungsformen konfiguriert der Pipeline-Steuerbefehl 914 den Pipeline-Zustand für die aktive Pipeline. In einer Ausführungsform wird der Pipeline-Steuerbefehl 914 für die Pipeline-Synchronisation und zum Löschen von Daten aus einem oder mehreren Cachespeichern innerhalb der aktiven Pipeline verwendet, bevor ein Batch von Befehlen verarbeitet wird.

[0120] In manchen Ausführungsformen werden Rückgabepufferzustandsbefehle 916 verwendet, um einen Satz von Rückgabepuffern für die jeweiligen Pipelines zum Schreiben von Daten zu konfigurieren. Einige Pipeline-Operationen erfordern die Zuweisung, Auswahl oder Konfiguration eines oder mehrerer Rückgabepuffer, in die die Operationen Zwischendaten während der Verarbeitung schreiben. In manchen Ausführungsformen verwendet der Grafikprozessor auch einen oder mehrere Rückgabepuffer, um Ausgabedaten zu speichern und eine Cross-Thread-Kommunikation auszuführen. In manchen Ausführungsformen beinhaltet der Rückgabepufferzustand 916 das Auswählen der Größe und Anzahl von Rückgabepuffern, die für einen Satz von Pipeline-Operationen zu verwenden sind.

[0121] Die übrigen Befehle in der Befehlssequenz unterscheiden sich basierend auf der aktiven Pipeline für Operationen. Basierend auf einer Pipeline-Bestimmung 920 wird die Befehlssequenz auf die 3D-Pipeline 922, beginnend mit dem 3D-Pipeline-Zustand 930, oder auf die Medien-Pipeline 924, beginnend mit dem Medien-Pipeline-Zustand 940, zugeschnitten.

[0122] Die Befehle zum Konfigurieren des 3D-Pipeline-Zustands 930 beinhalten 3D-Zustandseinstellbefehle für den Vertex-Pufferzustand, den Vertex-Elementzustand, den konstanten Farbzustand, den Tiefenpufferzustand und andere Zustandsvariablen, die zu konfigurieren sind, bevor 3D-Primitivenbefehle verarbeitet werden. Die Werte dieser Befehle werden zumindest teilweise basierend auf der jeweiligen verwendeten 3D-API bestimmt. In manchen Ausführungsformen sind Befehle zum 3D-Pipeline-Zustand 930 auch in der Lage, bestimmte Pipeline-Elemente gezielt zu deaktivieren oder zu umgehen, falls diese Elemente nicht verwendet werden.

[0123] In einigen Ausführungsformen wird der 3D-Primitivenbefehl 932 verwendet, um 3D-Primitiven, die von der 3D-Pipeline verarbeitet werden sollen, zu versenden. Befehle und assoziierte Parameter, die über den 3D-Primitivenbefehl 932 an den Grafikprozessor geleitet werden, werden an die Vertex-Abruffunktion in der Grafik-Pipeline weitergeleitet. Die Vertex-Abruffunktion verwendet die Daten des 3D-Primitivenbefehls 932, um Vertex-Datenstrukturen zu generieren. Die Vertex-Datenstrukturen werden in einem oder mehreren Rückgabepuffern gespeichert. In einigen Ausführungsformen wird der 3D-Primitivenbefehl 932 verwendet, um Vertex-Operationen an 3D-Primitiven über Vertex-Shader durchzuführen. Um Vertex-Shader zu verarbeiten, versendet die 3D-Pipeline 922 Shader-Ausführungs-Threads an Grafikprozessorausführungseinheiten.

[0124] In einigen Ausführungsformen wird die 3D-Pipeline 922 über einen Ausführungsbefehl 934 oder ein Ausführungsereignis ausgelöst. In manchen Ausführungsformen löst ein Registerschreibvorgang eine Befehlsausführung aus. In manchen Ausführungsformen wird die Ausführung über einen „go"- oder „kick"-Befehl in der Befehlssequenz ausgelöst. In einer Ausführungsform wird die Befehlsausführung unter Verwen-

dung eines Pipeline-Synchronisationsbefehls ausgelöst, um die Befehlssequenz durch die Grafik-Pipeline zu flushen. Die 3D-Pipeline führt eine Geometrieverarbeitung für die 3D-Primitiven aus. Sobald die Operationen abgeschlossen sind, werden die resultierenden geometrischen Objekte rasterisiert und färbt die Pixel-Engine die resultierenden Pixel. Weitere Befehle zum Steuern des Pixel-Shadings und der Pixel-Backend-Operationen können ebenfalls für diese Operationen enthalten sein.

[0125] In einigen Ausführungsformen folgt die Grafikprozessor-Befehlssequenz 910 dem Pfad der Medien-Pipeline 924, wenn Medienoperationen ausgeführt werden. Im Allgemeinen hängt die spezielle Verwendung und Art der Programmierung für die Medien-Pipeline 924 von den durchzuführenden Medien- oder Rechenoperationen ab. Spezifische Mediendecodieroperationen können während der Mediendecodierung auf die Medien-Pipeline ausgelagert werden. In manchen Ausführungsformen kann die Medien-Pipeline auch umgangen werden und kann die Mediendecodierung vollständig oder teilweise unter Verwendung von Ressourcen durchgeführt werden, die von einem oder mehreren Allzweckverarbeitungskernen bereitgestellt werden. In einer Ausführungsform beinhaltet die Medien-Pipeline auch Elemente für Operationen einer Allzweck-Grafikprozessoreinheit (GPGPU: General-Purpose Graphics Processor Unit), wobei der Grafikprozessor verwendet wird, um SIMD-Vektoroperationen unter Verwendung von Berechnungs-Shader-Programmen, die nicht explizit mit dem Rendering von Grafikprimitiven in Zusammenhang stehen, durchzuführen.

[0126] In manchen Ausführungsformen ist die Medien-Pipeline 924 auf eine ähnliche Weise wie die 3D-Pipeline 922 konfiguriert. Ein Satz von Befehlen zum Konfigurieren des Medien-Pipeline-Zustands 940 wird verteilt oder in einer Befehlswarteschlange vor Medienobjektbefehlen 942 platziert. In manchen Ausführungsformen beinhalten Befehle für den Medien-Pipeline-Zustand 940 Daten zum Konfigurieren der Medien-Pipeline-Elemente, die zum Verarbeiten der Medienobjekte verwendet werden. Dies schließt Daten zum Konfigurieren der Videodecodierungs- und Videocodierungslogik in der Medien-Pipeline wie etwa ein Codierungs- oder Decodierungsformat ein. In manchen Ausführungsformen unterstützen Befehle für den Medien-Pipeline-Zustand 940 auch die Verwendung eines oder mehrerer Zeiger auf „indirekte“ Zustandselemente, die einen Batch von Zustandseinstellungen enthalten.

[0127] In manchen Ausführungsformen liefern Medienobjektbefehle 942 Zeiger auf Medienobjekte zur Verarbeitung durch die Medien-Pipeline. Die Medienobjekte beinhalten Speicherpuffer, die zu verarbeitende Videodaten enthalten. In manchen Ausführungsformen müssen alle Medien-Pipeline-Zustände gültig sein, bevor sie einen Medienobjektbefehl 942 ausgeben. Sobald der Pipeline-Zustand konfiguriert ist und die Medienobjektbefehle 942 in die Warteschlange eingereiht sind, wird die Medien-Pipeline 924 über einen Ausführungsbefehl 944 oder ein äquivalentes Ausführungsereignis (z. B. Registerschreibvorgang) ausgelöst. Die Ausgabe von der Medien-Pipeline 924 kann dann durch Operationen nachverarbeitet werden, die durch die 3D-Pipeline 922 oder die Medien-Pipeline 924 bereitgestellt werden. In manchen Ausführungsformen werden GPGPU-Operationen auf ähnliche Art und Weise wie Medienoperationen konfiguriert und ausgeführt.

Grafiksoftwarearchitektur

[0128] **Fig. 10** veranschaulicht eine beispielhafte Grafiksoftwarearchitektur für ein Datenverarbeitungssystem 1000 gemäß manchen Ausführungsformen. In einigen Ausführungsformen beinhaltet die Softwarearchitektur eine 3D-Grafikanwendung 1010, ein Betriebssystem 1020 und mindestens einen Prozessor 1030. In einigen Ausführungsformen weist der Prozessor 1030 einen Grafikprozessor 1032 und einen oder mehrere Allzweckprozessorkerne 1034 auf. Die Grafikanwendung 1010 und das Betriebssystem 1020 werden jeweils im Systemspeicher 1050 des Datenverarbeitungssystems ausgeführt.

[0129] In einigen Ausführungsformen enthält die 3D-Grafikanwendung 1010 ein oder mehrere Shader-Programme, einschließlich Shader-Anweisungen 1012. Die Shader-Sprache-Anweisungen können in einer High-Level-Shader-Sprache sein, wie etwa der High-Level-Shader-Sprache (HLSL) von Direct3D, der OpenGL Shader-Sprache (GLSL) und so weiter. Die Anwendung weist auch ausführbare Anweisungen 1014 in einer Maschinensprache auf, die zur Ausführung durch den Allzweckprozessorkern 1034 geeignet sind. Die Anwendung beinhaltet auch Grafikobjekte 1016, die durch Vertexdaten definiert sind.

[0130] In einigen Ausführungsformen ist das Betriebssystem 1020 ein Microsoft® Windows®-Betriebssystem von der Microsoft Corporation, ein proprietäres UNIX-ähnliches Betriebssystem oder ein Open-Source-UNIX-ähnliches Betriebssystem, das eine Variante des Linux-Kernels verwendet. Das Betriebssystem 1020 kann eine Grafik-API 1022 unterstützen, wie etwa die Direct3D-API, die OpenGL-API oder die Vulkan-API. Wenn die Direct3D-API verwendet wird, verwendet das Betriebssystem 1020 einen Frontend-Shader-Compiler 1024, um alle Shader-Anweisungen 1012 in HLSL in eine Shader-Sprache niedrigerer Ebene zu kompilie-

ren. Die Kompilierung kann eine Just-in-Time(JIT)-Kompilierung sein, oder die Anwendung kann eine Shader-Vorkompilierung durchführen. In manchen Ausführungsformen werden High-Level-Shader während der Kompilierung der 3D-Grafikanwendung 1010 zu Low-Level-Shaders kompiliert. In manchen Ausführungsformen werden die Shader-Anweisungen 1012 in einer Zwischenform bereitgestellt, wie etwa als eine Version der SPIR (Standard Portable Intermediate Representation), die durch die Vulkan-API verwendet wird.

[0131] In einigen Ausführungsformen enthält ein Benutzermodus-Grafiktreiber 1026 einen Backend-Shader-Compiler xxxx, um die Shader-Anweisungen 1012 in eine hardwarespezifische Repräsentation umzuwandeln. Wenn die OpenGL-API verwendet wird, werden Shader-Anweisungen 1012 in der GLSL-Hochsprache zum Kompilieren an einen Benutzermodus-Grafiktreiber 1026 weitergegeben. In einigen Ausführungsformen verwendet der Benutzermodus-Grafiktreiber 1026 Betriebssystem-Kernelmodus-Funktionen 1028, um mit einem Kernelmodus-Grafiktreiber 1029 zu kommunizieren. In manchen Ausführungsformen kommuniziert der Kernelmodus-Grafiktreiber 1029 mit dem Grafikprozessor 1032, um Befehle und Anweisungen zu versenden.

IP-Kern-Implementierungen

[0132] Ein oder mehrere Aspekte mindestens einer Ausführungsform können durch einen repräsentativen Code implementiert werden, der auf einem maschinenlesbaren Medium gespeichert ist, das Logik innerhalb einer integrierten Schaltung, wie zum Beispiel einem Prozessor, repräsentiert und/oder definiert. Zum Beispiel kann das maschinenlesbare Medium Anweisungen beinhalten, die verschiedene Logik innerhalb des Prozessors repräsentieren. Wenn sie durch eine Maschine gelesen werden, können die Anweisungen bewirken, dass die Maschine die Logik herstellt, um die hierin beschriebenen Techniken durchzuführen. Derartige Repräsentationen, die als „IP-Kerne“ bekannt sind, sind wiederverwendbare Logikeinheiten für eine integrierte Schaltung, die auf einem greifbaren, maschinenlesbaren Medium als ein Hardwaremodell gespeichert werden können, das die Struktur der integrierten Schaltung beschreibt. Das Hardwaremodell kann an verschiedene Kunden oder Fertigungsanlagen geliefert werden, die das Hardwaremodell in Fertigungsmaschinen laden, die die integrierte Schaltung herstellen. Die integrierte Schaltung kann derart hergestellt sein, dass die Schaltung Operationen durchführt, die in Verbindung mit irgendeiner der hierin beschriebenen Ausführungsformen beschrieben sind.

[0133] **Fig. 11A** ist ein Blockdiagramm, das ein IP-Kern-Entwicklungssystem 1100 veranschaulicht, das verwendet werden kann, um eine integrierte Schaltung herzustellen, um Operationen gemäß einer Ausführungsform durchzuführen. Das IP-Kern-Entwicklungssystem 1100 kann verwendet werden, um modulare, wiederverwendbare Designs zu erzeugen, die in ein größeres Design integriert oder verwendet werden können, um eine gesamte integrierte Schaltung (z. B. eine integrierte SOC-Schaltung) zu konstruieren. Eine Designeinrichtung 1130 kann eine Softwaresimulation 1110 eines IP-Kern-Designs in einer höheren Programmiersprache (z. B. C/C++) erzeugen. Die Softwaresimulation 1110 kann dazu verwendet werden, das Verhalten des IP-Kerns unter Verwendung eines Simulationsmodells 1112 zu entwerfen, zu testen und zu verifizieren. Das Simulationsmodell 1112 kann Funktions-, Verhaltens- und/oder Timing-Simulationen beinhalten. Ein Registertransferebenen(RTL)-Design 1115 kann dann aus dem Simulationsmodell 1112 erzeugt oder synthetisiert werden. Das RTL-Design 1115 ist eine Abstraktion des Verhaltens der integrierten Schaltung, die den Fluss digitaler Signale zwischen Hardware-Registern modelliert, einschließlich der zugehörigen Logik, die unter Verwendung der modellierten digitalen Signale ausgeführt wird. Neben einem RTL-Design 1115 können auch Designs auf niedrigerer Ebene auf der Logikebene oder der Transistorebene erzeugt, entworfen oder synthetisiert werden. Daher können die besonderen Details des anfänglichen Designs und der Simulation variieren.

[0134] Das RTL-Design 1115 oder Äquivalent kann ferner durch die Designeinrichtung zu einem Hardwaremodell 1120 synthetisiert werden, das in einer Hardwarebeschreibungssprache (HDL) oder einer anderen Repräsentation von Daten des physischen Designs vorliegen kann. Die HDL kann weiter simuliert oder getestet werden, um das IP-Kern-Design zu verifizieren. Das IP-Kern-Design kann zur Lieferung an eine Drittfertigungsanlage 1165 unter Verwendung eines nichtflüchtigen Speichers 1140 (z. B. Festplatte, Flash-Speicher oder ein beliebiges nichtflüchtiges Speicherungsmedium) gespeichert werden. Alternativ dazu kann das IP-Kern-Design über eine drahtgebundene Verbindung 1150 oder eine drahtlose Verbindung 1160 übertragen werden (z. B. über das Internet). Die Fertigungsanlage 1165 kann dann eine integrierte Schaltung fertigen, die zumindest teilweise auf dem IP-Kern-Design basiert. Die gefertigte integrierte Schaltung kann dazu konfiguriert sein, Operationen gemäß mindestens einer hierin beschriebenen Ausführungsform durchzuführen.

[0135] Fig. 11B veranschaulicht eine Querschnittsseitenansicht einer Integrierte-Schaltung-Package-Baugruppe 1170 gemäß manchen hierin beschriebenen Ausführungsformen. Die Integrierte-Schaltung-Package-Baugruppe 1170 veranschaulicht eine Implementierung eines oder mehrerer Prozessoren oder einer oder mehrerer Beschleunigervorrichtungen, wie hierin beschrieben. Die Package-Baugruppe 1170 beinhaltet mehrere Hardwarelogikeinheiten 1172, 1174, die mit einem Substrat 1180 verbunden sind. Die Logik 1172, 1174 kann zumindest teilweise in konfigurierbarer Logik- oder Festfunktionalitätslogikhardware implementiert werden und kann einen oder mehrere Teile beliebiger des einen oder der mehreren Prozessorkerne, des einen oder der mehreren Grafikprozessoren oder anderer hierin beschriebener Beschleunigervorrichtungen beinhalten. Jede Logikeinheit 1172, 1174 kann in einem Halbleiter-Die implementiert und über eine Interconnect-Struktur 1173 mit dem Substrat 1180 gekoppelt werden. Die Interconnect-Struktur 1173 kann dazu konfiguriert sein, elektrische Signale zwischen der Logik 1172, 1174 und dem Substrat 1180 zu routen, und kann Interconnects, wie etwa unter anderem Kontakthügel oder Säulen, beinhalten. In einigen Ausführungsformen kann die Interconnect-Struktur 1173 dazu ausgelegt sein, elektrische Signale, zum Beispiel Eingabe/Ausgabe (E/A)-Signale und/oder Leistungs- oder Massesignale, zu routen, die mit dem Betrieb der Logik 1172, 1174 assoziiert sind. In einigen Ausführungsformen ist das Substrat 1180 ein epoxidbasiertes Laminatsubstrat. Das Substrat 1180 kann in anderen Ausführungsformen andere geeignete Arten von Substraten beinhalten. Die Package-Baugruppe 1170 kann über ein Package-Interconnect 1183 mit anderen elektrischen Vorrichtungen verbunden sein. Das Package-Interconnect 1183 kann mit einer Oberfläche des Substrats 1180 gekoppelt sein, um elektrische Signale zu anderen elektrischen Vorrichtungen, wie etwa einer Hauptplatine, einem anderen Chipsatz oder einem Mehrchipmodul zu routen.

[0136] In einigen Ausführungsformen sind die Logikeinheiten 1172, 1174 elektrisch mit einer Brücke 1182 gekoppelt, die dazu konfiguriert ist, elektrische Signale zwischen der Logik 1172, 1174 zu routen. Die Brücke 1182 kann eine dichte Interconnect-Struktur sein, die eine Route für elektrische Signale bereitstellt. Die Brücke 1182 kann ein Brückensubstrat beinhalten, das aus Glas oder einem geeigneten Halbleitermaterial gebildet ist. Elektrische Routing-Merkmale können auf dem Brückensubstrat ausgebildet sein, um eine Chip-zu-Chip-Verbindung zwischen der Logik 1172, 1174 bereitzustellen.

[0137] Obwohl zwei Logikeinheiten 1172, 1174 und eine Brücke 1182 veranschaulicht sind, können hierin beschriebene Ausführungsformen mehr oder weniger Logikeinheiten auf einem oder mehreren Dies beinhalten. Der eine oder die mehreren Dies können durch keine oder mehr Brücken verbunden sein, da die Brücke 1182 ausgelassen werden kann, wenn die Logik auf einem einzelnen Die enthalten ist. Alternativ dazu können mehrere Dies oder Logikeinheiten durch eine oder mehrere Brücken verbunden sein. Zusätzlich dazu können mehrere Logikeinheiten, Dies und Brücken in anderen möglichen Konfigurationen, einschließlich dreidimensionaler Konfigurationen, miteinander verbunden sein.

[0138] Fig. 11C veranschaulicht eine Package-Baugruppe 1190, die mehrere Einheiten von Hardwarelogik-Chiplets beinhaltet, die mit einem Substrat 1180 (z. B. Basis-Die) verbunden sind. Eine Grafikverarbeitungseinheit, ein Parallelprozessor und/oder ein Berechnungsbeschleuniger, wie hierin beschrieben, können aus verschiedenen Silizium-Chiplets bestehen, die separat hergestellt werden. In diesem Zusammenhang ist ein Chiplet eine zumindest teilweise integrierte Schaltung in einem Package, die verschiedene Logikeinheiten beinhaltet, die mit anderen Chiplets zu einem größeren Package zusammengebaut werden können. Ein diverser Satz von Chiplets mit unterschiedlicher IP-Kern-Logik kann zu einer einzigen Vorrichtung zusammengebaut werden. Darüber hinaus können die Chiplets mittels aktiver Interposer-Technologie in einen Basis-Die oder ein Basis-Chiplet integriert werden. Die hierin beschriebenen Konzepte ermöglichen die Verbindung und Kommunikation zwischen den verschiedenen Formen von IP innerhalb der GPU. IP-Kerne können unter Verwendung unterschiedlicher Prozesstechnologien hergestellt und während der Herstellung zusammengestellt werden, wodurch die Komplexität zum Konvergieren mehrerer IPs, insbesondere auf einem großen SoC mit mehreren IP-Varianten, für denselben Herstellungsprozess vermieden wird. Die Nutzung mehrerer Prozesstechnologien verkürzt die Markteinführungszeit und bietet eine kostengünstige Möglichkeit, mehrere Produkt-SKUs zu erstellen. Darüber hinaus sind die disaggregierten IPs für ein unabhängiges Ansteuern mit Leistung besser geeignet, Komponenten, die bei einer bestimmten Arbeitslast nicht verwendet werden, können ausgeschaltet werden, wodurch der Gesamtleistungsverbrauch reduziert wird.

[0139] Die Hardware-Logik-Chiplets können Spezialhardware-Logik-Chiplets 1172, Logik- oder E/A-Chiplets 1174 und/oder Speicher-Chiplets 1175 beinhalten. Die Hardware-Logik-Chiplets 1172 und die Logik- oder E/A-Chiplets 1174 können zumindest teilweise in konfigurierbarer Logik- oder Festfunktionalitätslogikhardware implementiert werden und können einen oder mehrere Teile beliebiger des einen oder der mehreren Prozessorkerne, des einen oder der mehreren Grafikprozessoren, der Parallelprozessoren oder anderer hie-

rin beschriebener Beschleunigervorrichtungen beinhalten. Die Speicher-Chiplets 1175 können DRAM(z. B. GDDR, HBM)-Speicher oder Cache(SRAM)-Speicher sein.

[0140] Jedes Chiplet kann als separater Halbleiter-Die gefertigt und über eine Interconnect-Struktur 1173 mit dem Substrat 1180 gekoppelt werden. Die Interconnect-Struktur 1173 kann dazu konfiguriert sein, elektrische Signale zwischen den verschiedenen Chiplets und der Logik innerhalb des Substrats 1180 zu routen. Die Interconnect-Struktur 1173 kann Interconnects wie etwa unter anderem Kontakthügel oder Säulen beinhalten. In manchen Ausführungsformen kann die Interconnect-Struktur 1173 dazu konfiguriert sein, elektrische Signale, wie etwa zum Beispiel Eingabe/Ausgabe(E/A)-Signale und/oder Leistungs- oder Massesignale, zu routen, die mit dem Betrieb der Logik, E/A und Speicher-Chiplets assoziiert sind.

[0141] In einigen Ausführungsformen ist das Substrat 1180 ein epoxidbasiertes Laminatsubstrat. Das Substrat 1180 kann in anderen Ausführungsformen andere geeignete Arten von Substraten beinhalten. Die Package-Baugruppe 1190 kann über ein Package-Interconnect 1183 mit anderen elektrischen Vorrichtungen verbunden sein. Das Package-Interconnect 1183 kann mit einer Oberfläche des Substrats 1180 gekoppelt sein, um elektrische Signale zu anderen elektrischen Vorrichtungen, wie etwa einer Hauptplatine, einem anderen Chipsatz oder einem Mehrchipmodul zu routen.

[0142] In manchen Ausführungsformen können ein Logik- oder E/A-Chiplet 1174 und ein Speicher-Chiplet 1175 elektrisch über eine Brücke 1187 gekoppelt sein, die dazu konfiguriert ist, elektrische Signale zwischen der Logik oder dem E/A-Chiplet 1174 und einem Speicher-Chiplet 1175 zu routen. Die Brücke 1187 kann eine dichte Interconnect-Struktur sein, die eine Route für elektrische Signale bereitstellt. Die Brücke 1187 kann ein Brückensubstrat beinhalten, das aus Glas oder einem geeigneten Halbleitermaterial gebildet ist. Elektrische Routing-Merkmale können auf dem Brückensubstrat ausgebildet sein, um eine Chip-zu-Chip-Verbindung zwischen dem Logik- oder E/A-Chiplet 1174 und einem Speicher-Chiplet 1175 bereitzustellen. Die Brücke 1187 kann auch als Siliziumbrücke oder Interconnect-Brücke bezeichnet werden. Zum Beispiel ist die Brücke 1187 in einigen Ausführungsformen eine eingebettete Multi-Die-Interconnect-Brücke (EMIB: Embedded Multi-Die Interconnect Bridge). In einigen Ausführungsformen kann die Brücke 1187 einfach eine direkte Verbindung von einem Chiplet zu einem anderen Chiplet sein.

[0143] Das Substrat 1180 kann Hardwarekomponenten für die E/A 1191, den Cachespeicher 1192 und andere Hardwarelogik 1193 beinhalten. Ein Fabric 1185 kann in das Substrat 1180 eingebettet sein, um eine Kommunikation zwischen den verschiedenen Logik-Chiplets und der Logik 1191, 1193 innerhalb des Substrats 1180 zu ermöglichen. In einer Ausführungsform können die E/A 1191, die Struktur 1185, der Cache, die Brücke und andere Hardwarelogik 1193 in einen Basis-Die integriert sein, der auf dem Substrat 1180 geschichtet ist.

[0144] In verschiedenen Ausführungsformen kann eine Package-Baugruppe 1190 weniger oder mehr Komponenten und Chiplets beinhalten, die durch ein Fabric 1185 oder eine oder mehrere Brücken 1187 miteinander verbunden sind. Die Chiplets innerhalb der Package-Baugruppe 1190 können in einer 3D- oder 2,5D-Anordnung angeordnet sein. Im Allgemeinen können Brückenstrukturen 1187 verwendet werden, um ein Punkt-zu-Punkt-Interconnect zwischen beispielsweise Logik- oder E/A-Chiplets und Speicher-Chiplets zu ermöglichen. Das Fabric 1185 kann verwendet werden, um die verschiedenen Logik- und/oder E/A-Chiplets (z. B. Chiplets 1172, 1174, 1191, 1193) miteinander, mit anderen Logik- und/oder E/A-Chiplets zu verbinden. In einer Ausführungsform kann der Cachespeicher 1192 innerhalb des Substrats als ein globaler Cache für die Package-Baugruppe 1190, ein Teil eines verteilten globalen Cache oder als ein dedizierter Cache für das Fabric 1185 fungieren.

[0145] **Fig. 11D** veranschaulicht eine Package-Baugruppe 1194 einschließlich austauschbarer Chiplets 1195 gemäß einer Ausführungsform. Die austauschbaren Chiplets 1195 können in standardisierte Steckplätze auf einem oder mehreren Basis-Chiplets 1196, 1198 montiert werden. Die Basis-Chiplets 1196, 1198 können über ein Brücken-Interconnect 1197 gekoppelt sein, das den anderen hierin beschriebenen Brücken-Interconnects ähnlich sein kann und beispielsweise eine EMIB sein kann. Speicher-Chiplets können auch über ein Brücken-Interconnect mit Logik- oder E/A-Chiplets verbunden sein. E/A- und Logik-Chiplets können über ein Interconnect-Fabric kommunizieren. Die Basis-Chiplets können jeweils einen oder mehrere Steckplätze in einem standardisierten Format für Logik oder E/A oder Speicher/Cache unterstützen.

[0146] In einer Ausführungsform können SRAM- und Leistungslieferschaltungen in einem oder mehreren der Basis-Chiplets 1196, 1198 gefertigt werden, die unter Verwendung einer anderen Prozesstechnologie relativ zu den austauschbaren Chiplets 1195, die auf den Basis-Chiplets gestapelt sind, gefertigt werden können.

Beispielsweise können die Basis-Chiplets 1196, 1198 unter Verwendung einer größeren Prozesstechnologie hergestellt werden, während die austauschbaren Chiplets unter Verwendung einer kleineren Prozesstechnologie hergestellt werden können. Einer oder mehrere der austauschbaren Chiplets 1195 können Speicher(z. B. DRAM)-Chiplets sein. Für die Package-Baugruppe 1194 können unterschiedliche Speicherdichten basierend auf der Leistung und/oder Performanz ausgewählt werden, die für das Produkt, das die Package-Baugruppe 1194 verwendet, angestrebt wird. Zusätzlich dazu können Logik-Chiplets mit einer anderen Anzahl von Typen von Funktionseinheiten zum Zeitpunkt des Zusammensetzens basierend auf der Leistung und/oder Performanz, die für das Produkt angestrebt wird, ausgewählt werden. Darüber hinaus können Chiplets, die IP-Logikkerne unterschiedlicher Typen enthalten, in die austauschbaren Chiplet-Steckplätze eingefügt werden, wodurch Hybridprozessordesigns ermöglicht werden, die IP-Blöcke unterschiedlicher Technologie Mix-and-Match-artig frei kombinieren können.

Beispielhafte integrierte System-on-Chip-Schaltung

[0147] Fig. 12-13 veranschaulichen beispielhafte integrierte Schaltungen und assoziierte Grafikprozessoren, die unter Verwendung eines oder mehrerer IP-Kerne hergestellt werden können, gemäß verschiedenen hierin beschriebenen Ausführungsformen. Zusätzlich zu den Veranschaulichungen können andere Logik und Schaltungen enthalten sein, einschließlich zusätzlicher Grafikprozessoren/-kerne, Peripherieschnittstellensteuerungen oder Allzweckprozessorkerne.

[0148] Fig. 12 ist ein Blockdiagramm, das eine beispielhafte integrierte System-on-Chip-Schaltung 1200 veranschaulicht, die unter Verwendung eines oder mehrerer IP-Kerne gefertigt werden kann, gemäß einer Ausführungsform. Die beispielhafte integrierte Schaltung 1200 enthält einen oder mehrere Anwendungsprozessoren 1205 (z. B. CPUs), mindestens einen Grafikprozessor 1210 und kann zusätzlich einen Bildprozessor 1215 und/oder einen Videoprozessor 1220 enthalten, von denen jeder ein modularer IP-Kern aus derselben oder mehreren verschiedenen Designeinrichtungen sein kann. Die integrierte Schaltung 1200 beinhaltet eine Peripherie- oder Buslogik, die eine USB-Steuerung 1225, eine UART-Steuerung 1230, eine SPI/SDIO-Steuerung 1235 und eine I2S/I2C-Steuerung 1240 aufweist. Außerdem kann die integrierte Schaltung eine Anzeigevorrichtung 1245 beinhalten, die mit einer HDMI(High Definition Multimedia Interface)-Steuerung 1250 und/oder einer MIPI(Mobile Industry Processor Interface)-Anzeigeschnittstelle 1255 gekoppelt ist. Die Speicherung kann durch ein Flash-Speichersubsystem 1260 bereitgestellt werden, das Flash-Speicher und eine Flash-Speichersteuerung beinhaltet. Eine Speicherschnittstelle kann über eine Speichersteuerung 1265 zum Zugriff auf SDRAM- oder SRAM-Speichervorrichtungen bereitgestellt werden. Einige integrierte Schaltungen weisen zusätzlich eine eingebettete Sicherheits-Engine 1270 auf.

[0149] Fig. 13-14 sind Blockdiagramme, die beispielhafte Grafikprozessoren zur Verwendung in einem SoC veranschaulichen, gemäß hierin beschriebenen Ausführungsformen. **Fig. 13** veranschaulicht einen beispielhaften Grafikprozessor 1310 einer integrierten System-on-Chip-Schaltung, der gemäß einer Ausführungsform unter Verwendung eines oder mehrerer IP-Kerne hergestellt werden kann. **Fig. 13B** veranschaulicht einen zusätzlichen beispielhaften Grafikprozessor 1340 einer integrierten System-on-Chip-Schaltung, der gemäß einer Ausführungsform unter Verwendung eines oder mehrerer IP-Kerne hergestellt werden kann. Der Grafikprozessor 1310 von **Fig. 13** ist ein Beispiel für einen Niederleistungs-Grafikprozessorkern. Der Grafikprozessor 1340 von **Fig. 13B** ist ein Beispiel für einen Grafikprozessorkern mit höherer Performanz. Jeder der Grafikprozessoren 1310, 1340 kann Varianten des Grafikprozessors 1210 von **Fig. 12** sein.

[0150] Wie in **Fig. 13** gezeigt, beinhaltet der Grafikprozessor 1310 einen Vertex-Prozessor 1305 und einen oder mehrere Fragment-Prozessoren 1315A-1315N (z. B. 1315A, 1315B, 1315C, 1315D bis 1315N-1 und 1315N). Der Grafikprozessor 1310 kann verschiedene Shader-Programme über separate Logik ausführen, sodass der Vertex-Prozessor 1305 zum Ausführen von Operationen für Vertex-Shader-Programme optimiert ist, während der eine oder die mehreren Fragment-Prozessoren 1315A-1315N Fragment(z. B. Pixel)-Shading-Operationen für Fragment- oder Pixel-Shader-Programme ausführen. Der Vertex-Prozessor 1305 führt die Vertex-Verarbeitungsstufe der 3D-Grafik-Pipeline aus und erzeugt Primitive und Vertex-Daten. Der eine oder die mehreren Fragment-Prozessoren 1315A-1315N verwenden die durch den Vertex-Prozessor 1305 erzeugten Primitiven und Vertex-Daten, um einen Framepuffer zu erzeugen, der auf einer Anzeigevorrichtung angezeigt wird. In einer Ausführungsform sind der eine oder die mehreren Fragment-Prozessoren 1315A-1315N dazu optimiert, Fragment-Shader-Programme, wie sie in der OpenGL-API bereitgestellt werden, auszuführen, die verwendet werden können, um ähnliche Operationen wie ein Pixel-Shader-Programm durchzuführen, wie es in der Direct-3D-API bereitgestellt wird.

[0151] Der Grafikprozessor 1310 enthält zusätzlich eine oder mehrere Speicherverwaltungseinheiten (MMUs: Memory Management Units) 1320A-1320B, den einen oder die mehreren Caches 1325A-1325B und das eine oder die mehreren Schaltung-Interconnects 1330A-1330B. Die eine oder die mehreren MMUs 1320A-1320B stellen eine Abbildung von virtuellen auf physische Adressen für den Grafikprozessor 1310 bereit, einschließlich für den Vertex-Prozessor 1305 und/oder den einen oder die mehreren Fragment-Prozessoren 1315A-1315N, die zusätzlich zu Vertex- oder Bild-/Texturdaten, die in dem einen oder den mehreren Caches 1325A-1325B gespeichert sind, Vertex- oder Bild-/Texturdaten referenzieren können, die im Speicher gespeichert sind. In einer Ausführungsform können die eine oder die mehreren MMUs 1320A-1320B mit anderen MMUs innerhalb des Systems synchronisiert sein, einschließlich einer oder mehrerer MMUs, die mit dem einen oder den mehreren Anwendungsprozessoren 1205, dem Bildprozessor 1215 und/oder dem Videoprozessor 1220 von **Fig. 12** assoziiert sind, sodass jeder Prozessor 1205-1220 an einem gemeinsam genutzten oder vereinheitlichten virtuellen Speichersystem teilnehmen kann. Das eine oder die mehreren Schaltung-Interconnects 1330A-1330B ermöglichen gemäß Ausführungsformen, dass der Grafikprozessor 1310 mit anderen IP-Kernen innerhalb des SoC eine Schnittstelle bildet, entweder über einen internen Bus des SoC oder über eine direkte Verbindung.

[0152] Wie in **Fig. 14** gezeigt, enthält der Grafikprozessor 1340 die eine oder die mehreren MMUs 1320A-1320B, den einen oder die mehreren Caches 1325A-1325B und das eine oder die mehreren Schaltung-Interconnects 1330A-1330B des Grafikprozessors 1310 von **Fig. 13A**. Der Grafikprozessor 1340 beinhaltet einen oder mehrere Shader-Kerne 1355A-1355N (z. B. 1455A, 1355B, 1355C, 1355D, 1355E, 1355F bis 1355N-1 und 1355N), was für eine vereinheitlichte Shader-Kernarchitektur sorgt, in der ein einzelner Kern oder Kerntyp alle Arten von programmierbarem Shader-Code ausführen kann, einschließlich Shader-Programmcode zum Implementieren von Vertex-Shadern, Fragment-Shadern und/oder Berechnungs-Shadern. Die genaue Anzahl der vorhandenen Shader-Kerne kann zwischen Ausführungsformen und Implementierungen variieren. Außerdem beinhaltet der Grafikprozessor 1340 einen Inter-Kern-Aufgabenmanager 1345, der als ein Thread-Dispatcher zum Versenden von Ausführungs-Threads an einen oder mehrere Shader-Kerne 1355A-1355N fungiert, und eine Kachelungseinheit 1358 zum Beschleunigen von Kacheloperationen für kachelbasiertes Rendering, wobei die Rendering-Operationen für eine Szene in einen Bildraum unterteilt sind, um zum Beispiel lokale räumliche Kohärenz innerhalb einer Szene auszunutzen oder um die Verwendung von internen Caches zu optimieren.

RAYTRACING MIT MASCHINELLEM LERNEN

[0153] Wie oben erwähnt, ist Raytracing (Strahlverfolgung) eine Grafikverarbeitungstechnik, bei der ein Lichttransport durch physisch basiertes Rendern simuliert wird. Eine der Schlüsseloperationen beim Raytracing ist das Verarbeiten einer Sichtbarkeitsanfrage, die ein Traversierungs- und Überschneidungstesten von Knoten in einer Hüllkörperhierarchie (BVH) erfordert.

[0154] Auf Ray- und Pfad-Tracing basierte Techniken berechnen Bilder durch Verfolgen von Strahlen und Pfaden durch jedes Pixel und durch Verwenden eines zufälligen Sampling, um fortgeschrittene Effekte, wie etwa Schatten, Glanz, indirekte Beleuchtung usw., zu berechnen. Das Verwenden von nur wenigen Samples ist schnell, produziert aber verrauschte Bilder, während das Verwenden vieler Samples Bilder hoher Qualität produziert, aber unerschwinglich ist.

[0155] Maschinelles Lernen beinhaltet eine beliebige Schaltungsanordnung, einen beliebigen Programmcode oder eine beliebige Kombination davon, die bzw. der in der Lage ist, die Leistungsfähigkeit einer spezifizierten Aufgabe progressiv zu verbessern oder progressiv genauere Vorhersagen oder Entscheidungen zu liefern. Manche Maschinenlern-Engines können diese Aufgaben durchführen oder diese Vorhersagen/Entscheidungen liefern, ohne explizit dazu programmiert zu sein, die Aufgaben durchzuführen oder die Vorhersagen/Entscheidungen zu liefern. Es gibt eine Vielzahl von Maschinenlerntechniken, darunter (jedoch nicht beschränkt auf) überwachtes und semi-überwachtes Lernen, unüberwachtes Lernen und bestärkendes Lernen.

[0156] In den letzten Jahren ist es zu einer Durchbruchlösung für das Ray-/Pfad-Tracing zur Echtzeitverwendung in Form von „Entrauschen“ - dem Prozess des Verwendens von Bildverarbeitungstechniken, um qualitativ hochwertige gefilterte/entrauschte Bilder aus rauschbehafteten Eingaben mit niedriger Sample-Anzahl zu erzeugen - gekommen. Die effektivsten Entrauschungstechniken beruhen auf Maschinenlerntechniken, bei denen eine Maschinenlern-Engine lernt, wie ein rauschbehaftetes Bild aussehen würde, wenn es mit mehr Samples berechnet worden wäre. Bei einer bestimmten Implementierung wird das maschinelle Lernen durch ein faltendes neuronales Netzwerk (CNN: Convolutional Neural Network) durchgeführt; die zugrundeliegen-

den Prinzipien der Erfindung sind jedoch nicht auf eine CNN-Implementierung beschränkt. Bei einer solchen Implementierung werden Trainingsdaten mit Eingaben mit geringer Sample-Anzahl und Ground-Truth produziert. Das CNN wird dahingehend trainiert, das konvergierte Pixel aus einer Nachbarschaft von rauschbehafteten Pixeleingaben um das betreffende Pixel herum vorherzusagen.

[0157] Obwohl sie nicht perfekt ist, hat sich diese KI-basierte Entrauschungstechnik als überraschend effektiv erwiesen. Der Vorbehalt ist jedoch, dass gute Trainingsdaten erforderlich sind, da das Netzwerk ansonsten die falschen Ergebnisse vorhersagen kann. Falls zum Beispiel ein Animationsfilmstudio ein Entrauschungs-CNN an älteren Filmen mit Szenen an Land trainiert und dann versucht, das trainierte CNN zu verwenden, um Frames aus einem neuen Film, der auf Wasser spielt, zu entrauschen, kommt es zu einer suboptimalen Durchführung der Entrauschungsoperation.

[0158] Zur Behandlung dieses Problems können Lerndaten während des Renderns dynamisch gesammelt werden und eine Maschinenlern-Engine, wie etwa ein CNN, kann basierend auf den Daten, mit denen sie gegenwärtig ausgeführt wird, kontinuierlich trainiert werden, wodurch die Maschinenlern-Engine für die vorliegende Aufgabe kontinuierlich verbessert wird. Daher kann immer noch eine Trainingsphase vor der Laufzeit durchgeführt werden, aber die Maschinenlern-Gewichtungen nach Bedarf während der Laufzeit weiter anpassen. Dadurch werden die hohen Kosten des Berechnens der Referenzdaten, die für das Training benötigt werden, vermieden, indem die Erzeugung von Lerndaten auf einen Teilbereich des Bildes in jedem Frame oder aller N Frames beschränkt wird. Insbesondere werden die rauschbehafteten Eingaben eines Frames zum Entrauschen des vollen Frames mit dem aktuellen Netzwerk erzeugt. Zusätzlich wird ein kleiner Bereich von Referenzpixeln erzeugt und zum kontinuierlichen Training verwendet, wie nachfolgend beschrieben.

[0159] Obwohl hierin eine CNN-Implementierung beschrieben ist, kann eine beliebige Form einer Maschinenlern-Engine verwendet werden, darunter unter anderem Systeme, die überwachtes Lernen (z. B. Erstellen eines mathematischen Modells eines Datensatzes, der sowohl die Eingaben als auch die gewünschten Ausgaben enthält), unüberwachtes Lernen (z. B. das die Eingabedaten für bestimmte Strukturtypen evaluiert) und/oder eine Kombination aus überwachtem und unüberwachtem Lernen durchführen.

[0160] Bestehende Entrauschungsimplementierungen arbeiten in einer Trainingsphase und einer Laufzeitphase. Während der Trainingsphase wird eine Netzwerktopologie definiert, die einen Bereich von NxN Pixeln mit verschiedenen Pro-Pixel-Datenkanälen, wie etwa Pixelfarbe, Tiefe, Normale, Normalenabweichung, Primitiv-IDs und Albedo, empfängt und eine endgültige Pixelfarbe erzeugt. Ein Satz von „repräsentativen“ Trainingsdaten wird durch Verwenden einer einem Frame entsprechenden Menge von Eingaben mit niedriger Sample-Anzahl und Referenzieren der „gewünschten“ Pixelfarben, die mit einer sehr hohen Sample-Anzahl berechnet werden, erzeugt. Das Netzwerk wird auf diese Eingaben hin trainiert, wodurch ein Satz von „idealen“ Gewichtungen für das Netzwerk erzeugt wird. In diesen Implementierungen werden die Referenzdaten verwendet, um die Gewichtungen des Netzwerks zu trainieren, um die Ausgabe des Netzwerks am nächsten an das gewünschte Ergebnis anzupassen.

[0161] Zur Laufzeit werden die gegebenen, vorberechneten idealen Netzwerkgewichtungen geladen und das Netzwerk wird initialisiert. Für jedes Frame wird ein Bild mit geringer Sample-Anzahl von Entrauschungseingaben (d. h. das gleiche wie für das Training verwendet) erzeugt. Für jedes Pixel wird die gegebene Nachbarschaft von Eingaben der Pixel durch das Netzwerk geleitet, um die „entrauschte“ Pixelfarbe vorherzusagen, wodurch ein entrauschtes Frame erzeugt wird.

[0162] **Fig. 15** veranschaulicht eine Anfangstrainingsimplementierung. Eine Maschinenlern-Engine 1500 (z. B. ein CNN) empfängt einen Bereich von N × N Pixeln als Bilddaten 1702 mit hoher Sample-Anzahl mit verschiedenen Pro-Pixel-Datenkanälen, wie etwa Pixelfarbe, Tiefe, Normale, Normalenabweichung, Primitiv-IDs und Albedo, und erzeugt endgültige Pixelfarben. Repräsentative Trainingsdaten werden durch Verwenden einer einem Frame entsprechenden Menge von Eingaben 1501 mit niedriger Sample-Anzahl erzeugt. Das Netzwerk wird auf diese Eingaben hin trainiert, wodurch ein Satz von „idealen“ Gewichtungen 1505 erzeugt wird, die die Maschinenlern-Engine 1500 anschließend verwendet, um Bilder mit niedriger Sample-Anzahl zur Laufzeit zu entrauschen.

[0163] Um die obigen Techniken zu verbessern, wird die Entrauschungsphase zum Erzeugen neuer Trainingsdaten in jedem Frame oder eines Teilsatzes von Frames (z. B. alle N Frames, wobei N = 2, 3, 4, 10, 25 usw.) erweitert. Insbesondere werden, wie in **Fig. 16** veranschaulicht, ein oder mehrere Bereiche in jedem Frame ausgewählt, hier bezeichnet als „neue Referenzbereiche“ 1602, die mit einer hohen Sample-Anzahl in einen separaten Puffer 1604 mit hoher Sample-Anzahl gerendert werden. Ein Puffer 1603 mit niedriger Sam-

ple-Anzahl speichert das Eingangsframe 1601 mit niedriger Sample-Anzahl (einschließlich des Bereichs 1604 mit geringer Sample-Anzahl, der dem neuen Referenzbereich 1602 entspricht).

[0164] Der Ort des neuen Referenzbereichs 1602 kann zufällig gewählt werden. Alternativ dazu kann der Ort des neuen Referenzbereichs 1602 auf eine vorgegebene Weise für jedes neue Frame angepasst werden (z. B. unter Verwendung einer vordefinierten Bewegung des Bereichs zwischen Frames, beschränkt auf einen vorgegebenen Bereich in der Mitte des Frames usw.).

[0165] Unabhängig davon, wie der neue Referenzbereich ausgewählt wird, wird er durch die Maschinenlern-Engine 1600 verwendet, um die trainierten Gewichtungen 1605, die zum Entrauschen verwendet werden, kontinuierlich zu verfeinern und zu aktualisieren. Insbesondere werden Referenzpixelfarben von jedem neuen Referenzbereich 1602 und rauschbehaftete Referenzpixeleingaben von einem entsprechenden Bereich 1607 mit niedriger Sample-Anzahl gerendert. Ein ergänzendes Training wird dann auf der Maschinenlern-Engine 1600 unter Verwendung des Referenzbereichs 1602 mit hoher Sample-Anzahl und des entsprechenden Bereichs 1607 mit niedriger Sample-Anzahl durchgeführt. Im Gegensatz zum anfänglichen Training wird dieses Training kontinuierlich während der Laufzeit für jeden neuen Referenzbereich 1602 durchgeführt - wodurch sichergestellt wird, dass die Maschinenlern-Engine 1600 präzise trainiert wird. Zum Beispiel können Pro-Pixel-Datenkanäle (z. B. Pixelfarbe, Tiefe, Normale, Normalenabweichung usw.) evaluiert werden, die die Maschinenlern-Engine 1600 verwendet, um Anpassungen an den trainierten Gewichtungen 1605 vorzunehmen. Wie im Trainingsfall (**Fig. 15**), wird die Maschinenlern-Engine 1600 auf einen Satz idealer Gewichtungen 1605 hin zum Entfernen von Rauschen aus dem Eingangsframe 1601 mit niedriger Sample-Anzahl trainiert, um das entrauschte Frame 1620 zu erzeugen. Die trainierten Gewichtungen 1605 werden jedoch basierend auf neuen Bildcharakteristiken neuer Typen von Eingangsframes 1601 mit niedriger Sample-Anzahl kontinuierlich aktualisiert.

[0166] Die Neutrainingsoperationen, die durch die Maschinenlern-Engine 1600 durchgeführt werden, können gleichzeitig in einem Hintergrundprozess auf der Grafikprozessoreinheit (GPU) oder dem Host-Prozessor ausgeführt werden. Die Renderschleife, die als eine Treiberkomponente und/oder eine GPU-Hardwarekomponente implementiert sein kann, kann kontinuierlich neue Trainingsdaten (z. B. in der Form neuer Referenzbereiche 1602) erzeugen, die sie in eine Warteschlange platziert. Der Hintergrundtrainingsprozess, der auf der GPU oder dem Host-Prozessor ausgeführt wird, kann kontinuierlich die neuen Trainingsdaten aus dieser Warteschlange lesen, die Maschinenlern-Engine 1600 neu trainieren und sie mit neuen Gewichtungen 1605 in geeigneten Intervallen aktualisieren.

[0167] Fig. 17 veranschaulicht ein Beispiel für eine solche Implementierung, bei der der Hintergrundtrainingsprozess 1700 durch die Host-CPU 1710 implementiert wird. Insbesondere verwendet der Hintergrundtrainingsprozess 1700 den neuen Referenzbereich 1602 mit hoher Sample-Anzahl und den entsprechenden Bereich 1604 mit niedriger Sample-Anzahl, um die trainierten Gewichtungen 1605 kontinuierlich zu aktualisieren, wodurch die Maschinenlern-Engine 1600 aktualisiert wird.

[0168] Wie in **Fig. 18A** für das nicht beschränkende Beispiel für ein Online-Spiel mit mehreren Spielern veranschaulicht, erzeugen verschiedene Host-Maschinen 1820-1822 individuell Referenzbereiche, die ein Hintergrundtrainingsprozess 1700A-C zu einem Server 1800 (z. B. einen Gaming-Server) überträgt. Der Server 1800 führt dann ein Training auf einer Maschinenlern-Engine 1810 durch Verwenden der neuen Referenzbereiche, die von jedem der Hosts 1821-1822 empfangen werden, durch, wobei die Gewichtungen 1805 aktualisiert werden, wie zuvor beschrieben. Er überträgt diese Gewichtungen 1805 zu den Host-Maschinen 1820, die die Gewichtungen 1605A-C speichern, wodurch jede einzelne Maschinenlern-Engine (nicht gezeigt) aktualisiert wird. Da dem Server 1800 eine große Anzahl von Referenzbereichen in einem kurzen Zeitraum bereitgestellt werden kann, kann er die Gewichtungen für eine beliebige gegebene Anwendung (z. B. ein Online-Spiel), die durch die Benutzer ausgeführt wird, effizient und präzise aktualisieren.

[0169] Wie in **Fig. 18B** veranschaulicht, können die verschiedenen Host-Maschinen neue trainierte Gewichtungen (z. B. basierend auf Trainings-/Referenzbereichen 1602, wie zuvor beschrieben) erzeugen und die neuen trainierten Gewichtungen mit einem Server 1800 (z. B. einem Gaming-Server) teilen oder alternativ dazu ein Peer-zu-Peer-Sharing-Protokoll verwenden. Eine Maschinenlernverwaltungskomponente 1810 auf dem Server erzeugt einen Satz kombinierter Gewichtungen 1805 unter Verwendung der neuen Gewichtungen, die von jeder der Host-Maschinen empfangen werden. Die kombinierten Gewichtungen 1805 können zum Beispiel ein Durchschnitt sein, der aus den neuen Gewichtungen erzeugt und kontinuierlich aktualisiert wird, wie hierin beschrieben. Sobald erzeugt, können Kopien der kombinierten Gewichtungen 1605A-C über-

tragen und auf jeder der Host-Maschinen 1820-1821 gespeichert werden, die dann die kombinierten Gewichtungen verwenden können, wie hierin beschrieben, um Entrauschungsoperationen durchzuführen.

[0170] Der Halbregelkreis-Aktualisierungsmechanismus kann auch vom Hardwarehersteller verwendet werden. Zum Beispiel kann das Referenznetzwerk als Teil des Treibers enthalten sein, der durch den Hardwarehersteller verteilt wird. Wenn der Treiber neue Trainingsdaten unter Verwendung der hierin beschriebenen Techniken erzeugt und diese kontinuierlich an den Hardwarehersteller zurückgibt, verwendet der Hardwarehersteller diese Informationen, um seine Maschinenlernimplementierungen für die nächste Treiberaktualisierung weiter zu verbessern.

[0171] In einer beispielhaften Implementierung (z. B. bei Batch-Film-Rendering auf einer Rendering-Farm) überträgt der Renderer die neu erzeugten Trainingsbereiche zu einem dedizierten Server oder einer dedizierten Datenbank (in der Rendering-Farm dieses Studios), der/die diese Daten von mehreren Rendering-Knoten im Laufe der Zeit aggregiert. Ein separater Prozess auf einer separaten Maschine verbessert kontinuierlich das dedizierte Entrauschungsnetzwerk des Studios, und neue Rendering-Aufgaben verwenden immer das neueste trainierte Netzwerk.

[0172] Ein Maschinenlernverfahren ist in **Fig. 19** veranschaulicht. Das Verfahren kann auf den hierin beschriebenen Architekturen implementiert werden, ist aber nicht auf irgendeine bestimmte System- oder Grafikverarbeitungsarchitektur beschränkt.

[0173] Bei 1901 werden als Teil der anfänglichen Trainingsphase Bilddaten mit niedriger Sample-Anzahl und Bilddaten mit hoher Sample-Anzahl für mehrere Bildframes erzeugt. Bei 1902 wird eine Maschinenlern-Entrauschungs-Engine unter Verwendung der Bilddaten mit hoher/niedriger Sample-Anzahl trainiert. Zum Beispiel kann ein Satz von Gewichtungen eines faltenden neuronalen Netzwerks, der mit Pixelmerkmalen assoziiert ist, gemäß dem Training aktualisiert werden. Es kann jedoch eine beliebige Maschinenlernarchitektur verwendet werden.

[0174] Bei 1903 werden zur Laufzeit Bildframes mit niedriger Sample-Anzahl zusammen mit mindestens einem Referenzbereich mit einer hohen Sample-Anzahl erzeugt. Bei 1904 wird der Referenzbereich mit hoher Sample-Anzahl durch die Maschinenlern-Engine und/oder separate Trainingslogik (z. B. Hintergrundtrainingsmodul 1700) verwendet, um das Training der Maschinenlern-Engine kontinuierlich zu verfeinern. Zum Beispiel kann der Referenzbereich mit hoher Sample-Anzahl in Kombination mit einem entsprechenden Abschnitt des Bildes mit niedriger Sample-Anzahl verwendet werden, um der Maschinenlern-Engine 1904 weiterhin beizubringen, wie eine Entrauschung am effektivsten durchzuführen ist. In einer CNN-Implementierung kann dies zum Beispiel Aktualisieren der mit dem CNN assoziierten Gewichtungen beinhalten.

[0175] Mehrere oben beschriebene Variationen können implementiert werden, wie etwa die Art und Weise, auf die die Rückkopplungsschleife zu der Maschinenlern-Engine konfiguriert ist, die Entitäten, die die Trainingsdaten erzeugen, die Art und Weise, auf die die Trainingsdaten zu der Trainings-Engine zurückgekoppelt werden, und wie das verbesserte Netzwerk den Rendering-Engines bereitgestellt wird. Obwohl die oben beschriebenen Beispiele ein kontinuierliches Training unter Verwendung eines einzigen Referenzbereichs durchführen, kann zusätzlich eine beliebige Anzahl von Referenzbereichen verwendet werden. Darüber hinaus können, wie zuvor erwähnt, die Referenzbereiche unterschiedliche Größen aufweisen, auf unterschiedlichen Anzahlen von Bildframes verwendet werden und können unter Verwendung unterschiedlicher Techniken (z. B. zufällig, gemäß einem vorbestimmten Muster usw.) an unterschiedlichen Orten innerhalb der Bildframes positioniert werden.

[0176] Zudem können, obgleich ein faltendes neuronales Netzwerk (CNN) als ein Beispiel für eine Maschinenlern-Engine 1600 beschrieben ist, die zugrundeliegenden Prinzipien der Erfindung unter Verwendung einer beliebigen Form einer Maschinenlern-Engine implementiert werden, die in der Lage ist, ihre Ergebnisse unter Verwendung neuer Trainingsdaten kontinuierlich zu verfeinern. Als Beispiel und nicht einschränkend beinhalten andere Maschinenlernimplementierungen das Gruppenverfahren der Datenbearbeitung (GMDH: Group Method of Data Handling), langen Kurzzeitspeicher, Deep-Reservoir-Computing, Deep-Belief-Netzwerke, Tensor-Deep-Stacking-Netzwerke und Deep-Predictive-Coding-Netzwerke, um einige zu nennen.

EINRICHTUNG UND VERFAHREN ZUM EFFIZIENTEN VERTEILTEN ENTRAUSCHEN

[0177] Wie oben beschrieben, ist das Entrauschen zu einem kritischen Merkmal für Echtzeit-Raytracing mit glatten, rauschlosen Bildern geworden. Rendern kann über ein verteiltes System auf mehreren Vorrichtungen

erfolgen, aber bisher arbeiten die existierenden Entrauschungsframeworks alle auf einer einzigen Instanz auf einer einzigen Maschine. Falls Rendern über mehrere Vorrichtungen durchgeführt wird, stehen ihnen möglicherweise nicht alle gerenderten Pixel zum Berechnen eines entrauschten Abschnitts des Bildes zur Verfügung.

[0178] Es wird ein verteilter Entrauschungsalgorithmus präsentiert, der sowohl mit Entrauschungstechniken auf Basis künstlicher Intelligenz (KI) als auch nicht-KI-basierten Entrauschungstechniken arbeitet. Bereiche des Bildes sein entweder von einer verteilten Render-Operation bereits über Knoten verteilt oder werden von einem einzelnen Framepuffer aufgeteilt und verteilt. Ghost-Bereiche benachbarter Bereiche, die zum Berechnen einer ausreichenden Entrauschung benötigt werden, werden bei Bedarf von benachbarten Knoten gesammelt, und die endgültigen resultierenden Kacheln werden zu einem endgültigen Bild zusammengesetzt.

VERTEILTE VERARBEITUNG

[0179] **Fig. 20** veranschaulicht mehrere Knoten 2021-2023, die Rendern durchführen. Obwohl der Einfachheit halber nur drei Knoten veranschaulicht sind, sind die zugrundeliegenden Prinzipien der Erfindung nicht auf eine bestimmte Anzahl von Knoten beschränkt. Tatsächlich kann ein einziger Knoten verwendet werden, um gewisse Ausführungsformen der Erfindung zu implementieren.

[0180] Die Knoten 2021-2023 rendern jeweils einen Abschnitt eines Bildes, was bei diesem Beispiel zu Bereichen 2011-2013 führt. Obwohl in **Fig. 20** rechteckige Bereiche 2011-2013 gezeigt sind, können Bereiche mit einer beliebigen Form verwendet werden und eine beliebige Vorrichtung kann eine beliebige Anzahl von Bereichen verarbeiten. Die Bereiche, die ein Knoten benötigt, um eine ausreichend glatte Entrauschungsoperation durchzuführen, werden als Ghost-Bereiche 2011-2013 bezeichnet. Mit anderen Worten repräsentieren die Ghost-Bereiche 2001-2003 die Gesamtheit von Daten, die erforderlich sind, um ein Entrauschen mit einem bestimmten Qualitätsgrad durchzuführen. Durch Verringern des Qualitätsgrads wird die Größe des Ghost-Bereichs und damit die benötigte Datenmenge verringert und durch Erhöhen des Qualitätsgrads wird der Ghost-Bereich und die Menge der entsprechenden benötigten Daten vergrößert.

[0181] Falls ein Knoten, wie etwa der Knoten 2021, eine lokale Kopie eines Abschnitts des Ghost-Bereichs 2001 aufweist, der erforderlich ist, um seinen Bereich 2011 mit einem spezifizierten Qualitätsgrad zu entrauschen, wird der Knoten die erforderlichen Daten von einem oder mehreren „angrenzenden“ Knoten abrufen, wie etwa von dem Knoten 2022, der einen Abschnitt des Ghost-Bereichs 2001 besitzt, wie veranschaulicht. Gleichermaßen wird, falls der Knoten 2022 eine lokale Kopie eines Abschnitts des Ghost-Bereichs 2002 aufweist, der erforderlich ist, um seinen Bereich 2012 mit dem spezifizierten Qualitätsgrad zu entrauschen, der Knoten 2022 die erforderlichen Ghost-Bereich-Daten 2032 von dem Knoten 2021 abrufen. Der Abruf kann über einen Bus, ein Interconnect, ein Hochgeschwindigkeits-Speicher-Fabric, ein Netzwerk (z. B. Hochgeschwindigkeits-Ethernet) durchgeführt werden oder kann sogar ein On-Chip-Interconnect in einem Mehrkern-Chip sein, das in der Lage ist, Renderarbeit auf mehrere Kerne zu verteilen (z. B. verwendet zum Rendern großer Bilder entweder mit extremen Auflösungen oder zeitlich variierend). Jeder Knoten 2021-2023 kann eine individuelle Ausführungseinheit oder einen spezifizierten Satz von Ausführungseinheiten innerhalb eines Grafikprozessors umfassen.

[0182] Die spezifische zu sendende Datenmenge ist von den verwendeten Entrauschungstechniken abhängig. Darüber hinaus können die Daten aus dem Ghost-Bereich beliebige Daten beinhalten, die zum Verbessern der Entrauschung jedes jeweiligen Bereichs benötigt werden. Zum Beispiel können die Ghost-Bereichsdaten Bildfarben/Wellenlängen, Intensität/Alpha-Daten und/oder Normalen beinhalten. Die zugrundeliegenden Prinzipien der Erfindung sind jedoch nicht auf einen bestimmten Satz von Ghost-Bereichsdaten beschränkt.

ZUSÄTZLICHE EINZELHEITEN

[0183] Für langsamere Netzwerke oder Interconnects kann eine Komprimierung dieser Daten unter Verwendung einer bestehenden verlustfreien oder verlustbehafteten Mehrzweckkomprimierung genutzt werden. Zu Beispielen gehören unter anderem zlib, gzip und Lempel-Ziv-Markov-Kettenalgorithmus (LZMA). Eine weitere inhaltsspezifische Komprimierung kann verwendet werden, indem angemerkt wird, dass die Delta-in-Strahltreffer-Informationen zwischen Frames recht spärlich sein können und nur die Samples, die zu diesem Delta beitragen, gesendet werden müssen, wenn der Knoten bereits die gesammelten Deltas von vorherigen Frames aufweist. Diese können selektiv zu Knoten gepusht werden, die diese Samples sammeln, i, oder Knoten

/ kann Samples von anderen Knoten anfordern. Für bestimmte Typen von Daten und Programmcode wird verlustfreie Komprimierung verwendet, während für andere Datentypen verlustbehaftete Daten verwendet werden.

[0184] Fig. 21 veranschaulicht zusätzliche Einzelheiten der Interaktionen zwischen den Knoten 2021-2022. Jeder Knoten 2021-2022 beinhaltet eine Raytracing-Renderschaltungsanordnung 2081-2082 zum Rendern der jeweiligen Bildbereiche 2011-2012 und Ghost-Bereiche 2001-2002. Entrauscher 2100-2111 führen Entrauschungsoperationen jeweils an den Bereichen 2011-2012 aus, für deren Rendern und Entrauschen die Knoten 2021-2022 jeweils verantwortlich sind. Die Entrauscher 2021-2022 können zum Beispiel Schaltungsanordnungen, Software oder eine beliebige Kombination davon umfassen, um jeweils die entrauschten Bereiche 2121-2122 zu erzeugen. Wie erwähnt sind die Entrauscher 2021-2022 bei der Erzeugung entrauschter Bereiche möglicherweise auf Daten innerhalb eines Ghost-Bereichs angewiesen, der einem anderen Knoten gehört (z. B. benötigt der Entrauscher 2100 möglicherweise Daten aus dem Ghost-Bereich 2002, der dem Knoten 2022 gehört).

[0185] Dementsprechend können die Entrauscher 2100-2111 die entrauschten Bereiche 2121-2122 unter Verwendung von Daten aus Bereichen 2011-2012 bzw. Ghost-Bereichen 2001-2002 erzeugen, von denen wenigstens ein Teil von einem anderen Knoten empfangen werden kann. Die Bereichsdatenmanager 2101-2102 können Datentransfers von Ghost-Bereichen 2001-2002 verwalten, wie hierin beschrieben. Komprimierer-/Dekomprimiereinheiten 2131-2132 können jeweils eine Komprimierung bzw. Dekomprimierung der Ghost-Bereichsdaten durchführen, die zwischen den Knoten 2021-2022 ausgetauscht werden.

[0186] Beispielsweise kann der Bereichsdatenmanager 2101 des Knotens 2021 auf Anforderung vom Knoten 2022 Daten von dem Ghost-Bereich 2001 an den Komprimierer/Dekomprimierer 2131 senden, der die Daten komprimiert, um komprimierte Daten 2106 zu erzeugen, die er zu dem Knoten 2022 überträgt, wodurch eine Bandbreite über das Interconnect, das Netzwerk, den Bus oder einen anderen Datenkommunikationslink reduziert wird. Der Komprimierer/Dekomprimierer 2132 des Knotens 2022 dekomprimiert dann die komprimierten Daten 2106 und der Entrauscher 2111 verwendet die dekomprimierten Ghost-Daten, um einen entrauschten Bereich 2012 höherer Qualität zu erzeugen, als dies nur mit Daten aus dem Bereich 2012 möglich wäre. Der Bereichsdatenmanager 2102 kann die dekomprimierten Daten aus dem Ghost-Bereich 2001 in einem Cache, einem Speicher, einer Registerdatei oder einer anderen Speicherung speichern, um sie dem Entrauscher 2111 zur Verfügung zu stellen, wenn der entrauschte Bereich 2122 erzeugt wird. Ein ähnlicher Satz von Operationen kann durchgeführt werden, um die Daten aus dem Ghost-Bereich 2002 an den Entrauscher 2100 auf dem Knoten 2021 bereitzustellen, der die Daten in Kombination mit Daten aus dem Bereich 2011 verwendet, um einen entrauschten Bereich 2121 mit höherer Qualität zu erzeugen.

ÜBERNEHMEN VON DATEN ODER RENDERN

[0187] Falls die Verbindung zwischen Vorrichtungen, wie etwa den Knoten 2021-2022, langsam ist (d. h. niedriger als eine Schwellenlatenz und/oder Schwellenbandbreite), kann es schneller sein, Ghost-Bereiche lokal zu rendern, anstatt die Ergebnisse von anderen Vorrichtungen anzufordern. Dies kann zur Laufzeit durch Verfolgung von Netzwerktransaktionsgeschwindigkeiten und linear extrapolierten Renderzeiten für die Ghost-Bereichsgröße bestimmt werden. In solchen Fällen, in denen es schneller ist, den gesamte Ghost-Bereich zu rendern, können im Endeffekt mehrere Vorrichtungen die gleichen Abschnitte des Bildes rendern. Die Auflösung des gerenderten Abschnitts der Ghost-Bereiche kann basierend auf der Varianz des Basisbereichs und dem bestimmten Unschärfegrad angepasst werden.

LASTAUSGLEICH

[0188] Statische und/oder dynamische Lastausgleichsschemen können verwendet werden, um die Verarbeitungslast auf die verschiedenen Knoten 2021-2023 zu verteilen. Zum dynamischen Lastausgleich kann die durch das Entrauschungsfilter bestimmte Varianz sowohl mehr Zeit beim Entrauschen benötigen, aber auch die Menge an Samples ansteuern, die zum Rendern eines bestimmten Bereichs der Szene verwendet werden, wobei geringe Varianz und unscharfe Bereiche des Bildes weniger Samples erfordern. Die spezifischen Knoten zugewiesenen spezifischen Bereiche können dynamisch basierend auf Daten von vorherigen Frames angepasst werden oder dynamisch über Vorrichtungen kommuniziert werden, während sie rendern, sodass alle Vorrichtungen die gleiche Arbeitsmenge aufweisen.

[0189] Fig. 22 veranschaulicht, wie eine Überwachungseinrichtung 2201-2202, die auf jedem jeweiligen Knoten 2021-2022 läuft, Leistungsfähigkeitsmetrikdaten sammelt, darunter unter anderem die Zeit, die ver-

braucht wird, um Daten über die Netzwerkschnittstelle 2211-2212 zu übertragen, die Zeit, die verbraucht wird, wenn ein Bereich (mit und ohne Ghost-Bereichsdaten) entrauscht wird, und die Zeit, die beim Rendern jedes Bereichs/Ghost-Bereichs verbraucht wird. Die Überwachungseinrichtungen 2201-2202 melden diese Leistungsfähigkeitsmetriken an einen Manager- oder Lastausgleicherknoten 2201 zurück, der die Daten analysiert, um die aktuelle Arbeitslast an jedem Knoten 2021-2022 zu identifizieren, und potenziell einen effizienteren Modus des Verarbeitens der verschiedenen entrauschten Bereiche 2121-2122 bestimmt. Der Managerknoten 2201 verteilt dann neue Arbeitslasten für neue Bereiche gemäß der detektierten Last an die Knoten 2021-2022. Zum Beispiel kann der Managerknoten 2201 mehr Arbeit zu jenen Knoten übertragen, die nicht stark belastet sind, und/oder Arbeit von jenen Knoten neu zuweisen, die überlastet sind. Zusätzlich dazu kann der Lastausgleicherknoten 2201 einen Rekonfigurationsbefehl übertragen, um die spezifische Art und Weise anzupassen, auf die Rendern und/oder Entrauschen durch jeden der Knoten durchgeführt wird (von denen manche Beispiele oben beschrieben sind).

BESTIMMUNG VON GHOST-BEREICHEN

[0190] Die Größen und Formen der Ghost-Bereiche 2001-2002 können basierend auf dem durch die Entrauscher 2100-2111 implementierten Entrauschungsalgorithmus bestimmt werden. Ihre jeweiligen Größen können dann basierend auf der detektierten Varianz der zu entrauschenden Samples dynamisch modifiziert werden. Der Lernalgorithmus, der zum KI-Entrauschen selbst verwendet wird, kann zum Bestimmen geeigneter Bereichsgrößen verwendet werden, oder in anderen Fällen, wie etwa einer bilateralen Unschärfe, bestimmt die vorbestimmte Filterbreite die Größe der Ghost-Bereiche 2001-2002. In einer beispielhaften Implementierung, die einen Lernalgorithmus verwendet, kann die Maschinenlern-Engine auf dem Managerknoten 2201 ausgeführt werden und/oder Teile des maschinellen Lernens können auf jedem der einzelnen Knoten 2021-2023 ausgeführt werden (siehe z. B. **Fig. 18A-B** und dazugehöriger Text oben).

ZUSAMMENTRAGEN DES ENDGÜLTIGEN BILDES

[0191] Das endgültige Bild kann erzeugt werden, indem die gerenderten und entrauschten Bereiche von jedem der Knoten 2021-2023 zusammengetragen werden, ohne dass die Ghost-Bereiche oder Normalen benötigt werden. In **Fig. 22** werden die entrauschten Bereiche 2121-2122 zum Beispiel zu dem Bereichsprozessor 2280 des Managerknotens 2201 übertragen, der die Bereiche kombiniert, um das endgültige entrauschte Bild 2290 zu erzeugen, das dann auf einer Anzeige 2290 angezeigt wird. Der Bereichsprozessor 2280 kann die Bereiche unter Verwendung einer Vielfalt von 2D-Compositing-Techniken kombinieren. Obwohl sie als separate Komponenten veranschaulicht sind, können der Bereichsprozessor 2280 und das entrauschte Bild 2290 integral mit der Anzeige 2290 sein. Die verschiedenen Knoten 2021-2022 können eine Direktsendetechnik verwenden, um die entrauschten Bereiche 2121-2122 zu übertragen, und möglicherweise verschiedene verlustbehaftete oder verlustlose Komprimierung der Bereichsdaten verwenden.

[0192] Das KI-Entrauschen ist immer noch eine kostspielige Operation und verlagert sich wie das Gaming in die Cloud. Von daher kann eine Verteilungsverarbeitung des Entrauschens über mehrere Knoten 2021-2022 hinweg erforderlich werden, um Echtzeit-Frameraten für traditionelles Gaming oder virtuelle Realität (VR) zu erreichen, was höhere Frameraten erfordert. Filmstudios rendern zudem häufig in großen Rendering-Farmen, die zur schnelleren Entrauschung genutzt werden können.

[0193] Ein beispielhaftes Verfahren zum Durchführen eines verteilten Renderns und Entrauschens ist in **Fig. 23** veranschaulicht. Das Verfahren kann im Kontext der oben beschriebenen Systemarchitekturen implementiert werden, ist aber nicht auf irgendeine spezielle Systemarchitektur beschränkt.

[0194] Bei 2301 wird Grafikarbeit an mehrere Knoten versendet, die Raytracing-Operationen durchführen, um einen Bereich eines Bildframes zu rendern. Jeder Knoten kann bereits Daten aufweisen, die erforderlich sind, um die Operationen im Speicher durchzuführen. Beispielsweise können sich zwei oder mehr der Knoten einen gemeinsamen Speicher teilen oder die lokalen Speicher der Knoten können bereits Daten von früheren Raytracing-Operationen gespeichert haben. Alternativ oder zusätzlich können gewisse Daten zu jedem Knoten übertragen werden.

[0195] Bei 2302 wird der „Ghost-Bereich“ bestimmt, der für einen spezifizierten Entrauschungsgrad (d. h. bei einem akzeptablen Performanceniveau) benötigt wird. Der Ghost-Bereich umfasst beliebige Daten, die zum Durchführen des spezifizierten Entrauschungsgrads erforderlich sind, einschließlich Daten, die einem oder mehreren anderen Knoten gehören.

[0196] Bei 2303 werden Daten bezüglich der Ghost-Bereiche (oder Abschnitte davon) zwischen Knoten ausgetauscht. Bei 2304 führt jeder Knoten eine Entrauschung an seinem jeweiligen Bereich durch (z. B. unter Verwendung der ausgetauschten Daten) und bei 2305 werden die Ergebnisse kombiniert, um das endgültige entrauschte Bildframe zu erzeugen.

[0197] Ein Managerknoten oder Primärknoten, wie in **Fig. 22** gezeigt, kann die Arbeit an die Knoten versenden und dann die durch die Knoten durchgeführte Arbeit kombinieren, um das endgültige Bildframe zu erzeugen. Eine Peer-basierte Architektur kann verwendet werden, bei der die Knoten Peers sind, die Daten austauschen, um das endgültige Bildframe zu rendern und zu entrauschen.

[0198] Die hierin beschriebenen Knoten (z. B. die Knoten 2021-2023) können Grafikverarbeitungsrechensysteme sein, die über ein Hochgeschwindigkeitsnetzwerk miteinander verbunden sind. Alternativ dazu können die Knoten einzelne Verarbeitungselemente sein, die mit einem Hochgeschwindigkeits-Speicher-Fabric gekoppelt sind. Alle Knoten können sich einen gemeinsamen virtuellen Speicherraum und/oder einen gemeinsamen physischen Speicher teilen. Alternativ dazu können die Knoten eine Kombination aus CPUs und GPUs sein. Zum Beispiel kann der oben beschriebene Managerknoten 2201 eine CPU und/oder auf der CPU ausgeführte Software sein und die Knoten 2021-2022 können GPUs und/oder auf den GPUs ausgeführte Software sein. Es können verschiedene Arten von Knoten verwendet werden, wobei die den zugrundeliegenden Prinzipien der Erfindung weiter erfüllt werden.

BEISPIELHAFTE IMPLEMENTIERUNGEN NEURONALER NETZWERKE

[0199] Es gibt viele Arten von neuronalen Netzwerken; eine einfache Art eines neuronalen Netzwerks ist ein vorwärtsgekoppeltes Netzwerk. Ein vorwärtsgekoppeltes Netzwerk kann als ein azyklischer Graph implementiert sein, in dem die Knoten in Schichten angeordnet sind. Typischerweise weist eine vorwärtsgekoppelte Netzwerktopologie eine Eingabeschicht und eine Ausgabeschicht auf, die durch mindestens eine verborgene Schicht getrennt sind. Die verborgene Schicht wandelt durch die Eingabeschicht empfangene Eingaben in eine Repräsentation um, die zum Erzeugen von Ausgaben in der Ausgabeschicht nützlich ist. Die Netzwerkknoten sind vollständig über Kanten mit den Knoten in angrenzenden Schichten verbunden, aber es gibt keine Kanten zwischen Knoten innerhalb jeder Schicht. Daten, die an den Knoten einer Eingabeschicht eines vorwärtsgekoppelten Netzwerks empfangen werden, werden über eine Aktivierungsfunktion, die die Zustände der Knoten jeder aufeinanderfolgenden Schicht in dem Netzwerk auf der Grundlage von Koeffizienten („Gewichtungen“) berechnet, die jeweils mit jeder der die Schichten verbindenden Kanten assoziiert sind, an die Knoten der Ausgabeschicht propagiert (d. h. „vorwärts gekoppelt“). Abhängig von dem durch den ausgeführten Algorithmus repräsentierten spezifischen Modell kann die Ausgabe von dem Neuronalnetzwerkalgorithmus verschiedene Formen annehmen.

[0200] Bevor ein Maschinenlernalgorithmus verwendet werden kann, um ein bestimmtes Problem zu modellieren, wird der Algorithmus unter Verwendung eines Trainingsdatensatzes trainiert. Das Trainieren eines neuronalen Netzwerks beinhaltet das Auswählen einer Netzwerktopologie, das Verwendung eines Satzes von Trainingsdaten, die ein durch das Netzwerk modelliertes Problem repräsentieren, und das Anpassen der Gewichtungen, bis das Netzwerkmodell für alle Instanzen des Trainingsdatensatzes mit einem minimalen Fehler arbeitet. Zum Beispiel wird während eines Trainingsprozesses mit überwachtem Lernen für ein neuronales Netzwerk die Ausgabe, die durch das Netzwerk als Reaktion auf die eine Instanz in einem Trainingsdatensatz repräsentierende Eingabe erzeugt wird, mit der als „korrekt“ gelabelten Ausgabe für diese Instanz verglichen, ein Fehlersignal, das die Differenz zwischen der Ausgabe und der gelabelten Ausgabe repräsentiert, wird berechnet, und die Gewichtungen, die mit den Verbindungen assoziiert sind, werden angepasst, um diesen Fehler zu minimieren, während das Fehlersignal rückwärts durch die Schichten des Netzwerks propagiert wird. Das Netzwerk wird als „trainiert“ betrachtet, wenn die Fehler für jede der aus den Instanzen des Trainingsdatensatzes erzeugten Ausgaben minimiert sind.

[0201] Die Genauigkeit eines Maschinenlernalgorithmus kann durch die Qualität des zum Trainieren des Algorithmus verwendeten Datensatzes erheblich beeinflusst werden. Der Trainingsprozess kann rechenintensiv sein und kann einen erheblichen Zeitaufwand auf einem herkömmlichen Allzweckprozessor erfordern. Dementsprechend wird eine Parallelverarbeitungshardware verwendet, um viele Arten von Maschinenlernalgorithmen zu trainieren. Dies ist besonders zum Optimieren des Trainings von neuronalen Netzwerken nützlich, da die Berechnungen, die beim Anpassen der Koeffizienten in neuronalen Netzwerken durchgeführt werden, sich auf natürliche Weise für parallele Implementierungen eignen. Insbesondere wurden viele Maschinenlernalgorithmen und Softwareanwendungen dahingehend angepasst, die Parallelverarbeitungshardware in Allzweck-Grafikverarbeitungsvorrichtungen zu verwenden.

[0202] Fig. 24 ist ein generalisiertes Diagramm eines Maschinenlern-Softwarestapels 2400. Eine Maschinenlernanwendung 2402 kann dazu konfiguriert sein, ein neuronales Netzwerk unter Verwendung eines Trainingsdatensatzes zu trainieren oder ein trainiertes tiefes neuronales Netzwerk zu verwenden, um Maschinenintelligenz zu implementieren. Die Maschinenlernanwendung 2402 kann eine Trainings- und Inferenzfunktionalität für ein neuronales Netzwerk und/oder spezialisierte Software beinhalten, die verwendet werden kann, um ein neuronales Netzwerk vor dem Einsatz zu trainieren. Die Maschinenlernanwendung 2402 kann eine beliebige Art von Maschinenintelligenz implementieren, darunter unter anderem Bilderkennung, Kartierung und Lokalisierung, autonome Navigation, Sprachsynthese, medizinische Bildgebung oder Sprachübersetzung.

[0203] Die Hardwarebeschleunigung für die Maschinenlernanwendung 2402 kann über ein Maschinenlern-Framework 2404 aktiviert werden. Das Maschinenlern-Framework 2404 kann auf hierin beschriebener Hardware implementiert werden, wie etwa dem Verarbeitungssystem 100, das die hierin beschriebenen Prozessoren und Komponenten umfasst. Die für **Fig. 24** beschriebenen Elemente mit den gleichen oder ähnlichen Namen wie die Elemente einer beliebigen anderen Figur hierin beschreiben die gleichen Elemente wie in den anderen Figuren, können auf ähnliche Weise arbeiten oder fungieren, können die gleichen Komponenten umfassen und können mit anderen Entitäten wie jene, die an anderer Stelle hierin beschrieben sind, verknüpft sein, sind jedoch nicht darauf beschränkt. Das Maschinenlern-Framework 2404 kann eine Bibliothek von Maschinenlernprimitiven bereitstellen. Maschinenlernprimitive sind grundlegende Operationen, die üblicherweise durch Maschinenlernalgorithmen durchgeführt werden. Ohne das Maschinenlern-Framework 2404 müssten Entwickler von Maschinenlernalgorithmen die mit dem Maschinenlernalgorithmus assoziierte Hauptrechenlogik erstellen und optimieren und dann bei Entwicklung neuer Parallelprozessoren die Rechenlogik erneut optimieren. Stattdessen kann die Maschinenlernanwendung dazu konfiguriert sein, die notwendigen Berechnungen unter Verwendung der Primitive durchzuführen, die von dem Maschinenlern-Framework 2404 bereitgestellt werden. Zu beispielhaften Primitiven gehören Tensorfaltungen, Aktivierungsfunktionen und Pooling, bei denen es sich um Rechenoperationen handelt, die während des Trainierens eines faltenden neuronalen Netzwerks (CNN: Convolutional Neural Network) durchgeführt werden. Das Maschinenlern-Framework 2404 kann auch Primitive bereitstellen, um grundlegende Unterprogramme für lineare Algebra, die durch viele Maschinenlernalgorithmen durchgeführt werden, wie etwa Matrix- und Vektoroperationen, zu implementieren.

[0204] Das Maschinenlern-Framework 2404 kann Eingabedaten, die von der Maschinenlernanwendung 2402 empfangen werden, verarbeiten und die geeignete Eingabe in ein Rechen-Framework 2406 erzeugen. Das Rechen-Framework 2406 kann die dem GPGPU-Treiber 2408 bereitgestellten zugrundeliegenden Anweisungen abstrahieren, um zu ermöglichen, dass das Maschinenlern-Framework 2404 eine Hardwarebeschleunigung über die GPGPU-Hardware 2410 nutzt, ohne dass das Maschinenlern-Framework 2404 genaue Kenntnisse über die Architektur der GPGPU-Hardware 2410 haben muss. Außerdem kann das Rechen-Framework 2406 eine Hardwarebeschleunigung für das Maschinenlern-Framework 2404 über viele verschiedene Arten und Generationen der GPGPU-Hardware 2410 ermöglichen.

GPGPU-Beschleunigung für maschinelles Lernen

[0205] Fig. 25 veranschaulicht ein Multi-GPU-Rechensystem 2500, das eine Variante des Verarbeitungssystems 100 sein kann. Daher wird mit der Offenbarung beliebiger Merkmale in Kombination mit dem Verarbeitungssystem 100 hierin auch eine entsprechende Kombination mit dem Multi-GPU-Rechensystem 2500offenbart, jedoch ohne Beschränkung darauf. Die Elemente von **Fig. 25** mit den gleichen oder ähnlichen Namen wie die Elemente einer beliebigen anderen Figur hierin beschreiben die gleichen Elemente wie in den anderen Figuren, können auf ähnliche Weise arbeiten oder fungieren, können die gleichen Komponenten umfassen und können mit anderen Entitäten wie jene, die an anderer Stelle hierin beschrieben sind, verknüpft sein, sind jedoch nicht darauf beschränkt. Das Multi-GPU-Rechensystem 2500 kann einen Prozessor 2502 beinhalten, der über einen Hostschnittstellen-Switch 2504 mit mehreren GPGPUs 2506A-D gekoppelt ist. Der Hostschnittstellen-Switch 2504 kann zum Beispiel eine PCI-Express-Switch-Vorrichtung sein, die den Prozessor 2502 mit einem PCI-Express-Bus koppelt, über den der Prozessor 2502 mit dem Satz von GPGPUs 2506A-D kommunizieren kann. Jede der mehreren GPGPUs 2506A-D kann eine Instanz der oben beschriebenen GPGPU sein. Die GPGPUs 2506A-D können über einen Satz von Hochgeschwindigkeits-Punkt-zu-Punkt-GPU-zu-GPU-Links 2516 miteinander verbunden sein. Die Hochgeschwindigkeits-GPU-zu-GPU-Links können mit jeder der GPGPUs 2506A-D über einen dedizierten GPU-Link verbunden sein. Die P2P-GPU-Links 2516 ermöglichen eine direkte Kommunikation zwischen jeder der GPGPUs 2506A-D, ohne dass eine Kommunikation über den Hostschnittstellenbus erforderlich ist, mit dem der Prozessor 2502 verbunden ist. Wenn der GPU-zu-GPU-Verkehr auf die P2P-GPU-Links geleitet wird, bleibt der Hostschnittstel-

lenbus für einen Systemspeicherzugriff oder zur Kommunikation mit anderen Instanzen des Multi-GPU-Rechensystems 2500 verfügbar, beispielsweise über eine oder mehrere Netzwerkvorrichtungen. Anstatt die GPGPUs 2506A-D über den Hostschnittstellen-Switch 2504 mit dem Prozessor 2502 zu verbinden, kann der Prozessor 2502 eine direkte Unterstützung für die P2P-GPU-Links 2516 beinhalten und somit direkt mit den GPGPUs 2506A-D verbunden werden.

Implementierungen neuronaler Netzwerke für maschinelles Lernen

[0206] Die hierin beschriebene Rechenarchitektur kann dazu konfiguriert sein, die Arten der parallelen Verarbeitung durchzuführen, die insbesondere zum Trainieren und Einsetzen von neuronalen Netzwerken für maschinelles Lernen geeignet sind. Ein neuronales Netzwerk kann als ein Netzwerk von Funktionen mit einer Graphenbeziehung verallgemeinert werden. Wie im Stand der Technik bekannt, gibt es viele verschiedene Arten von Implementierungen neuronaler Netzwerke, die beim maschinellen Lernen verwendet werden. Ein beispielhafter Typ eines neuronalen Netzwerks ist das vorwärtsgekoppelte Netzwerk, wie zuvor beschrieben.

[0207] Ein zweiter beispielhafter Typ eines neuronalen Netzwerks ist das faltende neuronale Netzwerk (CNN: Convolutional Neural Network). Ein CNN ist ein spezialisiertes neuronales vorwärtsgekoppeltes Netzwerk zum Verarbeiten von Daten mit einer bekannten gitterartigen Topologie, wie etwa Bilddaten. Dementsprechend werden CNNs üblicherweise für Computer-Vision- und Bilderkennungsanwendungen verwendet, sie können aber auch für andere Arten der Mustererkennung, wie etwa die Verarbeitung gesprochener und geschriebener Sprache, verwendet werden. Die Knoten in der CNN-Eingabeschicht sind in einem Satz von „Filtern“ organisiert (Merkmalsdetektoren, die von den rezeptiven Feldern in der Netzhaut inspiriert sind), und die Ausgabe jedes Filtersatzes wird an Knoten in aufeinanderfolgenden Schichten des Netzwerks propagiert. Die Berechnungen für ein CNN beinhalten das Anwenden der mathematischen Faltungsoperation auf jedes Filter, um die Ausgabe dieses Filters zu erzeugen. Die Faltung ist eine spezielle Art von mathematischer Operation, bei der zwei Funktionen eine dritte Funktion erzeugen, die eine modifizierte Version einer der beiden ursprünglichen Funktionen ist. In der Terminologie eines faltenden Netzwerks kann die erste Funktion der Faltung als Eingabe bezeichnet werden, während die zweite Funktion als Faltungskernel bezeichnet werden kann. Die Ausgabe kann als Feature-Map bezeichnet werden. Die Eingabe in eine Faltungsschicht kann beispielsweise ein mehrdimensionales Array von Daten sein, das die verschiedenen Farbkomponenten eines Eingabebilds definiert. Der Faltungskernel kann ein mehrdimensionales Array von Parametern sein, wobei die Parameter durch den Trainingsprozess für das neuronale Netzwerk angepasst werden.

[0208] Rekurrente neuronale Netzwerke (RNNs) sind eine Familie von neuronalen vorwärtsgekoppelten Netzwerken, die Rückkopplungsverbindungen zwischen Schichten enthalten. RNNs ermöglichen eine Modellierung sequenzieller Daten durch den Austausch von Parameterdaten über verschiedene Teile des neuronalen Netzwerks hinweg. Die Architektur für ein RNN beinhaltet Zyklen. Die Zyklen stellen den Einfluss eines gegenwärtigen Wertes einer Variablen auf ihren eigenen Wert zu einem zukünftigen Zeitpunkt dar, da zumindest ein Teil der Ausgangsdaten von dem RNN als eine Rückkopplung zum Verarbeiten einer nachfolgenden Eingabe in einer Sequenz verwendet wird. Diese Eigenschaft macht RNNs besonders nützlich für die Sprachverarbeitung, da Sprachdaten sehr variabel zusammengesetzt sein können.

[0209] Die nachstehend beschriebenen Figuren zeigen beispielhafte vorwärtsgekoppelte, CNN- und RNN-Netzwerke und beschreiben einen allgemeinen Prozess zum jeweiligen Trainieren und Einsetzen jedes dieser Typen von Netzwerken. Es versteht sich, dass diese Beschreibungen beispielhaft und nicht einschränkend sind und die veranschaulichten Konzepte allgemein auf tiefe neuronale Netzwerke und Maschinenlerntechniken im Allgemeinen angewendet werden können.

[0210] Die oben beschriebenen beispielhaften neuronalen Netzwerke können zum Durchführen von Deep Learning (Tiefes Lernen) verwendet werden. Deep Learning ist maschinelles Lernen unter Verwendung von tiefen neuronalen Netzwerken. Die tiefen neuronalen Netzwerke, die beim Deep Learning verwendet werden, sind künstliche neuronale Netzwerke, die aus mehreren verborgenen Schichten bestehen, im Gegensatz zu flachen neuronalen Netzwerken, die nur eine einzige verborgene Schicht beinhalten. Tiefere neuronale Netzwerke sind im Allgemeinen rechenintensiver zu trainieren. Die zusätzlichen verborgenen Schichten des Netzwerks ermöglichen jedoch eine mehrstufige Mustererkennung, die verglichen mit flachen Maschinenlerntechniken zu verringerten Ausgabefehlern führt.

[0211] Tiefe neuronale Netzwerke, die beim Deep Learning verwendet werden, beinhalten in der Regel ein Frontend-Netzwerk zur Durchführung einer Merkmalserkennung, das mit einem Backend-Netzwerk gekoppelt ist, das ein mathematisches Modell repräsentiert, das Operationen (z. B. Objektklassifizierung, Spracherkennung usw.) basierend auf der dem Modell bereitgestellten Merkmalsrepräsentation durchführen kann. Deep Learning ermöglicht ein Durchführen von maschinellem Lernen, ohne dass für das Modell eine manuelle Merkmalskonstruktion durchgeführt werden muss. Stattdessen können tiefe neuronale Netzwerke Merkmale basierend auf einer statistischen Struktur oder Korrelation innerhalb der Eingabedaten lernen. Die erlernten Merkmale können einem mathematischen Modell bereitgestellt werden, das detektierte Merkmale auf eine Ausgabe abbilden kann. Das durch das Netzwerk verwendete mathematische Modell ist im Allgemeinen für die spezifische durchzuführende Aufgabe spezialisiert, und andere Modelle werden verwendet, um andere Aufgaben durchzuführen.

[0212] Sobald das neuronale Netzwerk strukturiert ist, kann ein Lernmodell auf das Netzwerk angewendet werden, um das Netzwerk dahingehend zu trainieren, spezifische Aufgaben durchzuführen. Das Lernmodell beschreibt, wie die Gewichtungen innerhalb des Modells anzupassen sind, um den Ausgabefehler des Netzwerks zu reduzieren. Fehlerrückpropagation ist ein übliches Verfahren zum Trainieren neuronaler Netzwerke. Ein Eingabevektor wird dem Netzwerk zur Verarbeitung bereitgestellt. Die Ausgabe des Netzwerks wird unter Verwendung einer Verlustfunktion mit der gewünschten Ausgabe verglichen, und für jedes der Neuronen in der Ausgabeschicht wird ein Fehlerwert berechnet. Die Fehlerwerte werden dann rückwärts propagiert, bis jedes Neuron einen assoziierten Fehlerwert aufweist, der grob seinen Beitrag zur ursprünglichen Ausgabe repräsentiert. Das Netzwerk kann dann aus diesen Fehlern lernen, indem es einen Algorithmus, wie etwa den stochastischen Gradientenabstiegsalgorithmus, verwendet, um die Gewichtungen des neuronalen Netzwerks zu aktualisieren.

[0213] **Fig. 26-27** veranschaulichen ein beispielhaftes faltendes neuronales Netzwerk. **Fig. 26** veranschaulicht verschiedene Schichten innerhalb eines CNN. Wie in **Fig. 26** gezeigt, kann ein beispielhaftes CNN, das zum Modellieren einer Bildverarbeitung verwendet wird, eine Eingabe 2602 empfangen, die die Rot-, Grün- und Blau(RGB)-Komponenten eines Eingabebilds beschreibt. Die Eingabe 2602 kann durch mehrere Faltungsschichten (z. B. Faltungsschicht 2604, Faltungsschicht 2606) verarbeitet werden. Die Ausgabe von den mehreren Faltungsschichten kann gegebenenfalls durch einen Satz vollständig verbundener Schichten 2608 verarbeitet werden. Neuronen in einer vollständig verbundenen Schicht weisen vollständige Verbindungen mit allen Aktivierungen in der vorherigen Schicht auf, wie zuvor für ein vorwärtsgekoppeltes Netzwerk beschrieben. Die Ausgabe von den vollständig verbundenen Schichten 2608 kann dazu verwendet werden, ein Ausgabeergebnis von dem Netzwerk zu erzeugen. Die Aktivierungen innerhalb der vollständig verbundenen Schichten 2608 können unter Verwendung einer Matrixmultiplikation anstelle einer Faltung berechnet werden. Nicht alle CNN-Implementierungen verwenden vollständig verbundene Schichten. Zum Beispiel kann in einigen Implementierungen die Faltungsschicht 2606 eine Ausgabe für das CNN erzeugen.

[0214] Die Faltungsschichten sind spärlich verbunden, was sich von der herkömmlichen Neuronalnetzwerkkonfiguration unterscheidet, die in den vollständig verbundenen Schichten 2608 zu finden ist. Herkömmliche Neuronalnetzwerkschichten sind vollständig verbunden, sodass jede Ausgabeeinheit mit jeder Eingabeeinheit interagiert. Die Faltungsschichten sind jedoch spärlich verbunden, da die Ausgabe der Faltung eines Feldes (anstatt des jeweiligen Zustandswertes jedes der Knoten in dem Feld) in die Knoten der nachfolgenden Schicht eingegeben wird, wie veranschaulicht. Die mit den Faltungsschichten assoziierten Kernel führen Faltungsoperationen durch, deren Ausgabe an die nächste Schicht gesendet wird. Die Dimensionalitätsreduzierung, die in den Faltungsschichten durchgeführt + wird, ist ein Aspekt, der ermöglicht, dass das CNN zur Verarbeitung großer Bilder skaliert.

[0215] **Fig. 27** veranschaulicht beispielhafte Rechenstufen innerhalb einer Faltungsschicht eines CNN. Eine Eingabe in eine Faltungsschicht 2712 eines CNN kann in drei Stufen einer Faltungsschicht 2714 verarbeitet werden. Zu den drei Stufen können eine Faltungsstufe 2716, eine Detektorstufe 2718 und eine Pooling-Stufe 2720 gehören. Die Faltungsschicht 2714 kann dann Daten an eine nachfolgende Faltungsschicht ausgeben. Die letzte Faltungsschicht des Netzwerks kann Ausgabe-Feature-Map-Daten erzeugen oder eine Eingabe in eine vollständig verbundene Schicht bereitstellen, um beispielsweise einen Klassifizierungswert für die Eingabe in das CNN zu erzeugen.

[0216] In der Faltungsstufe 2716 werden mehrere Faltungen parallel durchgeführt, um einen Satz linearer Aktivierungen zu erzeugen. Die Faltungsstufe 2716 kann eine affine Transformation enthalten, bei der es sich um eine beliebige Transformation handelt, die als lineare Transformation plus eine Translation angegeben werden kann. Affine Transformationen beinhalten Rotationen, Translationen, Skalierungen und Kombina-

tionen dieser Transformationen. Die Faltungsstufe berechnet die Ausgabe von Funktionen (z. B. Neuronen), die mit spezifischen Regionen in der Eingabe verbunden sind, die als die mit dem Neuron assoziierte lokale Region bestimmt werden können. Die Neuronen berechnen ein Skalarprodukt zwischen den Gewichtungen der Neuronen und der Region in der lokalen Eingabe, mit der die Neuronen verbunden sind. Die Ausgabe von der Faltungsstufe 2716 definiert einen Satz linearer Aktivierungen, die durch aufeinanderfolgende Stufen der Faltungsschicht 2714 verarbeitet werden.

[0217] Die linearen Aktivierungen können durch eine Detektorstufe 2718 verarbeitet werden. In der Detektorstufe 2718 wird jede lineare Aktivierung durch eine nichtlineare Aktivierungsfunktion verarbeitet. Die nichtlineare Aktivierungsfunktion erhöht die nichtlinearen Eigenschaften des Gesamtnetzwerks, ohne die rezeptiven Felder der Faltungsschicht zu beeinflussen. Verschiedene Arten von nichtlinearen Aktivierungsfunktionen können verwendet werden. Eine bestimmte Art ist die rektifizierte lineare Einheit (ReLU: Rectified Linear Unit), die eine Aktivierungsfunktion verwendet, die als f(x)=max (0,x) definiert ist, sodass die Aktivierung bei null begrenzt wird.

[0218] Die Pooling-Stufe 2720 verwendet eine Pooling-Funktion, die die Ausgabe der Faltungsschicht 2706 durch eine Zusammenfassungsstatistik der nahegelegenen Ausgaben ersetzt. Die Pooling-Funktion kann dazu verwendet werden, eine Translationsinvarianz in das neuronale Netzwerk einzuführen, sodass kleine Translationen der Eingabe die gepoolten Ausgaben nicht verändern. Die Invarianz gegenüber lokaler Translation kann in Szenarien nützlich sein, in denen das Vorhandensein eines Merkmals in den Eingabedaten wichtiger ist als die genaue Position des Merkmals. Während der Pooling-Stufe 2720 können verschiedene Arten von Pooling-Funktionen verwendet werden, darunter Max-Pooling, Average-Pooling (Durchschnitts-Pooling) und L2-Norm-Pooling. Darüber hinaus beinhalten einige CNN-Implementierungen keine Pooling-Stufe. Stattdessen ersetzen solche Implementierungen eine zusätzliche Faltungsstufe, die ein erhöhtes Stride relativ zu vorherigen Faltungsstufen hat.

[0219] Die Ausgabe aus der Faltungsschicht 2714 kann dann durch die nächste Schicht 2722 verarbeitet werden. Die nächste Schicht 2722 kann eine zusätzliche Faltungsschicht oder eine der vollständig verbundenen Schichten 2708 sein. Zum Beispiel kann die erste Faltungsschicht 2704 von **Fig. 27** zu der zweiten Faltungsschicht 2706 ausgeben, während die zweite Faltungsschicht zu einer ersten Schicht der vollständig verbundenen Schichten 2808 ausgeben kann.

[0220] **Fig. 28** veranschaulicht ein beispielhaftes rekurrentes neuronales Netzwerk 2800. In einem rekurrenten neuronalen Netzwerk (RNN) beeinflusst der vorhergehende Zustand des Netzwerks die Ausgabe des aktuellen Zustands des Netzwerks. RNNs können auf vielfältige Weise unter Verwendung einer Vielzahl von Funktionen konstruiert werden. Bei der Verwendung von RNNs geht es im Allgemeinen um die Verwendung mathematischer Modelle zur Vorhersage der Zukunft basierend auf einer vorherigen Sequenz von Eingaben. Ein RNN kann beispielsweise zum Durchführen einer statistischen Sprachmodellierung verwendet werden, um ein kommendes Wort anhand einer vorherigen Wortfolge vorherzusagen. Das veranschaulichte RNN 2800 kann so beschrieben werden, dass es eine Eingabeschicht 2802, die einen Eingabevektor empfängt, verborgene Schichten 2804 zum Implementieren einer rekurrenten Funktion, einen Rückkopplungsmechanismus 2805 zum Ermöglichen eines „Memory" von vorherigen Zuständen und eine Ausgabeschicht 2806 zum Ausgeben eines Ergebnisses aufweist. Das RNN 2800 arbeitet auf Grundlage von Zeitschritten. Der Zustand des RNN zu einem gegebenen Zeitschritt wird basierend auf dem vorherigen Zeitschritt über den Rückkopplungsmechanismus 2805 beeinflusst. Für einen gegebenen Zeitschritt wird der Zustand der verborgenen Schichten 2804 durch den vorherigen Zustand und die Eingabe in dem aktuellen Zeitschritt definiert. Eine initiale Eingabe (x1) in einem ersten Zeitschritt kann durch die verborgene Schicht 2804 verarbeitet werden. Eine zweite Eingabe (x2) kann durch die verborgene Schicht 2804 unter Verwendung von Zustandsinformationen, die während der Verarbeitung der initialen Eingabe (x1) bestimmt werden, verarbeitet werden. Ein gegebener Zustand kann als s_t=f(Ux_t+ Ws_(t-1)) berechnet werden, wobei U und W Parametermatrizen sind. Die Funktion f ist allgemein eine Nichtlinearität, wie die Tangens-Hyperbelfunktion (Tanh) oder eine Variante der Rectifier-Funktion f(x)=max(0,x). Die spezifische mathematische Funktion, die in den verborgenen Schichten 2804 verwendet wird, kann jedoch abhängig von den spezifischen Implementierungsdetails des RNN 2800 variieren.

[0221] Zusätzlich zu den beschriebenen grundlegenden CNN- und RNN-Netzwerken können Variationen dieser Netzwerke ermöglicht werden. Eine beispielhafte RNN-Variante ist das LSTM-RNN (LSTM: Long Short Term Memory - langer Kurzzeitspeicher). LSTM-RNNs sind dazu in der Lage, langfristige Abhängigkeiten zu lernen, die für die Verarbeitung längerer Sprachsequenzen notwendig sein können. Eine Variante des CNN ist ein faltendes Deep-Belief-Netzwerk, das eine ähnliche Struktur wie ein CNN aufweist und ähnlich wie

ein Deep-Belief-Netzwerk trainiert wird. Ein Deep-Belief-Netzwerk (DBN) ist ein generatives neuronales Netzwerk, das aus mehreren Schichten stochastischer (zufälliger) Variablen besteht. DBNs können Schicht für Schicht mittels unüberwachtem Lernen mit Greedy-Ansatz trainiert werden. Die gelernten Gewichtungen des DBN können dann verwendet werden, um neuronale Vortrainings-Netzwerke bereitzustellen, indem ein optimaler initialer Satz von Gewichtungen für das neuronale Netzwerk bestimmt wird.

[0222] **Fig. 29** veranschaulicht Training und Einsatz eines tiefen neuronalen Netzwerks. Sobald ein gegebenes Netzwerk für eine Aufgabe strukturiert wurde, wird das neuronale Netzwerk unter Verwendung eines Trainingsdatensatzes 2902 trainiert. Verschiedene Trainings-Frameworks 2904 wurden entwickelt, um eine Hardwarebeschleunigung des Trainingsprozesses zu ermöglichen. Zum Beispiel kann das oben beschriebene Maschinenlern-Framework als ein Trainings-Framework konfiguriert sein. Das Trainings-Framework 2904 kann sich in ein untrainiertes neuronales Netzwerk 2906 einklinken und ermöglichen, dass das untrainierte neuronale Netzwerk unter Verwendung der hierin beschriebenen Parallelverarbeitungsressourcen trainiert wird, um ein trainiertes neuronales Netzwerk 2908 zu erzeugen.

[0223] Um den Trainingsprozess zu beginnen, können die initialen Gewichtungen zufällig oder durch Vortraining unter Verwendung eines Deep-Belief-Netzwerks gewählt werden. Der Trainingszyklus kann dann entweder auf überwachte oder auf unüberwachte Weise durchgeführt werden.

[0224] Überwachtes Lernen ist eine Lernmethode, bei der das Training als vermittelte Operation durchgeführt wird, z. B. wenn der Trainingsdatensatz 2902 Eingaben enthält, die mit der gewünschten Ausgabe für die Eingabe gepaart sind, oder wenn der Trainingsdatensatz Eingaben mit bekannter Ausgabe enthält und die Ausgabe des neuronalen Netzwerks manuell bewertet wird. Das Netzwerk verarbeitet die Eingaben und vergleicht die resultierenden Ausgaben mit einem Satz von erwarteten oder gewünschten Ausgaben. Fehler werden dann zurück durch das System propagiert. Das Trainings-Framework 2904 kann die Gewichtungen anpassen, die das untrainierte neuronale Netzwerk 2906 steuern. Das Trainings-Framework 2904 kann Werkzeuge bereitstellen, um zu überwachen, wie gut das untrainierte neuronale Netzwerk 2906 zu einem Modell hin konvergiert, das zum Erzeugen korrekter Antworten auf der Grundlage bekannter Eingabedaten geeignet ist. Der Trainingsprozess findet wiederholt statt, während die Gewichtungen des Netzwerks angepasst werden, um die durch das neuronale Netzwerk erzeugte Ausgabe zu verfeinern. Der Trainingsprozess kann fortgesetzt werden, bis das neuronale Netzwerk eine mit einem trainierten neuronalen Netzwerk 2908 assoziierte statistisch gewünschte Genauigkeit erreicht. Das trainierte neuronale Netzwerk 2908 kann dann eingesetzt werden, um eine beliebige Anzahl von Maschinenlernoperationen zu implementieren.

[0225] Unüberwachtes Lernen ist eine Lernmethode, bei der das Netzwerk versucht, sich selbst unter Verwendung nicht gelabelter Daten zu trainieren. Somit wird der Trainingsdatensatz 2902 für unüberwachtes Lernen Eingabedaten ohne zugehörige Ausgabedaten enthalten. Das untrainierte neuronale Netzwerk 2906 kann Gruppierungen innerhalb der nicht gelabelten Eingabe lernen und kann bestimmen, wie einzelne Eingaben mit dem gesamten Datensatz in Zusammenhang stehen. Unüberwachtes Training kann verwendet werden, um eine selbstorganisierende Map zu generieren, die eine Art trainiertes neuronales Netzwerk 2907 ist, das in der Lage ist, Operationen durchzuführen, die nützlich sind, um die Dimensionalität von Daten zu reduzieren. Unüberwachtes Training kann auch verwendet werden, um eine Anomaliedetektion durchzuführen, die die Identifizierung von Datenpunkten in einem Eingabedatensatz ermöglicht, die von den normalen Mustern der Daten abweichen.

[0226] Es können auch Variationen von überwachtem und unüberwachtem Training eingesetzt werden. Semi-überwachtes Lernen ist eine Technik, bei der der Trainingsdatensatz 2902 eine Mischung aus gelabelten und nicht gelabelten Daten mit gleicher Verteilung beinhaltet. Inkrementelles Lernen ist eine Variante des überwachten Lernens, bei der die Eingabedaten kontinuierlich verwendet werden, um das Modell weiter zu trainieren. Inkrementelles Lernen ermöglicht, dass das trainierte neuronale Netzwerk 2908 sich an die neuen Daten 2912 anpasst, ohne das Wissen zu vergessen, das dem Netzwerk bei einem initialen Training vermittelt wurde.

[0227] Unabhängig davon, ob er überwacht oder unüberwacht ist, kann der Trainingsprozess für besonders tiefe neuronale Netze für einen einzelnen Rechenknoten zu rechenintensiv sein. Anstatt einen einzelnen Rechenknoten zu verwenden, kann ein verteiltes Netzwerk von Rechenknoten verwendet werden, um den Trainingsprozess zu beschleunigen.

[0228] **Fig. 30A** ist ein Blockdiagramm, das verteiltes Lernen veranschaulicht. Verteiltes Lernen ist ein Trainingsmodell, das mehrere verteilte Rechenknoten verwendet, wie etwa die oben beschriebenen Knoten, um

überwachtes oder unüberwachtes Training eines neuronalen Netzwerks durchzuführen. Die verteilten Rechenknoten können jeweils einen oder mehrere Hostprozessoren und einen oder mehrere der Allzweck-Verarbeitungsknoten beinhalten, wie etwa eine hochparallele Allzweck-Grafikverarbeitungseinheit. Wie veranschaulicht, kann verteiltes Lernen mit Modellparallelität 3002, Datenparallelität 3004 oder einer Kombination aus Modell- und Datenparallelität durchgeführt werden.

[0229] Bei der Modellparallelität 3002 können verschiedene Rechenknoten in einem verteilten System Trainingsberechnungen für verschiedene Teile eines einzelnen Netzwerks durchführen. Zum Beispiel kann jede Schicht eines neuronalen Netzwerks durch einen anderen Verarbeitungsknoten des verteilten Systems trainiert werden. Zu den Vorteilen der Modellparallelität gehört die Möglichkeit, auf besonders große Modelle zu skalieren. Die Aufteilung der Berechnungen, die mit verschiedenen Schichten des neuronalen Netzwerks assoziiert sind, ermöglicht das Trainieren von sehr großen neuronalen Netzwerken, bei denen die Gewichtungen aller Schichten nicht in den Speicher eines einzelnen Rechenknotens passen würden. In einigen Fällen kann die Modellparallelität besonders nützlich sein, um ein unüberwachtes Training großer neuronaler Netzwerke durchzuführen.

[0230] Bei der Datenparallelität 3004 weisen die verschiedenen Knoten des verteilten Netzwerks eine vollständige Instanz des Modells auf, und jeder Knoten erhält einen anderen Teil der Daten. Die Ergebnisse aus den verschiedenen Knoten werden dann kombiniert. Obgleich verschiedene Ansätze zur Datenparallelität möglich sind, erfordern alle datenparallele Trainingsansätze eine Technik zur Kombination von Ergebnissen und zur Synchronisierung der Modellparameter zwischen den einzelnen Knoten. Zu beispielhaften Ansätzen zum Kombinieren von Daten gehören Parametermittelwertbildung und aktualisierungsbasierte Datenparallelität. Die Parametermittelwertbildung trainiert jeden Knoten an einem Teilsatz der Trainingsdaten und setzt die globalen Parameter (z. B. Gewichtungen, Biases) auf den Mittelwert der Parameter von jedem Knoten. Die Parametermittelwertbildung verwendet einen zentralen Parameterserver, der die Parameterdaten verwaltet. Die aktualisierungsbasierte Datenparallelität ist ähnlich wie die Parametermittelwertbildung, außer dass anstelle der Übertragung von Parametern von den Knoten zu dem Parameterserver die Aktualisierungen des Modells übertragen werden. Zudem kann die aktualisierungsbasierte Datenparallelität dezentral durchgeführt werden, wobei die Aktualisierungen komprimiert und zwischen Knoten übertragen werden.

[0231] Die kombinierte Modell- und Datenparallelität 3006 kann beispielsweise in einem verteilten System implementiert werden, in dem jeder Rechenknoten mehrere GPUs beinhaltet. Jeder Knoten kann eine vollständige Instanz des Modells aufweisen, wobei separate GPUs innerhalb jedes Knotens dazu verwendet werden, verschiedene Teile des Modells zu trainieren.

[0232] Verteiltes Training hat einen erhöhten Overhead im Vergleich zum Training auf einer einzelnen Maschine. Die hierin beschriebenen Parallelprozessoren und GPGPUs können jedoch jeweils verschiedene Techniken implementieren, um den Overhead des verteilten Trainings zu reduzieren, darunter Techniken zum Ermöglichen einer GPU-zu-GPU-Datenübertragung mit hoher Bandbreite und einer beschleunigten Ferndatensynch ron isation.

Beispielhafte Maschinenlernanwendungen

[0233] Maschinelles Lernen kann zur Lösung einer Vielzahl von technologischen Problemen eingesetzt werden, darunter unter anderem Computer Vision, autonomes Fahren und Navigation, Erkennung gesprochener Sprache und Sprachverarbeitung. Computer Vision ist traditionell eines der aktivsten Forschungsgebiete für Maschinenlernanwendungen. Anwendungen von Computer Vision reichen von der Reproduktion menschlicher visueller Fähigkeiten, wie etwa dem Erkennen von Gesichtern, bis hin zur Schaffung neuer Kategorien visueller Fähigkeiten. Zum Beispiel können Computer-Vision-Anwendungen dazu konfiguriert sein, Schallwellen aus den Vibrationen, die in den in einem Video sichtbaren Objekten induziert werden, erkennen. Parallelprozessor-beschleunigtes maschinelles Lernen ermöglicht es, Computer-Vision-Anwendungen unter Verwendung wesentlich größerer Trainingsdatensätze zu trainieren, als dies bisher möglich war, und ermöglicht es, Inferenzsysteme unter Verwendung von Niederleistungs-Parallelprozessoren einzusetzen.

[0234] Parallelprozessor-beschleunigtes maschinelles Lernen hat Anwendungen für autonomes Fahren, einschließlich Fahrspur- und Verkehrszeichenerkennung, Hindernisvermeidung, Navigation und Fahrkontrolle. Beschleunigte Maschinenlerntechniken können zum Trainieren von Fahrmodellen auf der Grundlage von Datensätzen verwendet werden, die die entsprechenden Reaktionen auf bestimmte Trainingseingaben definieren. Die vorliegend beschriebenen Parallelprozessoren können ein schnelles Training der zunehmend komplexen neuronalen Netzwerke ermöglichen, die für Lösungen zum autonomen Fahren verwendet wer-

den, und ermöglichen den Einsatz von Niederleistungs-Inferenzprozessoren in einer mobilen Plattform, die zur Integration in autonome Fahrzeuge geeignet ist.

[0235] Parallelprozessor-beschleunigte tiefe neuronale Netzwerke haben Maschinenlernansätze für die automatische Spracherkennung (Automatic Speech Recognition, ASR) ermöglicht. ASR beinhaltet die Erstellung einer Funktion, die die wahrscheinlichste sprachliche Sequenz angesichts einer akustischen Eingabesequenz berechnet. Beschleunigtes maschinelles Lernen unter Verwendung tiefer neuronaler Netzwerke hat es ermöglicht, die bisher für ASR verwendeten Hidden-Markov-Modelle (HMMs) und Gaußschen Mischmodelle (GMMs) zu ersetzen.

[0236] Parallelprozessor-beschleunigtes maschinelles Lernen kann auch zur Beschleunigung der Verarbeitung natürlicher Sprache verwendet werden. Automatische Lernprozeduren können statistische Inferenzalgorithmen nutzen, um Modelle zu erzeugen, die robust gegenüber fehlerhaften oder ungewohnten Eingaben sind. Zu beispielhaften Anwendungen für natürliche Sprachprozessoren gehört die automatische maschinelle Übersetzung zwischen menschlichen Sprachen.

[0237] Die für maschinelles Lernen verwendeten Parallelverarbeitungsplattformen können in Trainingsplattformen und Einsatzplattformen unterteilt werden. Trainingsplattformen sind im Allgemeinen hochparallel und beinhalten Optimierungen zur Beschleunigung von Multi-GPU-Einzelknoten-Training und Mehrfachknoten-Multi-GPU-Training. Beispielhafte Parallelprozessoren, die zum Trainieren geeignet sind, beinhalten die hochparallele Allzweck-Grafikverarbeitungseinheit und/oder die Multi-GPU-Rechensysteme, die hierin beschrieben werden. Eingesetzte Maschinenlernpattformen beinhalten dagegen im Allgemeinen Niederleistungs-Parallelprozessoren, die zur Verwendung in Produkten wie Kameras, autonomen Robotern und autonomen Fahrzeugen geeignet sind.

[0238] **Fig. 30B** veranschaulicht ein beispielhaftes Inferenz-Systemon-Chip (SOC) 3100, das zum Durchführen von Inferenzieren unter Verwendung eines trainierten Modells geeignet ist. Die Elemente von **Fig. 30B** mit den gleichen oder ähnlichen Namen wie die Elemente einer beliebigen anderen Figur hierin beschreiben die gleichen Elemente wie in den anderen Figuren, können auf ähnliche Weise arbeiten oder fungieren, können die gleichen Komponenten umfassen und können mit anderen Entitäten wie jene, die an anderer Stelle hierin beschrieben sind, verknüpft sein, sind jedoch nicht darauf beschränkt. Das SOC 3100 kann Verarbeitungskomponenten integrieren, darunter einen Medienprozessor 3102, einen Vision-Prozessor 3104, eine GPGPU 3106 und einen Mehrkernprozessor 3108. Das SOC 3100 kann zusätzlich einen On-Chip-Speicher 3105 beinhalten, der einen gemeinsam genutzten On-Chip-Datenpool ermöglichen kann, auf den jede der Verarbeitungskomponenten zugreifen kann. Die Verarbeitungskomponenten können für einen Niederleistungsbetrieb optimiert werden, um den Einsatz in einer Vielzahl von Maschinenlernplattformen zu ermöglichen, einschließlich autonomer Fahrzeuge und autonomer Roboter. Eine Implementierung des SOC 3100 kann zum Beispiel als Teil des Hauptsteuersystems für ein autonomes Fahrzeug verwendet werden. Wenn das SOC 3100 zur Verwendung in autonomen Fahrzeugen konfiguriert ist, ist das SOC dahingehend ausgelegt und konfiguriert, die relevanten funktionalen Sicherheitsstandards des Einsatzlandes zu erfüllen.

[0239] Während des Betriebs können der Medienprozessor 3102 und der Vision-Prozessor 3104 zusammenarbeiten, um Computer-Vision-Operationen zu beschleunigen. Der Medienprozessor 3102 kann eine latenzarme Decodierung mehrerer hochauflösender Videoströme (z. B. 4K, 8K) ermöglichen. Die decodierten Videoströme können in einen Puffer in dem On-Chip-Speicher 3105 geschrieben werden. Der Vision-Prozessor 3104 kann dann das decodierte Video parsen und vorläufige Verarbeitungsoperationen an den Frames des decodierten Videos als Vorbereitung der Verarbeitung der Frames unter Verwendung eines trainierten Bilderkennungsmodells durchführen. Beispielsweise kann der Vision-Prozessor 3104 die Faltungsoperationen für ein CNN beschleunigen, das zur Durchführung von Bilderkennung an den hochauflösenden Videodaten verwendet wird, während die Backend-Modellberechnungen durch die GPGPU 3106 durchgeführt werden.

[0240] Der Mehrkernprozessor 3108 kann Steuerlogik beinhalten, die bei der Sequenzierung und Synchronisierung von Datenübertragungen und gemeinsamen Speicheroperationen hilft, die durch den Medienprozessor 3102 und den Vision-Prozessor 3104 durchgeführt werden. Der Mehrkernprozessor 3108 kann zudem als Anwendungsprozessor fungieren, um Softwareanwendungen auszuführen, die die Inferenzrechenfähigkeit der GPGPU 3106 nutzen können. Zum Beispiel kann zumindest ein Teil der Navigations- und Fahrlogik in Software implementiert sein, die auf dem Mehrkernprozessor 3108 ausgeführt wird. Derartige Software kann direkt Rechenlasten an die GPGPU 3106 ausgeben oder die Rechenlasten können an den Mehrkernprozes-

sor 3108 ausgegeben werden, der zumindest einen Teil dieser Operationen an die GPGPU 3106 auslagern kann.

[0241] Die GPGPU 3106 kann Verarbeitungscluster, wie etwa eine Niederleistungskonfiguration der Verarbeitungscluster DPLAB06A-DPLAB06H, innerhalb der hochparallelen Allzweck-Grafikverarbeitungseinheit DPLAB00 beinhalten. Die Verarbeitungscluster innerhalb der GPGPU 3106 können Anweisungen unterstützen, die spezifisch zur Durchführung von Inferenzberechnungen in einem trainierten neuronalen Netzwerk optimiert sind. Die GPGPU 3106 kann beispielsweise Anweisungen zur Durchführung von Berechnungen mit geringer Präzision wie 8-Bit- und 4-Bit-Ganzzahl-Vektoroperationen unterstützen.

RAYTRACING-ARCHITEKTUR

[0242] Bei einer Implementierung beinhaltet der Grafikprozessor eine Schaltungsanordnung und/oder einen Programmcode zum Durchführen von Echtzeit-Raytracing. Ein dedizierter Satz von Raytracing-Kernen kann in dem Grafikprozessor enthalten sein, um die verschiedenen hierin beschriebenen Raytracing-Operationen durchzuführen, darunter Strahltraversierungs- und/oder Strahlüberschneidungsoperationen. Zusätzlich zu den Raytracing-Kernen können auch mehrere Sätze von Grafikverarbeitungskernen zum Durchführen programmierbarer Shading-Operationen und mehrere Sätze von Tensorkernen zum Durchführen von Matrixoperationen an Tensordaten enthalten sein.

[0243] Fig. 31 veranschaulicht einen beispielhaften Teil einer solchen Grafikverarbeitungseinheit (GPU) 3105, die dedizierte Sätze von Grafikverarbeitungsressourcen beinhaltet, die in Mehrkerngruppen 3100A-N angeordnet sind. Die Grafikverarbeitungseinheit (GPU) 3105 kann eine Variante des Grafikprozessors 300, der GPGPU 1340 und/oder eines beliebigen anderen hierin beschriebenen Grafikprozessors sein. Daher wird durch die Offenbarung beliebiger Merkmale für Grafikprozessoren auch eine entsprechende Kombination mit der GPU 3105 offenbart, jedoch ohne Beschränkung darauf. Zudem beschreiben die Elemente von **Fig. 31** mit den gleichen oder ähnlichen Namen wie die Elemente einer beliebigen anderen Figur hierin die gleichen Elemente wie in den anderen Figuren, können auf ähnliche Weise arbeiten oder fungieren, können die gleichen Komponenten umfassen und können mit anderen Entitäten wie jene, die an anderer Stelle hierin beschrieben sind, verknüpft sein, sind jedoch nicht darauf beschränkt. Während die Einzelheiten lediglich einer einzelnen Mehrkerngruppe 3100A bereitgestellt sind, versteht es sich, dass die anderen Mehrkerngruppen 3100B-N mit denselben oder ähnlichen Sätzen von Grafikverarbeitungsressourcen ausgerüstet sein können.

[0244] Wie veranschaulicht, kann eine Mehrkerngruppe 3100 A einen Satz von Grafikkernen 3130, einen Satz von Tensorkernen 3140 und einen Satz von Raytracing-Kernen 3150 beinhalten. Ein Scheduler/Dispatcher 3110 plant und sendet die Grafik-Threads zur Ausführung auf den verschiedenen Kernen 3130, 3140, 3150 aus. Ein Satz von Registerdateien 3120 speichert Operandenwerte, die von den Kernen 3130, 3140, 3150 verwendet werden, wenn die Grafik-Threads ausgeführt werden. Diese können zum Beispiel Ganzzahlregister zum Speichern von Ganzzahlwerten, Gleitkommaregister zum Speichern von Gleitkommawerten, Vektorregister zum Speichern von gepackten Datenelementen (Ganzzahl- und/oder Gleitkommadatenelementen) und Kachelregister zum Speichern von Tensor-/Matrixwerten beinhalten. Die Kachelregister können als kombinierte Sätze von Vektorregistern implementiert werden.

[0245] Ein oder mehrere Level-1(L1)-Caches und Textureinheiten 3160 speichern Grafikdaten, wie Texturdaten, Vertexdaten, Pixeldaten, Strahlendaten, Begrenzungsvolumendaten usw., lokal innerhalb jeder Mehrkerngruppe 3100A. Ein Level-2(L2)-Cache 3180, der durch alle oder eine Teilmenge der Mehrkerngruppen 3100A-N gemeinsam genutzt wird, speichert Grafikdaten und/oder Anweisungen für mehrere gleichzeitige Grafik-Threads. Wie veranschaulicht, kann der L2-Cache 3180 über mehrere Mehrkerngruppen 3100A-N hinweg gemeinsam genutzt werden. Eine oder mehrere Speichersteuerungen 3170 koppeln die GPU 3105 mit einem Speicher 3198, der ein Systemspeicher (z. B. DRAM) und/oder ein lokaler Grafikspeicher (z. B. GDDR6-Speicher) sein kann.

[0246] Eine Eingabe/Ausgabe(EA)-Schaltungsanordnung 3195 koppelt die GPU 3105 mit einer oder mehreren EA-Vorrichtungen 3195, wie etwa Digitalsignalprozessoren (DSPs), Netzwerksteuerungen oder Benutzereingabevorrichtungen. Ein On-Chip-Interconnect kann verwendet werden, um die E/A-Vorrichtungen 3190 mit der GPU 3105 und dem Speicher 3198 zu koppeln. Eine oder mehrere EA-Speicherverwaltungseinheiten (IOMMUs) 3170 der EA-Schaltungsanordnung 3195 koppeln die EA-Vorrichtungen 3190 direkt mit dem Systemspeicher 3198. Die IOMMU 3170 kann mehrere Sätze von Seitentabellen verwalten, um virtuelle Adres-

sen auf physische Adressen im Systemspeicher 3198 abzubilden. Zusätzlich können sich die EA-Vorrichtungen 3190, die CPU(s) 3199 und die GPU(s) 3105 denselben virtuellen Adressraum teilen.

[0247] Die IOMMU 3170 kann auch Virtualisierung unterstützen. In diesem Fall kann sie einen ersten Satz von Seitentabellen dahingehend verwalten, virtuelle Gast-/Grafikadressen auf physische Gast-/Grafikadressen abzubilden, und einen zweiten Satz von Seitentabellen dahingehend verwalten, die physischen Gast-/Grafikadressen auf physische System-/Hostadressen (z. B. innerhalb des Systemspeichers 3198) abzubilden. Die Basisadressen sowohl des ersten als auch des zweiten Satzes von Seitentabellen können in Steuerregistern gespeichert werden und bei einem Kontextwechsel ausgelagert werden (z. B. sodass der neue Kontext Zugriff auf den relevanten Satz von Seitentabellen erhält). Obgleich dies in **Fig. 31** nicht veranschaulicht ist, kann jeder der Kerne 3130, 3140, 3150 und/oder die Mehrkerngruppen 3100A-N Übersetzungspuffer (TLBs: Translation Lookaside Buffers) beinhalten, um Virtuell-Gast-zu-Physisch-Gast-Übersetzungen, Physisch-Gast-zu-Physisch-Host-Übersetzungen und Virtuell-Gast-zu-Physisch-Host-Übersetzungen zu cachen.

[0248] Die CPUs 3199, die GPUs 3105 und die EA-Vorrichtungen 3190 können auf einem einzelnen Halbleiterchip und/oder Chip-Package integriert sein. Der veranschaulichte Speicher 3198 kann auf demselben Chip integriert sein oder kann über eine chipexterne Schnittstelle mit den Speichersteuerungen 3170 gekoppelt sein. Bei einer Implementierung umfasst der Speicher 3198 einen GDDR6-Speicher, der denselben virtuellen Adressraum wie andere physische Systemebenenspeicher nutzt, obgleich die zugrundeliegenden Prinzipien der Erfindung nicht auf diese spezielle Implementierung beschränkt sind.

[0249] Die Tensorkerne 3140 können eine Vielzahl von Ausführungseinheiten beinhalten, die speziell dafür ausgelegt sind, Matrixoperationen durchzuführen, die die grundlegende Rechenoperation sind, die verwendet wird, um Deep-Learning-Operationen durchzuführen. Zum Beispiel können simultane Matrixmultiplikationsoperationen für neuronales Netzwerktraining und Inferenz verwendet werden. Die Tensorkerne 3140 können eine Matrixverarbeitung unter Verwendung einer Vielzahl von Operandenpräzisionen durchführen, darunter Gleitkomma mit einfacher Präzision (z. B. 32 Bit), Gleitkomma mit halber Präzision (z. B. 16 Bit), Ganzzahlwörter (16 Bit), Bytes (8 Bit) und Halbbytes (4 Bits). Eine Neuronalnetzwerkimplementierung kann auch Merkmale jeder gerenderten Szene extrahieren, potenziell Details von mehreren Frames kombinieren, um ein finales Bild mit hoher Qualität zu konstruieren.

[0250] Bei Deep-Learning-Implementierungen kann Parallelmatrix-Multiplikationsarbeit zur Ausführung auf den Tensorkernen 3140 geplant werden. Insbesondere erfordert das Training neuronaler Netzwerke eine signifikante Anzahl von Matrix-Skalarprodukt-Operationen. Um eine Innenproduktformulierung einer N × N × N Matrixmultiplikation zu verarbeiten, können die Tensorkerne 3140 mindestens N Skalarprodukt-Verarbeitungselemente beinhalten. Bevor die Matrixmultiplikation beginnt, wird eine ganze Matrix in Kachelregister geladen und wird pro Zyklus für N Zyklen mindestens eine Spalte einer zweiten Matrix geladen. In jedem Zyklus gibt es N Skalarprodukte, die verarbeitet werden.

[0251] Matrixelemente können in Abhängigkeit von der speziellen Implementierung mit unterschiedlichen Präzisionen gespeichert werden, darunter 16-Bit-Wörter, 8-Bit-Bytes (z. B. INT8) und 4-Bit-Halbbytes (z. B. INT4). Modi mit unterschiedlichen Präzisionen können für die Tensorkerne 3140 spezifiziert werden, um sicherzustellen, dass die effizienteste Präzision für unterschiedliche Arbeitslasten verwendet wird (z. B. wie etwa Inferenzieren von Arbeitslasten, die Quantisierung zu Bytes und Halbbytes tolerieren können).

[0252] Die Raytracing-Kerne 3150 können verwendet werden, um Raytracing-Operationen sowohl für Echtzeit-Raytracing- als auch für Nicht-Echtzeit-Raytracing-Implementierungen zu beschleunigen. Insbesondere können die Raytracing-Kerne 3150 eine Strahltraversierungs-/-überschneidungsschaltungsanordnung zum Durchführen einer Strahltraversierung unter Verwendung von Hüllkörperhierarchien (BVHs: Bounding Volume Hierarchies) und Identifizieren von Überschneidungen zwischen Strahlen und Primitiven, die in den BVH-Volumina enthalten sind, beinhalten. Die Raytracing-Kerne 3150 können auch Schaltungsanordnungen zum Durchführen von Tiefenprüfung und -Culling (z. B. unter Verwendung eines Z-Puffers oder einer ähnlichen Anordnung) beinhalten. Bei einer Implementierung führen die Raytracing-Kerne 3150 Traversierungs- und Überschneidungsoperationen in Übereinstimmung mit den hierin beschriebenen Bildentrauschungstechniken durch, von denen zumindest ein Teil auf den Tensorkernen 3140 ausgeführt werden kann. Zum Beispiel können die Tensorkerne 3140 ein neuronales Deep-Learning-Netzwerk implementieren, um eine Entrauschung von Frames durchzuführen, die durch die Raytracing-Kerne 3150 erzeugt werden. Die CPU(s) 3199, Grafikkerne 3130 und/oder Raytracing-Kerne 3150 können jedoch auch alle oder einen Teil der Entrauschungs- und/oder Deep-Learning-Algorithmen implementieren.

[0253] Zudem kann, wie oben beschrieben, ein verteilter Ansatz zur Rauschentfernung eingesetzt werden, bei dem sich die GPU 3105 in einer Rechenvorrichtung befindet, die über ein Netzwerk oder ein Hochgeschwindigkeits-Interconnect mit anderen Rechenvorrichtungen gekoppelt ist. Die miteinander verbundenen Rechenvorrichtungen können sich zusätzlich Neuronalnetzwerk-Lern-/-Trainingsdaten teilen, um die Geschwindigkeit zu verbessern, mit der das Gesamtsystem lernt, eine Entrauschung für unterschiedliche Arten von Bildframes und/oder unterschiedliche Grafikanwendungen durchzuführen.

[0254] Die Raytracing-Kerne 3150 können alle BVH-Traversierungen und Strahl-Primitiv-Überschneidungen verarbeiten, wodurch verhindert wird, dass die Grafikkerne 3130 mit tausenden Anweisungen pro Strahl überlastet werden. Jeder Raytracing-Kern 3150 kann einen ersten Satz spezialisierter Schaltungsanordnungen zum Durchführen von Begrenzungsrahmentests (z. B. für Traversierungsoperationen) und einen zweiten Satz spezialisierter Schaltungsanordnungen zum Durchführen der Strahl-Dreieck-Überschneidungstests (z. B. kreuzende Strahlen, die durchlaufen wurden) beinhalten. Somit kann die Mehrkerngruppe 3100A einfach eine Strahlsonde starten, und die Raytracing-Kerne 3150 führen unabhängig Strahltraversierung und -überschneidung durch und geben Trefferdaten (z. B. ein Treffer, kein Treffer, mehrere Treffer usw.) an den Thread-Kontext zurück. Die anderen Kerne 3130, 3140 können freigegeben werden, um andere Grafik- oder Rechenarbeit durchzuführen, während die Raytracing-Kerne 3150 die Traversierungs- und Überschneidungsoperationen durchführen.

[0255] Jeder Raytracing-Kern 3150 kann eine Traversierungseinheit zum Durchführen von BVH-Prüfungsoperationen und eine Überschneidungseinheit, die Strahl-Primitiv-Überschneidungsprüfungen durchführt, beinhalten. Die Überschneidungseinheit kann dann eine „Treffer"-, „Kein-Treffer"- oder „Mehrfachtreffer"-Antwort erzeugen, die sie dem geeigneten Thread bereitstellt. Während der Traversierungs- und Überschneidungsoperationen können die Ausführungsressourcen der anderen Kerne (z. B. Grafikkerne 3130 und Tensorkerne 3140) freigegeben werden, um andere Arten von Grafikarbeit durchzuführen.

[0256] Ein hybrider Rasterisierung/Raytracing-Ansatz kann auch verwendet werden, bei dem Arbeit zwischen den Grafikkernen 3130 und Raytracing-Kernen 3150 verteilt wird.

[0257] Die Raytracing-Kerne 3150 (und/oder andere Kerne 3130, 3140) können Hardwareunterstützung für einen Raytracing-Anweisungssatz, wie etwa DirectX Ray Tracing (DXR) von Microsoft, der einen DispatchRays-Befehl beinhaltet, sowie Strahlerzeugung, Nächstgelegener-Treffer-, Beliebiger-Treffer- und Fehltreffer-Shader, die die Zuweisung eindeutiger Sätze von Shadern und Texturen für jedes Objekt ermöglichen, beinhalten. Eine andere Raytracing-Plattform, die durch die Raytracing-Kerne 3150, Grafikkerne 3130 und Tensorkerne 3140 unterstützt werden kann, ist Vulkan 1.1.85. Es sei jedoch angemerkt, dass die zugrundeliegenden Prinzipien der Erfindung nicht auf irgendeine spezielle Raytracing-ISA beschränkt sind.

[0258] Im Allgemeinen können die verschiedenen Kerne 3150, 3140, 3130 einen Raytracing-Anweisungssatz unterstützen, der Anweisungen/Funktionen für Strahlerzeugung, Nächstgelegener-Treffer, Beliebiger-Treffer, Strahl-Primitiv-Überschneidung, Primitiv-weise und hierarchische Begrenzungsrahmenkonstruktion, Fehltreffer, Visit und Ausnahmen aufweist. Genauer gesagt können Raytracing-Anweisungen enthalten sein, um die folgenden Funktionen durchzuführen:

Strahlenerzeugung - Strahlenerzeugungsanweisungen können für jedes Pixel, jedes Sampling oder jede andere benutzerdefinierte Arbeitszuweisung ausgeführt werden.

Nächstgelegener-Treffer - Eine Nächstgelegener-Treffer-Anweisung kann ausgeführt werden, um den nächstgelegenen Schnittpunkt eines Strahls mit Primitiven innerhalb einer Szene zu lokalisieren.

Beliebiger-Treffer - Eine Beliebiger-Treffer-Anweisung identifiziert mehrere Überschneidungen zwischen einem Strahl und Primitiven innerhalb einer Szene, um potenziell einen neuen nächstgelegenen Schnittpunkt zu identifizieren.

Überschneidung - Eine Überschneidungsanweisung führt eine Strahl-Primitiv-Überschneidungsprüfung durch und gibt ein Ergebnis aus.

Primitiv-weise Begrenzungsrahmenkonstruktion - Diese Anweisung erstellt einen Begrenzungsrahmen um ein gegebenes Primitiv oder eine gegebene Gruppe von Primitiven herum (z. B. wenn eine neue BVH- oder andere Beschleunigungsdatenstruktur erstellt wird).

Fehltreffer - gibt an, dass ein Strahl alle Geometrie innerhalb einer Szene oder ein spezifiziertes Gebiet einer Szene verfehlt.

Visit - gibt die Nachfolgevolumina an, die ein Strahl durchlaufen wird.

Ausnahmen - beinhalten verschiedene Typen von Ausnahme-Handler (z. B. für verschiedene Fehlerbedingungen aufgerufen).

HIERARCHISCHE STRAHLENBÜNDEL-TRACING

[0259] Hüllkörperhierarchien werden üblicherweise verwendet, um die Effizienz zu verbessern, mit der Operationen an Grafik-Primitiven und anderen Grafikobjekten durchgeführt werden. Eine BVH ist eine hierarchische Baumstruktur, die basierend auf einem Satz geometrischer Objekte konstruiert wird. Am oberen Ende der Baumstruktur befindet sich der Root-Knoten, der alle geometrischen Objekte in einer gegebenen Szene einschließt. Die einzelnen geometrischen Objekte werden von Hüllkörpern umschlossen, die die Leaf-Knoten des Baums bilden. Diese Knoten werden dann als kleine Sätze gruppiert und innerhalb größerer Hüllkörper eingeschlossen. Diese wiederum werden ebenfalls gruppiert und rekursiv in anderen größeren Hüllkörpern eingeschlossen, was schließlich zu einer Baumstruktur mit einem einzigen Hüllkörper, der durch den Root-Knoten repräsentiert wird, am oberen Ende des Baums führt. Hüllkörperhierarchien werden verwendet, um eine Vielzahl von Operationen an Sätzen geometrischer Objekte effizient zu unterstützen, wie etwa Kollisionsdetektion, Primitiv-Culling und Strahltraversierungs-/-überschneidungsoperationen, die beim Raytracing verwendet werden.

[0260] In Raytracing-Architekturen werden Strahlen durch eine BVH traversiert, um Strahl-Primitiv-Überschneidungen zu bestimmen. Falls zum Beispiel ein Strahl nicht durch den Root-Knoten der BVH verläuft, dann schneidet der Strahl keines der Primitive, die durch die BVH eingeschlossen sind, und es ist keine weitere Verarbeitung für den Strahl mit Bezug auf diesen Satz von Primitiven erforderlich. Falls ein Strahl durch einen ersten Child-Knoten der BVH verläuft, aber nicht durch den zweiten Child-Knoten, dann muss der Strahl nicht hinsichtlich Primitive getestet werden, die durch den zweiten Child-Knoten eingeschlossen sind. Auf diese Weise stellt eine BVH einen effizienten Mechanismus zum Testen auf Strahl-Primitiv-Überschneidungen bereit.

[0261] Gruppen zusammenhängender Strahlen, die als „Strahlenbündel“ bezeichnet werden, können anstelle einzelner Strahlen hinsichtlich der BVH getestet werden. **Fig. 32** veranschaulicht ein beispielhaftes Strahlenbündel 3201, das durch vier unterschiedliche Strahlen umrissen wird. Jegliche Strahlen, die das durch die vier Strahlen definierte Patch 3200 schneiden, werden als innerhalb desselben Strahlenbündels liegend betrachtet. Während das Strahlenbündel 3201 in **Fig. 32** durch eine rechteckige Anordnung von Strahlen definiert ist, können Strahlenbündel auf verschiedene andere Weisen definiert sein, während sie weiterhin die zugrundeliegenden Prinzipien der Erfindung erfüllen (z. B. Kreise, Ellipsen usw.).

[0262] **Fig. 33** veranschaulicht, wie eine Raytracing-Engine 3310 einer GPU 3320 die hierin beschriebenen Strahlbündel-Tracing-Techniken implementiert. Insbesondere erzeugt die Strahlerzeugungsschaltungsanordnung 3304 mehrere Strahlen, für die Traversierungs- und Überschneidungsoperationen durchzuführen sind. Anstatt jedoch Traversierungs- und Überschneidungsoperationen an einzelnen Strahlen durchzuführen, werden Traversierungs- und Überschneidungsoperationen unter Verwendung einer Hierarchie von Strahlenbündeln 3307 durchgeführt, die durch eine Strahlenbündelhierarchie-Konstruktionsschaltungsanordnung 3305 erzeugt werden. Die Strahlenbündelhierarchie ist analog zur Hüllkörperhierarchie (BVH). Zum Beispiel stellt **Fig. 34** ein Beispiel für ein Primärstrahlenbündel 3400 bereit, das in mehrere unterschiedliche Komponenten unterteilt sein kann. Insbesondere kann das Primärstrahlenbündel 3400 in Quadranten 3401-3404 unterteilt sein und jeder Quadrant kann selbst in Unterquadranten, wie etwa Unterquadranten A-D innerhalb des Quadranten 3404, unterteilt werden. Die Aufteilung des Primärstrahlenbündels kann auf verschiedene Weise erfolgen. Zum Beispiel kann das Primärstrahlenbündel halbiert (anstatt in Quadranten unterteilt) werden und jede Hälfte kann halbiert werden usw. Unabhängig davon, wie die Aufteilungen vorgenommen werden, wird eine hierarchische Struktur auf ähnliche Weise wie eine BVH erzeugt, z. B. mit einem Root-Knoten, der das Primärstrahlenbündel 3400 repräsentiert, einer ersten Ebene von Child-Knoten, die jeweils durch einen Quadranten 3401-3404 repräsentiert werden, Child-Knoten der zweiten Ebene für jeden Unterquadranten A-D und so weiter.

[0263] Sobald die Strahlenbündelhierarchie 3307 konstruiert ist, kann die Traversierungs-/Überschneidungsschaltungsanordnung 3306 Traversierungs-/Überschneidungsoperationen unter Verwendung der Strahlenbündelhierarchie 3307 und der BVH 3308 durchführen. Insbesondere kann sie das Strahlenbündel hinsichtlich der BVH testen und Teile des Strahlenbündels, die keine Teile der BVH schneiden, aussortieren. Unter Verwendung der in **Fig. 34** gezeigten Daten, falls zum Beispiel die mit Teilbereichen 3402 und 3403 assoziierten Teilstrahlenbündel die BVH oder einen speziellen Zweig der BVH nicht schneiden, dann können sie mit Bezug auf die BVH oder den Zweig aussortiert werden. Die verbleibenden Teile 3401, 3404 können hin-

sichtlich der BVH getestet werden, indem eine Depth-First-Suche oder ein anderer Suchalgorithmus durchgeführt wird.

[0264] Ein Verfahren zum Raytracing ist in **Fig. 35** veranschaulicht. Das Verfahren kann im Kontext der oben beschriebenen Grafikverarbeitungsarchitekturen implementiert werden, ist aber nicht auf irgendeine spezielle Architektur beschränkt.

[0265] Bei 3500 wird ein Primärstrahlenbündel konstruiert, das mehrere Strahlen umfasst, und bei 3501 wird das Strahlenbündel unterteilt und hierarchische Datenstrukturen werden erzeugt, um eine Strahlenbündelhierarchie zu erstellen. Die Operationen 3500-3501 können als eine einzige integrierte Operation durchgeführt werden, die eine Strahlenbündelhierarchie aus mehreren Strahlen konstruiert. Bei 3502 wird die Strahlenbündelhierarchie mit einer BVH verwendet, um Strahlen (aus der Strahlenbündelhierarchie) und/oder Knoten/Primitive aus der BVH auszusortieren. Bei 3503 werden Strahl-Primitiv-Überschneidungen für die verbleibenden Strahlen und Primitive bestimmt.

VERLUSTBEHAFTETE UND VERLUSTFREIE PAKETKOMPRIMIERUNG IN EINEM VERTEILTEN RAYTRACING-SYSTEM

[0266] Raytracing-Operationen können über mehrere Rechenknoten verteilt sein, die über ein Netzwerk miteinander gekoppelt sind. **Fig. 36** veranschaulicht zum Beispiel einen Raytracing-Cluster 3600, der mehrere Raytracing-Knoten 3610-3613 umfasst, die Raytracing-Operationen parallel durchführen, wobei potenziell die Ergebnisse auf einem der Knoten kombiniert werden. In der veranschaulichten Architektur sind die Raytracing-Knoten 3610-3613 über ein Gateway kommunikativ mit einer clientseitigen Raytracing-Anwendung 3630 gekoppelt.

[0267] Eine der Schwierigkeiten bei einer verteilten Architektur ist die große Menge an paketisierten Daten, die zwischen jedem der Raytracing-Knoten 3610-3613 übertragen werden müssen. Sowohl verlustfreie Komprimierungstechniken als auch verlustbehaftete Komprimierungstechniken können verwendet werden, um die zwischen den Raytracing-Knoten 3610-3613 übertragenen Daten zu reduzieren.

[0268] Um verlustfreie Komprimierung zu implementieren, anstatt Pakete zu senden, die mit den Ergebnissen bestimmter Arten von Operationen gefüllt sind, werden Daten oder Befehle gesendet, die es dem Empfangsknoten ermöglichen, die Ergebnisse zu rekonstruieren. Zum Beispiel benötigen Operationen für stochastisch abgetastete Flächenlichter und Umgebungsverdeckung (AO: Ambient Occlusion) nicht notwendigerweise Richtungen. Folglich kann ein Übertragungsknoten einfach einen Zufalls-Seed senden, der dann durch den Empfangsknoten verwendet wird, um ein zufälliges Sampling durchzuführen. Falls zum Beispiel eine Szene über Knoten 3610-3612 verteilt ist, müssen, um das Licht 1 an den Punkten p1-p3 zu sampeln, nur die Licht-ID und Ursprünge an die Knoten 3610-3612 gesendet werden. Jeder der Knoten kann dann das Licht unabhängig stochastisch sampeln. Der Zufalls-Seed kann durch den Empfangsknoten erzeugt werden. Gleichermaßen können für Primärstrahl-Trefferpunkte Umgebungsverdeckung (AO) und Weichschatten-Sampling an Knoten 3610-3612 für nachfolgende Frames berechnet werden, ohne auf die ursprünglichen Punkte zu warten. Falls zudem bekannt ist, dass ein Strahlensatz zu derselben Punktlichtquelle gehen wird, können Anweisungen, die die Lichtquelle identifizieren, zu dem Empfangsknoten gesendet werden, der sie auf den Strahlensatz anwendet. Falls es als ein anderes Beispiel N Umgebungsverdeckungsstrahlen gibt, die zu einem einzigen Punkt übertragen werden, kann ein Befehl gesendet werden, um N Samples von diesem Punkt zu erzeugen.

[0269] Verschiedene zusätzliche Techniken können zur verlustbehafteten Komprimierung angewendet werden. Zum Beispiel kann ein Quantisierungsfaktor eingesetzt werden, um alle Koordinatenwerte zu quantisieren, die mit der BVH, den Primitiven und den Strahlen assoziiert sind. Außerdem können 32-Bit-Gleitkommawerte, die für Daten wie BVH-Knoten und Primitive verwendet werden, in 8-Bit-Ganzzahlwerte umgewandelt werden. In einer beispielhaften Implementierung werden die Grenzen von Strahlenpaketen in voller Präzision gespeichert, aber einzelne Strahlenpunkte P1-P3 werden als indizierte Offsets zu den Grenzen übertragen. Auf ähnliche Weise können mehrere lokale Koordinatensysteme erzeugt werden, die 8-Bit-Ganzzahlwerte als lokale Koordinaten verwenden. Der Ursprungsort jedes dieser lokalen Koordinatensysteme kann unter Verwendung der Werte voller Präzision (z. B. 32-Bit-Gleitkommawerte) codiert werden, wodurch effektiv das globale und das lokale Koordinatensystem verbunden werden.

[0270] Das Folgende ist ein Beispiel für verlustfreie Komprimierung. Ein Beispiel für ein Strahlendatenformat, das intern in einem Raytracing-Programm verwendet wird, ist wie folgt:

```
struct Ray

{
 uint32 pixId;
 uint32 materialID;
 uint32 instanceID;
 uint64 primitiveID;
 uint32 geometryID;
 uint32 lightID;
 float origin[3];
 float direction[3];
 float t0;
 float t;
 float time;
 float normal[3]; //verwendet für Geometrieüberschneidungen
 float u;
 float v;

 float wavelength;
 float phase; //Interferometrie
 float refractedOffset; //Schlieren-artig
 float amplitude;
 float weight;
};
```

[0271] Anstatt die Rohdaten für jeden und alle erzeugten Knoten zu senden, können diese Daten durch Gruppieren von Werten und durch Erzeugen impliziter Strahlen durch Verwenden anwendbarer Metadaten, wo möglich, komprimiert werden.

Bündelung und Gruppierung von Strahldaten

[0272] Flags können für gemeinsame Daten oder Masken mit Modifikatoren verwendet werden.

```
struct RayPacket
{
 uint32 size;
 uint32 flags;
 list<Ray> rays;
}
```

Beispielsweise:

RayPacket.rays= ray_1 bis ray_256

Alle Ursprünge sind gleich

[0273] Alle Strahldaten werden gepackt, mit der Ausnahme, dass nur ein einzelner Ursprung über alle Strahlen hinweg gespeichert wird. RayPacket.Flags wird für RAYPACKET_COMMON_ORIGIN eingestellt. Wenn RayPacket beim Empfang entpackt wird, werden Ursprünge von dem einzigen Ursprungswert gefüllt.

Nur einige Strahlen haben einen gemeinsamen Ursprung

[0274] Alle Strahldaten werden gepackt, mit Ausnahme von Strahlen, die gemeinsame Ursprünge haben. Für jede Gruppe eindeutiger gemeinsamer Ursprünge wird ein Operator gepackt, der die Operation identifiziert (gemeinsame Ursprünge), den Ursprung speichert und maskiert, welche Strahlen die Informationen gemeinsam nutzen. Eine solche Operation kann an beliebigen gemeinsam genutzten Werten unter Knoten durchgeführt werden, wie etwa Material-IDs, Primitiv-IDs, Ursprung, Richtung, Normale usw.

```
struct RayOperation
{
```

```
 uint8 operationID;
 void* value;
 uint64 mask;
}
```

Senden impliziter Strahlen

[0275] Häufig können Strahldaten auf der Empfangsseite mit minimalen Metainformationen abgeleitet werden, die zu ihrer Erzeugung verwendet werden. Ein sehr häufiges Beispiel ist das Erzeugen mehrerer Sekundärstrahlen zum stochastischen Sampeln einer Fläche. Anstatt dass der Sender einen Sekundärstrahl erzeugt, ihn sendet, und der Empfänger an ihm arbeitet, kann der Sender einen Befehl, dass ein Strahl erzeugt werden muss, mit jeglichen abhängigen Informationen senden und der Strahl wird auf der Empfangsseite erzeugt. Falls der Strahl zuerst durch den Sender erzeugt werden muss, um zu bestimmen, an welchen Empfänger er gesendet werden soll, wird der Strahl erzeugt und der Zufalls-Seed kann gesendet werden, um exakt den gleichen Strahl zu regenerieren.

[0276] Um zum Beispiel einen Trefferpunkt mit 64 Schattenstrahlen zu sampeln, die eine Flächenlichtquelle sampeln, schneiden alle 64 Strahlen Bereiche von derselben Berechnung N4. Es entsteht ein RayPacket mit gemeinsamem Ursprung und gemeinsamer Normalen. Mehr Daten könnten gesendet werden, falls erwünscht ist, dass der Empfänger den resultierenden Pixelbeitrag abschattet, aber für dieses Beispiel sei angenommen, dass nur zurückgeben werden soll, ob ein Strahl auf Daten eines anderen Knotens trifft. Eine RayOperation wird für eine Schattenstrahloperation erzeugt, dem der Wert der zu sampelnden lightID und des Zufallszahlen-Seed zugewiesen werden. Wenn N4 das Strahlenpaket empfängt, erzeugt es die vollständig gefüllten Strahldaten durch Füllen sämtlicher Strahlen mit den Daten des gemeinsamen Ursprungs und Einstellen der Richtung basierend auf dem stochastisch mit dem Zufallszahlen-Seed gesampelten lightID, um die gleichen Strahlen zu erzeugen, die der ursprüngliche Sender erzeugt hat. Wenn die Ergebnisse zurückgegeben werden, müssen nur binäre Ergebnisse für jeden Strahl zurückgegeben werden, was mittels einer Maske über die Strahlen vorgenommen werden kann.

[0277] Das Senden der ursprünglichen 64 Strahlen in diesem Beispiel hätte 104 Bytes * 64 Strahlen = 6656 Bytes verwendet. Würden auch die zurückkehrenden Strahlen in ihrer Rohform gesendet, so verdoppelt sich dies ebenfalls auf 13312 Bytes. Unter Verwendung verlustfreier Komprimierung mit nur Senden der Gemeinsamer-Strahlenursprungs-, Normalen- und Strahlerzeugungsoperation mit Seed und ID werden nur 29 Bytes gesendet, wobei 8 Bytes für die Wurde-Geschnitten-Maske zurückgegeben werden. Daraus resultiert eine über das Netzwerk zu sendende Datenkomprimierungsrate von ~360:1. Dies schließt keinen Overhead zum Verarbeiten der Nachricht selbst ein, der auf gewisse Weise identifiziert werden müsste, was jedoch der Implementierung überlassen wird. Andere Operationen können zum Neuberechnen von Strahlenursprung und Richtungen von der pixelID für Primärstrahlen, Neuberechnen von pixelIDs basierend auf den Bereichen in dem Strahlenpaket und für viele andere mögliche Implementierungen zum Neuberechnen von Werten durchgeführt werden. Ähnliche Operationen können für einen beliebigen einzelnen oder eine beliebige Gruppe von gesendeten Strahlen verwendet werden, einschließlich Schatten, Reflexionen, Brechung, Umgebungsverdeckung, Überschneidungen, Körperüberschneidungen, Schattierung, zurückgeworfene Reflexionen bei der Pfad-Tracing usw.

[0278] **Fig. 37** veranschaulicht zusätzliche Details für zwei Raytracing-Knoten 3710-3711, die eine Komprimierung und Dekomprimierung von Raytracing-Paketen durchführen. Wenn insbesondere eine erste Raytracing-Engine 3730 bereit ist, Daten zu einer zweiten Raytracing-Engine 3731 zu übertragen, führt eine Strahlkomprimierungsschaltungsanordnung 3720 eine verlustbehaftete und/oder verlustfreie Komprimierung der Raytracing-Daten durch, wie hierin beschrieben (z. B. Umwandeln von 32-Bit-Werten in 8-Bit-Werte, Ersetzen von Rohdaten mit Anweisungen zum Rekonstruieren der Daten usw.). Die komprimierten Strahlenpakete 3701 werden über ein lokales Netzwerk (z. B. ein 10 Gb/s, 100 Gb/s Ethernet-Netzwerk) von einer Netzwerkschnittstelle 3725 zu einer Netzwerkschnittstelle 3726 übertragen. Eine Strahldekomprimierungsschaltungsanordnung dekomprimiert dann die Strahlenpakete, falls angemessen. Sie kann zum Beispiel Befehle zum Rekonstruieren der Raytracing-Daten ausführen (z. B. unter Verwendung eines Zufalls-Seed, um ein Zufalls-Sampling für Beleuchtungsoperationen durchzuführen). Die Raytracing-Engine 3731 verwendet dann die empfangenen Daten, um Raytracing-Operationen durchzuführen.

[0279] In der umgekehrten Richtung komprimiert die Strahlkomprimierungsschaltungsanordnung 3741 Strahldaten, die Netzwerkschnittstelle 3726 überträgt die komprimierten Strahldaten über das Netzwerk (z. B. unter Verwendung der hierin beschriebenen Techniken), die Strahldekomprimierungsschaltungsanordnung

3740 dekomprimiert die Strahldaten, wenn nötig, und die Raytracing-Engine 3730 verwendet die Daten in Raytracing-Operationen. Obwohl sie in **Fig. 37** als eine separate Einheit veranschaulicht ist, kann die Strahldekomprimierungsschaltungsanordnung 3740-3741 jeweils in die Raytracing-Engines 3730-3731 integriert sein. Zum Beispiel können, sofern die komprimierten Strahldaten Befehle zum Rekonstruieren der Strahldaten umfassen, diese Befehle durch jede jeweilige Raytracing-Engine 3730-3731 ausgeführt werden.

[0280] Wie in **Fig. 38** veranschaulicht, kann die Strahlkomprimierungsschaltungsanordnung 3720 eine Verlustbehaftete-Komprimierung-Schaltungsanordnung 3801 zum Durchführen der hierin beschriebenen verlustbehafteten Komprimierungstechniken (z. B. Umwandeln von 32-Bit-Gleitkomma-Koordinaten in 8-Bit-Ganzzahl-Koordinaten) und eine Verlustfreie-Komprimierung-Schaltungsanordnung 3803 zum Durchführen der verlustfreien Komprimierungstechniken (z. B. Übertragen von Befehlen und Daten, um zu ermöglichen, dass eine Strahlneukomprimierungsschaltungsanordnung 3821 die Daten rekonstruiert) beinhalten. Die Strahldekomprimierungsschaltungsanordnung 3721 beinhaltet eine Verlustbehaftete-Dekomprimierung-Schaltungsanordnung 3802 und eine Verlustfreie-Dekomprimierung-Schaltungsanordnung 3804 zum Durchführen einer verlustfreien Dekomprimierung.

[0281] Ein anderes beispielhaftes Verfahren ist in **Fig. 39** veranschaulicht. Das Verfahren kann auf den hierin beschriebenen Raytracing-Architekturen oder anderen Architekturen implementiert werden, ist aber nicht auf irgendeine spezielle Architektur beschränkt.

[0282] Bei 3900 werden Strahldaten empfangen, die von einem ersten Raytracing-Knoten zu einem zweiten Raytracing-Knoten übertragen werden. Bei 3901 führt die Verlustbehaftete-Komprimierung-Schaltungsanordnung eine verlustbehaftete Komprimierung an ersten Raytracing-Daten durch und bei 3902 führt die Verlustfreie-Komprimierung-Schaltungsanordnung eine verlustfreie Komprimierung an zweiten Raytracing-Daten durch. Bei 3903 werden die komprimierten Raytracing-Daten zu einem zweiten Raytracing-Knoten übertragen. Bei 3904 führt die Verlustbehaftete-/Verlustfreie-Dekomprimierung-Schaltungsanordnung eine verlustbehaftete/verlustfreie Dekomprimierung der Raytracing-Daten durch und bei 3905 führt der zweite Raytracing-Knoten Raytracing-Operationen unter Verwendung der dekomprimierten Daten durch.

GRAFIKPROZESSOR MIT HARDWARE-BESCHLEUNIGTEM HYBRIDEM RAYTRACING

[0283] Als Nächstes wird eine hybride Render-Pipeline präsentiert, die Rasterisierung auf Grafikkernen 3130 und Raytracing-Operationen auf den Raytracing-Kernen 3150, Grafikkernen 3130 und/oder Kernen der CPU 3199 durchführt. Zum Beispiel können Rasterisierung und Tiefentest auf den Grafikkernen 3130 anstelle der Primär-Raycasting-Stufe durchgeführt werden. Die Raytracing-Kerne 3150 können dann Sekundärstrahlen für Strahlreflexionen, -brechungen und -schatten erzeugen. Außerdem werden gewisse Bereiche einer Szene, in denen die Raytracing-Kerne 3150 Raytracing-Operationen durchführen werden (z. B. basierend auf Materialeigenschaftsschwellen, wie etwa hohen Reflexionsgraden), ausgewählt, während andere Bereiche der Szene mit Rasterisierung auf den Grafikkernen 3130 gerendert werden. Diese hybride Implementierung kann für Echtzeit-Raytracing-Anwendungen verwendet werden, bei denen Latenz ein kritisches Problem ist.

[0284] Die unten beschriebene Strahltraversierungsarchitektur kann zum Beispiel eine programmierbare Schattierung und Steuerung der Strahltraversierung unter Verwendung vorhandener Single-Instruction-Multiple-Data(SIMD; Einzelanweisung-Mehrfachdaten)- und/oder Single-Instruction-Multiple-Thread(SIMT; Einzelanweisung-Multi-Thread)-Grafikprozessoren durchführen, während kritische Funktionen, wie etwa BVH-Traversierung und/oder -Überschneidungen, unter Verwendung dedizierter Hardware beschleunigt werden. Die SIMD-Belegung für inkohärente Pfade kann durch Umgruppieren von gespawnten Shadern an spezifischen Punkten während des Traversierens und vor dem Schattieren verbessert werden. Dies wird unter Verwendung dedizierter Hardware erreicht, die Shader dynamisch, chipintern, sortiert. Rekursion wird durch Aufteilen einer Funktion in Fortsetzungen, die bei Rückkehr ausgeführt werden, und Neugruppieren von Fortsetzungen vor der Ausführung für eine verbesserte SIMD-Belegung verwaltet.

[0285] Eine programmierbare Steuerung von Strahltraversierung/überschneidung wird durch Zerlegen der Traversierungsfunktionalität in eine innere Traversierung, die als Festfunktionshardware implementiert werden kann, und eine äußere Traversierung, die auf GPU-Prozessoren ausgeführt wird und eine programmierbare Steuerung durch benutzerdefinierte Traversierungs-Shader ermöglicht, erreicht. Der Aufwand des Transferierens des Traversierungskontextes zwischen Hardware und Software wird reduziert, indem der innere Traversierungszustand während des Übergangs zwischen innerer und äußerer Traversierung konservativ gekürzt wird.

[0286] Die programmierbare Steuerung des Raytracing kann anhand der unterschiedlichen in der folgenden Tabelle A aufgeführten Shader-Typen ausgedrückt werden. Für jeden Typ können mehrere Shader vorhanden sein. Beispielsweise kann jedes Material einen unterschiedlichen Treffer-Shader aufweisen.

TABELLE A

Shader-Typ	**Funktionalität**
Primär	Starten von Primärstrahlen
Treffer	BRDF-Sampling (BRDF: Bidirektionale Reflexionsverteilungsfunktion), Starten von Sekundärstrahlen
Beliebiger Treffer	Berechnung der Transmittanz für alpha-texturierte Geometrie
Fehltreffer	Berechnung der Strahldichte von einer Lichtquelle
Überschneidung	Überschneiden individueller Formen
Traversierung	Instanzauswahl und Transformation
Aufrufbar	Eine Mehrzweckfunktion

[0287] Rekursives Raytracing kann durch eine API-Funktion initiiert werden, die den Grafikprozessor befehligt, einen Satz von Primär-Shadern oder Überschneidungsschaltungsanordnungen zu starten, die Strahl-Szenen-Überschneidungen für Primärstrahlen spawnen können. Dies wiederum spawnt andere Shader, wie etwa Traversierung, Treffer-Shader oder Fehltreffer-Shader. Ein Shader, der einen Child-Shader spawnt, kann auch einen Rückgabewert von diesem Child-Shader empfangen. Aufrufbare Shader sind Mehrzweckfunktionen, die von einem anderen Shader direkt gespawnt werden können und auch Werte an den aufrufenden Shader zurückgeben können.

[0288] **Fig. 40** veranschaulicht eine Grafikverarbeitungsarchitektur, die eine Shader-Ausführungsschaltungsanordnung 4000 und eine Festfunktionsschaltungsanordnung 4010 beinhaltet. Das Mehrzweck-Ausführungshardwaresubsystem beinhaltet mehrere Single-Instruction-Multiple-Data(SIMD)- und/oder Single-Instruction-Multiple-Threads(SIMT)-Kerne/-Ausführungseinheiten (EUs) 4001 (d. h. jeder Kern kann mehrere Ausführungseinheiten umfassen), einen oder mehrere Sampler 4002 und einen Level-1(L1)-Cache 4003 oder eine andere Form von lokalem Speicher. Das Festfunktionshardwaresubsystem 4010 beinhaltet eine Nachrichteneinheit 4004, einen Scheduler 4007, eine Strahl-BVH-Traversierungs-/-Überschneidungsschaltungsanordnung 4005, eine Sortierungsschaltungsanordnung 4008 und einen lokalen L1-Cache 4006.

[0289] Im Betrieb versendet ein Primär-Dispatcher 4009 einen Satz von Primärstrahlen an den Scheduler 4007, der Arbeit auf Shader, die auf den SIMD/SIMT-Kernen/-EUs 4001 ausgeführt werden, verteilt. Die SIMD-Kerne/-EUs 4001 können Raytracing-Kerne 3150 und/oder Grafikkerne 3130 sein, die oben beschrieben sind. Die Ausführung der Primär-Shader spawnt zusätzliche durchzuführende Arbeit (z. B. die durch einen oder mehrere Child-Shader und/oder Festfunktionshardware auszuführen ist). Die Nachrichteneinheit 4004 verteilt Arbeit, die durch die SIMD-Kerne/-EUs 4001 gespawnt wird, an den Scheduler 4007, wobei sie nach Bedarf auf den Pool freier Stapel, die Sortierungsschaltungsanordnung 4008 oder die Strahl-BVH-Überschneidungsschaltungsanordnung 4005 zugreift. Falls die zusätzliche Arbeit an den Scheduler 4007 gesendet wird, wird sie zur Verarbeitung auf den SIMD/SIMT-Kernen/-EUs 4001 geplant. Vor dem Planen kann die Sortierungsschaltungsanordnung 4008 die Strahlen in Gruppen oder Bins sortieren, wie hierin beschrieben (z. B. Gruppieren von Strahlen mit ähnlichen Charakteristiken). Die Strahl-BVH-Überschneidungsschaltungsanordnung 4005 führt einen Überschneidungstest von Strahlen unter Verwendung von BVH-Volumina durch. Zum Beispiel kann die Strahl-BVH-Überschneidungsschaltungsanordnung 4005 Strahlkoordinaten mit jeder Ebene der BVH vergleichen, um Volumina zu identifizieren, die von dem Strahl geschnitten werden.

[0290] Shader können unter Verwendung einer Shader-Aufzeichnung, einer benutzerzugewiesenen Struktur, die einen Zeiger auf die Eingangsfunktion beinhaltet, herstellerspezifischer Metadaten und globaler Argumente zu dem Shader, ausgeführt durch die SIMD-Kerne/-EUs 4001, referenziert werden. Jede ausführende Instanz eines Shaders ist mit einem Aufrufstapel assoziiert, der verwendet werden kann, um Argumente zu speichern, die zwischen einem Parent-Shader und einem Child-Shader weitergeleitet werden. Aufrufstapel können auch Referenzen zu den Fortsetzungsfunktionen speichern, die ausgeführt werden, wenn ein Aufruf zurückkehrt.

[0291] **Fig. 41** veranschaulicht einen beispielhaften Satz zugewiesener Stapel 4101, der einen Primär-Shader-Stapel, einen Treffer-Shader-Stapel, einen Traversierung-Shader-Stapel, einen Fortsetzungsfunktionsstapel und einen Strahl-BVH-Überschneidungsstapel beinhaltet (die, wie beschrieben, durch Festfunktionshardware 4010 ausgeführt werden können). Neue Shader-Aufrufe können neue Stapel aus einem Pool 4102 freier Stapel implementieren. Die Aufrufstapel, z. B. Stapel, die in dem Satz zugewiesener Stapel enthalten sind, können in einem lokalen L1-Cache 4003, 4006 gecacht werden, um die Latenz von Zugriffen zu reduzieren.

[0292] Es kann eine endliche Anzahl von Aufrufstapeln geben, jeder mit einer festen Maximalgröße „Sstack“, die in einem zusammenhängenden Speicherbereich zugewiesen ist. Daher kann die Basisadresse eines Stapels direkt von einem Stapelindex (SID) als Basisadresse = SID * Sstack berechnet werden. Stapel-IDs können durch den Scheduler 4007 zugewiesen und freigegeben werden, wenn Arbeit auf die SIMD-Kerne/-EUs 4001 verteilt wird.

[0293] Der Primär-Dispatcher 4009 kann einen Grafikprozessor-Befehlsprozessor umfassen, der Primär-Shader als Reaktion auf einen Dispatch-Befehl von dem Host (z. B. einer CPU) versendet. Der Scheduler 4007 kann diese Dispatch-Anforderungen empfangen und startet einen Primär-Shader auf einem SIMD-Prozessor-Thread, falls er eine Stapel-ID für jede SIMD-Spur zuweisen kann. Stapel-IDs können aus dem Pool 4102 freier Stapel, der zu Beginn des Dispatch-Befehls initialisiert wird, zugewiesen werden.

[0294] Ein ausführender Shader kann einen Child-Shader spawnen, indem er eine Spawn-Nachricht an die Messaging-Einheit 4004 sendet. Dieser Befehl beinhaltet die Stapel-IDs, die mit dem Shader assoziiert sind, und beinhaltet auch einen Zeiger auf die Child-Shader-Aufzeichnung für jede aktive SIMD-Spur. Ein Parent-Shader kann diese Nachricht nur einmal für eine aktive Spur ausgeben. Nach dem Senden von Spawn-Nachrichten für alle relevanten Spuren kann der Parent-Shader beendet werden.

[0295] Ein Shader, der auf den SIMD-Kernen/-EUs 4001 ausgeführt wird, kann auch Festfunktionsaufgaben, wie etwa Strahl-BVH-Überschneidungen, unter Verwendung einer Spawn-Nachricht mit einem Shader-Aufzeichnungszeiger, der für die Festfunktionshardware reserviert ist, spawnen. Wie erwähnt, sendet die Messaging-Einheit 4004 gespawnte Strahl-BVH-Überschneidungsarbeit an die Strahl-BVH-Überschneidungsschaltungsanordnung 4005 mit fester Funktion und aufrufbare Shader direkt an die Sortierungsschaltungsanordnung 4008. Die Sortierungsschaltungsanordnung kann die Shader nach Shader-Aufzeichnungszeiger gruppieren, um ein SIMD-Batch mit ähnlichen Charakteristiken abzuleiten. Dementsprechend können Stapel-IDs von unterschiedlichen Parent-Shadern durch die Sortierungsschaltungsanordnung 4008 in demselben Batch gruppiert werden. Die Sortierungsschaltungsanordnung 4008 sendet gruppierte Batches an den Scheduler 4007, der auf die Shader-Aufzeichnung aus dem Grafikspeicher 2511 oder dem Last-Level-Cache (LLC) 4020 zugreift und den Shader auf einem Prozessor-Thread startet.

[0296] Fortsetzungen können als aufrufbare Shader behandelt werden und auch über Shader-Aufzeichnungen referenziert werden. Wenn ein Child-Shader gespawnt wird und Werte an den Parent-Shader zurückgibt, kann ein Zeiger auf die Fortsetzungs-Shader-Aufzeichnung auf dem Aufrufstapel 4101 gepusht werden. Wenn ein Child-Shader zurückkehrt, kann die Fortsetzungs-Shader-Aufzeichnung dann per Pop von dem Aufrufstapel 4101 entfernt werden und ein Fortsetzungs-Shader kann gespawnt werden. Optional können erzeugte Fortsetzungen die Sortierungseinheit ähnlich aufrufbaren Shadern durchlaufen und auf einem Prozessor-Thread gestartet werden.

[0297] Wie in **Fig. 42** veranschaulicht, gruppiert die Sortierungsschaltungsanordnung 4008 gespawnte Aufgaben nach Shader-Aufzeichnungszeigern 4201A, 4201B, 4201n, um SIMD-Batches zur Schattierung zu erzeugen. Die Stapel-IDs oder Kontext-IDs in einem sortierten Batch können aus unterschiedlichen Dispatches und unterschiedlichen Eingangs-SIMD-Spuren gruppiert werden. Eine Gruppierungsschaltungsanordnung 4210 kann das Sortieren unter Verwendung einer CAM-Struktur 4201 (CAM: Content Addressable Memory - inhaltsadressierbarer Speicher) durchführen, die mehrere Einträge umfasst, wobei jeder Eintrag mit einem Tag 4201 identifiziert ist. Wie erwähnt, kann das Tag 4201 ein entsprechender Shader-Aufzeichnungszeiger 4201A, 4201B, 4201n sein. Die CAM-Struktur 4201 kann eine begrenzte Anzahl von Tags (z. B. 32, 64, 128 usw.) speichern, die jeweils mit einem unvollständigen SIMD-Batch assoziiert sind, das einem Shader-Aufzeichnungszeiger entspricht.

[0298] Für einen eingehenden Spawn-Befehl weist jede SIMD-Spur eine entsprechende Stapel-ID (gezeigt als 16 Kontext-IDs 0-15 in jedem CAM-Eintrag) und einen Shader-Aufzeichnungszeiger 4201A-B,...n (agierend als ein Tag-Wert) auf. Die Gruppierungsschaltungsanordnung 4210 kann den Shader-Aufzeichnungszei-

ger für jede Spur mit den Tags 4201 in der CAM-Struktur 4201 vergleichen, um ein übereinstimmendes Batch zu finden. Wird ein übereinstimmendes Batch gefunden, kann die Stapel-ID/Kontext-ID zu dem Batch hinzugefügt werden. Andernfalls kann ein neuer Eintrag mit einem neuen Shader-Aufzeichnungszeiger-Tag erzeugt werden, wodurch möglicherweise ein älterer Eintrag mit einem unvollständigen Batch entfernt wird.

[0299] Ein ausführender Shader kann den Aufrufstapel freigeben, wenn er leer ist, indem er eine Freigabenachricht an die Nachrichteneinheit sendet. Die Freigabenachricht wird an den Scheduler weitergeleitet, der Stapel-IDs/Kontext-IDs für aktive SIMD-Spuren an den freien Pool zurückgibt.

[0300] Es wird ein hybrider Ansatz für Strahltraversierungsoperationen unter Verwendung einer Kombination aus Festfunktions-Strahltraversierung und Software-Strahltraversierung präsentiert. Folglich stellt sie die Flexibilität der Softwaretraversierung bereit, während die Effizienz der Festfunktionstraversierung beibehalten wird. **Fig. 43** zeigt eine Beschleunigungsstruktur, die für eine hybride Traversierung verwendet werden kann, die ein zweistufiger Baum mit einer einzigen BVH 4300 oberster Ebene und mehreren BVHs 4301 und 4302 unterster Ebene ist. Grafische Elemente sind rechts gezeigt, um innere Traversierungspfade 4303, äußere Traversierungspfade 4304, Traversierungsknoten 4305, Leaf-Knoten mit Dreiecken 4306 und Leaf-Knoten mit angepassten Primitiven 4307 anzugeben.

[0301] Die Leaf-Knoten mit Dreiecken 4306 in der BVH 4300 oberster Ebene können Dreiecke, Überschneidungs-Shader-Aufzeichnungen für angepasste Primitive oder Traversierungs-Shader-Aufzeichnungen referenzieren. Die Leaf-Knoten mit Dreiecken 4306 der BVHs 4301-4302 unterster Ebene können nur Dreiecke und Überschneidungs-Shader-Aufzeichnungen für angepasste Primitive referenzieren. Die Art der Referenz ist innerhalb des Leaf-Knotens 4306 codiert. Die innere Traversierung 4303 bezieht sich auf eine Traversierung innerhalb jeder BVH 4300-4302. Innere Traversierungsoperationen umfassen Berechnung von Strahl-BVH-Überschneidungen, und eine Traversierung über die BVH-Strukturen 4300-4302 hinweg ist als äußere Traversierung bekannt. Innere Traversierungsoperationen können effizient in Festfunktionshardware implementiert werden, während äußere Traversierungsoperationen mit akzeptabler Leistungsfähigkeit mit programmierbaren Shadern durchgeführt werden können. Folglich können innere Traversierungsoperationen unter Verwendung der Festfunktionsschaltungsanordnung 4010 durchgeführt werden und äußere Traversierungsoperationen können unter Verwendung der Shader-Ausführungsschaltungsanordnung 4000 einschließlich der SIMD/SIMT-Kerne/-EUs 4001 zum Ausführen programmierbarer Shader durchgeführt werden.

[0302] Es ist anzumerken, dass die SIMD/SIMT-Kerne/-EUs 4001 hierin der Einfachheit halber manchmal einfach als „Kerne“, „SIMD-Kerne“, „EUs“ oder „SIMD-Prozessoren“ bezeichnet werden. Gleichermaßen wird die Strahl-BVH-Traversierungs-/-Überschneidungsschaltungsanordnung 4005 mitunter einfach als eine „Traversierungseinheit“, „Traversierungs-/Überschneidungseinheit“ oder „Traversierungs-/Überschneidungsschaltungsanordnung“ bezeichnet. Wenn ein alternativer Begriff verwendet wird, ändert der bestimmte Name, der zum Bezeichnen der jeweiligen Schaltungsanordnung/Logik verwendet wird, die zugrundeliegenden Funktionen, die die Schaltungsanordnung/Logik durchführt, nicht, wie hierin beschrieben.

[0303] Obwohl dies zu Erläuterungszwecken in **Fig. 40** als eine einzige Komponente veranschaulicht ist, kann darüber hinaus die Traversierungs-/Überschneidungseinheit 4005 eine distinkte Traversierungseinheit und eine separate Überschneidungseinheit umfassen, die jeweils in einer Schaltungsanordnung und/oder Logik, wie hierin beschrieben, implementiert sein können.

[0304] Wenn ein Strahl einen Traversierungsknoten während einer inneren Traversierung schneidet, kann ein Traversierungs-Shader gespawnt werden. Die Sortierungsschaltungsanordnung 4008 kann diese Shader durch Shader-Aufzeichnungszeiger 4201A-B, n gruppieren, um ein SIMD-Batch zu erzeugen, das durch den Scheduler 4007 zur SIMD-Ausführung auf den Grafik-SIMD-Kernen/-EUs 4001 gestartet wird. Traversierungs-Shader können die Traversierung auf mehrere Weisen modifizieren, wodurch ein breites Spektrum von Anwendungen ermöglicht wird. Zum Beispiel kann der Traversierungs-Shader eine BVH mit einem gröberen Detaillierungsgrad (LOD: Level Of Detail) auswählen oder den Strahl transformieren, um Starrkörpertransformationen zu ermöglichen. Der Traversierungs-Shader kann dann eine innere Traversierung für die ausgewählte BVH spawnen.

[0305] Die innere Traversierung berechnet Strahl-BVH-Überschneidungen durch Traversieren der BVH und Berechnen von Strahl-Rahmen- und Strahl-Dreieck-Überschneidungen. Die innere Traversierung wird auf die gleiche Weise wie Shader gespawnt, indem eine Nachricht an die Messaging-Schaltungsanordnung 4004 gesendet wird, die die entsprechende Spawn-Nachricht an die Strahl-BVH-Überschneidungsschaltungsanordnung 4005 weiterleitet, die Strahl-BVH-Überschneidungen berechnet.

[0306] Der Stapel für innere Traversierung kann lokal in der Festfunktionsschaltungsanordnung 4010 (z. B. innerhalb des L1-Cache 4006) gespeichert sein. Wenn ein Strahl einen Leaf-Knoten schneidet, der einem Traversierungs-Shader oder einem Überschneidungs-Shader entspricht, kann die innere Traversierung beendet werden und der innere Stapel gekürzt werden. Der gekürzte Stapel kann zusammen mit einem Zeiger auf den Strahl und die BVH an einem Ort, der durch den aufrufenden Shader spezifiziert wird, in den Speicher geschrieben werden, und dann kann der entsprechende Traversierungs-Shader oder Überschneidungs-Shader gespawnt werden. Falls der Strahl beliebige Dreiecke während der inneren Traversierung schneidet, können die entsprechenden Trefferinformationen als Eingabeargumente an diese Shader geliefert werden, wie im nachstehenden Code gezeigt ist. Diese gespawnten Shader können durch die Sortierungsschaltungsanordnung 4008 gruppiert werden, um SIMD-Batches zur Ausführung zu erzeugen.

```
struct HitInfo {
   float barycentrics[2];
   float tmax;
   bool innerTravComplete;
   uint primID;
   uint geomID;
   ShaderRecord* leafShaderRecord;
}
```

[0307] Das Kürzen des inneren Traversierungsstapels reduziert die Kosten eines Überlaufs desselben in den Speicher. Es kann der in Restart Trail for Stackless BVH Traversal, High Performance Graphics (2010), S. 107-111 beschriebene Ansatz zum Kürzen des Stapels auf eine kleine Anzahl von Einträgen am oberen Ende des Stapels, einen 42-Bit-Neustartpfad und einen 6-Bit-Tiefenwert angewendet werden. Der Neustartpfad gibt Verzweigungen an, die bereits innerhalb der BVH genommen wurden, und der Tiefenwert gibt die Tiefe des Traversierens entsprechend dem letzten Stapeleintrag an. Dies sind ausreichende Informationen, um die innere Traversierung zu einem späteren Zeitpunkt wieder aufzunehmen.

[0308] Die innere Traversierung ist abgeschlossen, wenn der innere Stapel leer ist und keine BVH-Knoten mehr zu testen sind. In diesem Fall wird ein Außenstapel-Handler gespawnt, der das obere Glied des Außenstapels per Pop entfernt und die Traversierung wieder aufnimmt, wenn der Außenstapel nicht leer ist.

[0309] Die äußere Traversierung kann die Haupttraversierungszustandsmaschine ausführen und kann in Programmcode, der von der Shader-Ausführungsschaltungsanordnung 4000 ausgeführt wird, implementiert werden. Sie kann eine innere Traversierungsanfrage unter folgenden Bedingungen spawnen: (1) wenn ein neuer Strahl durch einen Treffer-Shader oder einen Primär-Shader gespawnt wird; (2) wenn ein Traversierungs-Shader eine BVH zur Traversierung auswählt; und (3) wenn ein Außenstapel-Handler die innere Traversierung für eine BVH wiederaufnimmt.

[0310] Wie in **Fig. 44** veranschaulicht, wird, bevor die innere Traversierung gespawnt wird, Platz in dem Aufrufstapel 4405 für die Festfunktionsschaltungsanordnung 4010 zum Speichern des gekürzten inneren Stapels 4410 zugewiesen. Die Offsets 4403-4404 zur Oberseite des Aufrufstapels und zum inneren Stapel werden im Traversierungszustand 4400 beibehalten, der ebenfalls im Speicher 2511 gespeichert wird. Der Traversierungszustand 4400 beinhaltet auch den Strahl im realen Raum 4401 und Objektraum 4402 sowie Trefferinformationen für das nächstgelegene schneidende Primitiv.

[0311] Der Traversierungs-Shader, der Überschneidungs-Shader und der Außenstapel-Handler werden alle durch die Strahl-BVH-Überschneidungsschaltungsanordnung 4005 gespawnt. Der Traversierungs-Shader weist auf dem Aufrufstapel 4405 zu, bevor er eine neue innere Traversierung für die BVH zweiter Ebene initiiert. Der Außenstapel-Handler ist ein Shader, der für die Aktualisierung der Trefferinformationen und die Wiederaufnahme ausstehender innerer Traversierungsaufgaben zuständig ist. Der Außenstapel-Handler ist auch zum Spawnen von Treffer- oder Fehltreffer-Shader zuständig, wenn die Traversierung abgeschlossen ist. Die Traversierung ist abgeschlossen, wenn es keine ausstehenden inneren Traversierungsanfragen zu spawnen gibt. Wenn die Traversierung abgeschlossen ist und eine Überschneidung gefunden wird, wird ein Treffer-Shader erzeugt; andernfalls wird ein Fehltreffer-Shader erzeugt.

[0312] Während das oben beschriebene hybride Traversierungsschema eine zweistufige BVH-Hierarchie verwendet, kann auch eine beliebige Anzahl von BVH-Stufen mit einer entsprechenden Änderung in der äußeren Traversierungsimplementierung implementiert werden.

[0313] Obwohl die Festfunktionsschaltungsanordnung 4010 oben zum Durchführen von Strahl-BVH-Überschneidungen beschrieben ist, können zusätzlich auch andere Systemkomponenten in der Festfunktionsschaltungsanordnung implementiert werden. Zum Beispiel kann der oben beschriebene Außenstapel-Handler ein interner (nicht benutzersichtbarer) Shader sein, der potenziell in der Festfunktions-BVH-Traversierungs-/-Überschneidungsschaltungsanordnung 4005 implementiert werden könnte. Diese Implementierung kann verwendet werden, um die Anzahl an versandten Shader-Stufen und Durchläufen zwischen der Festfunktionsüberschneidungshardware 4005 und dem Prozessor zu reduzieren.

[0314] Die hierin beschriebenen Beispiele ermöglichen programmierbare Schattierung und Strahltraversierungssteuerung unter Verwendung benutzerdefinierter Funktionen, die mit größerer SIMD-Effizienz auf bestehenden und zukünftigen GPU-Prozessoren ausgeführt werden können. Eine programmierbare Steuerung der Strahltraversierung ermöglicht mehrere wichtige Merkmale, wie etwa prozedurale Instanziierung, stochastische Detaillierungsgradauswahl, angepasste Primitiv-Überschneidungen und Lazy-BVH-Aktualisierungen.

[0315] Es wird auch eine programmierbare Multiple-Instruction-Multiple-Data(MIMD - Mehrfachanweisung-Mehrfachdaten)-Raytracing-Architektur, die spekulative Ausführung von Treffer- und Überschneidungs-Shadern unterstützt, bereitgestellt. Insbesondere fokussiert sich die Architektur auf das Reduzieren des Scheduling- und Kommunikations-Overhead zwischen den programmierbaren SIMD/SIMT-Kernen/-Ausführungseinheiten 4001, die oben mit Bezug auf **Fig. 40** beschrieben sind, und MIMD-Traversierungs-/-Überschneidungseinheiten 4005 mit fester Funktion in einer hybriden Raytracing-Architektur. Im Folgenden sind mehrere spekulative Ausführungsschemata von Treffer- und Überschneidungs-Shadern beschrieben, die in einem einzigen Batch von der Traversierungshardware versendet werden können, wodurch mehrere Traversierungs- und Shading-Durchläufe vermieden werden. Es kann eine dedizierte Schaltungsanordnung zum Implementieren dieser Techniken verwendet werden.

[0316] Die Ausführungsformen der Erfindung sind besonders vorteilhaft in Anwendungsfällen, in denen die Ausführung mehrerer Treffer- oder Überschneidungs-Shader aus einer Strahltraversierungsanfrage gewünscht wird, die bei Implementierung ohne dedizierte Hardwareunterstützung signifikanten Overhead verursachen würde. Diese beinhalten unter anderem die Nächste-k-Treffer-Anfrage (Starten eines Treffer-Shaders für die k nächstgelegenen Überschneidungen) und mehrere programmierbare Überschneidungs-Shader.

[0317] Die hier beschriebenen Techniken können als Erweiterungen der in **Fig. 40** veranschaulichten (und mit Bezug auf die **Fig. 40-44** beschriebenen) Architektur implementiert werden. Insbesondere bauen die vorliegenden Ausführungsformen der Erfindung auf dieser Architektur mit Verbesserungen auf, um die Leistungsfähigkeit der oben erwähnten Anwendungsfälle zu verbessern.

[0318] Eine Leistungsfähigkeitsbegrenzung hybrider Raytracing-Traversierungsarchitekturen besteht im Overhead des Startens von Traversierungsanfragen von den Ausführungseinheiten und im Overhead des Aufrufens programmierbarer Shader aus der Raytracing-Hardware. Wenn mehrere Treffer- oder Überschneidungs-Shader während der Traversierung desselben Strahls aufgerufen werden, erzeugt dieser Overhead „Ausführungsdurchläufe“ zwischen den programmierbaren Kernen 4001 und der Traversierungs-/Überschneidungseinheit 4005. Dadurch wird auch die Sortierungseinheit 4008 zusätzlich belastet, die die SIMD/-SIMT-Kohärenz aus den einzelnen Shader-Aufrufen extrahieren muss.

[0319] Einige Aspekte des Raytracing erfordern eine programmierbare Steuerung, die durch die verschiedenen Shader-Typen ausgedrückt werden kann, die in der obigen TABELLE A aufgelistet sind (d. h. Primär, Treffer, Beliebiger-Treffer, Fehltreffer, Überschneidung, Traversierung und Aufrufbar). Für jeden Typ können mehrere Shader vorhanden sein. Beispielsweise kann jedes Material einen unterschiedlichen Treffer-Shader aufweisen. Einige dieser Shader-Typen sind in der aktuellen Microsoft® Ray Tracing API definiert.

[0320] Zur Erinnerung: Rekursives Raytracing wird durch eine API-Funktion initiiert, die die GPU befehligt, einen Satz von Primär-Shadern zu starten, die Strahl-Szenen-Überschneidungen (implementiert in Hardware und/oder Software) für Primärstrahlen spawnen können. Dies kann wiederum andere Shader, wie etwa Traversierungs-, Treffer- oder Fehltreffer-Shader, spawnen. Ein Shader, der einen Child-Shader spawnt, kann auch einen Rückgabewert von diesem Shader empfangen. Aufrufbare Shader sind Mehrzweckfunktionen, die von einem anderen Shader direkt gespawnt werden können und auch Werte an den aufrufenden Shader zurückgeben können.

[0321] Strahltraversierung berechnet Strahl-Szenen-Überschneidungen durch Traversieren und Überschneiden von Knoten in einer Hüllkörperhierarchie (BVH: Bounding Volume Hierarchy). Jüngste Forschungen haben gezeigt, dass sich die Effizienz des Berechnens von Strahlen-Szenen-Überschneidungen unter Verwendung von Techniken, die sich besser für Festfunktionshardware eignen, wie etwa Arithmetik mit reduzierter Präzision, BVH-Komprimierung, Pro-Strahl-Zustandsmaschinen, dedizierte Überschneidungs-Pipelines und angepasste Caches, um mehr als eine Größenordnung verbessern lässt.

[0322] Die in **Fig. 40** gezeigte Architektur umfasst ein solches System, bei dem ein Array von SIMD/SIMT-Kernen/-Ausführungseinheiten 4001 mit einer Raytracing-/-Überschneidungseinheit 4005 mit fester Funktion interagiert, um ein programmierbares Raytracing durchzuführen. Programmierbare Shader werden auf SIMD/SIMT-Threads auf den Ausführungseinheiten/Kernen 4001 abgebildet, wobei SIMD/SIMT-Nutzung, Ausführung und Datenkohärenz für optimale Leistungsfähigkeit kritisch sind. Häufig wird die Kohärenz durch Strahlanfragen aufgelöst, wofür es verschiedene Gründe gibt, wie etwa:

- Traversierungsdivergenz: Die Dauer der BVH-Traversierung variiert stark
- zwischen Strahlen, die asynchrone Stahlverarbeitung begünstigen.
- Ausführungsdivergenz: Strahlen, die aus unterschiedlichen Spuren desselben SIMD/SIMT-Threads gespawnt werden, können zu unterschiedlichen Shader-Aufrufen führen.
- Datenzugriffsdivergenz: Strahlen, die unterschiedliche Oberflächen treffen, sampeln unterschiedliche BVH-Knoten und Primitive und Shader greifen zum Beispiel auf unterschiedliche Texturen zu. Eine Vielfalt anderer Szenarien kann Datenzugriffsdivergenz verursachen.

[0323] Die SIMD/SIMT-Kerne/-Ausführungseinheiten 4001 können Varianten von hierin beschriebenen Kernen/Ausführungseinheiten sein, einschließlich der Grafikkern(e) 415A-415B, Shader-Kerne 1355A-/V, Grafikkerne 3130, Grafikausführungseinheit 608, Ausführungseinheiten 852A-B oder beliebigen anderen hierin beschriebenen Kernen/Ausführungseinheiten. Die SIMD/SIMT-Kerne/-Ausführungseinheiten 4001 können anstelle des bzw. der Grafikkerne 415A-415B, Shader-Kerne 1355A-/V, Grafikkerne 3130, Grafikausführungseinheit 608, Ausführungseinheiten 852A-B oder beliebiger anderer hierin beschriebener Kerne/Ausführungseinheiten verwendet werden. Daher offenbart die Offenbarung beliebiger Merkmale in Kombination mit dem/den Grafikkern(en) 415A-415B, Shader-Kernen 1355A-/V, Grafikkernen 3130, Grafikausführungseinheit 608, Ausführungseinheiten 852A-B oder beliebigen anderen hierin beschriebenen Kernen/Ausführungseinheiten auch eine entsprechende Kombination mit den SIMD/SIMT-Kernen/-Ausführungseinheiten 4001 von **Fig. 40**, ist aber nicht darauf beschränkt.

[0324] Die Raytracing-/-überschneidungseinheit 4005 mit fester Funktion kann die ersten zwei Probleme lösen, indem jeder Strahl einzeln und außerhalb der Reihenfolge verarbeitet wird. Dies löst jedoch SIMD/-SIMT-Gruppen auf. Die Sortierungseinheit 4008 ist somit dafür verantwortlich, neue zusammenhängende SIMD/SIMT-Gruppen von Shader-Aufrufen zu bilden, die erneut an die Ausführungseinheiten zu versenden sind.

[0325] Die Vorteile einer solchen Architektur im Vergleich zu einer reinen softwarebasierten Raytracing-Implementierung direkt auf den SIMD/SIMT-Prozessoren sind leicht zu erkennen. Es gibt jedoch einen Overhead, der mit dem Messaging zwischen den SIMD/SIMT-Kernen/-Ausführungseinheiten 4001 (hierin manchmal einfach als SIMD/SIMT-Prozessoren oder Kerne/EUs bezeichnet) und der MIMD-Traversierungs-/-Überschneidungseinheit 4005 assoziiert ist. Des Weiteren extrahiert die Sortierungseinheit 4008 möglicherweise keine perfekte SIMD/SIMT-Nutzung aus inkohärenten Shader-Aufrufen.

[0326] Anwendungsfälle können identifiziert werden, bei denen Shader-Aufrufe während des Traversierens besonders häufig sein können. Es sind Verbesserungen für hybride MIMD-Raytracing-Prozessoren beschrieben, um den Kommunikations-Overhead zwischen den Kernen/EUs 4001 und den Traversierungs-/Überschneidungseinheiten 4005 signifikant zu reduzieren. Dies kann insbesondere bei der Auffindung der k-nächstgelegenen Überschneidungen und der Implementierung programmierbarer Überschneidungs-Shader von Vorteil sein. Es ist jedoch zu beachten, dass die hier beschriebenen Techniken nicht auf irgendein spezielles Verarbeitungsszenario beschränkt sind.

[0327] Eine Zusammenfassung der Kosten hoher Ebene des Raytracing-Kontextwechseln zwischen den Kernen/EUs 4001 und der Festfunktions-Traversierungs-/Überschneidungseinheit 4005 ist unten bereitgestellt. Der größte Teil des Performance-Overheads wird durch diese beiden Kontextwechsel jedes mal verursacht, wenn der Shader-Aufruf während der Einstrahltraversierung erforderlich ist.

[0328] Jede SIMD/SIMT-Spur, die einen Strahl startet, erzeugt eine Spawn-Nachricht an die Traversierungs-/Überschneidungseinheit 4005, die mit einer zu traversierenden BVH assoziiert ist. Die Daten (Strahltraversierungskontext) werden über die Spawn-Nachricht und den (gecachten) Speicher an die Traversierungs-/Überschneidungseinheit 4005 weitergeleitet. Wenn die Traversierungs-/Überschneidungseinheit 4005 bereit ist, der Spawn-Nachricht einen neuen Hardware-Thread zuzuweisen, lädt sie den Traversierungszustand und führt eine Traversierung an der BVH durch. Es gibt auch Einrichtungsaufwand, der vor dem ersten Traversierungsschritt an der BVH durchgeführt werden muss.

[0329] **Fig. 45** veranschaulicht einen Betriebsablauf einer programmierbaren Raytracing-Pipeline. Die grau unterlegten Elemente, zu denen Traversierung 4502 und Überschneidung 4503 gehören, können in einer Festfunktionsschaltungsanordnung implementiert werden, während die verbleibenden Elemente mit programmierbaren Kernen/Ausführungseinheiten implementiert werden können.

[0330] Ein Primärstrahl-Shader 4501 sendet bei 4502 Arbeit an die Traversierungsschaltungsanordnung, die den oder die aktuellen Strahlen durch die BVH (oder eine andere Beschleunigungsstruktur) traversiert. Wenn ein Leaf-Knoten erreicht wird, ruft die Traversierungsschaltungsanordnung bei 4503 die Überschneidungsschaltungsanordnung auf, die bei Identifikation einer Strahl-Dreieck-Überschneidung bei 4504 einen Beliebiger-Treffer-Shader aufruft (der Ergebnisse zurück an die Traversierungsschaltungsanordnung liefern kann, wie angegeben).

[0331] Alternativ dazu kann die Traversierung beendet werden, bevor ein Leaf-Knoten erreicht wird, und bei 4507 ein Nächstgelegener-Treffer-Shader (falls ein Treffer aufgezeichnet wurde) oder bei 4506 ein Fehltreffer-Shader (im Fall eines Fehltreffers) aufgerufen werden.

[0332] Wie bei 4505 angegeben, kann ein Überschneidungs-Shader aufgerufen werden, falls die Traversierschaltungsanordnung einen Leaf-Knoten mit angepasstem Primitiv erreicht. Ein angepasstes Primitiv kann ein jegliches Nicht-Dreieck-Primitiv sein, wie zum Beispiel ein Polygon oder ein Polyeder (z. B. Tetraeder, Voxel, Hexaeder, Keile, Pyramiden oder ein anderes „unstrukturiertes“ Volumen). Der Überschneidungs-Shader 4505 identifiziert beliebige Überschneidungen zwischen dem Strahl und dem angepassten Primitiv für den Beliebiger-Treffer-Shader 4504, der eine Beliebiger-Treffer-Verarbeitung implementiert.

[0333] Wenn die Hardwaretraversierung 4502 eine programmierbare Stufe erreicht, kann die Traversierungs-/Überschneidungseinheit 4005 eine Shader-Dispatch-Nachricht an einen relevanten Shader 4505-4507 erzeugen, die einer einzigen SIMD-Spur der zum Ausführen des Shaders verwendeten Ausführungseinheit(en) entspricht. Da Dispatches in einer beliebigen Reihenfolge von Strahlen auftreten und sie in den aufgerufenen Programmen divergent sind, kann die Sortierungseinheit 4008 mehrere Dispatch-Aufrufe akkumulieren, um kohärente SIMD-Batches zu extrahieren. Der aktualisierte Traversierungszustand und die optionalen Shader-Argumente können durch die Traversierungs-/Überschneidungseinheit 4005 in den Speicher 2511 geschrieben werden.

[0334] Bei dem Problem k-nächster Überschneidungen wird ein Nächstgelegener-Treffer-Shader 4507 für die ersten k Überschneidungen ausgeführt. Auf herkömmliche Weise würde dies Beenden der Strahltraversierung bei Auffinden der nächstgelegenen Überschneidung, Aufrufen eines Treffer-Shaders und Spawnen eines neuen Strahls aus dem Treffer-Shader bedeuten, um die nächste nächstgelegene Überschneidung zu finden (mit versetztem Strahlenursprung, sodass dieselbe Überschneidung nicht erneut auftritt). Es ist leicht zu erkennen, dass diese Implementierung k Strahl-Spawns für einen einzigen Strahl erfordern würde. Eine andere Implementierung arbeitet mit Beliebiger-Treffer-Shadern 4504, die für alle Überschneidungen aufgerufen werden und eine globale Liste nächster Überschneidungen pflegen, unter Verwendung einer Einfügungssortieroperation. Das Hauptproblem bei diesem Ansatz besteht darin, dass es keine Obergrenze von Beliebiger-Treffer-Shader-Aufrufen gibt.

[0335] Wie erwähnt, kann ein Überschneidungs-Shader 4505 auf Nicht-Dreieck- (angepasste) Primitive aufgerufen werden. In Abhängigkeit von dem Ergebnis des Überschneidungstests und dem Traversierungszustand (ausstehende Knoten- und Primitiv-Überschneidungen) kann das Traversieren desselben Strahls nach der Ausführung des Überschneidungs-Shaders 4505 fortgesetzt werden. Daher kann das Auffinden des nächstgelegenen Treffers mehrere Durchläufe zur Ausführungseinheit erfordern.

[0336] Ein Fokus kann auch auf die Reduktion von SIMD-MIMD-Kontextwechseln für Überschneidungs-Shader 4505 und Treffer-Shader 4504, 4507 durch Änderungen an der Traversierungshardware und dem Shader-Scheduling-Modell gelegt werden. Zuerst stellt die Strahltraversierungsschaltungsanordnung 4005 Sha-

der-Aufrufe zurück, indem sie mehrere potenzielle Aufrufe akkumuliert und diese in einem größeren Batch versendet. Außerdem können gewisse Aufrufe, die sich als unnötig erweisen, in dieser Phase aussortiert werden. Des Weiteren kann der Shader-Scheduler 4007 mehrere Shader-Aufrufe aus demselben Traversierungskontext zu einem einzigen SIMD-Batch aggregieren, was zu einer einzigen Strahl-Spawn-Nachricht führt. Bei einer beispielhaften Implementierung setzt die Traversierungshardware 4005 den Traversierungs-Thread aus und wartet auf die Ergebnisse mehrerer Shader-Aufrufe. Dieser Betriebsmodus wird hierin als „spekulative“ Shader-Ausführung bezeichnet, da er das Dispatch mehrerer Shader ermöglicht, von denen einige nicht aufgerufen werden können, wenn sequenzielle Aufrufe verwendet werden.

[0337] **Fig. 46A** veranschaulicht ein Beispiel, bei dem die Traversierungsoperation auf mehrere angepasste Primitive 4650 in einem Unterbaum trifft, und **Fig. 46B** veranschaulicht, wie dies mit drei Überschneidungs-Dispatch-Zyklen C1-C3 gelöst werden kann. Insbesondere kann der Scheduler 4007 drei Zyklen erfordern, um die Arbeit an den SIMD-Prozessor 4001 zu übermitteln, und die Traversierungsschaltungsanordnung 4005 erfordert drei Zyklen, um die Ergebnisse an die Sortierungseinheit 4008 zu liefern. Der Traversierungszustand 4601, der von der Traversierungsschaltungsanordnung 4005 benötigt wird, kann in einem Speicher, wie etwa einem lokalen Cache (z. B. einem L1-Cache und/oder L2-Cache), gespeichert werden.

A. Zurückgestellte Raytracing-Shader-Aufrufe

[0338] Die Art und Weise, auf die der Hardwaretraversierungszustand 4601 verwaltet wird, um die Akkumulation mehrerer potenzieller Überschneidungs- oder Trefferaufrufe in einer Liste zu ermöglichen, kann auch modifiziert werden. Zu einer gegebenen Zeit während der Traversierung kann jeder Eintrag in der Liste verwendet werden, um einen Shader-Abruf zu erzeugen. Zum Beispiel können die k-nächsten Überschneidungen auf der Traversierungshardware 4005 und/oder in dem Traversierungszustand 4601 im Speicher akkumuliert werden, und Treffer-Shader können für jedes Element aufgerufen werden, falls die Traversierung abgeschlossen ist. Für Treffer-Shader können mehrere potenzielle Überschneidungen für einen Unterbaum in der BVH akkumuliert werden.

[0339] Für den Anwendungsfall mit nächsten k besteht der Vorteil dieses Ansatzes darin, dass anstelle von k-1 Durchläufen zu dem/der SIMD-Kern/-EU 4001 und k-1 neuen Strahl-Spawn-Nachrichten alle Treffer-Shader von demselben Traversierungs-Thread während einer einzigen Traversierungsoperation auf der Traversierungsschaltungsanordnung 4005 aufgerufen werden. Eine Herausforderung für potenzielle Implementierungen ist, dass es nicht unbedeutend ist, die Ausführungsreihenfolge von Treffer-Shadern zu garantieren (der standardmäßige „Durchlauf“-Ansatz garantiert, dass der Treffer-Shader der nächstgelegenen Überschneidung zuerst ausgeführt wird usw.). Dies kann entweder durch die Synchronisation der Treffer-Shader oder die Lockerung der Reihenfolge behandelt werden.

[0340] Für den Anwendungsfall mit Überschneidungs-Shader weiß die Traversierungsschaltungsanordnung 4005 im Voraus nicht, ob ein gegebener Shader einen positiven Überschneidungstest zurückgeben würde. Es ist jedoch möglich, mehrere Überschneidungs-Shader spekulativ auszuführen, und wenn mindestens einer ein positives Trefferergebnis zurückgibt, wird er in den globalen nächsten Treffer eingegliedert. Spezielle Implementierungen müssen eine optimale Anzahl von zurückgestellten Überschneidungstests finden, um die Anzahl von Dispatch-Aufrufen zu reduzieren und gleichzeitig zu verhindern, dass zu viele redundante Überschneidungs-Shader aufgerufen werden.

B. Aggregierte Shader-Aufrufe von der Traversierungsschaltungsanordnung

[0341] Wenn mehrere Shader von demselben Strahl-Spawn auf der Traversierungsschaltungsanordnung 4005 versendet werden, können Verzweigungen in dem Fluss des Strahltraversierungsalgorithmus erzeugt werden. Dies kann für Überschneidungs-Shader problematisch sein, da der Rest des BVH-Traversierens von dem Ergebnis aller versendeten Überschneidungstests abhängt. Dies bedeutet, dass eine Synchronisierungsoperation notwendig ist, um auf das Ergebnis der Shader-Aufrufe zu warten, was auf asynchroner Hardware herausfordernd sein kann.

[0342] Zwei Punkte des Zusammenführens der Ergebnisse der Shader-Aufrufe können Folgende sein: der SIMD-Prozessor 4001 und die Traversierungsschaltungsanordnung 4005. In Bezug auf den SIMD-Prozessor 4001 können mehrere Shader ihre Ergebnisse unter Verwendung von Standardprogrammierungsmodellen synchronisieren und aggregieren. Eine relativ einfache Weise, dies zu tun, besteht darin, globale Atomics zu verwenden und Ergebnisse in einer gemeinsam genutzten Datenstruktur in dem Speicher zu aggregieren, wo Überschneidungsergebnisse mehrerer Shader gespeichert werden könnten. Dann kann der letzte Shader die

Datenstruktur auflösen und die Traversierungsschaltungsanordnung 4005 zurück aufrufen, um die Traversierung fortzusetzen.

[0343] Es kann auch ein effizienterer Ansatz implementiert werden, bei dem die Ausführung mehrerer Shader-Aufrufe auf Spuren desselben SIMD-Threads auf dem SIMD-Prozessor 4001 beschränkt wird. Die Überschneidungstests werden dann unter Verwendung von SIMD/SIMT-Reduktionsoperationen lokal reduziert (anstatt auf globale Atomics angewiesen zu sein). Diese Implementierung kann darauf beruhen, dass eine neue Schaltungsanordnung innerhalb der Sortierungseinheit 4008 einen kleinen Batch von Shader-Aufrufen in demselben SIMD-Batch belässt.

[0344] Die Ausführung des Traversierungs-Threads kann ferner auf der Traversierungsschaltungsanordnung 4005 ausgesetzt werden. Unter Verwendung des herkömmlichen Ausführungsmodells wird, wenn ein Shader während des Traversierens versendet wird, der Traversierungs-Thread beendet und der Strahltraversierungszustand in dem Speicher gespeichert, um die Ausführung anderer Strahl-Spawn-Befehle zu ermöglichen, während die Ausführungseinheiten 4001 die Shader verarbeiten. Falls der Traversierungs-Thread lediglich ausgesetzt wird, muss der Traversierungszustand nicht gespeichert zu werden, und kann auf jedes Shader-Ergebnis separat warten. Diese Implementierung kann eine Schaltungsanordnung zum Vermeiden von Blockierungen und zum Bereitstellen einer ausreichenden Hardwarenutzung beinhalten.

[0345] **Fig. 47-48** veranschaulichen Beispiele eines zurückgestellten Modells, das einen einzelnen Shader-Aufruf auf den SIMD-Kernen/-Ausführungseinheiten 4001 mit drei Shadern 4701 aufruft. Wenn sie bewahrt werden, werden alle Überschneidungstests innerhalb derselben SIMD/SIMT-Gruppe ausgewertet. Infolgedessen kann die naheliegendste Überschneidung auch auf den programmierbaren Kernen/Ausführungseinheiten 4001 berechnet werden.

[0346] Wie erwähnt, kann die gesamte oder ein Teil der Shader-Aggregation und/oder Zurücksetzung durch die Traversierungs-/Überschneidungsschaltungsanordnung 4005 und/oder den Kern/EU-Scheduler 4007 durchgeführt werden. **Fig. 47** veranschaulicht, wie eine Shader-Zurücksetzungs-/Aggregator-Schaltungsanordnung 4706 innerhalb des Schedulers 4007 ein Scheduling von Shadern, die mit einem bestimmten SIMD/-SIMT-Thread/einer bestimmten SIMD/SIMT-Spur assoziiert sind, zurücksetzen kann, bis ein spezifiziertes Auslöseereignis aufgetreten ist. Bei Detektion des Auslöseereignisses versendet der Scheduler 4007 die mehreren aggregierten Shader in einem einzigen SIMD/SIMT-Batch an die Kerne/EUs 4001.

[0347] **Fig. 48** veranschaulicht, wie die Shader-Zurücksetzungs-/Aggregator-Schaltungsanordnung 4805 innerhalb der Traversierungs-/Überschneidungsschaltungsanordnung 4005 ein Scheduling von Shadern, die mit einem speziellen SIMD-Thread/einer speziellen SIMD-Spur assoziiert sind, zurücksetzen kann, bis ein spezifiziertes Auslöseereignis aufgetreten ist. Bei Detektion des Auslöseereignisses übermittelt die Traversierungs-/Überschneidungsschaltungsanordnung 4005 die aggregierten Shader an die Sortierungseinheit 4008 in einem einzigen SIMD/SIMT-Batch.

[0348] Es wird jedoch angemerkt, dass die Shader-Zurücksetz- und Aggregationstechniken innerhalb verschiedener anderer Komponenten, wie etwa der Sortierungseinheit 4008, implementiert sein können oder über mehrere Komponenten verteilt sein können. Zum Beispiel kann die Traversierungs-/Überschneidungsschaltungsanordnung 4005 einen ersten Satz von Shader-Aggregationsoperationen durchführen und der Scheduler 4007 kann einen zweiten Satz von Shader-Aggregationsoperationen durchführen, um sicherzustellen, dass Shader für einen SIMD-Thread effizient auf den Kernen/EUs 4001 geplant werden.

[0349] Das „Auslöseereignis“, um zu veranlassen, dass die aggregierten Shader an die Kerne/EU versendet werden, kann ein Verarbeitungsereignis sein, wie etwa eine bestimmte Anzahl akkumulierter Shader, oder eine minimale Latenz, die mit einem bestimmten Thread assoziiert ist. Alternativ oder zusätzlich kann das Auslöseereignis ein zeitliches Ereignis sein, wie etwa eine gewisse Dauer ab der Zurücksetzung des ersten Shaders oder eine bestimmte Anzahl von Prozessorzyklen. Andere Variablen, wie etwa die aktuelle Arbeitslast auf den Kernen/EUs 4001 und der Traversierungs-/Überschneidungseinheit 4005, können auch durch den Scheduler 4007 evaluiert werden, um zu bestimmen, wann der SIMD/SIMT-Batch von Shadern zu versenden ist.

[0350] Unterschiedliche Ausführungsformen der Erfindung können durch Verwenden unterschiedlicher Kombinationen der obigen Ansätze implementiert werden, basierend auf der bestimmten verwendeten Systemarchitektur und den Anforderungen der Anwendung.

RAYTRACING-ANWEISUNGEN

[0351] Die unten beschriebenen Raytracing-Anweisungen sind in einer Anweisungssatzarchitektur (ISA) enthalten, die durch die CPU 3199 und/oder die GPU 3105 unterstützt wird. Wenn sie durch die CPU ausgeführt werden, können die Single-Instruction-Multiple-Data(SIMD)-Anweisungen Vektor-/gepackte Quell- und Zielregister verwenden, um die beschriebenen Operationen durchzuführen, und können durch einen CPU-Kern decodiert und ausgeführt werden. Wenn sie durch eine GPU 3105 ausgeführt werden, können die Anweisungen durch Grafikkerne 3130 ausgeführt werden. Zum Beispiel können beliebige der oben beschriebenen Ausführungseinheiten (EUs) 4001 die Anweisungen ausführen. Alternativ oder zusätzlich können die Anweisungen durch eine Ausführungsschaltungsanordnung auf den Raytracing-Kernen 3150 und/oder den Tensorkernen 3140 ausgeführt werden.

[0352] **Fig. 49** veranschaulicht eine Architektur zum Ausführen der unten beschriebenen Raytracing-Anweisungen. Die veranschaulichte Architektur kann in einem oder mehreren der oben beschriebenen Kerne 3130, 3140, 3150 (siehe z. B. **Fig. 31** und assoziierter Text) integriert sein oder kann in einer anderen Prozessorarchitektur enthalten sein.

[0353] Im Betrieb ruft eine Befehlsabrufeinheit 4903 Raytracing-Anweisungen 4900 aus dem Speicher 3198 ab und ein Decodierer 4995 decodiert die Anweisungen. In einer Implementierung decodiert der Decodierer 4995 Anweisungen, um ausführbare Operationen (z. B. Mikrooperationen oder Uops in einem mikrocodierten Kern) zu erzeugen. Alternativ dazu können manche oder alle der Raytracing-Anweisungen 4900 ohne Decodierung ausgeführt werden, und somit ist ein Decodierer 4904 nicht erforderlich.

[0354] Bei jeder Implementierung plant und versendet ein Scheduler/Dispatcher 4905 die Anweisungen (oder Operationen) über einen Satz von Funktionseinheiten (FUs) 4910-4912 hinweg. Die veranschaulichte Implementierung beinhaltet eine Vektor-FU 4910 zum Ausführen von Single-Instruction-Multiple-Data(SIMD)-Befehlen, die gleichzeitig an mehreren gepackten Datenelementen arbeiten, die in Vektorregistern 4915 gespeichert sind, und eine Skalar-FU 4911 zum Arbeiten an Skalarwerten, die in einem oder mehreren Skalarregistern 4916 gespeichert sind. Eine optionale Raytracing-FU 4912 kann an gepackten Datenwerten, die in den Vektorregistern 4915 gespeichert sind, und/oder Skalarwerten, die in den Skalarregistern 4916 gespeichert sind, arbeiten. Bei einer Implementierung ohne eine dedizierte FU 4912 können die Vektor-FU 4910 und möglicherweise die Skalar-FU 4911 die unten beschriebenen Raytracing-Anweisungen durchführen.

[0355] Die verschiedenen FUs 4910-4912 greifen auf Raytracing-Daten 4902 (z. B. Traversierungs-/Überschneidungsdaten) zu, die benötigt werden, um die Raytracing-Anweisungen 4900 von den Vektorregistern 4915, dem Skalarregister 4916 und/oder dem lokalen Cache-Subsystem 4908 (z. B. einem L1-Cache) auszuführen. Die FUs 4910-4912 können auch Zugriffe auf den Speicher 3198 über Lade- und Speicheroperationen durchführen, und das Cache-Subsystem 4908 kann unabhängig arbeiten, um die Daten lokal zu cachen.

[0356] Obwohl die Raytracing-Anweisungen verwendet werden können, um die Leistungsfähigkeit für Strahltraversierung/-überschneidung und BVH-Konstruktionen zu erhöhen, können sie auch auf andere Bereiche anwendbar sein, wie etwa Hochleistungsberechnung (HPC) und Mehrzweck-GPU(GPGPU)-Implementierungen.

[0357] In den folgenden Beschreibungen wird der Begriff Doppelwort mitunter mit dw abgekürzt und ein vorzeichenloses Byte mit ub abgekürzt. Zusätzlich dazu können sich die Quell- und Zielregister, auf die unten Bezug genommen wird (z. B. src0, src1, dest usw.), auf Vektorregister 4915 oder in manchen Fällen auf eine Kombination von Vektorregistern 4915 und Skalarregistern 4916 beziehen. Falls typischerweise ein Quell- oder Zielwert, der durch eine Anweisung verwendet wird, gepackte Datenelemente beinhaltet (z. B. wobei eine Quelle oder ein Ziel N Datenelemente speichert), werden Vektorregister 4915 verwendet. Andere Werte können Skalarregister 4916 oder Vektorregister 4915 verwenden.

Dequantisierung

[0358] Ein Beispiel für die Dequantisierungsanweisung „dequantisiert“ zuvor quantisierte Werte. Beispielsweise können in einer Raytracing-Implementierung bestimmte BVH-Unterbäume quantisiert werden, um Speicherungs- und Bandbreitenanforderungen zu reduzieren. Die Dequantisierungsanweisung kann die Form dequantize dest src0 src1 src2 annehmen, wobei das Quellregister src0 N vorzeichenlose Bytes speichert, das Quellregister src1 ein vorzeichenloses Byte speichert, das Quellregister src2 1 Gleitkommawert speichert und das Zielregister dest N Gleitkommawerte speichert. Alle diese Register können Vektorregister

4915 sein. Alternativ können src0 und dest Vektorregister 4915 sein, und src1 und src2 können Skalarregister 4916 sein.

[0359] Die folgende Code-Sequenz definiert eine bestimmte Implementierung der Dequantisierungsanweisung:

```
for (int i = 0; i < SIMD_WIDTH) {
    if (execMaske[i]) {
        dst[i] = src2[i] + Idexp(convert_to_float(src0[i]),src1);
    }
}
```

Bei diesem Beispiel multipliziert Idexp einen Gleitkommawert mit doppelter Präzision mit einer spezifizierten ganzzahligen Potenz von zwei (d. h. Idexp(x, exp) = x * 2^{exp}). Falls in dem obigen Code der Ausführungsmaskenwert, der mit dem aktuellen SIMD-Datenelement assoziiert ist (execMask[i])), auf 1 gesetzt wird, dann wird das SIMD-Datenelement am Ort i in src0 in einen Gleitkommawert umgewandelt und mit der ganzzahligen Potenz des Werts in src1 multipliziert ($2^{src1\text{-}Wert}$) und dieser Wert wird zu dem entsprechenden SIMD-Datenelement in src2 addiert.

Selektives Min oder Max

[0360] Eine Selective-Min-or-Max-Anweisung kann entweder eine Min- oder eine Max-Operation pro Spur durchführen (d. h. das Minimum oder Maximum eines Satzes von Werten zurückgeben), wie durch ein Bit in einer Bitmaske angegeben. Die Bitmaske kann die Vektorregister 4915, Skalarregister 4916 oder einen separaten Satz von Maskenregistern (nicht gezeigt) verwenden. Die folgende Codesequenz definiert eine bestimmte Implementierung der Min/Max-Anweisung: sel_min_max dest src0 src1 src2, wobei src0 N Doppelwörter speichert, src1 N Doppelwörter speichert, src2 ein Doppelwort speichert und das Zielregister N Doppelwörter speichert.

[0361] Die folgende Codesequenz definiert eine besondere Implementierung der Selective-Min/Max-Anweisung:

```
for (int i = 0; i < SIMD_WIDTH) {
    if (execMask[i]) {
    dst[i] = (1 < < i) & src2 ? min(src0[i],src1[i]) :
    max(src0[i],src1[i]);
    }
}
```

In diesem Beispiel wird der Wert von (1 < < i) & src2 (eine um i nach links verschobene 1 UND-verknüpft mit src2) verwendet, um entweder das Minimum des i-ten Datenelements in src0 und src1 oder das Maximum des i-ten Datenelements in src0 und src1 auszuwählen. Die Operation für das i-te Datenelement wird nur durchgeführt, wenn der Ausführungsmaskenwert, der mit dem aktuellen SIMD-Datenelement (execMask[i])) assoziiert ist, auf 1 gesetzt ist.

Shuffle-Index-Anweisung

[0362] Eine Shuffle-Index-Anweisung kann einen beliebigen Satz von Eingabespuren auf die Ausgabespuren kopieren. Für eine SIMD-Breite von 32 kann diese Anweisung mit einem niedrigeren Durchsatz ausgeführt werden. Diese Anweisung hat die folgende Form: shuffle_index dest src0 src1 <optional Flag>, wobei src0 N Doppelwörter speichert, src1 N vorzeichenlose Bytes (d. h. den Indexwert) speichert und dest N Doppelwörter speichert.

[0363] Die folgende Codesequenz definiert eine besondere Implementierung der Shuffle-Index-Anweisung:

```
  for (int i = 0; i < SIMD_WIDTH) {

   uint8_t srcLane = srcl.index[i];

   if (execMask[i]) {
```

```
  bool invalidLane = srcLane < 0 || srcLane >= SIMD_WIDTH ||
!execMask[srcLaneMod];

  if (FLAG) {
   invalidLane |= flag[srcLaneMod];
  }

  if (invalidLane) {
   dst[i] = src0[i];
  }
  else {
   dst[i] = src0[srcLane];
  }
 }
}
```

[0364] In dem obigen Code identifiziert der Index in src1 die aktuelle Spur. Falls der i-te Wert in der Ausführungsmaske auf 1 gesetzt ist, dann wird eine Prüfung durchgeführt, um sicherzustellen, dass sich die Quellspur innerhalb des Bereichs von 0 bis zur SIMD-Breite befindet. Falls ja, wird das Flag gesetzt (srcLaneMod), und das Datenelement i des Ziels wird gleich dem Datenelement i von src0 gesetzt. Falls sich die Spur innerhalb des Bereichs befindet (d. h. gültig ist), wird der Indexwert von src1 (srcLane0) als ein Index in src0 verwendet (dst[i] = src0[srcLane]).

Immediate-Shuffle-Up/Dn/XOR-Anweisung

[0365] Eine Immediate-Shuffle-Anweisung kann Eingabedatenelemente/-spuren basierend auf einem Immediate der Anweisung mischen. Der Immediate kann eine Verschiebung der Eingabespuren um 1, 2, 4, 8 oder 16 Positionen basierend auf dem Wert des Immediate spezifizieren. Optional kann ein zusätzliches Skalarquellregister als Füllwert spezifiziert werden. Wenn der Quellspurindex ungültig ist, wird der Füllwert (falls bereitgestellt) zu dem Datenelementort in dem Ziel gespeichert. Falls kein Füllwert bereitgestellt ist, wird der Datenelementort auf alles 0 gesetzt.

[0366] Ein Flag-Register kann als eine Quellmaske verwendet werden. Falls das Flag-Bit für eine Quellspur auf 1 gesetzt ist, kann die Quellspur als ungültig markiert werden und die Anweisung kann fortfahren.

[0367] Beispiele für verschiedene Implementierungen der Immediate-Shuffle-Anweisung sind Folgende:

shuffle_<up/dn/xor>_<1/2/4/8/16> dest src0 <optional src1> <optional Flag>

shuffle_<up/dn/xor>_<1/2/4/8/16> dest src0 <optional src1> <optional Flag>

Bei dieser Implementierung speichert src0 N Doppelwörter, src1 speichert ein Doppelwort für den Füllwert (falls vorhanden) und dest speichert N Doppelwörter, die das Ergebnis umfassen.

[0368] Die folgende Codesequenz definiert eine besondere Implementierung der Immediate-Shuffle-Anweisung:

```
for (int i = 0; i < SIMD_WIDTH) {
 int8_t srcLane;
 switch(SHUFFLE_TYPE) {
 case UP:
  srcLane = i - SHIFT;
 case DN:
  srcLane = i + SHIFT;
 case XOR:
  srcLane = i ^ SHIFT;
 }

 if (execMask[i]) {
  bool invalidLane = srcLane < 0 || srcLane >= SIMD_WIDTH ||
!execMask[srcLane];
```

```
    if (FLAG) {
     invalidLane |= flag[srcLane];
    }

    if (invalidLane) {
     if (SRC1)
      dst[i] = src1;
     else
      dst[i] = 0;
    }
    else {
     dst[i] = src0[srcLane];
    }
   }
  }
```

[0369] Hier werden die Eingabedatenelemente/-spuren um 1, 2, 4, 8 oder 16 Positionen basierend auf dem Wert des Immediate verschoben. Das Register src1 ist ein zusätzliches Skalarquellregister, das als Füllwert verwendet wird, der an dem Datenelementort im Ziel gespeichert wird, wenn der Quellspurindex ungültig ist. Falls kein Füllwert bereitgestellt wird und der Quellspurindex ungültig ist, wird der Datenelementort im Ziel auf 0en gesetzt. Das Flag-Register (FLAG) wird als Quellmaske verwendet. Falls das Flag-Bit für eine Quellspur auf 1 gesetzt ist, wird die Quellspur als ungültig markiert und die Anweisung fährt wie oben beschrieben fort.

Indirect-Shuffle-Up/Dn/XOR-Anweisung

[0370] Die Indirect-Shuffle-Anweisung weist einen Quelloperanden (srcl) auf, der die Abbildung von Quellspuren auf Zielspuren steuert. Die Indirect-Shuffle-Anweisung kann folgende Form haben:

shuffle_<up/dn/xor> dest src0 src1 <optional Flag>

wobei src0 N Doppelwörter speichert, src1 1 Doppelwort speichert und dest N Doppelwörter speichert.

[0371] Die folgende Codesequenz definiert eine besondere Implementierung der Immediate-Shuffle-Anweisung:

```
for (int i = 0; i < SIMD_WIDTH) {
 int8_t srcLane;
 switch(SHUFFLE_TYPE) {
 case UP:
  srcLane = i - src1;
 case DN:
  srcLane = i + src1;
 case XOR:
  srcLane = i ^ src1;
}
  if (execMask[i]) {
   bool invalidLane = srcLane < 0 || srcLane >= SIMD_WIDTH ||
 !execMask[srcLane];

   if (FLAG) {
    invalidLane |= flag[srcLane];
   }

   if (invalidLane) {
    dst[i] = 0;
   }
   else {
    dst[i] = src0[srcLane];
   }
```

```
    }
  }
```

[0372] Somit arbeitet die Indirect-Shuffle-Anweisung auf eine ähnliche Weise wie die oben beschriebene Immediate-Shuffle-Anweisung, aber die Abbildung von Quellspuren auf Zielspuren wird durch das Quellregister src1 anstatt durch den Immediate gesteuert.

Cross-Lane-Min/Max-Anweisung

[0373] Eine Cross-Lane-Minimum/Maximum-Anweisung kann für Gleitkomma- und Ganzzahldatentypen unterstützt werden. Die Cross-Lane-Minimum-Anweisung kann die Form lane_min dest src0 haben und die Cross-Lane-Maximum-Anweisung kann die Form lane_max dest src0 haben, wobei src0 N Doppelwörter speichert und dest 1 Doppelwort speichert.

[0374] Beispielsweise definiert die folgende Codesequenz eine besondere Implementierung des Cross-Lane-Minimum:

```
dst = src[0];

for (int i = 1; i < SIMD_WIDTH) {


  if (execMask[i]) {
   dst = min(dst, src[i]);
  }
}
```

In diesem Beispiel wird der Doppelwortwert in der Datenelementposition i des Quellregisters mit dem Datenelement in dem Zielregister verglichen und das Minimum der beiden Werte wird in das Zielregister kopiert. Die Cross-Lane-Maximum-Anweisung arbeitet im Wesentlichen auf die gleiche Weise, wobei der einzige Unterschied darin besteht, dass das Maximum des Datenelements in Position i und des Zielwerts ausgewählt wird.

Cross-Lane-Min/Max-Index-Anweisung

[0375] Eine Cross-Lane-Minimum-Index-Anweisung kann die Form lane_min_index dest src0 haben und die Cross-Lane-Maximum-Index-Anweisung kann die Form lane_max_index dest src0 haben, wobei src0 N Doppelwörter speichert und dest 1 Doppelwort speichert.

[0376] Beispielsweise definiert die folgende Codesequenz eine besondere Implementierung der Cross-Lane-Minimum-Index-Anweisung:

```
dst_index = 0;
tmp = src[0]

for (int i = 1; i < SIMD_WIDTH) {

 if (src[i] < tmp && execMask[i])
 {
   tmp = src[i];
   dst_index = i;
 }
}
```

In diesem Beispiel wird der Zielindex von 0 auf eine SIMD-Breite inkrementiert, was das Zielregister überspannt. Falls das Ausführungsmaskenbit gesetzt ist, dann wird das Datenelement an Position i in dem Quellregister zu einem temporären Speicherort (tmp) kopiert und der Zielindex wird auf Datenelementposition i gesetzt.

Cross-Lane-Sorting-Network-Anweisung

[0377] Eine Cross-Lane-Sorting-Network-Anweisung kann alle N Eingabeelemente unter Verwendung eines N-breiten (stabilen) Sortiernetzwerks sortieren, entweder in aufsteigender Reihenfolge (sortnet_min) oder in absteigender Reihenfolge (sortnet_max). Die Min/Max-Versionen der Anweisung können die Formen sortnet_min dest src0 bzw. sortnet_max dest src0 haben. In einer Implementierung speichern src0 und dest N Doppelwörter. Die Min/Max-Sortierung wird an den N Doppelwörtern von src0 durchgeführt und die aufsteigend geordneten Elemente (für min) oder absteigend geordneten Elemente (für max) werden in ihren jeweiligen sortierten Reihenfolgen in dest gespeichert. Ein Beispiel für eine die Anweisung definierende Code-Sequenz ist: dst = apply_N wide sorting_network_min/max(src0).

Cross-Lane-Sorting-Network-Index-Anweisung

[0378] Eine Cross-Lane-Sorting-Network-Index-Anweisung kann alle N Eingabeelemente unter Verwendung eines N-breiten (stabilen) Sortiernetzwerks sortieren, gibt aber den Permutationsindex zurück, entweder in aufsteigender Reihenfolge (sortnet_min) oder in absteigender Reihenfolge (sortnet_max). Die Min/Max-Versionen der Anweisung können die Formen sortnet_min_index dest src0 und sortnet_max_index dest src0 haben, wobei src0 und dest jeweils N Doppelwörter speichern. Ein Beispiel für eine die Anweisung definierende Codesequenz ist dst = apply_N_wide_sorting_network_min/max_index(src0).

[0379] Ein Verfahren zum Ausführen einer beliebigen der obigen Anweisungen ist in **Fig. 50** veranschaulicht. Das Verfahren kann auf den oben beschriebenen spezifischen Prozessorarchitekturen implementiert werden, ist aber nicht auf irgendeine spezielle Prozessor- oder Systemarchitektur beschränkt.

[0380] Bei 5001 werden Anweisungen eines Primärgrafik-Threads auf Prozessorkernen ausgeführt. Dies kann zum Beispiel einen beliebigen der oben beschriebenen Kerne (z. B. Grafikkerne 3130) beinhalten. Wenn eine Raytracing-Arbeit innerhalb des Primärgrafik-Threads erreicht wird, bestimmt bei 5002, werden die Raytracing-Anweisungen an die Raytracing-Ausführungsschaltungsanordnung ausgelagert, die in der Form einer Funktionseinheit (FU) vorliegen kann, wie oben mit Bezug auf **Fig. 49** beschrieben, oder die sich in einem dedizierten Raytracing-Kern 3150 befinden kann, wie mit Bezug auf **Fig. 31** beschrieben.

[0381] Bei 5003 werden die Raytracing-Anweisungen decodiert und aus dem Speicher abgerufen und bei 5005 werden die Anweisungen in ausführbare Operationen decodiert (z. B. in einer Ausführungsform, die einen Decodierer erfordert). Bei 5004 werden die Raytracing-Anweisungen zur Ausführung durch eine Raytracing-Schaltungsanordnung geplant und versendet. Bei 5005 werden die Raytracing-Anweisungen durch die Raytracing-Schaltungsanordnung ausgeführt. Die Anweisungen können zum Beispiel auf den oben beschriebenen FUs (z. B. Vektor-FU 4910, Raytracing-FU 4912 usw.) und/oder den Grafikkernen 3130 oder Raytracing-Kernen 3150 versendet und ausgeführt werden.

[0382] Wenn die Ausführung für eine Raytracing-Anweisung abgeschlossen ist, werden die Ergebnisse bei 5006 gespeichert (z. B. zurück in den Speicher 3198 gespeichert) und bei 5007 wird der Primärgrafik-Thread benachrichtigt. Bei 5008 werden die Raytracing-Ergebnisse innerhalb des Kontexts des Primär-Threads verarbeitet (z. B. aus dem Speicher gelesen und in Grafikrendergebnisse integriert).

[0383] In Ausführungsformen kann sich der Begriff „Engine“ oder „Modul“ oder „Logik“ auf eine anwendungsspezifische integrierte Schaltung (ASIC), eine elektronische Schaltung, einen Prozessor (gemeinsam genutzt, dediziert oder als Gruppe) und/oder einen Speicher (gemeinsam genutzt, dediziert oder als Gruppe) beziehen, Teil davon sein oder solches beinhalten, der/die ein oder mehrere Software- oder Firmware-Programme, eine kombinatorische Logikschaltung und/oder andere geeignete Komponenten, die die beschriebene Funktionalität bereitstellen, ausführen. In Ausführungsformen kann eine Engine, ein Modul oder eine Logik in Firmware, Hardware, Software oder einer beliebigen Kombination von Firmware, Hardware und Software implementiert sein.

EINRICHTUNG UND VERFAHREN ZUM ASYNCHRONEN RAYTRACING

[0384] Ausführungsformen der Erfindung beinhalten eine Kombination einer Festfunktionsbeschleunigungsschaltungsanordnung und einer Mehrzweckverarbeitungsschaltungsanordnung zum Durchführen eines Raytracing. Beispielsweise können bestimmte Operationen in Bezug auf die Strahltraversierung einer Hüllkörperhierarchie (BVH) und Überschneidungstests durch die Festfunktionsbeschleunigungsschaltungsanordnung durchgeführt werden, während mehrere Ausführungsschaltungen verschiedene Formen von Raytracing-Sha-

dern (z. B. Beliebiger-Treffer-Shader, Überschneidungs-Shader, Fehltreffer-Shader usw.) ausführen. Eine Ausführungsform beinhaltet duale Speicherbänke mit hoher Bandbreite, die mehrere Einträge zum Speichern von Strahlen und entsprechende duale Stapel zum Speichern von BVH-Knoten umfassen. In dieser Ausführungsform wechselt die Traversierungsschaltungsanordnung zwischen den dualen Strahlbänken und Stapeln, um einen Strahl bei jedem Taktzyklus zu verarbeiten. Außerdem beinhaltet eine Ausführungsform eine Prioritätsauswahlschaltungsanordnung/-logik, die zwischen internen Knoten, nichtinternen Knoten und Primitiven unterscheidet und diese Informationen verwendet, um die Verarbeitung der BVH-Knoten und der durch die BVH-Knoten begrenzten Primitive intelligent zu priorisieren.

[0385] Eine bestimmte Ausführungsform reduziert den Hochgeschwindigkeitsspeicher, der für die Traversierung benötigt wird, unter Verwendung eines kurzen Stapels, um eine begrenzte Anzahl von BVH-Knoten während Traversierungsoperationen zu speichern. Diese Ausführungsform beinhaltet eine Stapelverwaltungsschaltungsanordnung/-logik zum effizienten Durchführen von Push- und Pop-Vorgängen von Einträgen zu und von dem kurzen Stapel, um sicherzustellen, dass die erforderlichen BVH-Knoten verfügbar sind. Außerdem werden Traversierungsoperationen verfolgt, indem Aktualisierungen an einer Verfolgungsdatenstruktur durchgeführt werden. Wenn die Traversierungsschaltungsanordnung/-logik pausiert wird, kann sie die Verfolgungsdatenstruktur heranziehen, um Traversierungsoperationen an demselben Ort innerhalb der BVH zu beginnen, an dem sie zuletzt aktiv war, und das Verfolgen von in einer Datenverfolgungsstruktur gehaltenen Daten wird durchgeführt, sodass die Traversierungsschaltungsanordnung/-logik neu starten kann.

[0386] Fig. 51 veranschaulicht eine Ausführungsform, die eine Shader-Ausführungsschaltungsanordnung 4000 zum Ausführen von Shader-Programmcode und Verarbeiten von assoziierten Raytracing-Daten 4902 (z. B. BVH-Knotendaten und Strahldaten), eine Raytracing-Beschleunigungsschaltungsanordnung 5110 zum Durchführen von Traversierungs- und Überschneidungsoperationen und einen Speicher 3198 zum Speichern von Programmcode und assoziierten Daten, die durch die RT-Beschleunigungsschaltungsanordnung 5110 und die Shader-Ausführungsschaltungsanordnung 4000 verarbeitet werden, umfasst.

[0387] In einer Ausführungsform beinhaltet die Shader-Ausführungsschaltungsanordnung 4000 mehrere Kerne/Ausführungseinheiten 4001, die Shader-Programmcode ausführen, um verschiedene Formen von datenparallelen Operationen auszuführen. In einer Ausführungsform können die Kerne/Ausführungseinheiten 4001 zum Beispiel eine einzige Anweisung über mehrere Spuren ausführen, wobei jede Instanz der Anweisung an Daten arbeitet, die in einer anderen Spur gespeichert sind. Bei einer SIMT-Implementierung ist zum Beispiel jede Instanz der Anweisung mit einem anderen Thread assoziiert. Während der Ausführung speichert ein L1-Cache gewisse Raytracing-Daten für effizienten Zugriff (z. B. Daten, auf die kürzlich oder häufig zugegriffen wurde).

[0388] Ein Satz von Primärstrahlen kann an den Scheduler 4007 versendet werden, der Arbeit für Shader plant, die durch die Kerne/EUs 4001 ausgeführt werden. Die Kerne/EUs 4001 können Raytracing-Kerne 3150, Grafikkerne 3130, CPU-Kerne 3199 oder andere Arten von Schaltungsanordnungen sein, die in der Lage sind, Shader-Programmcode auszuführen. Ein oder mehrere Primärstrahl-Shader 5101 verarbeiten die Primärstrahlen und spawnen zusätzliche Arbeit, die durch die Raytracing-Beschleunigungsschaltungsanordnung 5110 und/oder die Kerne/EUs 4001 durchzuführen sind (z. B. die durch einen oder mehrere Child-Shader auszuführen sind). Neue Arbeit, die durch den Primärstrahl-Shader 5101 oder andere Shader gespawnt wird, die durch die Kerne/EUs 4001 ausgeführt werden, kann an eine Sortierungsschaltungsanordnung 4008 verteilt werden, die die Strahlen in Gruppen oder Bins sortiert, wie hierin beschrieben (z. B. Gruppieren von Strahlen mit ähnlichen Charakteristiken). Der Scheduler 4007 plant dann die neue Arbeit auf den Kernen/EUs 4001.

[0389] Andere Shader, die ausgeführt werden können, beinhalten Beliebiger-Treffer-Shader 4514 und Nächstgelegener-Treffer-Shader 4507, die Trefferergebnisse verarbeiten, wie oben beschrieben (z. B. Identifizieren eines beliebigen Treffers bzw. des nächstgelegenen Treffers für einen gegebenen Strahl). Ein Fehltreffer-Shader 4506 verarbeitet Strahlenfehltreffer (z. B. wenn ein Strahl den Knoten/das Primitiv nicht schneidet). Wie erwähnt, können die verschiedenen Shader unter Verwendung einer Shader-Aufzeichnung referenziert werden, der einen oder mehrere Zeiger, herstellerspezifische Metadaten und globale Argumente beinhalten kann. In einer Ausführungsform werden Shader-Aufzeichnungen durch Shader-Aufzeichnung-Identifikatoren (SRI: Shader Record Identifiers) identifiziert. In einer Ausführungsform ist jede ausführende Instanz eines Shaders mit einem Aufrufstapel 5203 assoziiert, der Argumente speichert, die zwischen einem Parent-Shader und einem Child-Shader weitergeleitet werden. Aufrufstapel 5121 können auch Referenzen zu Fortsetzungsfunktionen speichern, die ausgeführt werden, wenn ein Aufruf zurückkehrt.

[0390] Die Strahltraversierungsschaltungsanordnung 5102 traversiert jeden Strahl durch Knoten einer BVH, wobei die Hierarchie der BVH (z. B. durch Parent-Knoten, Child-Knoten und Leaf-Knoten) herunterbearbeitet wird, um Knoten/Primitive zu identifizieren, die durch den Strahl traversiert werden. Die Strahl-BVH-Überschneidungsschaltungsanordnung 5103 führt Überschneidungstests von Strahlen durch, wobei sie Trefferpunkte auf Primitiven bestimmt, und erzeugt Ergebnisse als Reaktion auf die Treffer. Die Traversierungsschaltungsanordnung 5102 und die Überschneidungsschaltungsanordnung 5103 können Arbeit aus dem einen oder den mehreren Aufrufstapeln 5121 abrufen. Innerhalb der Raytracing-Beschleunigungsschaltungsanordnung 5110 können Aufrufstapel 5121 und assoziierte Raytracing-Daten 4902 innerhalb eines lokalen Raytracing-Cache (RTC) 5107 oder einer anderen lokalen Speicherungsvorrichtung für einen effizienten Zugriff durch die Traversierungsschaltungsanordnung 5102 und die Überschneidungsschaltungsanordnung 5103 gespeichert werden. Eine spezielle Ausführungsform, die unten beschrieben ist, beinhaltet Strahlenbänke mit hoher Bandbreite (siehe z. B. **Fig. 52A**).

[0391] Die Raytracing-Beschleunigungsschaltungsanordnung 5110 kann eine Variante der verschiedenen hierin beschriebenen Traversierungs-/Überschneidungsschaltungen sein, darunter die Strahl-BVH-Traversierungs-/-Überschneidungsschaltung 4005, die Traversierungsschaltung 4502 und die Überschneidungsschaltung 4503 sowie die Raytracing-Kerne 3150. Die Raytracing-Beschleunigungsschaltungsanordnung 5110 kann anstelle der Strahl-BVH-Traversierungs-/-Überschneidungsschaltung 4005, der Traversierungsschaltung 4502 und der Überschneidungsschaltung 4503 sowie der Raytracing-Kerne 3150 oder einer beliebigen anderen Schaltungsanordnung/Logik zum Verarbeiten von BVH-Stapeln und/oder Durchführen einer Traversierung/Überschneidung verwendet werden. Daher wird durch die Offenbarung beliebiger Merkmale in Kombination mit der Strahl-BVH-Traversierungs-/-Überschneidungsschaltung 4005, der Traversierungsschaltung 4502 und der Überschneidungsschaltung 4503 sowie den Raytracing-Kernen 3150, die hierin beschrieben sind, auch eine entsprechende Kombination mit der Raytracing-Beschleunigungsschaltungsanordnung 5110 offenbart, ohne darauf beschränkt zu sein.

[0392] Unter Bezugnahme auf **Fig. 52A** beinhaltet eine Ausführungsform der Strahltraversierungsschaltungsanordnung 5102 eine erste und eine zweite Strahlspeicherbank 5201 bzw. 5202, wobei jede Bank mehrere Einträge zum Speichern entsprechender mehrerer eingehender Strahlen 5206, die aus dem Speicher geladen werden, umfasst. Entsprechende erste und zweite Stapel 5203 bzw. 5204 umfassen ausgewählte BVH-Knotendaten 5290-5291, die aus dem Speicher gelesen und lokal zur Verarbeitung gespeichert werden. Wie hierin beschrieben, sind die Stapel 5203-5204 in einer Ausführungsform „kurze“ Stapel, die eine begrenzte Anzahl von Einträgen zum Speichern von BVH-Knotendaten umfassen (z. B. in einer Ausführungsform sechs Einträge). Obwohl sie getrennt von den Strahlbänken 5201-5202 veranschaulicht sind, können die Stapel 5203-5204 auch innerhalb der entsprechenden Strahlbänke 5201-5202 geführt werden. Alternativ können die Stapel 5203-5204 in einem separaten lokalen Speicher oder Cache gespeichert werden.

[0393] Eine Ausführungsform der Traversierungsverarbeitungsschaltungsanordnung 5210 alterniert zwischen den zwei Bänken 5201-5202 und Stapeln 5203-5204, wenn der nächste zu verarbeitende Strahl und Knoten ausgewählt werden (z. B. auf eine Ping-Pong-Weise). Zum Beispiel kann die Traversierungsverarbeitungsschaltungsanordnung 5210 einen neuen Strahl/BVH-Knoten aus einer alternativen Strahlbank/einem alternativen Strahlstapel bei jedem Taktzyklus auswählen, wodurch ein hocheffizienter Betrieb sichergestellt wird. Es sei jedoch darauf hingewiesen, dass diese spezielle Anordnung nicht notwendig ist, um die der Erfindung zugrundeliegenden Prinzipien zu erfüllen.

[0394] In einer Ausführungsform gleicht ein Strahlzuweiser 5205 den Eintritt eingehender Strahlen 5206 in die erste bzw. zweite Speicherbank 5201-5202 basierend auf aktuellen relativen Werten eines Satzes von Bankzuweisungszählern 5220 aus. Bei einer Ausführungsform führen die Bankzuweisungszähler 5220 einen Zählwert der Anzahl von nicht traversierten Strahlen in jeder der ersten und zweiten Speicherbank 5201-5202. Zum Beispiel kann ein erster Bankzuweisungszähler inkrementiert werden, wenn der Strahlzuweiser 5205 einen neuen Strahl zu der ersten Bank 5201 hinzufügt, und dekrementiert werden, wenn ein Strahl von der ersten Bank 5201 verarbeitet wird. Gleichermaßen kann der zweite Bankzuweisungszähler inkrementiert werden, wenn der Strahlzuweiser 5205 einen neuen Strahl zu der zweiten Bank 5201 hinzufügt, und dekrementiert werden, wenn ein Strahl von der zweiten Bank 5201 verarbeitet wird.

[0395] In einer Ausführungsform weist der Strahlzuweiser 5205 einer mit dem kleineren Zählerwert assoziierten Bank den aktuellen Strahl zu. Falls die zwei Zähler gleich sind, kann der Strahlzuweiser 5205 entweder eine der Banken auswählen oder eine andere Bank auswählen als diejenige, die beim letzten Mal ausgewählt wurde, als die Zähler gleich wären. In einer Ausführungsform wird jeder Strahl in einem Eintrag einer

der Bänke 5201-5202 gespeichert und jede Bank umfasst 32 Einträge zum Speichern von bis zu 32 Strahlen. Die der Erfindung zugrunde liegenden Prinzipien sind jedoch nicht auf diese Einzelheiten beschränkt.

[0396] **Fig. 52B** veranschaulicht vier Prozesse 5251-5254, die in einer Ausführungsform ausgeführt werden, um die Strahlspeicherbänke 5201-5202 und Stapel 5203-5204 zu verwalten. In einer Ausführungsform sind die vier Prozesse 5251-5254 unterschiedliche Implementierungen oder Konfigurationen eines gemeinsamen Satzes von Programmcode (hierin manchmal als „TraceRay“ bezeichnet). Der anfängliche Prozess 5251 kann ausgeführt werden, um den Strahl 5261 zu lesen und eine neue Top-Down-Traversierung einer BVH ausgehend von dem Root-Knoten durchzuführen. Die Alloc(Zuweisung)-Funktion modifiziert Steuerbits und startet entsprechende Leseanforderungen an den Raytracing-Stapel. Um den neuen Eintrag zuzuweisen, setzt Alloc insbesondere das Gültig(VLD)-Bit und setzt das Räumbereitschafts(Evict_Rdy)-Bit zurück. In dem Bankeintrag für den Strahl werden das Datenpräsenz(DP)-Bit und das Dirty-Bit zurückgesetzt. Das DP-Bit in dem entsprechenden Stapeleintrag wird gesetzt. Für die entsprechende Hitinfo wird das DP-Bit gesetzt und das Dirty-Bit zurückgesetzt. Das DP-Bit und das Shader-Aufzeichnungskennungs(SRI)-DP-Bit, die mit den Knotendaten assoziiert sind, werden zurückgesetzt.

[0397] Der Instanzprozess 5252 führt eine Traversierung innerhalb eines der Knoten der BVH (außer dem Root-Knoten) durch und liest den Strahl und den zuvor festgelegten Treffer 5262. In einer Ausführungsform, wenn einer der Treffer-Shader einen Treffer zwischen dem Strahl und einem Primitiv identifiziert, wird der Festlegungsprozess 5253 ausgeführt, um Ergebnisse festzulegen, wobei der Strahl, der potenzielle Treffer und der Stapel 5263 gelesen werden. Alternativ dazu wird der Fortfahrprozess 5254 ausgeführt, um das Traversieren des Strahls fortzusetzen, wobei der Strahl, der festgelegte Treffer und der Stapel 5264 gelesen werden.

[0398] Unter verschiedenen Umständen muss die Traversierungsschaltungsanordnung 5002 Traversierungsoperationen pausieren und den aktuellen Strahl und assoziierte BVH-Knoten speichern, etwa wenn ein Shader eine Sequenz von Operationen durchführen muss. Falls zum Beispiel ein nicht-opakes Objekt getroffen wird oder eine prozedurale Textur, speichert die Traversierungsschaltungsanordnung 5002 den Stapel 5203-5204 im Speicher und führt den erforderlichen Shader aus. Sobald der Shader die Verarbeitung des Treffers (oder anderer Daten) abgeschlossen hat, stellt die Traversierungsschaltungsanordnung 5002 den Zustand der Strahlbänke 5201-5202 und Stapel 5203-5204 aus dem Speicher wieder her.

[0399] In einer Ausführungsform überwacht ein Traversierungs-/Stapelverfolger 5248 kontinuierlich Traversierungs- und Stapeloperationen und speichert Neustartdaten in einem Verfolgungsarray 5249. Falls zum Beispiel die Traversierungsschaltungsanordnung 5002 bereits die Knoten N, N0, N1, N2 und N00 traversiert hat und Ergebnisse erzeugt hat, dann aktualisiert der Traversierungs-/Stapelverfolger 5248 das Verfolgungsarray, um anzugeben, dass das Traversieren dieser Knoten abgeschlossen ist, und/oder um den nächsten zu verarbeitenden Knoten aus dem Stapel anzugeben. Wenn die Traversierungsschaltungsanordnung 5002 neu gestartet wird, liest sie die Neustartdaten aus dem Verfolgungsarray 5249, sodass sie die Traversierung in der korrekten Phase neu starten kann, ohne BVH-Knoten erneut zu traversieren (und Zyklen zu verschwenden). Die in dem Verfolgungsarray 5249 gespeicherten Neustartdaten werden manchmal als der „Neustartpfad“ oder „RST“ bezeichnet.

[0400] Wie in **Fig. 52B** angegeben, verwalten die verschiedenen TraceRay-Prozesse 5251-5254 die Zuweisung in die und aus den Strahlspeicherbänken 5201-5202 über eine oder mehrere Funktionen. Wie für den anfänglichen Prozess 5251 veranschaulicht, setzt eine Alloc-Funktion das gültige Bit (VLD) in einem Speicherbankeintrag (wodurch angegeben wird, dass der Eintrag nun einen gültigen Strahl enthält) und setzt das Räumungsbereitschafts-Flag zurück (Rst) (wodurch angegeben wird, dass die Strahldaten nicht geräumt werden sollten). Die Ray-Funktion speichert den Strahl in dem ausgewählten Eintrag und setzt das Datenpräsenz(DP)-Bit (wodurch angegeben wird, dass Strahldaten in dem Eintrag gespeichert sind) und das Dirty-Bit (wodurch angegeben wird, dass die Daten nicht modifiziert wurden) zurück. Beim Lesen des Strahls aus der Speicherbank setzt die Stapelfunktion das DP-Bit und ruft den relevanten BVH-Knoten aus dem Stapel ab (z. B. den Root-Knoten im Fall des anfänglichen Prozesses 5251 und einen anderen Knoten im Fall des Instanzprozesses 5252). Die Hitinfo-Funktion setzt das Dirty-Bit zurück und setzt das DP-Bit für die Anfänglich-Funktion 5251 oder setzt es für alle anderen Funktionen zurück. In einer Ausführungsform produziert Hitinfo Daten, die einen Strahltreffer wiedergeben. Die Knotenfunktion setzt das DP-Bit und das SRI(Shader-Aufzeichnungskennung)-DP, wobei es sich um das DP für die Shader-Aufzeichnungskennung handelt, zurück. Eine Ausführungsform führt eine Kernelstartzeiger(KSP: Kernel Start Pointer)-Suche durch, um sicherzustellen, dass KSP ungleich null ist. Falls dies der Fall ist, wird eine andere Handhabung für nicht-opake Vierecke implementiert.

[0401] In einer Ausführungsform wird, sobald ein Strahleintrag in einer der Speicherbänke 5201-5202 zugewiesen wurde, ein Abruf durchgeführt, um die Knotendaten (und potenziell andere Daten) aus dem Stapel, der mit dem Strahl assoziiert ist, abzurufen. In einer Ausführungsform wird ein Stapel für jeden Strahl geführt, der den Arbeitsdatensatz für den aktuellen Knoten umfasst, durch den der Strahl traversiert wird.

[0402] Bei Fortschreiten zur nächsten Ebene in der BVH (z. B. wenn bestimmt wird, dass der Strahl einen Parent-Knoten schneidet), werden die Child-Knoten sortiert und auf den Stapel 5203-5204 gepusht. Die Child-Knoten werden sequenziell von dem Stapel abgerufen und verarbeitet, um Child-Knoten zu identifizieren, die der Strahl traversiert (Traversierungs-„Treffer“). In einer Ausführungsform wird der Stapel immer dann in einem Speicher oder einem lokalen Cache/Speicher gespeichert, wenn es einen Handoff zwischen der RT-Beschleunigungsschaltungsanordnung 5110 und den Shadern 4504, 4506, 4507, 5101, 5105 gibt.

[0403] Wenn ein Leaf-Knoten, der ein Viereck oder Dreieck (oder einen anderen Primitivtyp) umfasst, durch die Traversierungsschaltungsanordnung 5102 identifiziert wird, gibt er diese Informationen an die Überschneidungsschaltungsanordnung 5103 weiter, die einen Überschneidungstest an dem Viereck bzw. Dreieck durchführt. Falls das Primitiv kein Viereck oder Dreieck ist, beendet die Traversierungsschaltungsanordnung in einer Implementierung die Traversierung und gibt die Steuerung an den Nächstgelegenen-Treffer-Shader 4507 (falls ein Treffer detektiert wird) oder den Fehltreffer-Shader 4506 (falls kein Treffer detektiert wird) zurück. Bei einer Implementierung, bei der die Überschneidungsschaltungsanordnung 5103 dazu ausgelegt ist, Überschneidungen für eine Vielzahl von Primitiven zusätzlich zu Vierecken und Dreiecken (z. B. Linien, Bögen, Kreise usw.) durchzuführen, dann wird die Traversierschaltungsanordnung 5102 Leaf-Knoten für diese Primitive an die Überschneidungsschaltungsanordnung 5103 weiterleiten.

[0404] In einer Ausführungsform wird, wenn eine Hardware- oder Softwarekomponente eine Leseanforderung an den Speicher 3198 oder den Cache erzeugt, ein 16-Bit-Tag verwendet, um Informationen über den Datentyp und den Anfordernden bereitzustellen. Zum Beispiel kann ein Zweibitcode spezifizieren, ob die Anforderung für einen Strahl, Stapeldaten, Trefferdaten, Knotendaten aus der BVH oder einen beliebigen anderen Datentyp ist. Wenn der Strahl, Stapel und Hitinfo aus dem Speicher zurückgegeben wurden, wird der Strahl durch einen oder mehrere BVH-Knoten traversiert und Überschneidungstests wie oben beschrieben durchgeführt.

[0405] Ein oder mehrere Stapel 5203-5204 und Strahlen 5206 werden bei unterschiedlichen Verarbeitungsstufen von dem Speicher geladen. Zum Beispiel kann der anfängliche Prozess 5251 und/oder der Instanzprozess 5252 erfordern, dass eine neue BVH zur Traversierung geladen werden soll. Unter diesen Umständen kann der Stapel 5203-5204 auf den oberen Knoten (oder „Root“-Knoten) der BVH initialisiert werden. Für eine Strahlfortsetzung 5254 innerhalb einer BVH kann der Stapel 5203-5204 von dem Speicher geladen und erweitert werden. Sobald der Stapel 5203-5204 vorbereitet wurde, werden Knotendaten aus dem Stapel abgerufen (eine Operation, die nachstehend mitunter als Proc_Node_Fetch bezeichnet wird).

[0406] In einer Ausführungsform werden Knotendaten abgerufen, indem parallele Anforderungen für zwei nichtinnere (NI-)Knoten und zwei innere Knoten gestartet werden. **Fig. 53** veranschaulicht eine solche Ausführungsform, bei der eine NI-Knoten-Prioritätsauswahllogik (PRISEL: PRIority SELection) 5311 duale NI-Knoten anfordert: einen ersten NI-Knoten 5301 von Bank 0 und einen zweiten NI-Knoten 5302 von Bank 1. Gleichzeitig fordert die Innerer-Knoten-PRISEL-Logik 5312 duale innere Knoten an: einen ersten Knoten 5303 von Bank 0 und einen zweiten Knoten 5304 von Bank 1.

[0407] In einer Ausführungsform priorisiert die NI-Knoten-Prioritätsauswahllogik (PRISEL) 5311 den ersten NI-Knoten 5301 oder den zweiten NI-Knoten 5302, wobei das priorisierte Ergebnis in dem Raytracing-Cache (RTC) gespeichert wird. Gleichermaßen fordert die Innerer-Knoten-PRISEL-Logik 5312 duale innere Knoten an und wählt ein priorisiertes Ergebnis aus einem ersten inneren Knoten 5303 und einem zweiten inneren Knoten 5304 aus.

[0408] Jede Instanz der Prioritätsauswahllogik 5311-5312 priorisiert einen der nichtinneren BVH-Knoten 5301-5302 und einen der inneren BVH-Knoten 5303-5304 aus einer anderen Bank, falls möglich. In einer Ausführungsform wird nur eine Anforderung aus jeder Bank ausgewählt (z. B. eine der Anforderungen 5302 und 5304 und eine der Anforderungen 5301 und 5303). Das Starten dieser Anforderungen kann auch das Stapeldatenpräsenz(DP)-Bit zurücksetzen, wie angegeben, sodass dieser Eintrag als Reaktion auf eine Knotenabrufoperation nicht abgerufen wird. In einer Ausführungsform wird für die Instanzaufrufoperation das Datenpräsenz(DP)-Bit des Strahls zurückgesetzt, wenn die Instanzanforderung gesendet wird, und schließlich gesetzt, wenn der Strahl nach dem Knotenabruf transformiert wird.

[0409] In einer Ausführungsform wird node_info beim Start von Lesevorgängen geschrieben und die Adresse/das Tag wird wie folgt für die Leseanforderungen berechnet:

i. rtt_rtc_rd_addr[47:6] = rt_ray.rt_ray_ctrl.root_node_ptr[47:6] + curr_stack.child_offset; (Hinweis: Der Child-Offset auf dem Knoten ist immer gegenüber aktuellem BVH-Root-Knoten)

ii. rtt_rtc_rd_tag[6:0] = {RTT_INST, rtt_alloc_entry[5:0]};

iii. node. node_info = curr_stack.node_info.

[0410] In einer Ausführungsform setzen die zurückgegebenen Knotendaten das DP-Bit für den Knoten und den Stapel.

[0411] Die folgenden Fälle können basierend auf dem gelesenen Tag unterschieden werden:

A. Innerer Knoten: Dies schreibt in den Knoten

B. Instanz: Dies aktualisiert rt_ray.rt_ray_ctrl für BVH (1) der nächsten Ebene und schreibt die Knotenstruktur.

i. root_node_ptr = node_return.StartNodePtr

ii. hitgrp_srbase_ptr = rt_ray_ctrl.hitgrp_srbase_ptr + rt_ray_ctrl.srstride*node_return.instancecontributiontohitgrpindex

iii. hitgrp_sr_stride = rt_ray_ctrl.srstride* rt_ray_ctrl.shade_indx_mult

iv. inst_leaf_ptr = rt_ray.rt_ray_ctrl.root_node_ptr + stack.current_node.child_offset → Nur Logisches Einsehen, Einholen und Speichern der Knotenabrufadresse während der Instanzknoten-Abrufanforderung selbst

v. {miss_sr_ptr, shader_indx_mult, mask} = {rt_ray[0].rt_ray_ctrl.miss_sr_ptr, rt_ray[0]. rt_ray_ctrl. shader_indx_mult, rt_ray[0].rt_ray_ctr!.mask} □ Preserve BVH[0]

vi. flag[0] = rt_ray[0].rt_ray_ctrl.flag[0] | (~rt_ray[0].rt_ray_ctr|.flag[1] & Node_Return.flag[2]); → Opaque entweder über Strahl- oder Instanz-Flag bewahren (nur wenn Strahl-Flag nicht zwangsweise nicht-opak ist)

vii. flag[1] = (rt_ray[0].rt_ray_ctrl.flag[1]) | (~rt_ray[0].rt_ray_ctr|.flag[0] & Node_Return.flag[3]); → Non-Opaque entweder über Strahl- oder Instanz-Flag bewahren (nur wenn Strahl-Flag nicht zwangsweise opak ist)

viii. flag[3:2] = rt_ray[0].rt_ray_ctrl.flag[3:2]; → (ERSTEN TREFFER annehmen und Suche beenden oder Nächstgelegener-Treffer-Shader überspringen) BVH[0] bewahren

ix. flag[5:4] = Node_Return.flag[0] ? 2'd0: rt_ray[0].rt_ray_ctrl.flag[5:4];->Dreiecks-Culling wird über Instanz deaktiviert

x. flag[8:6] = rt_ray[0].rt_ray_ctrl.flag[8:6]; → (Überschneidungs-Shader deaktivieren, Opaque aussortieren oder Non-Opaque aussortieren) BVH[0] bewahren

xi. node.node_ctrl = Für Instanz nicht erforderlich

xii. node.node_data = {'0, node_rtn.obj2world_p, world2obj_vzyx};

C. Viereck: Dies aktualisiert den Knoten wie folgt:

i. node.node_ctrl = {node_rtn.leafDesc.last, node_rtn.leafDesc.PrimIndex1Delta[15:0], node_rtn.leafDesc.PrimIndex0[31:0], node_rtn.shader_indx};

ii. node.node_data = {'0, Quad_mode, J[2:0], V[3:0]}; → Quad_mode = node_rtn.leafDesc.PrimIndexlDelta[15:0] != '0;

[0412] Basierend auf dem Strahl-Flag, Instanz-Flag und dem Geometrie-Flag zeigt die in **Fig. 55A** gezeigte Opak-/Nicht-Opak-Handhabungstabelle das resultierende Flag an, das verwendet werden soll, wenn die Knotendaten aufgerufen werden (Opaque oder Non-Opaque). Wie in der Tabelle angegeben, haben Strahl-Flags immer Vorrang. Zudem schließen sich manche der Zustände gegenseitig aus. In einer Ausführungsform werden diese in Hardware mit der Priorität exklusiver Bits gehandhabt. In einer Implementierung wird, falls cull_opaque und force_opaque beide gesetzt sind, die zugehörige Geometrie automatisch aussortiert.

opaque = rt_ray.rt_ray_ctrl.flag[0] | quad.flag[0]; (Man beachte, dass der pro BVH-Ebene gespeicherte Strahl bereits die Instanz-Flags berücksichtigt)

nopaque = rt_ray.rt_ray_ctrl.flag[1] | ~quad.flag[0];

[0413] Fig. 55B ist eine Tabelle, die eine Strahl-Flag-Handhabung und Ausnahmen gemäß einer Ausführungsform zeigt. Hier basiert die Entscheidung zum Culling auf einer Kombination des Strahl-Flags, des Instanz-Flags und des Geometrie-Flags.

cull_opaque = rt_ray.rt_ray_ctrl.flag[6] & (rt_ray.rt_ray_ctrl.flag[0] | quad.flag[0]);

cull_nopaque = rt_ray.rt_ray_ctrl.flag[7] & (rt_ray.rt_ray_ctrl.flag[1] | ~quad.flag[0]);

cull = cull_opaque | cull_nopaque;

[0414] Ein maskenbasiertes Culling kann in einer Ausführungsform wie folgt implementiert werden:

mask_kill = ~|(rtc_rtt_rd_rtn.mask & rtc_rtt_rd_rtn.data.mask);

[0415] Fig. 55C ist eine Tabelle, die abschließendes Culling gemäß einer Ausführungsform zeigt. Die Strahl-Flags, die (cull_opaque und force_opaque) oder (cull_non_opaque und force_non_opaque) sind, schließen sich gegenseitig aus. In dieser Gleichung berücksichtigt das Strahl-Flag jedoch auch das Instanz-Flag, das opaque/non-opaque setzen kann. Nur Geometrie kann aussortiert werden, während sowohl Instanz als auch Geometrie maskiert werden können.

[0416] Wie in **Fig. 56** veranschaulicht, wird in einer Ausführungsform basierend auf der Evaluierung der oben beschriebenen cull- und mask_kill-Einstellungen bei 5601 oder 5602 Early Out bestimmt und das Ergebnis entweder bei 5603 an die Knotenspeicherung und/oder bei 5604 an den Stapel gesendet.

[0417] Sobald die Knotendaten bereit sind, können Rahmen-/Überschneidungstests durchgeführt werden. Dies wird bei einer Ausführungsform durch einen Prozess erreicht, der hierin als Ray_Test_Proc bezeichnet wird, bei dem zwei gleichzeitige zugrundeliegende Prozesse laufen, einer zum Füllen des Vierecks/der Instanz (QI) und einen anderen zum Durchführen der Rahmen-/Überschneidungstests. In einer in **Fig. 57** veranschaulichten Implementierung startet Ray_Test_Proc zwei parallele Instanzen einer Prioritätsauswahllogik (PRISEL) 5701-5702: eine Viereck/Instanz-PRISEL 5701 zum Anfordern und Wählen zwischen einem Viereck/einer Instanz 5711 von Bank 0 und einem/einer zweiten Viereck/Instanz 5712 von Bank 1, und eine Innerer-Knoten-PRISEL 5702 zum Anfordern und Wählen zwischen einem inneren Knoten aus der Bank 0 5713 und einem inneren Knoten aus der Bank 1 5714.

[0418] In einer Ausführungsform priorisiert die Viereck-/Instanzprioritätsauswahllogik 5701 den ersten QI-Knoten 5711 oder den zweiten QI-Knoten 5712, wobei das priorisierte Ergebnis in der Raytracing-Warteschlange (RTQ: Ray Tracing Queue) zur weiteren Verarbeitung (z. B. Überschneidungstests) gespeichert wird. Gleichermaßen priorisiert die Innerer-Knoten-PRISEL-Logik 5702 einen der inneren BVH-Knoten 5713-5714, an dem ein Raytracing-Traversierungs(RTT)-Rahmentest durchgeführt wird. In einer Ausführungsform wird nur eine Anforderung aus jeder Bank ausgewählt (z. B. eine der Anforderungen 5711 und 5712 und eine der Anforderungen 5713 und 5714). Das Starten dieser Anforderungen kann auch das Stapeldatenpräsenz(DP)-Bit zurücksetzen, wie angegeben, sodass dieser Eintrag als Reaktion auf eine Knotenabrufoperation nicht abgerufen wird. In einer Ausführungsform wird für die Instanzaufrufoperation das Datenpräsenz(DP)-Bit des Strahls zurückgesetzt, wenn die Instanzanforderung gesendet wird, und schließlich gesetzt, wenn der Strahl nach dem Knotenabruf transformiert wird.

[0419] Als Teil dieses Prozesses wird für jeden Viereck-Test-Dispatch, bei dem der Knotentyp nicht-opak ist, der Shader-Aufzeichnungskennung-Null-Lookup als ein Bindless Thread Dispatch (BTD) basierend auf der folgenden Shader-Aufzeichnungskennung-Lookup-Adresse gesendet:

sri_null_lookup_ptr[47:3] = 2*(Ray.hitGroupSRBasePtr + Node.leafDesc.ShaderIndex*ray.SRStride) + 1;

sri_null_lookup_tag[7:0] = {1'd0, RTT_INST, rtt_alloc_entry[5:0]};

[0420] In einer Ausführungsform ist ein Viereck(Quad)/Instanz(QI)-Entkopplungs-FIFO enthalten, um Zeitlicher-Stapel-FIFO-Voll-Bedingungen aufzulösen und synchrone Aktualisierungen an Hitinfo/Ray mit einem Push in das Stapel-FIFO zu implementieren (siehe z. B. Stapel-FIFO 6001 in **Fig. 60**). Dies erfolgt, damit Ray/Hitinfo in nachfolgenden Prozessen ein garantiertes Datenpräsenz(DP)-Bit gesetzt hat. Es sei ange-

merkt, dass Ray/Hitinfo eine feste hohe Priorität zugewiesen werden kann, wenn es mit Speicherschreibvorgängen kollidiert.

[0421] Die Rückgabe von RTQ kann zu einer Instanz (z. B. einer Instanztransformation) oder einem Viereck (d. h. Traversierungs-/Überschneidungstestergebnisse) auf zwei getrennten Schnittstellen führen. Nachstehend finden sich die zwei Rückgabe-FIFOs, die in einer Ausführungsform zum Verarbeiten von Ergebnissen verwendet werden:

a. Instanzrückgabe-FIFO: Update rt_ray.rt_ray_data = rtq_rt_ray_data; ray_dirty[Entry] = 1;

b. Viereck-Rückgabe-FIFO:

i. Falls das Vierecke nicht-opak ist und (T_{far} < P_{rev}_T_{far}) → SRI_NULL_DP prüfen, um einen Pop-(Auslese-)Vorgang am Viereck/Instanz(QI)-Entkopplungs-FIFO durchzuführen. Es ist zu beachten, dass in einer Ausführungsform der Hitinfo-Schreibvorgang aus dem Raytracing-Warteschlangen(RTQ)-FIFO eine höhere Priorität gegenüber MemHitInfo aufweist.

1. Falls (KSP_NULL = 1) → Behandeln des nicht-opaken Vierecks so, als ob es opak wäre, und T_{far} aktualisieren.

2. Falls (KSP_NULL != 1) -7

♦ Schreiben des potenziellen Hitinfo in den Speicher mit auf 1 gesetztem Valid-Bit.

♦ Lesen von T, U, V, Leaf Type, PrimLeafIndex und Front Face aus der RTQ.

♦ Lesen von PrimIndexDelta, PrimleafPtr aus NodeData. instanceLeafPtr aus Ray Data aktualisieren.

♦ hitGroupRecPtr wie oben berechnet

ii. Falls das Viereck nicht-opak ist und (T_{far} < P_{rev}_T_{far}) -7

♦ Aktualisieren des festgelegten Hitinfo mit Valid = 1.

♦ Lesen von T, U, V, Leaf Type, PrimLeafIndex, Front Face aus der RTQ.

♦ Lesen von PrimIndexDelta, PrimleafPtr aus NodeData.

♦ Aktualisieren von instanceLeafPtr aus rt_ray.rt_ray_ctrl

♦ hitGroupRecPtr wie für oben berechnet

[0422] In einer Ausführungsform kann die Rückgabe von dem Raytracing-Traversierungs(RTT)-Rahmenüberschneidungstest zur weiteren Verarbeitung in das Stapel-0/1-(5203/5204)-FIFO 6001 gepusht werden.

[0423] **Fig. 58** und die **Fig. 59A-B** veranschaulichen ein Beispiel für eine BVH-Strahlverarbeitung durch Verwenden eines „kurzen“ Stapels (wie z. B. Stapel 5203 oder 5204, die eine begrenzte Anzahl von lokalen Stapeleinträgen beinhalten). Ein kurzer Stapel wird verwendet, um eine Hochgeschwindigkeitsspeicherung in Kombination mit intelligenten Knotenverwaltungstechniken zu bewahren, um eine hocheffiziente Sequenz von Traversierungsoperationen bereitzustellen. In dem veranschaulichten Beispiel beinhaltet der kurze Stapel 5203 Einträge für sechs BVH-Knoten. Die der Erfindung zugrunde liegenden Prinzipien können jedoch unter Verwendung unterschiedlich großer kurzer Stapel implementiert werden.

[0424] Die Operationen 5949-5972 führen während der BVH-Traversierung Push- und Pop-Vorgänge an Stapeleinträgen durch. In einer Ausführungsform werden die Operationen 5949-5972 an dem Stapel 5203 durch eine Stapelverarbeitungsschaltungsanordnung 5120 (siehe **Fig. 51**) ausgeführt. Eine spezifische Traversierungssequenz ist beginnend mit dem Root-BVH-Knoten N 5900 auf BVH-Ebene 0 gezeigt.

[0425] Bei 5949 wird der Stapel 5203 mit dem Knoten N initialisiert, der dann von dem Stapel mittels Pop entnommen und verarbeitet wird, was in Treffern H0-H2 resultiert, die Child-Knoten N0-N2 5901-5903 auf Level 1 der BVH umfassen (d. h. „Treffer“, was bedeutet, dass der Strahl die drei Child-Knoten N0-N2 5901-5903 traversiert). Die drei Child-Knoten-Treffer 5901-5902 werden basierend auf der Trefferdistanz sortiert und in der sortierten Reihenfolge auf den Stapel 5203 gepusht (Operation 5950). Dementsprechend werden in dieser Ausführungsform, wann immer ein neuer Satz von Child-Knoten evaluiert wird, diese basierend auf der Trefferdistanz sortiert und in der sortierten Reihenfolge (d. h. mit den näher gelegenen Child-Knoten oben auf dem Stapel) in den Stapel 5203 geschrieben.

[0426] Der erste Child-Knoten N0 5901 (d. h. der nächstgelegene Child-Knoten) wird von dem Stapel 5203 mittels Pop entnommen und verarbeitet, was in drei weiteren Child-Knotentreffern N00-N02 5911-5913 auf Ebene 2 der BVH resultiert (die „Ebene“ wird manchmal als die „Tiefe“ der BVH-Knoten bezeichnet), die sortiert und zu dem Stapel 5203 gepusht werden (Operation 5951).

[0427] Der Child-Knoten N00 5911 wird mittels Pop aus dem Stapel entnommen und verarbeitet, was zu einem einzelnen Treffer führt, der einen einzelnen Child-Knoten N000 5920 auf Ebene 3 der BVH umfasst (Operation 5952). Dieser Knoten wird einem Pop-Vorgang unterzogen und verarbeitet, was zu sechs Treffern N0000-N0005 5931-5936 auf Ebene 4 führt, die sortiert und auf den Stapel 5203 gepusht werden (Operation 5953). Um Platz innerhalb des kurzen Stapels 5203 zu schaffen, werden die Knoten N1, N2, N02, N01 entfernt, wie angegeben (d. h. um den kurzen Stapel auf sechs Einträge zu beschränken). Der erste sortierte Knoten N0000 5931 wird einem Pop-Vorgang unterzogen und verarbeitet, wodurch drei Treffer N00000-N00002 5931-5933 auf Ebene 5 der BVH erzeugt werden (Operation 5954). Knoten N0005 wird entfernt, um auf dem kurzen Stapel 5203 Platz für die neuen Knoten zu schaffen.

[0428] In einer Ausführungsform wird jedes Mal, wenn ein Knoten aus dem kurzen Stapel 5203 entfernt wird, dieser in den Speicher zurück gespeichert. Er wird dann zu einem späteren Zeitpunkt (z. B. wenn es Zeit ist, den Knoten gemäß der Traversierungsoperation zu verarbeiten) erneut in den kurzen Stapel 5203 geladen.

[0429] Die Verarbeitung fährt in **Fig. 59A** fort, in der die Knoten N00001 und N00002 auf Ebene 5 der BVH einem Pop-Vorgang unterzogen und verarbeitet werden (Operationen 5955-5956). Die Knoten N0001, N0002, N0003 und N0004 auf Ebene 4 werden dann einem Pop-Vorgang unterzogen und verarbeitet (Operationen 5957-5960), was zu einem leeren kurzen Stapel 5203 führt.

[0430] Somit führt eine Pop-Operation zum Abruf des Root-BVH-Knotens, Knoten N, gemäß dem Neustartpfad (RST) (Operation 5961). Die drei Child-Treffer N0, N1, N2 aus Ebene 1 werden wieder sortiert und auf den kurzen Stapel gepusht (Operation 5962). Der Knoten N0 wird dann einem Pop-Vorgang unterzogen und verarbeitet, gefolgt von den Knoten N00, N000 und N0005 (Operationen 5963-5965). Knoten N01 wird einem Pop-Vorgang unterzogen und verarbeitet (Operation 5966), gefolgt von Knoten N02, Knoten N2 und Knoten N1 (Operationen 5967-5970), was wiederum zu einem leeren kurzen Stapel führt. Folglich wird der nächste Ebene-2-Knoten N11 mittels Pop-Vorgang von dem kurzen Stapel entfernt und verarbeitet, wodurch die Traversierung abgeschlossen wird (d. h. da der Knoten N11 keinen Treffer zur Folge hatte).

[0431] Wie erwähnt, aktualisiert eine Ausführungsform eines Traversierungsverfolgers 5248 das Verfolgungsarray 5249, das den Child-Knoten/Unterbaum in jeder Ebene der BVH-Hierarchie identifiziert, die gegenwärtig traversiert wird. In einer Implementierung ist die Länge des Verfolgungsarrays 5249 gleich der Tiefe der BVH (6 in dem veranschaulichten Beispiel), und jeder Eintrag in dem Verfolgungsarray 5249 beinhaltet einen Indexwert, der den Child-Unterbaum identifiziert, der aktuell traversiert wird. In einer spezifischen Implementierung beinhaltet jeder Eintrag in dem Verfolgungsarray 5249 für eine N-breite BVH (d. h. in der jeder innere Knoten N Child-Knoten referenziert) einen log2(N) -Bitwert, um die Child-Knoten/Unterbäume zu identifizieren. In einer Ausführungsform wurden Child-Knoten/Unterbäume, denen ein Index zugewiesen ist, der kleiner als der aktuelle Child-Index ist, vollständig traversiert, und werden daher in dem Fall eines Neustarts nicht erneut besucht. In einer Ausführungsform wird, wenn das letzte geschnittene Child traversiert wird, der Child-Index auf den Maximalwert gesetzt, um anzuzeigen, dass es keine Einträge mehr auf dem Stapel gibt.

[0432] Der kurze Traversierungsstapel 5203 kann die obersten wenigen Einträge des Stapels in einem kreisförmigen Array speichern. In einer Implementierung beinhaltet jeder Stapeleintrag in dem kurzen Traversierungsstapel 5203 einen Offset zu einem Knoten, sonstige Informationen, wie etwa den Knotentyp (innerer, Primitiv, Instanz usw.) sowie ein Bit, das anzeigt, ob dieses Child der letzte (am weitesten entfernte) geschnittene Child-Knoten in einem Parent-Knoten ist. Diese speziellen Einzelheiten sind jedoch nicht erforderlich, um mit den der Erfindung zugrundeliegenden Prinzipien übereinzustimmen.

[0433] **Fig. 60** veranschaulicht eine Ausführungsform der Stapelverarbeitungsschaltungsanordnung/-logik 5120 zum Durchführen von Stapelverwaltung und Traversierungsoperationen, wie oben beschrieben. Ein Stapel-FIFO 6001 wird mit beliebigen Child-BVH-Knoten 6000 geladen, die eine Verarbeitung erfordern. Wenn zum Beispiel ein Rahmentest oder Viereckstest durch die Traversierungsverarbeitungsschaltungsanordnung 5210 abgeschlossen wird, werden die Ergebnisse in das Stapel-FIFO 6001 gepusht und zum Aktualisieren des Stapels 5203 verwendet. Dies kann zum Beispiel Aktualisierungen an der Hitinfo, wie etwa dem Satz von Child-Knoten 6000, die mit einem bestimmten Treffer assoziiert sind, beinhalten.

[0434] Die Stapelverarbeitungsschaltungsanordnung/-logik 6003 liest Einträge aus dem Stapel 5203 mit Daten, die zum Verarbeiten jedes Eintrags erforderlich sind, einschließlich einer Angabe darüber, ob der BVH-Knoten ein innerer Knoten oder ein Leaf-Knoten ist, und assoziierten Indexdaten. Falls der Knoten ein Leaf-Knoten/Viereck ist, dann können die Daten Viereck-Deskriptoren und Indizes sowie Shader-Index-Daten beinhalten. Die Stapelverarbeitungsschaltungsanordnung/-logik 6003 führt dann die hierin beschriebenen Stapelverarbeitungsoperationen aus, wie etwa Identifizieren neuer mit einem Treffer assoziierter Knoten, und Sortieren der Knoten basierend auf der Trefferdistanz. Obwohl dies als eine separate Entität veranschaulicht ist, kann die Stapelverarbeitungsschaltungsanordnung/-logik 6003 innerhalb der Traversierungsschaltungsanordnung 5102 integriert sein.

[0435] Wie angegeben, erzeugt die Stapelverarbeitungsschaltungsanordnung/-logik 6003 Stapelaktualisierungen 6011, wenn sie die Verarbeitung jedes BVH-Knotens aus dem Stapel 5203 abschließt. Nach dem Lesen eines Eintrags aus dem Stapel 5203 kann sie zum Beispiel die verschiedenen Steuerbits aktualisieren, wie etwa das Datenpräsenz(DP)-Bit und das Gültig(VLD)-Bit. **Fig. 60** veranschaulicht, dass das Räumbereitschafts- und das Datenpräsenzbit 6010 gesetzt sind. Eine entsprechende Stapelaktualisierung 6011 kann auch an den Stapel 5203 gesendet werden (was z. B. ermöglicht, dass alte Einträge entfernt werden, um Platz für neue Child-Knoten zu schaffen).

[0436] Stapelaktualisierungen können über eine Arbitrierungsschaltungsanordnung 6012 gesteuert werden, die zwischen Aktualisieren des Stapels 5203 mit den aktuellen Verarbeitungsaktualisierungen 6011, Füllen des Stapels 5203 aus dem Speicher mit einem oder mehreren neuen BVH-Child-Knoten (Mem Fill) und Durchführen einer anfänglichen Zuweisung zu dem Stapel aus dem Speicher (z. B. beginnend mit dem Root-Knoten und einem oder mehreren Child-Knoten) auswählt.

[0437] In einer Ausführungsform können, wenn ein(e) Viereck/Instanz/innerer Knoten auf dem Stapel verarbeitet wird, eine oder mehrere der folgenden Operationen durchgeführt werden:

i. Räumung des Stapeleintrags aufgrund mehrerer Bedingungen, wie etwa Herunterbewegen der Instanz für eine neue BVH, Verarbeiten eines Trefferprozedurals, eines beliebigen Treffer-Shaders usw.

ii. Freigeben des Strahleintrags, wenn der Stapel aufgrund eines Trefferprozedurals und/oder Beliebiger-Treffer-Shaders geräumt wird.

iii. Freigeben des Cache-Eintrags, wenn dieser Stapel aufgrund eines Trefferprozedurals und/oder Beliebiger-Treffer-Shaders geräumt wird.

iv. Aktualisieren der Strahlsteuerung (nur BVH), falls der Strahl über das Instanz-Leaf zu der neuen BVH abwärtsgeleitet werden muss.

[0438] **Fig. 61A-B** veranschaulichen Tabellen zum Konfigurieren von Lese-/Schreibports und Einstellen von Steuerbits für alle Raytracing-Traversierungsstrukturen. Insbesondere sind beispielhafte Unterstrukturen, vertikale Strukturen und Lese/Schreib-Aktionen für Strahlen 6101, Treffer 6102 und Stapel 6103 gezeigt. Es ist jedoch anzumerken, dass die zugrundeliegenden Prinzipien der Erfindung nicht auf diese spezifischen Datenstrukturen/Operationen beschränkt sind.

EINRICHTUNG UND VERFAHREN FÜR HOCHWERTIGE STRAHLVERFOLGTE DETAILGRADÜBERGÄNGE

[0439] Auf Grafikverarbeitungsarchitekturen kann sich das „Level-of-Detail“ (LOD - Detailgrad) auf die Auswahl von Mesh-Auflösungen basierend auf Variablen, wie etwa dem Abstand von der Kamera, beziehen. LOD-Techniken werden verwendet, um Speicherverbrauch zu reduzieren und Grafikverarbeitungsfunktionen, wie etwa geometrisches Aliasing in Spielen, zu verbessern. Zum Beispiel werden die Details eines hochauflösenden Mesh möglicherweise nicht benötigt, wenn das Mesh weit von der aktuellen Perspektive des Benutzers entfernt ist.

[0440] Bei rasterisierungsbasierten Implementierungen werden glatte Übergänge zwischen LODs durch Verwenden von „stochastischen LOD“-Techniken ermöglicht, wie etwa beschrieben in Lloyd et al., Implementing Stochastic Levels of Detail with Microsoft DirecX Raytracing (15. Juni 2020). Ohne diese stochastischen Techniken kann der Übergang zwischen LODs zu ablenkenden Artefakten führen, bei denen sich das Erscheinungsbild von Objekten plötzlich ändert, wenn ein neuer LOD ausgewählt wird. Durch Verwenden stochastischer LODs wird eine Querauflösung zwischen LOD-Graden durch eine zufällige Zuordnung von Pixeln

zu einem der LODs, die an dem Übergang beteiligt sind, ausgeführt (z. B. entweder der LOD mit höherer Auflösung oder der LOD mit niedrigerer Auflösung).

[0441] Die obige Lösung verwendet eine binäre Maske und einen binären Vergleichswert, um acht Übergangsschritte für stochastische LOD-Übergänge beim Überblenden von einem ersten LOD („LOD0“) zu einem zweiten LOD („LOD1“) zu erreichen. In dieser Implementierung werden eine 8-Bit-Strahlmaske und eine 8-Bit-Instanzmaske logisch UND-verknüpft, um zu bestimmen, ob eine Instanz traversiert werden muss. Diese 8-Bit-Masken und die assoziierten bitweisen Logikoperationen resultieren in begrenzten LOD-Übergangsfähigkeiten. Wenn zum Beispiel ein Übergang zwischen LODO und LOD1 eines Objekts vorgenommen wird, wobei LODO einen Bruchwert von 0,25 aufweist und LOD1 einen Bruchwert von 0,75 aufweist (basierend auf dem Kameraabstand), würde die Maske für die Instanz auf LODO gesetzt werden, um nur 2 Zufallsbit (0,25 von 8 Bit) freizugeben. Die Instanzmaske für LOD1 würde auf das Binärkomplement der Maske von LODO gesetzt, wobei 6 Bit freigegeben sind. Für einen jeglichen gegebenen Strahl wird ein Zufallsbit in der Strahlmaske ausgewählt, um eine Zufallsauswahl von entweder LODO (mit einer Wahrscheinlichkeit von 0,25) und LOD1 (mit einer Wahrscheinlichkeit von 0,75) zu erreichen. Da jedoch nur eines von acht Bit ausgewählt wird, gibt es nur 8 Zwischenschritte zum Übergang zwischen LODO und LOD1.

[0442] Wie in **Fig. 62** gezeigt, wird in einer Ausführungsform der Erfindung eine N-Bit-Vergleichsoperationsmaske 6220 an den LOD-Selektor 6205 bereitgestellt, die als ein Binärwert behandelt wird, um eine durchzuführende Vergleichsoperation zu bestimmen. Die ausgewählte Vergleichsoperation wird verwendet, um mit der Referenz zu vergleichen, um mehr Übergangs-LOD-Schritte zu gestatten. In einer Ausführungsform ist die Vergleichsoperation von kleiner-gleich (less_equal) und größer-als (greater) ausgewählt, obwohl die der Erfindung zugrundeliegenden Prinzipien nicht auf diese spezifischen Vergleichsoperationen beschränkt sind. In einer Implementierung werden 8 Bit verwendet (N=8), wobei 7 der Bits einen vorzeichenlosen ganzzahligen Wert in dem Bereich von [0..127] definieren, wodurch 128 Übergangsschritte für LOD-Überblendung ermöglicht werden, und 1 Bit die Vergleichsoperation angibt (z. B. wenn auf 0 gesetzt, dann wird eine less_equal-Operation ausgeführt, und wenn auf 1 gesetzt, wird die greater-Operation ausgeführt). In einer Ausführungsform kann dem LOD-Selektor 6205 auch eine Strahlvergleichsmaske 6221 in dem Bereich [0..127] als ein zusätzlicher Strahlparameter bereitgestellt werden.

[0443] Die folgende Codesequenz verdeutlicht, wie die Strahltraversierung auf diese neue Vergleichsmaske in einer Ausführungsform reagiert:

```
if (ray.InstanceMask & instance.InstanceMask)
{
       if(
      ( instance.ComparisonMode == less_equal &&
      instance.ComparisonMask <= ray.ComparisonMask) ||
      (instance.ComparisonMode == greater && instance.ComparisonMask
      > ray.ComparisonMask)
      )
      {
     traverseInstance(Instance);
     }
}
```

[0444] In der obigen Codesequenz prüft die erste IF-Aussage, ob die Binärmasken eine Traversierung in die aktuelle Instanz ermöglichen. Falls ja, prüft die zweite IF-Aussage dann die Vergleichsmoduseinstellung hinsichtlich der Werte für die Instanzvergleichsmaske (z. B. Vergleichsoperationsmaske 6220) und die Strahlvergleichsmaske 6221.

[0445] Zurückkehrend zu dem obigen LOD-Übergangsbeispiel werden für die Instanz von LODO mit einem Bruchwert von 0,25 die ersten 7 Bit auf einen Wert von 31 (=int(0,25*127)) gesetzt, und das letzte Bit wird auf 0 gesetzt (was die less_equal-Operation angibt). Für die Instanz von LOD1 mit einem Bruchwert von 0,75 werden die ersten 7 Bit auf einen Wert von 31 (=int(1,0-0,75)*127) gesetzt, und das letzte Bit wird auf 1 gesetzt (was die greater-Operation angibt). Somit gibt es für diese Implementierung, falls eine gleichmäßig verteilte Zufallszahl in dem Bereich [0 .. 127] als eine Strahlvergleichsmaske erzeugt wird, bis zu 127 Übergangsschritte, die durch den LOD-Selektor 6205 zum Übergehen zwischen LODO und LOD1 ausgewählt werden können.

[0446] Obwohl die oben dargelegten spezifischen Einzelheiten zu dem Zweck der Erläuterung verwendet werden, können die zugrundeliegenden Prinzipien der Erfindung mit anderen Einzelheiten implementiert werden. Zum Beispiel können andere Vergleichsoperatoren anstelle von oder zusätzlich zu less_equal und greater verwendet werden. Zum Beispiel können auch Vergleichsoperatoren wie not_equal, equal, less und greater_equal (größer-gleich) verwendet werden. Eine Implementierung beinhaltet ein Strahl-Flag und ein Instanz-Flag, das UND-verknüpfte Strahlmasken deaktiviert, und die Verwendung dieser Bits als Vergleichsmasken ermöglicht.

[0447] Ausführungsformen der Erfindung beinhalten eine Kombination einer Festfunktionsbeschleunigungsschaltungsanordnung und einer Mehrzweckverarbeitungsschaltungsanordnung zum Durchführen eines Raytracing. Beispielsweise können bestimmte Operationen in Bezug auf die Strahltraversierung einer Hüllkörperhierarchie (BVH) und Überschneidungstests durch die Festfunktionsbeschleunigungsschaltungsanordnung durchgeführt werden, während mehrere Ausführungsschaltungen verschiedene Formen von Raytracing-Shadern (z. B. Beliebiger-Treffer-Shader, Überschneidungs-Shader, Fehltreffer-Shader usw.) ausführen. Eine Ausführungsform beinhaltet duale Speicherbänke mit hoher Bandbreite, die mehrere Einträge zum Speichern von Strahlen und entsprechende duale Stapel zum Speichern von BVH-Knoten umfassen. In dieser Ausführungsform wechselt die Traversierungsschaltungsanordnung zwischen den dualen Strahlbänken und Stapeln, um einen Strahl bei jedem Taktzyklus zu verarbeiten. Außerdem beinhaltet eine Ausführungsform eine Prioritätsauswahlschaltungsanordnung/-logik, die zwischen internen Knoten, nichtinternen Knoten und Primitiven unterscheidet und diese Informationen verwendet, um die Verarbeitung der BVH-Knoten und der durch die BVH-Knoten begrenzten Primitive intelligent zu priorisieren.

BESCHLEUNIGUNGSDATENSTRUKTURKOMPRIMIERUNG

[0448] Die Konstruktion von Beschleunigungsdatenstrukturen ist einer der wichtigsten Schritte beim effizienten Raytracing-Rendern. In jüngster Zeit ist die hierin ausführlich beschriebene Hüllkörperhierarchie(BVH)-Beschleunigungsstruktur die für diesen Zweck am meisten verwendete Struktur geworden. Die BVH ist eine hierarchische Baumstruktur, die dazu dient, Geometrie räumlich so zu indizieren und zu organisieren, dass Strahl/Primitiv-Überschneidungsabfragen sehr effizient gelöst werden können. Die Fähigkeit, diese Abfragen zu lösen, ist eine der kritischsten Operationen für Raytracing-Rendern. Obwohl die unten beschriebenen Ausführungsformen der Erfindung an einer BVH-Struktur arbeiten, sind die zugrundeliegenden Prinzipien der Erfindung nicht auf eine BVH beschränkt. Diese Ausführungsformen können auf jegliche andere Beschleunigungsdatenstruktur mit ähnlichen relevanten Merkmalen angewendet werden.

[0449] Das Erzeugen einer BVH wird typischerweise als „Konstruieren“ oder „Building“ der BVH bezeichnet. Obwohl eine Anzahl von BVH-Konstruktionsalgorithmen vorgeschlagen wurden, werden Top-Down-BVH-Builder vorwiegend verwendet, um eine hohe Rendereffizienz sowohl für Echtzeit- als auch Offline-Renderanwendungen zu erreichen. Top-Down-BVH-Building-Algorithmen führen in der Regel ein oder mehrere temporäre Arrays während der Konstruktion. Diese Arrays halten Daten, die notwendig sind, um die Geometrie zu sortieren/zu organisieren, um die BVH-Struktur zu erzeugen. Diese Arrays werden während des Building mehrmals gelesen und/oder geschrieben (typischerweise 1-2 mal pro Ebene der BVH-Hierarchie). Da diese Arrays häufig von erheblicher Größe sind, ist dieser Prozess bandbreitenintensiv. Somit haben Verbesserungen der BVH-Building-Rechenleistung, wie sie etwa von einem Hardware-BVH-Builder erwartet werden könnten, wahrscheinlich nur eine begrenzte Auswirkung, wenn dieses Bandbreitenproblem unbehandelt bleibt.

[0450] Eine Ausführungsform der Erfindung beinhaltet ein Komprimierungsschema für die temporären Daten, die durch viele Top-Down-BVH-Builder geführt werden. Der Zweck dieses Komprimierungsschemas ist es, die für die BVH-Konstruktion erforderliche Bandbreite zu reduzieren, wodurch eine schnellere und effizientere BVH-Konstruktion ermöglicht wird. Es ist jedoch anzumerken, dass die Ausführungsformen der Erfindung für andere Arten von BVH-Builder und mit anderen Arten von Beschleunigungsdatenstrukturen, wie etwa kd-Bäumen, verwendet werden können.

[0451] Viele Top-Down-BVH-Builder führen zwei primäre Datentypen während des BVH-Build: (1) einen achsenausgerichteten Begrenzungsrahmen (Axis Aligned Bounding Box, AABB) für jedes Primitiv, das an dem BVH-Build beteiligt ist; und (2) einen mit jedem Primitiv assoziierten vorzeichenlosen ganzzahligen Index, der auf einen dieser AABBs und/oder auf den ursprünglichen Primitiv, von dem der AABB erzeugt wurde, zeigt.

[0452] Eine Ausführungsform der Erfindung nutzt ein Structure-of-Arrays(SOA)-Layout zum Kombinieren jedes AABB mit einem einzigen ganzzahligen Index. Die AABBs werden in einem Array und die ganzzahligen

Indizes in einem zweiten Array geführt. Nur das Index-Array muss umgeordnet werden, um eine BVH-Konstruktion zu erreichen. Ein Speichern der Build-Daten auf diese Weise führt zu einer Anzahl von Vorteilen. In diesem Layoutschema sind die AABB-Daten größtenteils Nurlesedaten, und AABB-Schreibbandbreite fällt für den Großteil des Build-Prozesses nicht an.

[0453] Durch die Verwendung einer SOA-Struktur müssen nur die AABBs gelegentlich während des Build komprimiert werden. Tatsächlich müssen die AABB-Daten in Abhängigkeit von der Implementierung möglicherweise nur einmal vor dem Build als ein Vorprozess komprimiert werden. Da der Build durch Partitionieren der Index-Arrays durchgeführt wird, werden diese in einer Ausführungsform der Erfindung auf jeder Ebene des Build erneut komprimiert.

[0454] Durch Arbeiten mit komprimierten Versionen dieser Arrays anstelle ihrer herkömmlichen, unkomprimierten Gegenstücke wird die für den BVH-Build erforderliche Bandbreite reduziert. Die komprimierten Versionen der Arrays werden temporär gespeichert und nur zum Zweck des Build verwendet. Sie werden verworfen, sobald der Build abgeschlossen ist, wodurch eine BVH entsteht, die auf die ursprüngliche Eingabeliste von Primitiven verweist.

[0455] Ein wichtiges Merkmal der hierin beschriebenen Komprimierungstechniken ist, dass sie cachezeilenbewusst sind. Beide der komprimierten Arrays werden als ein Array von Komprimierungsblöcken mit fester Größe gespeichert, wobei die Größe eine ganze Zahl von Cachezeilen ist. Diese Zahl ist größer oder gleich eins. Die Komprimierungsblöcke jedes der beiden Arraytypen müssen nicht gleich groß sein. Diese zwei Blocktypen werden hierin als AABB-Komprimierungsblöcke und Indexkomprimierungsblöcke bezeichnet.

[0456] Es wird angemerkt, dass die zugrundeliegenden Prinzipien der Erfindung nicht erfordern, dass die Größe der Blöcke eine ganze Zahl von Cachezeilen ist. Vielmehr ist dies eines von mehreren hierin beschriebenen optionalen Merkmalen. Bei einer unten beschriebenen Ausführungsform wird diese Funktionalität durch die Variablen AABBCompressionBlockSizeBytes und IndexCompressionBlockSizeBytes in Tabellen B bzw. D gesteuert.

[0457] Da die räumliche Ausdehnung jedes Knotens und die Anzahl von Primitiven, auf die jeder Knoten verweist, im Allgemeinen abnehmen wird, wenn der Top-Down-Build von dem Root zu den Leaves der Baumstruktur fortschreitet, können unterschiedliche Repräsentationen der AABBs an unterschiedlichen Konstruktionsstufen geeignet sein. Zum Beispiel kann die Präzision der komprimierten AABBs an den oberen Ebenen des Baums weniger kritisch sein, wohingegen genauere Repräsentationen an den unteren Ebenen erforderlich sein können, um eine angemessene Baumqualität aufrechtzuerhalten. Es kann deshalb ausreichend sein, verlustbehaftete Komprimierung nahe des Root des Baums zu verwenden, um Bandbreiteneinsparungen zu maximieren, und zu einer unkomprimierten, verlustfreien Repräsentation der Primitive für die niedrigeren Ebenen zu wechseln. Dies unterteilt den BVH-Build in mindestens zwei Phasen, die in **Fig. 63** veranschaulicht sind: eine obere Phase 6301 für Knoten auf oder oberhalb einer spezifizierten Ebene der Hierarchie (Knoten 0, 1, 8) und eine untere Phase 6302 für Knoten unterhalb der spezifizierten Ebene (Knoten 2-7, 9-14). Ein Mehrebenen-Build kann auf eine solche Weise fortfahren, dass die Gesamtheit einer Hierarchie höherer Ebene (z. B. der „obere“ Abschnitt in **Fig. 63**) aufgebaut wird, bevor ein jeglicher Knoten in den unteren Ebenen aufgebaut wird, oder der Build der Ebenen kann verschachtelt sein. Falls eine obere Ebene vollständig vor jeglichen unteren Ebenen aufgebaut wird, können Knoten, die auf einer unteren Ebene des Build aufgeteilt werden müssen, auf einer Struktur, wie etwa einer Warteschlange, die in einem späteren Stadium partitioniert werden soll, gespeichert werden.

[0458] Alternativ zu der Verwendung einer Kopie der AABBs mit voller Präzision für die unteren Ebenen 6302 ist es eine andere Variation des Schemas, die AABBs während des Build zur Verwendung bei dem Building der unteren Ebenen „erneut zu komprimieren“. Dadurch kann Geometrie relativ zum Umfang einzelner Unterbäume komprimiert werden. Da einzelne Unterbäume im Allgemeinen eine geringere räumliche Ausdehnung im Vergleich zu dem Root-Knoten repräsentieren, kann dies der Genauigkeit der komprimierten Repräsentation bzw. der Effizienz der Komprimierung zugute kommen. Ein ähnliches Muster für einen komprimierten Mehrebenen-Build wird in der aktuellen Forschung beobachtet. Die Aufteilung 6300 zwischen verschiedenen Konstruktionsphasen kann gemäß einer Vielzahl von Knotencharakteristiken definiert werden. Eine Ausführungsform verwendet eine feste Anzahl von Primitiven, die als ein Schwellenwert agieren.

[0459] Eine Variation, die bei manchen Ausführungsformen der Erfindung verwendet wird, sieht stattdessen vor, nur einen Einzelebenen-Build einzusetzen. Zum Beispiel könnte eine einzelne komprimierte Repräsentation der Build-Daten verwendet werden, um den gesamten Baum aufzubauen.

I. AABB-Komprimierung

[0460] In einer Ausführungsform der Erfindung ist die Eingabe in die AABB-Komprimierungslogik (die in Hardware und/oder Software implementiert sein kann) ein Array von unkomprimierten Primitiven, und die Ausgabe ist ein Array von AABB-Komprimierungsblöcken, die eine feste Größe haben und mit einer gewissen Anzahl von Cachezeilen ausgerichtet sind. Da das effektive AABB-Komprimierungsverhältnis in einem beliebigen bestimmten Bereich des Mesh stark datenabhängig ist, packt eine Ausführungsform eine variable Anzahl von AABBs pro AABB-Komprimierungsblock.

[0461] Wie in **Fig. 64** gezeigt, ist eine Ausführungsform des Komprimierungsblocks 6400 in zwei Hauptteilen organisiert: Metadaten 6401 und Vektorresiduen 6402. Die Metadaten 6401 liefern blockweise Informationen und Konstanten, die erforderlich sind, um die Vektorresiduen 6402 in eine Liste von AABBs zu decodieren. Die Vektorresiduen 6402 speichern den Großteil der komprimierten Informationen, die zum Repräsentieren der AABBs verwendet werden. Jedes dieser Elemente wird unten ausführlicher besprochen.

[0462] Kurz gesagt, wird in einer Ausführungsform Delta-Komprimierung verwendet. Ein seedVector umfasst einen Baseline-Satz von AABB-Werten und die Vektorresiduen 6402 liefern Offsets zu diesen Baseline-Werten, um jeden AABB zu rekonstruieren. Der numResiduals-Wert spezifiziert die Anzahl der Vectorresiduen 6402, und der residualSizeVector spezifiziert die Größe der Residuen 6402.

Globale AABB-Komprimierungskonstanten

[0463] Zusätzlich zu den blockweisen Konstanten, die in jedem Komprimierungsblock 6400 gespeichert sind, kann ein Satz von globalen AABB-Komprimierungskonstanten Informationen bezüglich aller Blöcke in dem gesamten Komprimierungsprozess speichern. Diese sind in Tabelle B für eine bestimmte Implementierung zusammengefasst.

TABELLE B

Konstante	**Beschreibung**
NQ {X, Y, Z}	Drei Werte, die die Anzahl der Bits bezeichnen, die zur Quantisierung von Vertex-Komponenten in jeder der drei Raumdimensionen verwendet werden.
AABBCompressionBlockSizeBytes	Größe in Bytes eines AABB-Komprimierungsblocks. Dieser Wert wird typischerweise auf eine bestimmte Anzahl von Cachezeilen ausgerichtet sein.
maxAABBsPerBlock	Die maximal erlaubte Anzahl von AABBs in einem AABB-Komprimierungsblock. Diese Konstante wird zusammen mit der globalen Komprimierungskonstante num ResidualVectorsPerPrimitive verwendet, um die Anzahl von Bits zu bestimmen, die für den in **Fig. 64** gezeigten numResiduals-Wert benötigt werden.
numResidualVectorsPerPrimitive	Dieser Wert verfolgt die Anzahl der Residuenvektoren, die zur Repräsentation eines AABB in den komprimierten Blöcken verwendet werden. Ein regulärer AABB besteht normalerweise aus zwei 3D-Vektoren, min und max. Es ist jedoch möglich, dass die Repräsentation des AABB in eine Struktur mit einer unterschiedlichen Anzahl von Vektoren transformiert werden kann. Ein Beispiel dafür wird in dem späteren Abschnitt über AABB-Residuenberechnung diskutiert, wobei ein Paar von 3D-Vektoren in einen einzelnen 6D-Vektor transformiert wird. Es ist erforderlich, dass der Komprimierungsalgorithmus diesen Wert verfolgt, um eine Anzahl von Kernoperationen korrekt auszuführen.
residualNumDimensions	Diese Konstante wird verwendet, um zu verfolgen, wie viele Dimensionen die

Konstante	Beschreibung
	Residuenvektoren an dem Punkt haben werden, an dem sie zu den AABB-Komprimierungsblöcken hinzugefügt werden. Dieser Wert wird benötigt, da es möglich ist, dass die 3D-AABB-Dater bei der Komprimierung in eine unterschiedliche Anzahl von Dimensionen transformiert werden.

AABB-Komprimierungsfluss

[0464] Eine Ausführungsform des AABB-Komprimierungsprozesses schließt wiederum der Reihe nach Iterieren durch das Eingabe-Array von Primitiven und Ausgeben eines Arrays von AABB-Komprimierungsblöcken 6400 ein. Das Ausgabe-Array enthält eine minimale Anzahl von AABB-Komprimierungsblöcken 6400, die benötigt werden, um den AABB der Primitiven in komprimierter Form zu repräsentieren.

[0465] **Fig. 65** veranschaulicht einen Prozess gemäß einer bestimmten Ausführungsform. Wie erwähnt, ist der Komprimierungsprozess nicht auf irgendeine Architektur beschränkt, und kann in Hardware, Software oder jeglicher Kombination davon implementiert werden.

[0466] Bei 6501 wird ein Array von Primitiven für einen BVH-Build bereitgestellt. Bei 6502 wird das nächste Primitiv in dem Array (z. B. das erste Primitiv zu Beginn des Prozesses) ausgewählt und sein AABB wird auf Komprimierung evaluiert. Falls der AABB in den aktuellen Komprimierungsblock passt, bestimmt bei 6503 (z. B. basierend auf seinen Min/Max-Daten), dann wird der AABB bei 6504 zu dem aktuellen Komprimierungsblock hinzugefügt. Wie erwähnt, kann dies das Bestimmen von Residuenwerten für den AABB durch Berechnen der Abstände zu einem existierenden Basisvektor innerhalb des Komprimierungsblocks (z. B. dem seedVector) beinhalten.

[0467] In einer Ausführungsform wird, falls der AABB des Primitiv nicht in den Komprimierungsblock passt, der aktuelle Komprimierungsblock bei 6510 finalisiert, und in Speicher innerhalb des Ausgabe-Arrays gespeichert. Bei 6511 wird ein neuer Komprimierungsblock durch Verwenden des AABB des Primitiv initialisiert. In einer Ausführungsform wird der Primitiven-AABB als der Seed-Vektor für den neuen Komprimierungsblock verwendet. Residuen können dann für nachfolgende Primitiven-AABB basierend auf Abständen zu dem neuen Seed-Vektor erzeugt werden. In einer Implementierung wird das erste Residuum, das für den zweiten AABB erzeugt wird, basierend auf Abstandswerten zu den Seed-Vektorwerten bestimmt. Das zweite Residuum wird dann für den dritten AABB basierend auf Abständen zu dem ersten Residuum bestimmt. Es wird somit eine Laufdifferenz, wie unten ausführlicher beschrieben, gespeichert. Sobald das aktuelle Primitiv komprimiert ist, kehrt der Prozess zu 6502 zurück, wo das nächste Primitiv in dem Array zur Komprimierung ausgewählt wird.

[0468] Somit wird beim Besuchen jedes Primitiv sein AABB bestimmt (z. B. als ein Gleitwert). Dann wird eine Reihe von Operationen an dem AABB durchgeführt, um eine Komprimierung zu erreichen, und das komprimierte Ergebnis wird zu dem aktuellen AABB-Komprimierungsblock in dem Ausgabe-Array hinzugefügt. Falls der komprimierte AABB passt, wird er zu dem aktuellen Block hinzugefügt, und der Prozess geht weiter zu dem nächsten AABB. Falls der AABB nicht passt, wird der aktuelle AABB-Komprimierungsblock abgeschlossen und ein neuer AABB-Komprimierungsblock wird in dem Ausgabe-Array initialisiert. Auf diese Weise wird die Anzahl komprimierter Blöcke, die zum Speichern des AABB benötigt werden, minimiert.

[0469] Der Pseudocode unten in TABELLE C zeigt den Fluss der AABB-Komprimierung gemäß einer bestimmten Ausführungsform der Erfindung. Es ist jedoch anzumerken, dass die zugrundeliegenden Prinzipien der Erfindung nicht notwendigerweise auf diese Einzelheiten beschränkt sind.

[0470] Wie in der Pseudocodesequenz gezeigt, wird für jeden AABB-Komprimierungsblock eine ganze Zahl in ein separates Array (blockOffsets) geschrieben, das die Position in dem ursprünglichen Primitiv-Array aufzeichnet, an der jeder AABB-Komprimierungsblock startet (d. h. den ersten Primitiv-AABB, den es enthält). Das blockOffsets-Array wird während des Build zum Auflösen der ursprünglichen Primitiv-ID verwendet, die der komprimierte Block repräsentiert.

AABB-Residuenberechnung

[0471] In einer Ausführungsform durchläuft jeder Eingabe-AABB einen Satz von Stufen, um ihn zu komprimieren, bevor er zu einem komprimierten Block hinzugefügt wird, was in den in **Fig. 64** gezeigten Vektorresiduen resultiert. Der Prozess wird als der Code in Zeile 26 von Tabelle C erfasst, wobei der CompressionCore verwendet wird, um den AABB in eine Liste komprimierter Vektoren umzuwandeln.

```
1:  uint numBoxesEncoded = 0;
2:  uint blockStartIndex = 0;
3:  uint currentBlock = 0;
4:  CompressedAABBBlock compressedBlocks = []
5:  uint blockOffsets = []
6:  uint totalNumBoxes = geometry.getNumPrimitives();
7:  uint maxBitsPerBlock =AABBCompressionBlockSizeBytes * 8;
8:  uint numBitsRequiredCurrentBlock = 0;
9:
10: while(numBoxesEncoded < totalNumBoxes)
11: {
12:  CompressionCore cCore;
13:  InitBlock(compressedBlocks, currentBlock);
14:  blockOffsets.append(numBoxesEncoded);
15:  blockStartIndex = numBoxesEncoded;
16:  numBitsRequiredCurrentBlock = 0;
17:
18:  while(numBitsRequiredCurrentBlock < maxBitsPerBlock &&
19:        numBoxesEncoded < totalNumBoxes &&
20:       (numBoxesEncoded - blockStartIndex) < maxAABBsPerBlock)
21:  {
22:     Primitive p = geometry.getPrimitive(numEncoded);
23:     AABB box = p.getBoundingBox();
24:
25:     Vector compressedVectors = [];
26:     compressedVectors = cCore.compress(box);
27:     numBitsRequiredCurrentBlock =
28:        TestAddToBlock(compressedBlocks[currentBlock],compressedVectors);
29:
30:     if(numBitsRequiredCurrentBlock <= maxBitsPerBlock)
31:     {
32:        CommitToBlock(compressedBlocks[currentBlock], compressedVectors);
33:        numBoxesEncoded++;
34:     }
35:     else
36:        break;
37:  }
38:
39:  FinalizeBlock(compressedBlocks[currentBlock++]);
40: }
41:
42: if(numBoxesEncoded - blockStartIndex > 0)
43:   FinalizeBlock(compressedBlocks[currentBlock++]);
```

TABELLE C

[0472] In einer Ausführungsform tritt die Komprimierung eines AABB in den folgenden Stufen auf: (1) Quantisierung, (2) Transformation und (3) Vorhersage/Delta-Codierung.

1. Quantisierung

[0473] In einer Ausführungsform werden die Gleitkomma-AABB-Werte zuerst durch Verwenden einer festen Anzahl von Bits pro Achse zu einer vorzeichenlosen ganzzahligen Repräsentation quantisiert. Dieser Quantisierungsschritt kann auf eine Vielzahl von Weisen durchgeführt werden. Zum Beispiel werden in einer Implementierung für jede Achse i die folgenden Werte bestimmt:

$$L_i = S_{max,i} - S_{min,i}$$

$$N_{B,i} = 2^{NQ_i}$$

$$VU_{min,i} = \left(VF_{min,i} - S_{min,i}\right) / L_i \times N_{B,i}$$

$$VU_{max,i} = \left(VF_{max,i} - S_{min,i}\right) / L_i \times N_{B,i}$$

wobei S_{min} und S_{max} die Minimum- und Maximumkoordinaten des gesamten Geometriesatzes sind, für die eine BVH aufgebaut werden soll, $N_{B,i}$ die Anzahl von Zellen in dem quantisierten Gitter in der i-ten Achse ist, NQ_i dem Wert in Tabelle B entspricht, VU_{min} und VU_{max} die Minimum- und Maximumkoordinaten des quantisierten AABB sind, VF_{min} und VF_{max} die Minimum- und Maximumkoordinaten des ursprünglichen Gleitkomma-AABB sind, und der Subskript i eine gegebene Achse ($i \in \{x,y,z\}$) bezeichnet. Da eine Gleitkommaberechnung Fehler einführen kann, sollten die Zwischenwerte auf- oder abgerundet werden, um die Werte von VU_{min} zu minimieren, und die Werte von VU_{max} zu maximieren. Die Werte können auch in eine ganze Zahl umgewandelt und auf den gültigen Bereich beschränkt werden, um zu gewährleisten, dass sich ein hieb- und stichfester AABB innerhalb des AABB des gesamten Geometriesatzes befindet.

[0474] S_{min} und S_{max} könnten auch den Umfang eines Teilsatzes der Geometrie (z. B. eines Unterbaums innerhalb einer größeren BVH) repräsentieren. Dies könnte beispielsweise bei einem komprimierten Mehrebenen-Build gemäß **Fig. 63** auftreten.

2. Transformation

[0475] In einer Ausführungsform wird eine Transformationsstufe implementiert, in der Daten in eine Form transformiert werden, die sich besser für eine Komprimierung eignet. Obwohl eine Vielzahl von Transformationen verwendet werden kann, setzt eine Ausführungsform der Erfindung eine neuartige Transformation ein, die hierin als Position-Ausmaß-Transformation bezeichnet wird, die VU_{min} und VU_{max} zu einem einzelnen 6-dimensionalen (6D) Vektor pro Primitiv, VT, kombiniert, wie unten gezeigt:

$$E_x = VU_{max,x} - VU_{min,x} \quad E_y = VU_{max,y} - VU_{min,y} \quad E_2 = VU_{max,z} - VU_{max,z} - VU_{min,z}$$

$$V_T = \left(VU_{min,x}, VU_{min,y}, VU_{min,z}, E_x, E_y, E_z\right)$$

wobei $VU_{min}\{x,y,z\}$ und $\backslash VU_{max}\{x,y,z\}$ die Komponenten von VU_{min} bzw. VU_{max} sind. Im Wesentlichen ermöglicht diese Transformation, die Positions- und Ausmaß-/Größencharakteristiken des AABB separat in den verbleibenden Komprimierungsstufen zu behandeln. Wie erwähnt, können auch andere Transformationen verwendet werden.

3. Prädiktions-/Delta-Codierung

[0476] In einer Implementierung wird eine herkömmliche Delta-Codierungstechnik verwendet, um eine gute Komprimierungsleistung zu erreichen. In einer Ausführungsform wird der erste Vektor in jedem Komprimierungsblock als ein „Seed“-Vektor bezeichnet, und in dem AABB-Komprimierungsblock 6400 verbatim gespeichert, wie in **Fig. 64** gezeigt. Für nachfolgende Vektoren wird eine Laufdifferenz der Werte (d. h. Residuen 6402) gespeichert. Dies entspricht einem Prädiktionsschema, wobei die Prädiktion für den nächsten Eingabevektor in der Sequenz immer der vorherige Eingabevektor ist, und der Residuenwert die Differenz zwischen dem aktuellen und dem vorherigen Eingabevektor ist. Die Residuenwerte 6402 sind in dieser Ausführungsform also vorzeichenbehaftete Werte, was ein zusätzliches Vorzeichenbit erfordert. Verschiedene andere Prädiktions-/Delta-Codierungen können verwendet werden, wobei diese noch mit den der Erfindung zugrundeliegenden Prinzipien übereinstimmen.

[0477] Eine Ausführungsform speichert die Residuenwerte 6402 mit der Mindestanzahl an benötigten Bits, um die Komprimierung zu maximieren. Basierend auf der Größe der Residuenwerte am Ende der Residuen-Codierungsschritte wird eine gewisse Anzahl von Bits für jede der Vektordimensionen erforderlich sein, um den in dieser Dimension vorliegenden Wertebereich aufzunehmen.

[0478] Die Anzahl an benötigten Bits wird in einem Residuengrößenvektor (RSV: Residual Size Vector) gespeichert, wie in den Metadaten 6401 in **Fig. 64** veranschaulicht ist. Der RSV ist für einen gegebenen Komprimierungsblock 6400 festgelegt, sodass alle Werte in einer gegebenen Dimension eines bestimmten Blocks dieselbe Anzahl von Bits für ihre Residuen 6402 verwenden.

[0479] Der in jedem Element des RSV gespeicherte Wert ist einfach die minimale Anzahl von Bits, die benötigt wird, um den gesamten Bereich von Residuenwerten in der Dimension als eine vorzeichenbehaftete Zahl zu speichern. Während ein gegebener AABB-Komprimierungsblock (d. h. die Zeilen 18 - 37 von Tabelle C) komprimiert wird, wird ein laufendes Maximum der Anzahl von Bits, die benötigt werden, um alle bisher betrachteten Vektoren aufzunehmen, aufrechterhalten. Der RSV wird für jeden neu hinzugefügten AABB (d. h. CommitToBlock, Zeile 32 von Tabelle C) bestimmt, und in den Metadaten der Komprimierungsblöcke gespeichert.

[0480] Um zu prüfen, ob ein neuer AABB in den aktuellen Block passen wird (d. h. TestAddToBlock, Zeile 28 von Tabelle C und Operation 6503 in **Fig. 65**), wird der erwartete neue RSV, der durch Hinzufügen des neuen AABB auftreten würde, berechnet, der erwartete RSV-Vektor summiert, und dann dieser Wert mit der Gesamtanzahl von Residuen, die in dem Block existieren würden, wenn der neue AABB hinzugefügt würde, multipliziert. Falls dieser Wert innerhalb des Budgets liegt, das zum Speichern von Residuen verfügbar ist (d. h. kleiner oder gleich der Gesamtblockgröße minus der Größe der Metadaten 6401), kann er zu dem aktuellen Block hinzugefügt werden. Falls nicht, wird ein neuer Komprimierungsblock initialisiert.

Entropie-Codierung

[0481] Eine Ausführungsform der Erfindung beinhaltet einen zusätzlichen Schritt zur AABB-Residuenberechnung, der eine Entropie-Codierung der Residuen nach der Prädiktions-/Delta-Codierung beinhaltet. Die der Erfindung zugrundeliegenden Prinzipien sind nicht auf diese bestimmte Implementierung beschränkt.

Vorsortier-/Umordnungsfähigkeit

[0482] Als ein optionaler Vorprozess kann die Eingabegeometrie sortiert/umgeordnet werden, um räumliche Kohärenz zu verbessern, was die Komprimierungsleistung verbessern kann. Das Sortieren kann auf eine Vielzahl von Weisen durchgeführt werden. Eine Möglichkeit, dies zu erreichen, ist die Verwendung einer Morton-Code-Sortierung. Eine solche Sortierung wird bereits als wichtiger Schritt in anderen BVH-Buildern verwendet, um räumliche Kohärenz in der Geometrie vor Extraktion einer Hierarchie zu fördern.

[0483] Die komprimierten AABBs können in einer beliebigen gewünschten Reihenfolge geschrieben werden, wobei es jedoch dann, wenn die AABBs umgeordnet/sortiert werden, notwendig ist, ein zusätzliches Array von ganzen Zahlen zu speichern, das die Sortierungsreihenfolge aufzeichnet. Das Array besteht aus einem einzelnen ganzzahligen Index pro Primitiv. Der Build kann mit dem primären Index fortfahren, der verwendet wird, um die umgeordnete Liste von Primitiven zu referenzieren. Wenn die ursprüngliche Primitiv-ID benötigt wird (wie etwa wenn die Inhalte eines Leaf-Knotens geschrieben werden), muss der primäre Index verwendet werden, um die ursprüngliche Primitiv-ID in dem zusätzlichen Array nachzuschlagen, um sicherzustellen, dass der Baum die ursprüngliche Eingabegeometrieliste korrekt referenziert.

II. AABB-Dekomprimierung

[0484] In einer Ausführungsform wird eine Dekomprimierung des AABB für jeweils einen gesamten AABB-Komprimierungsblock 6400 durchgeführt. Die Residuendaten werden zuerst rekonstruiert, indem die Metadaten 6401 des Komprimierungsblocks 6400 untersucht werden, und die gespeicherten Residuen basierend auf diesen Informationen interpretiert werden (z. B. Hinzufügen der Abstandswerte zu dem Seed-Vektor und früheren Residuenwerten in der Sequenz). Dann wird die Inverse jeder der AABB-Komprimierungsstufen durchgeführt, um die durch den Komprimierungsblock repräsentierten Einfachpräzision-Gleitkomma-AABB zu dekomprimieren.

[0485] Eine Ausführungsform implementiert eine Variation des Dekomprimierungsschritts bei BVH-Buildern, die präzisionsreduzierte Konstruktionstechniken einsetzen, die auf eine komprimierte Hierarchieausgabe ausgerichtet sind. Solche präzisionsreduzierten Builder sind in der gleichzeitig anhängigen Anmeldung mit dem Titel „An Architecture for Reduced Precision Bounding Volume Hierarchy Construction“, Seriennummer 16/746,636, eingereicht am 17. Januar 2020, die an den Abtretungsempfänger der vorliegenden Anmeldung abgetreten wurde, beschrieben. Ein präzisionsreduzierter Builder führt viel seiner Berechnung in einem präzi-

sionsreduzierten ganzzahligen Raum aus. Infolgedessen richtet eine Ausführungsform der Erfindung den Quantisierungsschritt der hierin beschriebenen AABB-Residuenberechnung mit der Quantisierung aus, die in dem präzisionsreduzierten Builder eingesetzt wird. Die AABBs können dann nur auf eine ganze Zahl dekomprimiert werden, die mit dem Koordinatenraum eines jeglichen Knotens, der gegenwärtig durch den präzisionsreduzierten Builder verarbeitet wird, ausgerichtet ist. Eine ähnliche Variation kann mit einem Builder implementiert werden, der keine komprimierte Hierarchie ausgibt, sondern eine Quantisierung von Vertices durchführt.

III. Indexkomprimierung

[0486] In einer Ausführungsform der Erfindung wird das Index-Array in ein Array von Indexkomprimierungsblöcken komprimiert. **Fig. 66** veranschaulicht eine Ausführungsform eines Indexkomprimierungsblocks 6610, der Metadaten 6603 und Index-Residuen 6602 umfasst. Das Index-Array unterscheidet sich von dem AABB-Array, da es erneut komprimiert werden muss, da die Indizes während des Build-Prozesses partitioniert/umgeordnet werden.

[0487] In vielen herkömmlichen BVH-Buildern werden Indizes als vorzeichenlose ganze Zahlen repräsentiert, im Allgemeinen mit einem Index pro Primitiv. Das Index-Array hat die Aufgabe, auf Primitiv-AABBs zu zeigen. Jedem AABB/Primitiv kann eine feste Größe in Speicher zugewiesen werden. Es ist daher möglich, zufällig auf ein beliebiges Primitiv p oder einen beliebigen AABB a in den Arrays zuzugreifen. Wenn AABB-Komprimierung jedoch zu einer variablen Anzahl von AABBs pro Cachezeile führt, ist es nicht einfach, den AABB-Komprimierungsblock, der ein gegebenes Primitiv speichert, nach der Komprimierung zu bestimmen. Die Speicherung herkömmlicher Indizes ist daher nicht kompatibel mit den hierin beschriebenen AABB-Komprimierungsblöcken.

[0488] In einer Ausführungsform der Erfindung ermöglichen die Indizierungstechniken, die verwendet werden, um den Ort von Primitiv-AABBs zu identifizieren, auch eine Komprimierung der Indizes selbst. Zwei neue Techniken werden unten als Block-Offset-Indizierung (Block Offset Indexing - BOI) und Hierarchische Bitvektorindizierung (Hierarchical Bit-Vector Indexing - HBI) bezeichnet. Diese Indizierungsimplementierungen können in den verschiedenen Ausführungsformen der Erfindung allein oder in Kombination verwendet werden. Außerdem können beide Indizierungstechniken als Teil eines Mehrebenen-Build gemäß **Fig. 63** verwendet werden und beide Arten von Indizes können auch als Teil desselben BVH-Build verwendet werden. Diese Indizierungstechniken ermöglichen, dass der BVH-Build auf eine ähnliche Weise wie ein herkömmlicher BVH-Build, jedoch mit komprimierten Repräsentationen sowohl des AABB als auch der entsprechenden Index-Arrays verläuft.

Globale Indexkomprimierungskonstanten

[0489] Indexkomprimierung verwendet einen Satz globaler Indexkomprimierungskonstanten, die für alle Indexkomprimierungsblöcke gelten. Beide der unten beschriebenen Indexkomprimierungsschemen nutzen gemeinsam dieselben globalen Konstanten, die in Tabelle D unten zusammengefasst sind.

TABELLE D

Konstante	**Beschreibung**
IndexCompressionBlockSizeBytes	Größe in Bytes eines Indexkomprimierungsblocks. Dieser Wert wird typischerweise auf eine bestimmte Anzahl von Cachezeilen ausgerichtet sein.
maxIndicesPerBlock	Die maximale Anzahl der in einem Indexkomprimierungsblock erlaubten Indizes. Dieser Wert bestimmt die Anzahl von Bits, die benötigt werden, um die Anzahl von Indizes, die durch einen gegebenen Block repräsentiert werden, zu speichern.

Block-Offset-Indizierung

[0490] In einer Block-Offset-Indizierung (BOI) wird der reguläre Einzelganzzahlindex zu einer Struktur geändert, die zwei ganze Zahlen enthält, von denen eine den Komprimierungsblock 6400 identifiziert, und eine

einen Offset zum Identifizieren der Primitiv-AABB-Daten innerhalb des Komprimierungsblocks 6400 umfasst. Eine Ausführungsform der neuen Datenstruktur wird gemäß folgender Codesequenz erzeugt:

```
struct blockOffsetIndex
{
    uint blockIdx;
    uint blockOffset;
}
```

Hier speichert blockIdx einen Index zu einem AABB-Komprimierungsblock, und blockOffset referenziert einen spezifischen Primitiv-AABB innerhalb des Blocks (d. h. blockIdx in Kombination mit blockOffset stellt die Adresse des Primitiv-AABB bereit). Diese Informationen reichen aus, um einen bestimmten AABB innerhalb seines Komprimierungsblocks während eines Build vollständig zu referenzieren.

[0491] In einer Ausführungsform wird eine dieser Strukturen für jedes Primitiv in dem BVH-Build erzeugt, sodass die Größe der Liste vorhersagbar ist. Bei einer variablen Anzahl von AABBs pro AABB-Komprimierungsblock wird es jedoch eine variable Anzahl dieser Indexstrukturen für jeden dieser Komprimierungsblöcke geben (z. B. werden nicht alle möglichen Werte von blockOffset für jeden AABB-Komprimierungsblock existieren). Um das Array von Block-Offset-Indizes korrekt zu initialisieren, ist es daher notwendig, auf das blockOffsets-Array (siehe z. B. die Codesequenz in Tabelle C) zu verweisen, von dem die Anzahl von Primitiven in jedem AABB-Komprimierungsblock entweder gleichzeitig mit der AABB-Komprimierung oder als ein Nachprozess zu dieser bestimmt werden kann. Sobald initialisiert, können die Block-Offset-Indizes im Wesentlichen auf dieselbe Weise behandelt werden, wie herkömmliche Indizes, die in herkömmlichen BVH-Buildern zu finden sind.

[0492] Einzelganzzahlindizes, die in herkömmlichen BVH-Buildern verwendet werden, haben typischerweise eine Größe von 4 Byte. In einer Ausführungsform werden 26 Bit für blockIdx verwendet, und 6 Bit werden für blockOffset verwendet. In einer alternativen Ausführungsform werden kleinere Anzahlen von Bits für jede Variable verwendet, um den Gesamtspeicherumfang zu reduzieren. Da in einer Ausführungsform eine feste Größe für blockOffset gewählt werden muss, beschränkt dies die maximale Anzahl an Primitiven pro AABB-Komprimierungsblock. Bei 6 Bit können pro AABB-Komprimierungsblock maximal 64 Primitive repräsentiert werden.

[0493] Der verbleibende Punkt, um Block-Offset-Indizierung zu behandeln, ist, wie eine Komprimierung erreicht werden kann. Block-Offset-Indizes werden delta-codiert, und der Reihe nach in Indexkomprimierungsblöcke gepackt. Jeder Block wird mit so vielen Indizes wie möglich gepackt, und ein neuer Indexkomprimierungsblock wird jedes Mal dann gestartet, wenn der vorherige die Kapazität erreicht. Dies wird auf eine sehr ähnliche Weise wie die AABB-Komprimierungsblöcke (wie in Tabelle C gezeigt) ausgeführt, was zu einer variablen Anzahl von Indizes pro Indexkomprimierungsblock führt.

[0494] Fig. 66 veranschaulicht ein Beispiel für einen Block-Offset-Indexkomprimierungsblock 6610, der Metadaten 6603 umfasst, die die Anzahl von Indizes zusätzlich zu einem Residuengrößenvektor und einem Seed-Vektor identifizieren. In einer Ausführungsform wird eine Zweikanal-Codierung für die Indexresiduen 6602 verwendet, wobei die blockIdx- und blockOffset-Werte separat delta-komprimiert werden. Ähnlich den AABB-Komprimierungsblöcken speichert der Indexkomprimierungsblock 6610 eine Anzeige der Anzahl von Indizes in dem Block, die Anzahl von Bits für die Residuen (als den Residuengrößenvektor), und einen Seed-Vektor, der einen ersten Seed-Vektor für blockIdx und einen zweiten Seed-Vektor für blockOffset umfasst. Die Indexresiduenwerte 6602 umfassen ein Differenzwertepaar, das aus der Komprimierung resultiert. Zum Beispiel kann ein Indexresiduenwert einen ersten Differenzwert, der eine Differenz zwischen dem aktuellen Eingabe-blockIdx-Wert und einem vorherigen Eingabe-blockIdx-Wert repräsentiert, und einen zweiten Differenzwert, der eine Differenz zwischen dem aktuellen Eingabe-blockOffset-Wert und einem vorherigen Eingabe-blockOffset-Wert repräsentiert, umfassen. Die ersten blockIdx- und blockOffset-Werte in der Sequenz sind in dem seedVector-Feld verbatim gespeichert, das den Vektor repräsentiert, von dem der erste Residuenwert berechnet wird.

Hierarchische Bitvektorindexierung

[0495] Eine Ausführungsform der Erfindung verwendet eine andere Primitivindexkomprimierungstechnik, die als hierarchische Bitvektorindizierung (HBI) bezeichnet wird, die allein oder in Kombination mit Block-Offset-

Indizierung (BOI) verwendet werden kann. HBI unterscheidet sich sowohl von herkömmlichen ganzzahligen Indizes als auch von BOI darin, dass ein einzelner HBI-Index mehrere Primitive auf einmal referenzieren kann. Tatsächlich kann ein HBI-Index bis zu einen gesamten AABB-Komprimierungsblock referenzieren.

[0496] Eine erweiterte Struktur dieses Indextyps ist in den **Fig. 67AB** gezeigt. Jeder HBI-Index 6700 besteht aus zwei Elementen. blockIdx 6708 zeigt auf einen gegebenen AABB-Komprimierungsblock, der demselben Zweck wie das entsprechende Element in Block-Offset-Indizes dient. Die zweite Komponente ist ein Bitvektor 6701, der eine Anzahl von Bits aufweist, die gleich der maximalen Anzahl von AABBs ist, die in einem AABB-Komprimierungsblock erlaubt sind (d. h. maxAABBsPerBlock). Jedes Bit in dem Bitvektor 6701 gibt an, ob das entsprechende Element in dem AABB-Komprimierungsblock durch diesen Index referenziert wird. Falls zum Beispiel das dritte Bit in dem Bitvektor eine ‚1' ist, bedeutet dies, dass der dritte AABB/das dritte Primitiv des AABB-Komprimierungsblocks durch den HBI-Index referenziert wird. Falls das Bit 0' ist, dann wird dieser AABB/dieses Primitiv nicht referenziert.

[0497] Im Gegensatz zu BOI-Indizes wird ein einzelner HBI-Index 6700 pro AABB-Komprimierungsblock erzeugt, wenn das Array initialisiert wird. Die blockIdx-Werte 6708 werden startend bei 0 auf aufsteigende Werte gesetzt, und die anfänglichen Bitvektoren 6701 werden alle auf 1 gesetzt. Bei Partitionierung in dem Top-Down-Builder kann der Index, falls alle Primitive, die durch einen gegebenen HBI-Index 6700 referenziert werden, alle auf derselben Seite der Aufteilungsebene liegen, so wie sie sind in eine Seite der Liste partitioniert werden, ähnlich einem herkömmlichen ganzzahligen Index. Falls der HBI-Index 6700 jedoch Primitive auf beiden Seiten einer Aufteilungsebene referenziert, dann muss der Index in zwei neue HBI-Indizes aufgeteilt werden, wobei ein HBI-Index in jeder der zwei neuen Indexunterlisten platziert wird, die der linken und rechten Partition entsprechen. Um einen HBI-Index aufzuteilen, wird der Index dupliziert, und die Bitvektoren 6701 werden in jeder Kopie des Index aktualisiert, um die Primitiven wiederzugeben, die die zwei neuen Indizes referenzieren. Dies bedeutet, dass die Anzahl von HBI-Indizes in dem Array wachsen kann, und die Duplikation von Indizes in gewisser Weise der Handhabung räumlicher Aufteilungen in manchen herkömmlichen BVH-Buildern ähnelt. Eine einfache Art, die potenziell wachsende Liste zu handhaben, ist es, einfach eine „Worst-Case"-Speichermenge zuzuweisen.

[0498] HBI-Indizes 6700 können durch Verwenden von Delta-Komprimierung an den blockIdx-Komponenten 6708 in Indexkomprimierungsblöcke gepackt werden. Zusätzlich bieten HBI-Indizes auch eine hierarchische Komprimierungsmöglichkeit, von der sich ihr Name ableitet. Bei einem jeglichen HBI-Index, der keine Aufteilungsebene überspannt, sind alle Elemente seines Bitvektors gleich ‚1'. Wenn HBI-Indizes in Indexkomprimierungsblöcke gepackt werden, kann ein Einzelbit-Flag (hierin manchmal als Bitvektorbelegungs-Flag bezeichnet) verwendet werden, um anzugeben, dass der gesamte Bitvektor nur aus 1en besteht („all 1s"). Ein Wert von ‚0' gibt an, dass der Bitvektor verbatim in dem Block gespeichert ist, und ein Wert von ‚1' gibt an, dass der Vektor nur aus 1en besteht und somit überhaupt nicht gespeichert ist (mit Ausnahme des Flags). HBI-Indizes leiten somit eine Komprimierung anhand von zwei Techniken ab: Delta-Codierung und hierarchische Bitvektoren. Wie BOI-Indizes werden HBI-Indizes auf eine sehr ähnliche Weise wie AABB-Komprimierungsblöcke auch in Komprimierungsblöcke gepackt. Um dies korrekt durchzuführen, muss die Komprimierungsoperation auch die Indexbitvektoren überwachen, um zu entscheiden, ob jegliche Bitvektoren verbatim gespeichert werden müssen, und dies bei der erforderlichen Größe für den Block berücksichtigen.

[0499] **Fig. 67B** zeigt, wie eine Sequenz von HBI-Indizes in einen HBI-Komprimierungsblock 6710 codiert werden kann, der Residuendaten 6704 und Metadaten 6701 beinhaltet. In dieser Ausführungsform beinhalten die Residuendaten blockIdx-Residuen 6702 und hierarchische Zugehörigkeitsbitvektoren 6703. HBI-Indizierung soll nahe dem oberen Ende der Hierarchie oder nahe dem oberen Ende von Unterbäumen arbeiten, für die die AABB-Komprimierungsblöcke kürzlich neu komprimiert wurden, gemäß einer Mehrebenen-Build-Situation von **Fig. 63**. Dies liegt daran, dass HBI-Indizes im Vergleich zu anderen Indizierungsverfahren durch Änderung der räumlichen Kohärenz in den AABB-Komprimierungsblöcken direkter beeinflusst werden. Obwohl HBI-Indizes Komprimierung bereitstellen, kann die „Worst-Case"-Situation tatsächlich in einer Expansion der Indexdaten (bis zu einer Obergrenze) resultieren. Ein Übergang zu Block-Offset-Indizierung (BOI) oder herkömmlichen ganzzahligen Indizes während des Build kann diese Situation vermeiden, und kann effektiver sein, falls in jüngster Zeit keine Neukomprimierung durchgeführt wurde.

Indexübergänge zwischen Build-Ebenen

[0500] Falls entweder BOI- oder HBI-Indizes in einem BVH-Build verwendet werden, und der Build zu einer anderen Stufe übergeht (gemäß einer Mehrebenen-Build-Situation von **Fig. 63**), ist es notwendig, die Indizes

zu einer Form zu decodieren, die für die nächste Build-Stufe geeignet ist. Zum Beispiel wird es in dem einfachen Fall des Verwendens von Block-Offset-Indizes für die oberen Ebenen des Baums und des Übergangs von einer komprimierten AABB-Repräsentation zu einer herkömmlichen AABB-Repräsentation notwendig sein, die Block-Offset-Indizes in herkömmliche ganzzahligen Indizes zu decodieren. Die Block-Offset-Indizes können nach dem Übergang verworfen werden.

[0501] Ein ähnlicher Übergang muss für die HBI-Indizierung und für einen Übergang zwischen zwei komprimierten Build-Ebenen auftreten, die unterschiedliche AABB-Komprimierungskonfigurationen einsetzen. Der Übergangsprozess ist relativ einfach, da sowohl Block-Offset-Indizes als auch hierarchische Bitvektorindizes alternative Codierungen derselben zugrundeliegenden Informationen repräsentieren, und auch immer zu herkömmlichen ganzzahligen Indizes decodiert werden können, die den ursprünglichen Satz von Primitiven referenzieren.

Partitionieren von komprimierten Index-Arrays

[0502] In Top-Down-BVH-Build-Vorgängen ist es notwendig, die Liste ganzzahliger Indizes zu partitionieren/-sortieren, um während des Build zu rekurrieren, und damit die Indexordnung die Baumstruktur wiedergibt. In herkömmlichen BVH-Buildern ist dieser Schritt unkompliziert, da die Indizes eine regelmäßige, unkomprimierte Datenstruktur sind. Die hierin beschriebenen Ausführungsformen der Erfindung resultieren jedoch in einer neuen Herausforderung, die darin besteht, dass eine Liste von Indexkomprimierungsblöcken anstelle einer Liste von Indizes partitioniert werden muss. Außerdem ist eine Vorhersage der Anzahl von Blöcken erst möglich, nachdem alle Indizes komprimiert sind. Da die Indizes nach jedem Partitionierungsschritt erneut komprimiert werden, besteht diese Herausforderung während des gesamten Build.

[0503] Obwohl es nicht möglich ist, die Größe des komprimierten Index-Arrays im Voraus vorherzusagen, kann der maximalen Größe des Arrays eine Obergrenze gesetzt werden, falls die Anzahl an Indizes, die komprimiert werden sollen, bekannt ist. In einem Top-Down-BVH-Builder ist die Anzahl von Indizes in jedem von einer Knotenpartition resultierenden Index-Subarray typischerweise bekannt, bevor die Partitionierung auftritt, sodass eine Obergrenze für beide Subarrays bei jedem Partitionierungsschritt abgeleitet werden kann.

[0504] In dem Fall von BOI tritt die maximale Größe des Arrays auf, wenn keine Komprimierung der Indizes durch Delta-Komprimierung erreicht wird. Durch Berücksichtigen der Größe der Metadaten für einen Block ist es möglich, die maximale Anzahl von Blöcken und somit die maximale Größe in Bytes vorherzusagen.

[0505] In dem Fall einer HBI-Indizierung tritt die maximale Größe auf, wenn keine Delta-Komprimierung des blockIdx erreicht wird, und die HBI-Indizes werden in einem solchen Ausmaß aufgeteilt, dass jeder HBI-Index nur ein einzelnes Primitiv repräsentiert (nur ein Bit ist in jedem Indexbitvektor gesetzt). Durch Berücksichtigen sämtlicher Metadaten, einschließlich des zusätzlichen Bits, das für die erste Ebene des hierarchischen Bitvektors verwendet wird (das Bitvektorbelegungs-Flag), kann die maximale Anzahl von Blöcken und somit die maximale Größe in Bytes für eine gegebene Anzahl von Primitiven berechnet werden.

[0506] Da der Größe des Arrays eine Obergrenze gesetzt werden kann, wird eine einfache Technik verwendet, um das Indexkomprimierungsblock-Array durch Verwenden eines Array-Paares zu partitionieren. Beide Arrays sind auf die maximal mögliche Größe basierend auf dem Indextyp bemessen, wie zuvor in diesem Abschnitt diskutiert. Zu Beginn des Build wird ein Satz anfänglicher Indizes in eines der Arrays in dem Paar geschrieben. Für jede Ebene werden Blöcke von einem Array gelesen, interpretiert, und neu komprimierte Blöcke in das zweite Array, die die partitionierten Indizes wiedergeben, hinaus geschrieben. Bei der Rekursion können die Rollen jedes der Arrays vertauscht werden, wobei stets von dem Array gelesen wird, das gerade geschrieben wurde. Da sich die Reihenfolge der Indizes ändert, um die Partitionierung wiederzugeben, werden die Index-Arrays kontinuierlich neu komprimiert.

[0507] Da die maximale Anzahl von Blöcken in einer Partition vorhergesagt werden kann, kann jedes Subarray, das aus einer Partition resultiert, in eine Position des anderen Arrays geschrieben werden, sodass immer die maximale Größe aufgenommen werden kann. Dies kann effektiv zu „Lücken“ in den Arrays führen, wobei dennoch eine Bandbreitenkomprimierung erzielt wird. Falls Indizes auf diese Weise partitioniert werden, kann der BVH-Builder den Start/das Ende der aktuellen Build-Aufgabe hinsichtlich der Indexkomprimierungsblöcke, die seine Primitiven referenzieren, sowie die Anzahl von Primitiven in der Build-Aufgabe verfolgen.

Räumliche Aufteilungen

[0508] Eine weit verbreitete Technik zur Verbesserung der BVH-Traversierungseffizienz ist in einigen Fällen die Verwendung räumlicher Aufteilungen. Da die AABBs nicht auf jeder Ebene des Build neu komprimiert werden, ist es schwierig, räumliche Aufteilungen, die während des Build selbst auftreten (wie in einigen verwandten Arbeiten zu sehen ist), in das Komprimierungsschema aufzunehmen. Das Komprimierungsschema sollte jedoch mit einem Voraufteilungsansatz gemäß anderen früheren Designs kompatibel sein. Solche Schemen liefern einen Satz von AABBs an den BVH-Build, und erfordern im Allgemeinen wenig oder keine Modifikation an dem Build selbst.

[0509] Eine Möglichkeit, diese Voraufteilungsschemen mit den Ausführungsformen der Erfindung zu kombinieren, besteht darin, das Array von Gleitkomma-AABBs im Voraus vorzubereiten, einschließlich aller aufgeteilten Primitiven (anstatt sie gemäß Zeile 23 von Tabelle C zu berechnen), und ein Array von IDs, die sie mit den ursprünglichen Primitiven zurück verknüpfen, beizubehalten. Dann könnten die BOI- oder HBI-Indizes oder herkömmliche Indizes verwendet werden, um diese AABBs während des Build zu referenzieren, und sie bei Bedarf (wie etwa beim Schreiben von Leaf-Knoten) zurück mit den ursprünglichen Primitiven zu verknüpfen.

[0510] **Fig. 68** veranschaulicht eine Ausführungsform einer Raytracing-Engine 8000 einer GPU 2505 mit Komprimierungshardwarelogik 6810 und Dekomprimierungshardwarelogik 6808 zum Ausführen der hierin beschriebenen Komprimierungs- und Dekomprimierungstechniken. Es ist jedoch anzumerken, dass **Fig. 68** viele spezifische Einzelheiten beinhaltet, die nicht erforderlich sind, um mit den zugrundeliegenden Prinzipien der Erfindung übereinzustimmen.

[0511] Es ist BVH-Builder 6807 gezeigt, der eine BVH basierend auf einem aktuellen Satz von Primitiven 6806 (z. B. assoziiert mit einem aktuellen Grafikbild) konstruiert. In einer Ausführungsform arbeitet die BVH-Komprimierungslogik 6810 mit dem BVH-Builder 6807 zusammen, um gleichzeitig die zugrundeliegenden Daten zu komprimieren, die durch den BVH-Builder 6807 verwendet werden, um eine komprimierte Version der Daten 6812 zu erzeugen. Insbesondere beinhaltet die Komprimierungslogik 6810 einen Begrenzungsrahmenkomprimierer 6825 zum Erzeugen von AABB-Komprimierungsblöcken 6400, und einen Indexkomprimierer 6826 zum Erzeugen von Indexkomprimierungsblöcken 6610, wie hierin beschrieben. Obwohl in **Fig. 68** als eine separate Einheit veranschaulicht, kann die Komprimierungslogik 6810 innerhalb des BVH-Builder 6807 integriert sein. Umgekehrt ist ein BVH-Builder zur Erfüllung der zugrundeliegenden Prinzipien der Erfindung nicht erforderlich.

[0512] Wenn eine Systemkomponente unkomprimierte Daten 6814 erfordert (wie z. B. der BVH-Builder 6807), implementiert die Dekomprimierungslogik 6808 die hierin beschriebenen Techniken, um die komprimierten Daten 6812 zu dekomprimieren. Insbesondere dekomprimiert der Indexdekomprimierer 6836 die Indexkomprimierungsblöcke 6610, und der Begrenzungsrahmendekomprimierer 6835 dekomprimiert die AABB-Komprimierungsblöcke 6400, um unkomprimierte AABBs der unkomprimierten Daten 6814 zu erzeugen. Andere Systemkomponenten können dann auf die unkomprimierten Daten 6814 zugreifen.

[0513] Die verschiedenen in **Fig. 68** veranschaulichten Komponenten können in Hardware, Software oder einer beliebigen Kombination davon implementiert sein. Zum Beispiel können gewisse Komponenten auf einer oder mehreren der Ausführungseinheiten 4001 ausgeführt werden, während andere Komponenten, wie etwa die Traversierungs-/Überschneidungseinheit 6803, in Hardware implementiert sein können.

[0514] Außerdem können die Primitiven 6806, die komprimierten Daten 6812 und die unkomprimierten Daten 6814 in einem lokalen Speicher/Cache 6898 und/oder einem Systemspeicher (nicht gezeigt) gespeichert werden. Zum Beispiel kann in einem System, das gemeinsam genutzten virtuellen Speicher (Shared Virtual Memory - SVM) unterstützt, der virtuelle Speicherraum über einen oder mehrere lokale Speicher und den physischen Systemspeicher hinweg abgebildet werden. Wie oben erwähnt, können die BVH-Komprimierungsblöcke basierend auf der Größe von Cachezeilen in der Cache-Hierarchie erzeugt werden (z. B., um einen oder mehrere Komprimierungsblöcke pro Cachezeile einzupassen).

EINRICHTUNG UND VERFAHREN ZUR KOMPRIMIERUNG EINES VERSCHOBENEN MESH

[0515] Eine Ausführungsform der Erfindung führt ein Pfad-Tracing aus, um fotorealistische Bilder durch Verwenden von Raytracing für Sichtbarkeitsabfragen zu rendern. In dieser Implementierung werden Strahlen von einer virtuellen Kamera geworfen, und durch eine simulierte Szene verfolgt. Ein zufälliges Sampling wird

dann durchgeführt, um inkrementell ein endgültiges Bild zu berechnen. Das zufällige Sampling beim Pfad-Tracing veranlasst, dass Rauschen in dem gerenderten Bild erscheint, das entfernt werden kann, indem es ermöglicht wird, mehrere Samples zu erzeugen. Die Samples können bei dieser Implementierung Farbwerte sein, die von einem einzelnen Strahl resultieren.

[0516] In einer Ausführungsform sind die für Sichtbarkeitsabfragen verwendeten Raytracing-Operationen auf Hüllkörperhierarchien (BVHs) (oder eine andere hierarchische 3D-Anordnung) angewiesen, die über die Szenenprimitiven (z. B. Dreiecke, Vierecke usw.) in einer Vorverarbeitungsphase erzeugt werden. Durch Verwenden einer BVH kann der Renderer schnell den nächstgelegenen Überschneidungspunkt zwischen einem Strahl und einem Primitiv bestimmen.

[0517] Beim Beschleunigen dieser Strahlabfragen in Hardware (wie z. B. mit der hierin beschriebenen Traversierungs-/Überschneidungsschaltungsanordnung) können Speicherbandbreitenprobleme aufgrund der Menge abgerufener Dreiecksdaten auftreten. Glücklicherweise wird viel der Komplexität in modellierten Szenen durch Verschiebungsabbildung erzeugt, wobei eine gleichmäßige Basisoberflächenrepräsentation, wie etwa eine Unterteilungsoberfläche, durch Verwenden von Unterteilungsregeln fein tesselliert wird, um ein tesselliertes Mesh 6991 zu erzeugen, wie in **Fig. 69A** gezeigt. Auf jeden Vertex des fein tessellierten Mesh wird Verschiebungsfunktion 6992 angewendet, die typischerweise entweder nur entlang der geometrischen Normalen der Basisoberfläche oder in eine beliebige Richtung verschiebt, um ein Verschiebungs-Mesh 6993 zu erzeugen. Der Betrag der Verschiebung, der der Oberfläche hinzugefügt wird, ist in der Reichweite begrenzt; somit sind sehr große Verschiebungen von der Basisoberfläche selten.

[0518] Eine Ausführungsform der Erfindung komprimiert verschiebungsabgebildete Meshes effektiv durch Verwenden einer verlustbehafteten hieb- und stichfesten Komprimierung. Insbesondere quantisiert diese Implementierung die Verschiebung in Bezug auf ein grobes Basis-Mesh, das mit dem Basisunterteilungs-Mesh übereinstimmen kann. In einer Ausführungsform können die ursprünglichen Vierecke des Basisunterteilungs-Mesh durch Verwenden bilinearer Interpolation in ein Gitter mit derselben Genauigkeit wie die Verschiebungsabbildung unterteilt werden.

[0519] **Fig. 69B** veranschaulicht Komprimierungsschaltungsanordnung/-logik 6900, die ein verschiebungsabgebildetes Mesh 6902 gemäß den hierin beschriebenen Ausführungsformen komprimiert, um ein komprimiertes verschobenes Mesh 6910 zu erzeugen. In der veranschaulichten Ausführungsform erzeugt die Verschiebungsabbildungsschaltungsanordnung/-logik 6911 das verschiebungsabgebildete Mesh 6902 von einer Basisunterteilungsoberfläche. **Fig. 70A** veranschaulicht ein Beispiel, in dem eine Primitiv-Oberfläche 7000 fein tesselliert ist, um die Basisunterteilungsoberfläche 7001 zu erzeugen. Eine Verschiebungsfunktion wird auf die Vertices der Basisunterteilungsfläche 7001 angewendet, um eine Verschiebungsabbildung 7002 zu erzeugen.

[0520] Zurückkehrend zu **Fig. 69B** quantisiert in einer Ausführungsform ein Quantisierer 6912 das verschiebungsabgebildete Mesh 6902 relativ zu grobem Basis-Mesh 6903, um ein komprimiertes verschobenes Mesh 6910 zu erzeugen, das ein 3D-Verschiebungs-Array 6904 und Basiskoordinaten 6905 umfasst, die mit dem groben Basis-Mesh 6903 assoziiert sind. Beispielhaft und nicht einschränkend veranschaulicht **Fig. 70B** einen Satz von Differenzvektoren d1-d4 7022, die jeweils mit einem unterschiedlichen verschobenen Vertex v1-v4 assoziiert sind.

[0521] In einer Ausführungsform ist das grobe Basis-Mesh 7003 das Basisunterteilungs-Mesh 6301. Alternativ unterteilt ein Interpolierer 6921 die ursprünglichen Vierecke des Basisunterteilungs-Mesh durch Verwenden einer bilinearen Interpolation in ein Gitter mit derselben Genauigkeit wie die Verschiebungsabbildung.

[0522] Der Quantisierer 6912 bestimmt die Differenzvektoren d1-d4 7022 von jedem groben Basis-Vertex zu einem entsprechenden verschobenen Vertex v1-v4 und kombiniert die Differenzvektoren 7022 in dem 3D-Verschiebungs-Array 6904. Auf diese Weise wird das verschobene Gitter nur durch Verwenden der Koordinaten des Vierecks (Basiskoordinaten 6905) und des Arrays von 3D-Verschiebungsvektoren 6904 definiert. Es ist anzumerken, dass diese 3D-Verschiebungsvektoren 6904 nicht notwendigerweise mit den Verschiebungsvektoren übereinstimmen, die zum Berechnen der ursprünglichen Verschiebung 7002 verwendet werden, da ein Modellierungswerkzeug normalerweise das Viereck nicht durch Verwenden einer bilinearen Interpolation unterteilen würde, und komplexere Unterteilungsregeln anwenden würde, um gleichmäßige Oberflächen zum Verschieben zu erzeugen.

[0523] Wie in **Fig. 70C** veranschaulicht, werden Gitter von zwei benachbarten Vierecke 7090-7091 nahtlos zusammengefügt, da entlang des Rands 7092 beide Vierecke 7090-7091 genau dieselben Vertex-Orte v5-v8 evaluieren werden. Da auch die entlang der Kante 7092 gespeicherten Verschiebungen für benachbarte Vierecke 7090-7091 identisch sind, wird die verschobene Oberfläche keinerlei Risse haben. Diese Eigenschaft ist signifikant, da dies insbesondere bedeutet, dass die Genauigkeit der gespeicherten Verschiebungen für eine gesamtes Mesh beliebig reduziert werden kann, was in einem verbundenen verschobenen Mesh mit niedrigerer Qualität resultiert.

[0524] In einer Ausführungsform werden Gleitkommazahlen mit halber Präzision verwendet, um die Verschiebungen zu codieren (z. B. 16-Bit-Gleitkommawerte). Alternativ oder zusätzlich wird eine gemeinsam genutzte Exponentenrepräsentation verwendet, die nur einen Exponenten für alle drei Vertex-Komponenten und drei Mantissen speichert. Ferner können, da das Ausmaß der Verschiebung normalerweise recht gut begrenzt ist, die Verschiebungen eines Mesh durch Verwenden von Festpunktkoordinaten codiert werden, die durch eine Konstante skaliert sind, um einen ausreichenden Bereich zum Codieren aller Verschiebungen zu erhalten. Während eine Ausführungsform der Erfindung bilineare Patches als Basis-Primitiven unter Verwendung von nur flachen Dreiecken verwendet, verwendet eine andere Ausführungsform Dreieckspaare, um jedes Viereck zu handhaben.

[0525] Ein Verfahren gemäß einer Ausführungsform der Erfindung ist in **Fig. 71** veranschaulicht. Das Verfahren kann auf den hierin beschriebenen Architekturen implementiert werden, ist aber nicht auf irgendeine spezielle Prozessor- oder Systemarchitektur beschränkt.

[0526] Bei 7101 wird ein verschiebungsabgebildetes Mesh von einer Basisunterteilungsoberfläche erzeugt. Zum Beispiel kann eine Primitiv-Oberfläche fein tesselliert werden, um die Basisunterteilungsoberfläche zu erzeugen. Bei 7102 wird ein Basis-Mesh erzeugt oder identifiziert (wie z. B. das Basisunterteilungs-Mesh in einer Ausführungsform).

[0527] Bei 7103 wird eine Verschiebungsfunktion auf die Vertices der Basisunterteilungsoberfläche angewendet, um ein 3D-Verschiebungs-Array von Differenzvektoren zu erzeugen. Bei 7104 werden die mit dem Basis-Mesh assoziierten Basiskoordinaten erzeugt. Wie erwähnt, können die Basiskoordinaten in Kombination mit den Differenzvektoren verwendet werden, um das verschobene Gitter zu rekonstruieren. Bei 7105 wird das komprimierte verschobene Mesh einschließlich des 3D-Verschiebungs-Array und der Basiskoordinaten gespeichert.

[0528] Das nächste Mal, wenn das Primitiv von der Speicherung oder dem Speicher gelesen wird, bestimmt bei 6506, wird das verschobene Gitter von dem komprimierten verschobenen Mesh bei 7103 erzeugt. Zum Beispiel kann das 3D-Verschiebungs-Array auf die Basiskoordinaten angewendet werden, um das verschobene Mesh zu rekonstruieren.

VERBESSERTE VERLUSTBEHAFTETE KOMPRIMIERUNG EINES VERSCHOBENEN MESH UND HARDWARE-BVH-TRAVERSIERUNG/-ÜBERSCHNEIDUNG FÜR VERLUSTBEHAFTETE GITTER-PRIMITIVE

[0529] Komplexe dynamische Szenen sind für Echtzeit-Raytracing-Implementierungen herausfordernd. Prozedurale Oberflächen, Skinning-Animationen usw. erfordern Aktualisierungen von Triangulations- und Beschleunigungsstrukturen in jedem Frame, noch bevor der erste Strahl gestartet wird.

[0530] Anstatt nur ein bilineares Patch als Basis-Primitiv zu verwenden, erweitert eine Ausführungsform der Erfindung den Ansatz, bikubische Viereck- oder Dreieck-Patches zu unterstützen, die an den Patch-Grenzen hieb- und stichfest ausgewertet werden müssen. In einer Implementierung wird ein Bitfeld zu dem verlustbehafteten Gitter-Primitiv hinzugefügt, das angibt, ob ein implizites Dreieck gültig ist oder nicht. Eine Ausführungsform beinhaltet auch einen modifizierten Hardware-Block, der den existierenden Tessellator erweitert, um verlustbehaftete verschobene Meshes (z. B. wie oben mit Bezug auf die **Fig. 69A-71** beschrieben) direkt zu erzeugen, die dann in dem Speicher gespeichert werden.

[0531] In einer Implementierung nimmt eine Hardware-Erweiterung der BVH-Traversierungseinheit ein verlustbehaftetes Gitter-Primitiv als Eingabe, und extrahiert dynamisch Begrenzungsrahmen für Untersätze von implizit referenzierten Dreiecken/Vierecken. Die extrahierten Begrenzungsrahmen sind in einem Format, das mit der Strahl-Rahmen-Prüfungsschaltungsanordnung der BVH-Traversierungseinheit (z. B. der unten beschriebenen Strahl-/Rahmen-Traversierungseinheit 8930) kompatibel ist. Das Ergebnis der Strahl- gegen-

über dynamisch erzeugter Begrenzungsrahmen-Überschneidungsprüfung wird an die Strahl-Viereck/Dreieck-Überschneidungseinheit 8940 weitergeleitet, die die relevanten Dreiecke, die in dem Begrenzungsrahmen enthalten sind, extrahiert und diese schneidet.

[0532] Eine Implementierung beinhaltet auch eine Erweiterung der verlustbehafteten Gitter-Primitive durch Verwenden von indirekt referenzierten Vertex-Daten (ähnlich anderen Ausführungsformen), wodurch der Speicherverbrauch durch gemeinsames Nutzen von Vertex-Daten über benachbarte Gitter-Primitive hinweg reduziert wird. In einer Ausführungsform wird eine modifizierte Version des Hardware-BVH-Dreieck-Überschneidungsblocks darauf aufmerksam gemacht, dass die Eingabe Dreiecke von einem verlustbehafteten verschobenen Mesh sind, was es ihm ermöglicht, eine Kantenberechnung für benachbarte Dreiecke wiederzuverwenden. Eine Erweiterung wird auch zu der verlustbehafteten Komprimierung des verschobenen Mesh hinzugefügt, um eine Bewegungsunschärfegeometrie zu handhaben.

[0533] Wie oben beschrieben, wird unter der Annahme, dass die Eingabe ein Gitter-Mesh beliebiger Abmessungen ist, dieses Eingabegitter-Mesh zuerst in kleinere Untergitter mit einer festen Auflösung unterteilt, wie etwa 4x4 Vertices, wie in **Fig. 72** veranschaulicht.

[0534] Wie in **Fig. 73** gezeigt, wird nun in einer Ausführungsform eine verlustbehaftete 4x4-Gitter-Primitiv-Struktur (GridPrim) basierend auf den 4x4-Eingabe-Vertices berechnet. Eine Implementierung arbeitet gemäß der folgenden Codesequenz:

```
    struct GridPrim
{
 PrimLeafDesc leafDesc; // 4B
 uint32_t primIndex; // 4B
 float3       vertex[4]; //48B
 struct {
      exp   : 7; // gemeinsam genutzter Exponent
      disp_x : 5;
      disp_y: 5;
      disp_z : 5;
 }          disp_mag [16];     // 44B
}; // 64 Bytes insgesamt
```

[0535] In einer Implementierung verbrauchen diese Operationen 100 Bytes: 18 Bits von PrimLeafDesc können reserviert werden, um einzelne Dreiecke zu deaktivieren, z. B. eine Bitmaske von (in oben-nach-unten-, Links-Rechts-Reihenfolge) 000000000100000000b würde das in **Fig. 74** gezeigte hervorgehobene Dreieck 7401 deaktivieren.

[0536] Implizite Dreiecke können entweder 3x3 Vierecke (4x4 Vertices) oder mehr Dreiecke sein. Viele von diesen fügen sich zusammen und bilden ein Mesh. Die Maske gibt an, ob gewünscht wird, das Dreieck zu schneiden. Falls ein Loch erreicht wird, werden die einzelnen Dreiecke pro 4x4-Gitter deaktiviert. Dies ermöglicht eine höhere Präzision und einen erheblich reduzierten Speicherverbrauch: ~5,5 Byte/Dreieck, was eine sehr kompakte Repräsentation ist. Im Vergleich dazu nimmt jedes Dreieck, falls ein lineares Array mit voller Präzision gespeichert wird, 48 und 64 Bytes.

[0537] Wie in **Fig. 75** veranschaulicht, tesselliert ein Hardware-Tessellator 7550 Patches zu Dreiecken in 4x4 Einheiten, und speichert sie in einem Speicher, sodass BVHs über ihnen aufgebaut werden können, und sie strahlverfolgt werden können. In dieser Ausführungsform wird der Hardware-Tessellator 7550 modifiziert, um verlustbehaftete verschobene Gitter-Primitive direkt zu unterstützen. Anstatt einzelne Dreiecke zu erzeugen und sie an die Rasterisierungseinheit weiterzuleiten, kann die Hardware-Tessellationseinheit 7550 direkt verlustbehaftete Gitter-Primitive erzeugen, und sie in dem Speicher speichern.

[0538] Eine Erweiterung der Hardware-BVH-Traversierungseinheit 7550, die ein verlustbehaftetes Gitter-Primitiv als Eingabe nimmt, und spontan Begrenzungsrahmen für Untersätze implizit referenzierter Dreiecke/Vierecke extrahiert. In dem in **Fig. 76** gezeigten Beispiel werden neun Begrenzungsrahmen 7601A-I, einer für jedes Viereck, von dem verlustbehafteten Gitter extrahiert, und als ein spezieller neun-breiter BVH-Knoten an die Hardware-BVH-Traversierungseinheit 7550 weitergeleitet, um eine Strahl-Rahmen-Überschneidung auszuführen.

[0539] Die Prüfung aller 18 Dreiecke nacheinander ist sehr aufwändig. Unter Bezugnahme auf **Fig. 77** extrahiert eine Ausführungsform einen Begrenzungsrahmen 7601A-I für jedes Viereck (obwohl dies nur ein Beispiel ist; eine beliebige Anzahl von Dreiecken könnte extrahiert werden). Wenn ein Teilsatz von Dreiecken gelesen wird, und Begrenzungsrahmen berechnet werden, wird ein N-breiter BVH-Knoten 7700 erzeugt - ein Child-Knoten 7601A-I für jedes Viereck. Diese Struktur wird dann an die Hardware-Traversierungseinheit 7710 weitergeleitet, die Strahlen durch die neu konstruierte BVH traversiert. Somit wird in dieser Ausführungsform das Gitter-Primitiv als ein impliziter BVH-Knoten verwendet, von dem die Begrenzungsrahmen bestimmt werden können. Wenn ein Begrenzungsrahmen erzeugt wird, ist bekannt, dass er zwei Dreiecke enthält. Wenn die Hardware-Traversierungseinheit 7710 bestimmt, dass ein Strahl einen der Begrenzungsrahmen 7601A-I traversiert, wird dieselbe Struktur zu dem Strahl-Dreieck-Überschneider 7715 geleitet, um zu bestimmen, welcher Begrenzungsrahmen getroffen wurde. Das heißt, falls der Begrenzungsrahmen getroffen wurde, werden Überschneidungsprüfungen für die in dem Begrenzungsrahmen enthaltenen Dreiecke durchgeführt.

[0540] In einer Ausführungsform der Erfindung werden diese Techniken als ein Pre-Culling-Schritt für die Strahl-Dreieck-Traversierung 7710 und die Überschneidungseinheiten 7710 verwendet. Die Überschneidungsprüfung ist wesentlich kostengünstiger, wenn nur durch Verwenden der BVH-Knotenverarbeitungseinheit auf die Dreiecke geschlossen werden kann. Für jeden geschnittenen Begrenzungsrahmen 7601A-I werden die zwei jeweiligen Dreiecke an eine Raytracing-Dreieck-/-Viereckü berschneidungseinheit 7715 weitergeleitet, um die Strahl-Dreieck-Überschneidungsprüfungen durchzuführen.

[0541] Die oben beschriebenen Gitter-Primitiven- und impliziten BVH-Knotenverarbeitungstechniken können innerhalb jeglicher der hierin beschriebenen Traversierungs-/Überschneidungseinheiten (wie z. B. der unten beschriebenen Strahl/Rahmen-Traversierungseinheit 8930) integriert oder als ein Vorverarbeitungsschritt zu diesen verwendet werden.

[0542] In einer Ausführungsform werden Erweiterungen eines solchen verlustbehafteten 4x4-Gitter-Primitivs verwendet, um eine Bewegungsunschärfeverarbeitung mit zwei Zeitschritten zu unterstützen. Ein Beispiel ist in der folgenden Codesequenz bereitgestellt:

```
struct GridPrimMB
{
 PrimLeafDesc leafDesc; // 4B
 uint32_t primIndex; // 4B
 float3 vertex_time0[4]; // 48B
 float3 vertex_time1[4]; // 48B

 // insgesamt 32 Bytes bis hier
 struct {
   exp  : 6; // gemeinsam genutzter Exponent
   disp_x: 6;
   disp_y: 6;
   disp_z : 6;
 } disp_mag_time0[16],disp_mag_time1[16];       // 2x48B

}; // 8 + 96 + 96 Bytes insgesamt
```

[0543] Bewegungsunschärfeoperationen sind analog zum Simulieren der Verschlusszeit in einer Kamera. Um diesen Effekt strahlverfolgen zu können, gibt es bei Bewegen von t0 zu t1 zwei Repräsentationen eines Dreiecks, eine für t0 und eine für t1. In einer Ausführungsform wird zwischen ihnen eine Interpolation durchgeführt (z. B. Interpolieren der Primitiv-Repräsentationen zu jedem der zwei Zeitpunkte linear bei 0,5).

[0544] Der Nachteil von Beschleunigungsstrukturen, wie etwa Hüllkörperhierarchien (BVH) und k-d-Bäumen ist es, dass sie sowohl Zeit als auch Speicher benötigen, um aufgebaut und gespeichert zu werden. Eine Möglichkeit, diesen Overhead zu reduzieren, besteht darin, eine Art von Komprimierung und/oder Quantisierung der Beschleunigungsdatenstruktur einzusetzen, was besonders gut für BVH funktioniert, die natürlich für eine konservative inkrementelle Codierung geeignet sind. Auf der positiven Seite kann dies die Größe der Beschleunigungsstruktur, die häufig die Größe von BVH-Knoten halbiert, erheblich reduzieren. Auf der negativen Seite fällt beim Komprimieren der BVH-Knoten auch Overhead an, der in unterschiedliche Kategorien fallen kann. Erstens gibt es die offensichtlichen Kosten des Dekomprimierens jedes BVH-Knotens während

der Traversierung; zweitens, insbesondere für hierarchische Codierungsschemen, verkompliziert die Notwendigkeit, Parent-Informationen zu verfolgen, die Stapeloperationen geringfügig; und drittens bedeutet konservatives Quantisieren der Grenzen, dass die Begrenzungsrahmen etwas weniger eng sind als unkomprimierte, wodurch eine messbare Zunahme der Anzahl von Knoten und Primitiven ausgelöst wird, die traversiert bzw. geschnitten werden müssen.

[0545] Das Komprimieren der BVH durch lokale Quantisierung ist ein bekanntes Verfahren zum Reduzieren ihrer Größe. Ein n-breiter BVH-Knoten enthält die achsenausgerichteten Begrenzungsrahmen (AABB) seiner „n“ Children in Gleitkommaformat mit einfacher Präzision. Lokale Quantisierung drückt die „n“ Child-AABBs relativ zu dem AABB des Parent aus, und speichert diesen Wert in einem quantisierten, z. B. 8-Bit-Format, wodurch die Größe des BVH-Knotens reduziert wird.

[0546] Die lokale Quantisierung der gesamten BVH führt mehrere Overhead-Faktoren ein, da (a) die dequantisierten AABBs gröber als die ursprünglichen Gleitkomma-AABBs mit einfacher Präzision sind, wodurch zusätzliche Traversierungs- und Überschneidungsschritte für jeden Strahl eingeführt werden, und (b) die Dequantisierungsoperation selbst aufwändig ist, was einen Overhead zu jedem Strahltraversierungsschritt hinzufügt. Aufgrund dieser Nachteile werden komprimierte BVH nur in spezifischen Anwendungsszenarien verwendet, und sind nicht weit verbreitet.

[0547] Eine Ausführungsform der Erfindung setzt Techniken zum Komprimieren von Leaf-Knoten für Hair-Primitive in einer Hüllkörperhierarchie ein, wie in der gleichzeitig anhängigen Anmeldung mit dem Titel „Apparatus and Method for Compressing Leaf Nodes of Bounding Volume Hierarchies“, Seriennummer 16/236,185, eingereicht am 28. Dezember 2018, die an den Abtretungsempfänger der vorliegenden Anmeldung abgetreten wurde, beschrieben. Insbesondere werden, wie in der gleichzeitig anhängigen Anmeldung beschrieben, mehrere Gruppen orientierter Primitive zusammen mit einem Parent-Begrenzungsrahmen gespeichert, wodurch eine Child-Zeigerspeicherung in dem Leaf-Knoten eliminiert wird. Ein orientierter Begrenzungsrahmen wird dann für jedes Primitiv durch Verwenden von 16-Bit-Koordinaten, die bezüglich einer Ecke des Parent-Rahmens quantisiert sind, gespeichert. Schließlich wird eine quantisierte Normale für jede Primitiv-Gruppe gespeichert, um die Orientierung anzugeben. Dieser Ansatz kann zu einer erheblichen Reduzierung der Bandbreite und der Speichergrundfläche für BVH-Hair-Primitive führen.

[0548] In einigen Ausführungsformen werden BVH-Knoten komprimiert (z. B. für eine 8-breite BVH), indem der Parent-Begrenzungsrahmen gespeichert wird, und N Child-Begrenzungsrahmen (z. B. 8 Children) relativ zu diesem Parent-Begrenzungsrahmen durch Verwenden einer geringeren Präzision codiert werden. Ein Nachteil des Anwendens dieser Idee auf jeden Knoten einer BVH ist es, dass an jedem Knoten etwas Dekomprimierungs-Overhead eingeführt wird, wenn Strahlen durch diese Struktur traversiert werden, was die Leistungsfähigkeit reduzieren kann.

[0549] Um dieses Problem zu adressieren, verwendet eine Ausführungsform der Erfindung die komprimierten Knoten nur auf der untersten Ebene der BVH. Dies stellt einen Vorteil der höheren BVH-Ebenen bereit, die bei optimaler Leistungsfähigkeit laufen (d. h. sie werden so oft berührt, wie Rahmen groß sind, aber es gibt sehr wenige von ihnen), und die Komprimierung auf den niedrigeren/niedrigsten Ebenen ist auch sehr effektiv, da die meisten Daten der BVH in der niedrigsten Ebene (den niedrigsten Ebenen) liegen.

[0550] Zusätzlich wird in einer Ausführungsform Quantisierung auch für BVH-Knoten angewendet, die orientierte Begrenzungsrahmen speichern. Wie unten diskutiert, sind die Operationen etwas komplizierter als für achsenausgerichtete Begrenzungsrahmen. In einer Implementierung wird die Verwendung von komprimierten BVH-Knoten mit orientierten Begrenzungsrahmen mit dem Verwenden der komprimierten Knoten nur auf der niedrigsten Ebene (oder niedrigeren Ebenen) der BVH kombiniert.

[0551] Somit verbessert sich eine Ausführungsform an vollständig komprimierten BVHs durch Einführen einer einzelnen fest zugeordneten Ebene komprimierter Leaf-Knoten, während reguläre, unkomprimierte BVH-Knoten für innere Knoten verwendet werden. Eine Motivation hinter diesem Ansatz ist es, dass fast alle Einsparungen an Komprimierung von den niedrigsten Ebenen einer BVH (die insbesondere für 4-breite und 8-breite BVHs auf die überwiegende Mehrzahl aller Knoten ausmachen) kommen, während der größte Teil des Overhead von inneren Knoten kommt. Infolgedessen liefert das Einführen einer einzelnen Ebene fest zugeordneter „komprimierter Leaf-Knoten“ nahezu dieselben (und in einigen Fällen noch bessere) Komprimierungsgewinne, wie eine vollständig komprimierte BVH, während nahezu dieselbe Traversierungsleistungsfähigkeit wie eine unkomprimierte aufrechterhalten wird.

[0552] Fig. 80 veranschaulicht eine beispielhafte Raytracing-Engine 8000, die die hierin beschriebenen Leaf-Knoten-Komprimierungs- und -dekomprimierungsoperationen durchführt. In einer Ausführungsform umfasst die Raytracing-Engine 8000 eine Schaltungsanordnung eines oder mehrerer der oben beschriebenen Raytracing-Kerne. Alternativ kann die Raytracing-Engine 8000 auf den Kernen der CPU oder auf anderen Typen von Grafikkernen (z. B. GFX-Kernen, Tensorkernen usw.) implementiert werden.

[0553] In einer Ausführungsform erzeugt ein Strahlerzeuger 8002 Strahlen, die eine Traversierungs-/Überschneidungseinheit 8003 durch eine mehrere Eingabe-Primitive 8006 umfassende Szene verfolgt. Zum Beispiel kann eine App, wie etwa ein Virtual-Reality-Spiel, Streams von Befehlen erzeugen, von denen die Eingabe-Primitive 8006 erzeugt werden. Die Traversierungs-/Überschneidungseinheit 8003 traversiert die Strahlen durch eine BVH 8005, die durch einen BVH-Builder 8007 erzeugt wird, und identifiziert Trefferpunkte, an denen die Strahlen ein oder mehrere der Primitiven 8006 schneiden. Obwohl als eine einzelne Einheit veranschaulicht, kann die Traversierungs-/Überschneidungseinheit 8003 eine Traversierungseinheit umfassen, die mit einer spezifischen Überschneidungseinheit gekoppelt ist. Diese Einheiten können in einer Schaltungsanordnung, Software/Befehlen, die durch die GPU oder CPU ausgeführt werden, oder einer beliebigen Kombination davon implementiert werden.

[0554] In einer Ausführungsform beinhaltet die BVH-Verarbeitungsschaltungsanordnung/-logik 8004 einen BVH-Builder 8007, der die BVH 8005 wie hierin beschrieben basierend auf den räumlichen Beziehungen zwischen Primitiven 8006 in der Szene erzeugt. Zusätzlich beinhaltet die BVH-Verarbeitungsschaltungsanordnung/-logik 8004 einen BVH-Komprimierer 8009 und einen BVH-Dekomprimierer 8009 zum Komprimieren bzw. Dekomprimieren der Leaf-Knoten, wie hierin beschrieben. Die folgende Beschreibung konzentriert sich zu dem Zweck der Veranschaulichung auf 8-breite BVHs (BVH8).

[0555] Wie in **Fig. 81** veranschaulicht, enthält eine Ausführungsform von einzelnem 8-breitem BVH-Knoten 8100A 8 Begrenzungsrahmen 8101-8108 und 8 (64bit) Child-Zeiger/Referenzen 8110, die auf die Begrenzungsrahmen/Leaf-Daten 8101-8108 zeigen. In einer Ausführungsform führt ein BVH-Komprimierer 8025 eine Codierung aus, bei der die 8 Child-Begrenzungsrahmen 8101A-8108A relativ zu dem Parent-Begrenzungsrahmen 8100A ausgedrückt und zu einheitlichen 8-Bit-Werten quantisiert werden, die als Begrenzungsrahmen-Leaf-Daten 8101B-8108B gezeigt sind. Der quantisierte 8-breite BVH-, QBVH8-Knoten 8100B, wird durch die BVH-Komprimierung 8125 durch Verwenden eines Start- und Erweiterungswerts codiert, der als zwei 3-dimensionale Vektoren mit einfacher Präzision (2 × 12 Bytes) gespeichert wird. Die acht quantisierten Child-Begrenzungsrahmen 8101B-8108B werden als 2 mal 8 Bytes für die Unter- und Obergrenzen der Begrenzungsrahmen pro Dimension (insgesamt 48 Bytes) gespeichert. Es ist anzumerken, dass sich dieses Layout von existierenden Implementierungen unterscheidet, da das Ausmaß in voller Präzision gespeichert wird, was im Allgemeinen engere Grenzen bereitstellt, jedoch mehr Platz benötigt.

[0556] In einer Ausführungsform dekomprimiert ein BVH-Dekomprimierer 8026 den QBVH8-Knoten 8100B wie folgt. Die dekomprimierten Untergrenzen in Dimension i können durch QBVH8.starti+(byte-to-float) QBVH8 .lower$_{i}$*QBVH8.extend$_{i}$ berechnet werden, was auf der CPU 4099 fünf Anweisungen pro Dimension und Rahmen erfordert: 2 Ladungen (start,extend), Byte-to-int-Ladung + Aufwärtsumwandlung, Int-to-Floating-Umwandlung und eine Multiplikations-Addition. In einer Ausführungsform wird die Dekomprimierung für alle 8 quantisierten Child-Begrenzungsrahmen 8101B-8108B parallel durch Verwenden von SIMD-Anweisungen ausgeführt, was einen Overhead von etwa 10 Anweisungen zu der Strahl-Knoten-Überschneidungsprüfung hinzufügt, was sie mindestens mehr als doppelt so aufwändig macht wie in dem unkomprimierten Standardknotenfall. In einer Ausführungsform werden diese Anweisungen auf den Kernen der CPU 4099 ausgeführt. Alternativ wird ein vergleichbarer Satz von Anweisungen durch die Raytracing-Kerne 4050 ausgeführt.

[0557] Ohne Zeiger benötigt ein QBVH8-Knoten 72 Bytes, während ein unkomprimierter BVH8-Knoten 192 Bytes benötigt, was in einem Reduktionsfaktor von 2,66x resultiert. Mit 8 (64bit) Zeigern reduziert sich der Reduktionsfaktor auf 1,88x, was es erforderlich macht, die Speicherungskosten für die Handhabung von Leaf-Zeigern zu adressieren.

[0558] In einer Ausführungsform, wenn nur die Leaf-Schicht der BVH8-Knoten in QBVH8-Knoten komprimiert wird, beziehen sich alle Child-Zeiger der 8 Children 8101-8108 nur auf Leaf-Primitiv-Daten. In einer Implementierung wird diese Tatsache ausgenutzt, indem alle referenzierten Primitiv-Daten direkt nach dem QBVH8-Knoten 8100B selbst gespeichert werden, wie in **Fig. 81** veranschaulicht. Dies ermöglicht es, die vollen 64bit Child-Zeiger 8110 der QBVH8 auf nur 8-Bit-Offsets 8122 zu reduzieren. Falls die Primitiv-Daten eine feste Größe haben, werden in einer Ausführungsform die Offsets 8122 vollständig übersprungen, da sie

direkt von dem Index des geschnittenen Begrenzungsrahmens und dem Zeiger zu dem QBVH8-Knoten 8100B selbst berechnet werden können.

[0559] Beim Verwenden eines Top-down-BVH8-Builders erfordert das Komprimieren von nur der BVH8-Leaf-Ebene nur geringe Modifikationen des Build-Prozesses. In einer Ausführungsform werden diese Build-Modifikationen in dem BVH-Builder 8007 implementiert. Während der rekursiven Build-Phase verfolgt der BVH-Builder 8007, ob die aktuelle Anzahl an Primitiven unter einem bestimmten Schwellenwert liegt. In einer Implementierung ist N × M die Schwelle, wobei N sich auf die Breite der BVH bezieht und M die Anzahl an Primitiven innerhalb eines BVH-Leaf ist. Für einen BVH8-Knoten und zum Beispiel vier Dreiecke pro Leaf beträgt die Schwelle 32. Daher tritt für alle Unterbäume mit weniger als 32 Primitiven die BVH-Verarbeitungsschaltungsanordnung/-logik 8004 in einen speziellen Code-Pfad ein, wo sie den Oberflächenheuristik(Surface Area Heuristic - SAH)-basierten Aufteilungsprozess fortsetzt, aber einen einzelnen QBVH8-Knoten 8100B erzeugt. Wenn der QBVH8-Knoten 8100B schließlich erzeugt wird, sammelt der BVH-Komprimierer 8009 dann alle referenzierten Primitiv-Daten, und kopiert sie unmittelbar hinter dem QBVH8-Knoten.

[0560] Die tatsächliche BVH8-Traversierung, die durch den Raytracing-Kern 8150 oder die CPU 8199 durchgeführt wird, wird von der Leaf-Ebenen-Komprimierung nur geringfügig beeinflusst. Im Wesentlichen wird der QBVH8-Knoten 8100B auf Leaf-Ebene als ein erweiterter Leaf-Typ behandelt (z. B. ist er als ein Leaf markiert). Dies bedeutet, dass die reguläre BVH8-Top-Down-Traversierung fortgesetzt wird, bis ein QBVH-Knoten 8100B erreicht ist. An diesem Punkt wird eine einzelne Strahl-QBVH-Knoten-Überschneidung ausgeführt, und für alle seine gekreuzten Children 8101B bis 8108B wird der jeweilige Leaf-Zeiger rekonstruiert, und reguläre Strahl-Primitiv-Überschneidungen werden ausgeführt. Interessanterweise kann die Ordnung der geschnittenen Children 8101B-8108B der QBVH basierend auf dem Überschneidungsabstand keinen messbaren Vorteil bereitstellen, da in den meisten Fällen ohnehin nur ein einzelnes Child durch den Strahl geschnitten wird.

[0561] Eine Ausführungsform des Komprimierungsschemas auf Leaf-Ebene ermöglicht sogar eine verlustfreie Komprimierung der tatsächlichen Primitiv-Leaf-Daten durch Extrahieren gemeinsamer Merkmale. Dreiecke innerhalb eines BVH-Knotens mit komprimiertem Leaf (CLBVH) nutzen zum Beispiel sehr wahrscheinlich Vertices/Vertex-Indizes und Eigenschaften wie dieselbe objectID gemeinsam. Durch das Speichern dieser gemeinsam genutzten Eigenschaften nur einmal pro CLBVH-Knoten und das Verwenden kleiner lokaler Byte-großer Indizes in den Primitiven wird der Speicherverbrauch weiter reduziert.

[0562] In einer Ausführungsform werden die Techniken zum wirksamen Einsetzen gemeinsamer räumlich kohärenter geometrischer Merkmale innerhalb eines BVH-Leaf auch für andere komplexere Primitiv-Typen verwendet. Primitive, wie zum Beispiel Hair-Segmente, nutzen wahrscheinlich eine gemeinsame Richtung pro BVH-Leaf. In einer Ausführungsform implementiert der BVH-Komprimierer 8009 ein Komprimierungsschema, das diese gemeinsame Richtungseigenschaft berücksichtigt, um orientierte Begrenzungsrahmen (OBB), die sich als sehr nützlich für das Begrenzen langer diagonaler Primitiv-Typen gezeigt haben, effizient zu komprimieren.

[0563] Die hierin beschriebenen komprimierten BVHs auf Leaf-Ebene führen BVH-Knotenquantisierung nur auf der niedrigsten BVH-Ebene ein, und ermöglichen daher zusätzliche Speicherreduzierungsoptimierungen, während die Traversierungsleistungsfähigkeit einer unkomprimierten BVH bewahrt wird. Da nur BVH-Knoten auf der niedrigsten Ebene quantisiert werden, zeigen alle ihre Children auf Leaf-Daten 8101B bis 8108B, die zusammenhängend in einem Speicherblock oder einer oder mehreren Cachezeilen 8098 gespeichert sein können.

[0564] Die Idee kann auch auf Hierarchien angewendet werden, die orientierte Begrenzungsrahmen (OBB) verwenden, die typischerweise zur Beschleunigung des Renderns von Hair-Primitiven verwendet werden. Um eine bestimmte Ausführungsform zu veranschaulichen, werden die Speicherreduzierungen in einem typischen Fall einer 8-breiten Standard-BVH über Dreiecke evaluiert.

[0565] Das Layout eines 8-breiten BVH-Knotens 8100 ist in der folgender Kernsequenz repräsentiert:

```
struct BVH8Node {
    float lowerX[8], upperX[8];
    // 8 × Ober- und Untergrenzen in der X-Dimension
    float lowerY[8], upperY[8];
    // 8 × Ober- und Untergrenzen in der Y-Dimension
```

```
    float lowerZ[8], upperZ[8];
    // 8 × Ober- und Untergrenzen in der Z-Dimension
    void *ptr[8];
    // 8 × 64bit Zeiger zu den 8 Kind-Knoten oder Leaf-Daten
};
```

und benötigt 276 Bytes an Speicher. Das Layout eines 8-breiten quantisierten Standardknotens kann definiert sein als:

```
struct QBVH8Node {
    Vec3f start, scale;
    char lowerX[8], upperX[8];
    // 8 × Byte-quantisierte Ober-/Untergrenzen in der X-Dimension
    char lowerY[8], upperY[8];
    // 8 × Byte-quantisierte Ober-/Untergrenzen in der Y-Dimension
    char lowerZ[8], upperZ[8];
    // 8 × Byte-quantisierte Ober-/Untergrenzen in der Z-Dimension
    void *ptr[8];
    // 8 × 64bit Zeiger zu den 8 Kind-Knoten oder Leaf-Daten
};
```

und benötigt 136 Bytes.

[0566] Da nur quantisierte BVH-Knoten auf Leaf-Ebene verwendet werden, zeigen alle Child-Zeiger tatsächlich auf Leaf-Daten 8101A bis 8108A. In einer Ausführungsform werden durch Speichern des quantisierten Knotens 8100B und aller Leaf-Daten 8101B-8108B, auf den dessen Children in einem einzelnen kontinuierlichen Speicherblock 8098 zeigen, die 8 Child-Zeiger in dem quantisierten BVH-Knoten 8100B entfernt. Das Speichern der Child-Zeiger reduziert das quantisierte Knoten-Layout auf:

```
struct QBVH8NodeLeaf {
Vec3f start, scale;
    // Startposition, Vektor des Parent-AABB erweitern
    char lowerX[8], upperX[8];
    // 8 × Byte-quantisierte Ober- und Untergrenzen in der X-
    Dimension
    char lowerY[8], upperY[8];
    // 8 × Byte-quantisierte Ober- und Untergrenzen in der Y-
    Dimension
    char lowerZ[8], upperZ[8];
    // 8 × Byte-quantisierte Ober- und Untergrenzen in der Z-
    Dimension
};
```

der nur 72 Byte benötigt. Aufgrund des kontinuierlichen Layouts in dem Speicher/Cache 8098 kann der Child-Zeiger des i-ten Childs nun einfach berechnet werden durch: childPtr(i) = addr(QBVH8NodeLeaf) + sizeof (QBVH8NodeLeaf) + i * sizeof(LeafDataType).

[0567] Da die Knoten auf niedrigster Ebene der BVH mehr als die Hälfte der gesamten Größe der BVH ausmachen, stellt die hierin beschriebene Nur-Leaf-Ebene-Komprimierung eine Reduzierung auf 0,5 + 0,5 * 72/256 = 0,64x der ursprünglichen Größe bereit.

[0568] Zusätzlich treten der Overhead, gröbere Grenzen zu haben, und die Kosten des Dekomprimierens quantisierter BVH-Knoten selbst nur auf der BVH-Leaf-Ebene auf (im Gegensatz zu allen Ebenen, wenn die gesamte BVH quantisiert wird). Somit wird der oft recht signifikante Traversierungs- und Überschneidungs-Overhead aufgrund gröberer Grenzen (eingeführt durch Quantisierung) weitgehend vermieden.

[0569] Ein weiterer Vorteil der Ausführungsformen der Erfindung ist eine verbesserte Hardware- und Software-Vorabrufeffizienz. Dies resultiert aus der Tatsache, dass alle Leaf-Daten in einem relativ kleinen kontinuierlichen Block von Speicher- oder Cachezeile(n) gespeichert sind.

[0570] Da die Geometrie auf der BVH-Leaf-Ebene räumlich kohärent ist, ist es sehr wahrscheinlich, dass alle Primitive, die durch einen QBVH8NodeLeaf-Knoten referenziert werden, gemeinsame Eigenschaften/Merkmale, wie etwa objectID, einen oder mehrere Vertices usw., gemeinsam nutzen. Infolgedessen reduziert eine Ausführungsform der Erfindung die Speicherung weiter durch Entfernen von Primitiv-Daten-Duplizierung. Zum Beispiel können ein Primitiv und assoziierte Daten nur einmal pro QBVH8NodeLeaf-Knoten gespeichert werden, wodurch der Speicherverbrauch für Leaf-Daten weiter reduziert wird.

[0571] Die effektive Begrenzung von Hair-Primitiven wird unten als ein Beispiel für signifikante Speicherreduzierungen beschrieben, die durch Ausnutzung gemeinsamer Geometrieeigenschaften auf BVH-Leaf-Ebene realisiert werden. Um ein Hair-Primitiv, das eine lange, jedoch dünne Struktur ist, die im Raum orientiert ist, genau zu begrenzen, ist es ein wohlbekannter Ansatz, einen orientierten Begrenzungsrahmen zu berechnen, um die Geometrie eng zu begrenzen. Zuerst wird ein Koordinatenraum berechnet, der auf die Hair-Richtung ausgerichtet ist. Zum Beispiel kann bestimmt werden, dass die z-Achse in die Hair-Richtung zeigt, während die x- und y-Achse senkrecht auf der z-Achse stehen. Durch Verwenden dieses orientierten Raums kann nun ein Standard-AABB verwendet werden, um das Hair-Primitiv eng zu begrenzen. Das Schneiden eines Strahls mit einer solchen orientierten Grenze erfordert zuerst das Transformieren des Strahls in den orientierten Raum, und dann das Ausführen einer Standard-Strahl/Rahmen-Überschneidungsprüfung.

[0572] Ein Problem mit diesem Ansatz ist sein Speicherverbrauch. Die Transformation in den orientierten Raum erfordert 9 Gleitkommawerte, während das Speichern des Begrenzungsrahmens zusätzliche 6 Gleitkommawerte erfordert, was insgesamt 60 Bytes ergibt.

[0573] In einer Ausführungsform der Erfindung komprimiert der BVH-Komprimierer 8025 diesen orientierten Raum und Begrenzungsrahmen für mehrere Hair-Primitive, die einander räumlich nahe sind. Diese komprimierten Grenzen können dann innerhalb der komprimierten Leaf-Ebene gespeichert werden, um die in dem Leaf gespeicherten Hair-Primitive eng zu begrenzen. Der folgende Ansatz wird in einer Ausführungsform zur Komprimierung der orientierten Grenzen verwendet. Der orientierte Raum kann durch die normierten Vektoren v_x, v_y, und v_z ausgedrückt werden, die zueinander orthogonal sind. Die Transformation eines Punktes p in diesen Raum funktioniert durch seine Projektion auf diese Achsen:

$$p_x = dot(v_x, p)$$

$$p_y = dot(v_y, p)$$

$$p_z = dot(v_z, p)$$

[0574] Da die Vektoren v_x, v_y, und v_z normiert werden, liegen ihre Komponenten in dem Bereich [-1,1]. Diese Vektoren werden daher unter Verwendung von vorzeichenbehafteten 8 Bit-Festkommazahlen quantisiert anstatt unter Verwendung von vorzeichenbehafteten 8 Bit-Ganzzahlen und eines konstanten Maßstabs. Auf diese Weise werden quantisierte v_x', v_y', und v_y' erzeugt. Dieser Ansatz reduziert den zur Codierung des orientierten Raums benötigten Speicher von 36 Bytes (9 Gleitkommawerte) auf nur 9 Bytes (9 Festkommazahlen mit je 1 Byte).

[0575] In einer Ausführungsform wird der Speicherverbrauch des orientierten Raums weiter reduziert, indem die Tatsache genutzt wird, dass alle Vektoren zueinander orthogonal sind. Somit müssen nur zwei Vektoren gespeichert werden (z. B. p_y' und p_z') und es kann p_x' = cross(p_y' , p_z') berechnet werden, wodurch die erforderliche Speicherung weiter auf nur sechs Bytes reduziert wird.

[0576] Was übrig bleibt, ist die Quantisierung des AABB innerhalb des orientierten quantisierten Raums. Problematisch hierbei ist, dass das Projizieren eines Punktes p auf eine komprimierte Koordinatenachse dieses Raumes (z. B. durch Berechnen vondot(v_x', p)) Werte eines potenziell großen Bereichs liefert (da Werte p typischerweise als Gleitkommazahlen codiert sind). Aus diesem Grund müssten Gleitkommazahlen zum Codieren der Grenzen verwendet werden, wodurch potenzielle Einsparungen reduziert werden.

[0577] Zur Lösung dieses Problems transformiert eine Ausführungsform der Erfindung zunächst das mehrfache Hair-Primitiv in einen Raum, in dem seine Koordinaten in dem Bereich [0, 1/V3] liegen. Dies kann durch

Bestimmen des realraum-achsenausgerichteten Begrenzungsrahmens b der mehreren Hair-Primitive und Verwenden einer Transformation T erfolgen, die zuerst um b.lower nach links translatiert, und dann um 1/max(b.size.x, b.size.y.b.size.z) in jeder Koordinate skaliert:

$$T(p) = \frac{1}{\sqrt{3}}(p - b.lower) / \max(b.size.x, b.size.y, b.size.z)$$

[0578] Eine Ausführungsform gewährleistet, dass die Geometrie nach dieser Transformation in dem Bereich [0, 1/^3] bleibt, da dann eine Projektion eines transformierten Punktes auf einen quantisierten Vektor p_x', p_y' oder p_z' in dem Bereich [-1,1] bleibt. Das heißt, der AABB der Kurvengeometrie kann quantisiert werden, wenn er durch Verwenden von T transformiert wird, und dann in den quantisierten orientierten Raum transformiert wird. In einer Ausführungsform wird vorzeichenbehaftete 8-Bit-Festkomma-Arithmetik verwendet. Aus Präzisionsgründen können jedoch vorzeichenbehaftete 16-Bit-Festkommazahlen verwendet werden (z. B. codiert durch Verwenden von vorzeichenbehafteten 16-Bit-Ganzzahlen und einer konstanten Skala). Dies reduziert den Speicherbedarf zum Codieren des achensausgerichteten Begrenzungsrahmens von 24 Bytes (6 Gleitkommawerten) auf nur 12 Bytes (6 Wörter) plus dem Offset b.lower (3 Gleitkommas) und der Skala (1 Gleitkomma), die für mehrere Hair-Primitive gemeinsam genutzt werden.

[0579] Wenn zum Beispiel 8 Hair-Primitive begrenzt werden sollen, reduziert diese Ausführungsform den Speicherverbrauch von 8*60 Bytes = 480 Bytes auf nur 8*(6+12)+3*4+4 = 160 Bytes, was eine Reduzierung um das Dreifache ist. Das Überschneiden eines Strahls mit diesen quantisierten orientierten Grenzen funktioniert, indem zuerst der Strahl durch Verwenden der Transformation T transformiert wird, dann der Strahl durch Verwenden quantisierter v_x', v_y' und v_z' projiziert wird. Schließlich wird der Strahl mit dem quantisierten AABB geschnitten.

[0580] Der oben beschriebene Fat-Leaves-Ansatz stellt eine Möglichkeit für noch mehr Komprimierung bereit. Unter der Annahme, dass es einen impliziten einzelnen float3-Zeiger in dem Fat-BVH-Leaf gibt, der auf die gemeinsam genutzten Vertex-Daten mehrerer benachbarter GridPrims zeigt, kann der Vertex in jedem Gitter-Primitiv indirekt durch Byte-große Indizes („vertex_index_*“) adressiert werden, wodurch gemeinsames Nutzen von Vertices ausgenutzt wird. In **Fig. 78** werden Vertices 7801-7802 gemeinsam genutzt - und in voller Präzision gespeichert. In dieser Ausführungsform werden die gemeinsam genutzten Vertices 7801-7802 nur einmal gespeichert, und Indizes werden gespeichert, die auf ein Array zeigen, das die eindeutigen Vertices enthält. Somit werden anstelle von 48 Bytes nur 4 Bytes pro Zeitstempel gespeichert. Die Indizes in der folgenden Codesequenz werden zur Identifizierung der gemeinsam genutzten Vertices verwendet.

```
struct GridPrimMBIndexed
{
      PrimLeafDesc leafDesc; // 4B
        uint32_t   primIndex; // 4B
        uint8_t    vertex_index_time0[4]; / / 4B
        uint8_t    vertex_index_time1[4]; 4B
// insgesamt 16 Bytes bis hier
        struct {
         exp : 5; // gemeinsam genutzter Exponent
         disp_x : 5;
         disp_y: 5;
         disp_z : 5;
        } disp_mag_time0[16],disp_mag_time1[16];      // 80
       Bytes
}; // 96 Bytes insgesamt
```

[0581] In einer Ausführungsform werden gemeinsam genutzte Kanten von Primitiven nur einmal evaluiert, um Verarbeitungsressourcen zu sparen. In **Fig. 79** wird zum Beispiel angenommen, dass ein Begrenzungsrahmen aus den hervorgehobenen Vierecken besteht. Statt alle Dreiecke individuell zu überschneiden, führt eine Ausführungsform der Erfindung Strahl-Kanten-Berechnungen einmal für jede der drei gemeinsam genutzten Kanten aus. Die Ergebnisse der drei Strahl-Kanten-Berechnungen werden somit über die vier Dreiecke gemeinsam genutzt (d. h., es wird nur eine Strahl-KantenBerechnung für jede gemeinsam genutzte Kante durchgeführt). Zusätzlich werden in einer Ausführungsform die Ergebnisse in einem On-Chip-Speicher

(z. B. einem Scratch-Speicher/-Cache, auf den die Überschneidungseinheit direkt zugreifen kann) gespeichert.

ATOMICS FÜR GRAPHIKEN UND DATENSTRUKTUREN

[0582] Ein „Atomic“ ist ein Satz von Operationen, die als eine einzelne Einheit abgeschlossen werden müssen. Bestimmte Atomics wären für die Grafikverarbeitungsleistungsfähigkeit vorteilhaft, insbesondere wenn Berechnungs-Shader ausgeführt werden. Eine Ausführungsform der Erfindung beinhaltet eine Vielzahl neuer Atomics, um die Grafikverarbeitungsleistungsfähigkeit zu verbessern, einschließlich:

- Atomics, die Clamping durchführen
- ‚z-geprüfte' Atomic-Schreibvorgänge
- ‚z-geprüfte' Atomic-Akkumulation
- Atomics für Ringpuffer

I. Atomics für Clamping

[0583] Eine Ausführungsform eines Clamping-Atomic spezifiziert ein Ziel (destination), einen Typwert (type value) und minimale und maximale Clamping-Werte. Ein Clamping-Atomic kann beispielsweise die folgende Form annehmen:

InterlockedAddClamp(destination, type value, type min, type max)

[0584] Die obige Clamping-Operation fügt atomar einen Wert zu dem Ziel hinzu, und führt dann Clamping auf die spezifizierten Minimal- und Maximalwerte durch (z. B. durch Einstellen auf das Maximum für jegliche Werte oberhalb des Maximums, und Einstellen auf das Minimum für jegliche Werte unterhalb von min).

[0585] Clamping-Atomic-Werte können 32 Bit, 64 Bit oder eine beliebige andere Datengröße sein. Darüber hinaus können Clamping-Atomics an verschiedenen Datentypen arbeiten, einschließlich, jedoch nicht beschränkt auf uint, float, 2xfp16, float2, und 4xfp16.

II. „Z-geprüfte“ gestreute Schreibvorgänge

[0586] Z-geprüfte gestreute Schreibvorgänge können für eine Vielzahl von Anwendungen verwendet werden, einschließlich zum Beispiel:

- gestreutes Cubemap-Rendering/Voxelisierung (z. B. für Umgebungssonden);
- gestreute unvollkommene reflektierende Schattenabbildungen (Reflective Shadow Maps - RSM) (ähnlich unvollkommenen Schattenabbildungen, jedoch für indirekte Beleuchtung); und
- globale Beleuchtung im Stil dynamischer diffuser globaler Beleuchtung durch gestreute „Umgebungssonden“-Aktualisierungen.

[0587] Das Folgende ist ein Beispiel für eine Vergleichsaustauschanweisung, die in einer Ausführungsform der Erfindung ausgeführt werden kann:

InterlockedCmpXChg_type_cmp_op()

type = int, uint, float

cmp_op = less, greater, equal, less_equal, greater_equal, not_equal

z. B.: InterlockedDepthCmpXChg_float_less_equal()

[0588] Ein beispielhaftes 64-Bit-Zielregister 8201 ist in **Fig. 82A** veranschaulicht, das einen 32-Bit-Tiefenwert 8202 und 32-Bit-Nutzdaten 8203 speichert. Im Betrieb tauscht der obige Vergleichsaustauschbefehl nur Nutzdaten und Tiefe aus, falls der neue Gleitkomma-Tiefenwert kleiner oder gleich dem gespeicherten Gleitkommawert ist. In einer Ausführungsform sind die cmpxchg-Atomics „entfernte“ Atomics, was bedeutet, dass der tatsächliche Vergleich und die Atomic-Aktualisierung nicht durch die EU, die die Anweisung ausgegeben hat,

sondern stattdessen durch einen Logikblock nahe dem LLC (oder der Speichersteuerung), der/die die Daten speichert, durchgeführt wird.

Beispielhafte HLSL(High Level Shading Language)-Intrinsik für Schreib-Lese-Byte-Adresspuffer (RWByteAddressBuffers)

[0589] In einer Ausführungsform ist nur der HighCompValue von dem Typ, der mit den hohen 32 Bits in dem 64-Bit-Ziel verglichen werden soll. Für den Rest wird angenommen, dass er in vorzeichenlose 32-Bit-Ganzzahl (asuint()) umgewandelt wird:

void RWByteAddressBuffer::InterlockedCompareHighExchange_Less(uint byteAddress64, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ LessEqual(uint byteAddress64, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ Greater(uint byteAddress64, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ GreaterEqual(uint byteAddress64, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer: :InterlockedCompareHighExchange_Equal(uint byteAddress64, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ NotEqual(uint byteAddress64, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_Less(uint byteAddress64, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ LessEqual(uint byteAddress64, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ Greater(uint byteAddress64, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ GreaterEqual(uint byteAddress64, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_Equal(uint byteAddress64, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ NotEqual(uint byteAddress64, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_Less(uint byteAddress64, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ LessEqual(uint byteAddress64, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ Greater(uint byteAddress64, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ GreaterEqual(uint byteAddress64, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ Equal(uint byteAddress64, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareHighExchange_ NotEqual(uint byteAddress64, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

Beispielhafte HLSL-Intrinsik für Ziel R

[0590] HighCompValue ist von dem Typ, der mit den hohen 32 Bit an dem 64-Bit-Ziel verglichen werden soll. Für den Rest wird angenommen, dass er durch Verwenden von asuint() umgewandelt wird.

[0591] Alle diese Instrinsiken nehmen einen ‚dest'-Parameter vom Typ ‚R', der entweder eine Ressourcenvariable oder eine gemeinsam genutzte Speichervariable sein kann. Eine Ressourcenvariable ist eine skalare Referenz auf eine Ressource, die Indizierung oder Feldreferenzen beinhaltet. Eine gemeinsam genutzte Speichervariable ist eine, die mit dem ‚groupshared'-Schlüsselwort definiert ist. In jedem Fall muss der Typ uint2 oder uint64 sein. Wenn ‚R' ein Variablentyp mit gemeinsam genutztem Speicher ist, wird die Operation an dem value'-Parameter und dem gemeinsam genutzten Speicherregister, das durch „dest" referenziert wird, durchgeführt. Wenn „R" ein Ressourcenvariablentyp ist, wird die Operation an dem value'-Parameter und dem Ressourcenort, der durch „dest" referenziert wird, durchgeführt. Das Ergebnis wird in dem gemeinsam genutzten Speicherregister oder dem durch „dest" referenzierten Ressourcenort gespeichert:

void InterlockedCompareHighExchange_Less(R dest, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_LessEqual(R dest, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_Greater(R dest, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_GreaterEqual(R dest, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_Equal(R dest, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_NotEqual(R dest, uint uHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_Less(R dest, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_LessEqual(R dest, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_Greater(R dest, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_GreaterEqual(R dest, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_Equal(R dest, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_NotEqual(R dest, int iHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_Less(R dest, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_LessEqual(R dest, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_Greater(R dest, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_GreaterEqual(R dest, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_Equal(R dest, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareHighExchange_NotEqual(R dest, float fHighCompVal, uint2 HighAndLowVal, out uint2 HighAndLowOrgVal)

III. „Z-geprüfte" gestreute Akkumulation

[0592] Im Folgenden werden zwei Ausführungsformen mit Bezug auf die **Fig. 82B-C** beschrieben. **Fig. 82B** veranschaulicht ein 64-Bit-Zielregister, das einen 32-Bit-Tiefenwert und einen 32-Bit-Nutzdatenwert spei-

chert. **Fig. 82C** veranschaulicht ein 64-Bit-Ziel, das einen 32-Bit-Tiefenwert und zwei 16-Bit-Gleitkommawerte speichert. Das Folgende ist ein beispielhaftes Atomic:

InterlockedCmpAdd_type1_type2_cmp_op()

• type1 = int, uint, float

• type2 = int, uint, float, 2xfp16

• cmp_op = less, greater, equal, less_equal, greater_equal, not_equal

• z. B.: InterlockedCmpAccum float_2xfp16_less()

• falls der neue float-Tiefenwert < als der gespeicherte float-Tiefenwert ist:

2. Austauschen des gespeicherten Tiefenwerts gegen den neuen

3. Dest.Payload.lowfp16 += InputPayload.lowfp16

4. Dest.Payload.highfp16 += InputPayload.highfp16

Neue HLSL-Intrinsik für RWByteAddressBuffers

[0593] Nur der HighCompValue ist von dem Typ, der mit den hohen 32 Bits an dem 64-Bit-Ziel verglichen werden soll. Die AddLowVal kann vom Typ „float", „int", „uint" und min16float2' sein:

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Less(uint byteAddress64, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _LessEqual(uint byteAddress64, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Greater(uint byteAddress64, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _GreaterEqual(uint byteAddress64, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Equal(uint byteAddress64, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _NotEqual(uint byteAddress64, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Less(uint byteAddress64, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _LessEqual(uint byteAddress64, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Greater(uint byteAddress64, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _GreaterEqual(uint byteAddress64, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Equal(uint byteAddress64, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _NotEqual(uint byteAddress64, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Less(uint byteAddress64, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _LessEqual(uint byteAddress64, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Greater(uint byteAddress64, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _GreaterEqual(uint byteAddress64, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _Equal(uint byteAddress64, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void RWByteAddressBuffer::InterlockedCompareExchangeHighAddLow _NotEqual(uint byteAddress64, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

Vorgeschlagene neue HLSL-Intrinsiken für Ziel R

[0594] Nur der HighCompValue ist von dem Typ, der mit den hohen 32 Bit an dem 64-Bit-Ziel verglichen werden soll. Die AddLowVal kann vom Typ „float", „int", „uint" und min16float2' sein:

void InterlockedCompareExchangeHighAddLow_LessEqual(R dest, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_Greater(R dest, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_GreaterEqual(R dest, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_Equal(R dest, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_NotEqual(R dest, uint uHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_Less(R dest, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_LessEqual(R dest, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_Greater(R dest, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_GreaterEqual(R dest, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_Equal(R dest, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_NotEqual(R dest, int iHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_Less(R dest, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_LessEqual(R dest, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_Greater(R dest, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_GreaterEqual(R dest, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_Equal(R dest, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

void InterlockedCompareExchangeHighAddLow_NotEqual(R dest, float fHighCompVal, type AddLowVal, out uint2 HighAndLowOrgVal)

IV. Atomics für Ringpuffer

[0595] Ein Ringpuffer (oder kreisförmiger Puffer) ist eine Datenstruktur, die einen einzelnen Puffer mit fester Größe umfasst, der so arbeitet, als wäre er Ende-zu-Ende verbunden. Üblicherweise werden kreisförmige

Puffer zum Puffern von Datenströmen verwendet. Eine Ausführungsform der Erfindung beinhaltet Atomics zum Anhängen und Entfernen per Pop von Einträgen zu und von Ringpuffern.

[0596] Anfänglich sind AppendIndex und PopFrontIndex 0. Um atomares Anhängen oder Entfernen per Pop zu ermöglichen, verwendet eine Ausführungsform spezielle 64-Bit-Atomics. Mit diesen Atomics können GPU-Threads zum Beispiel ein Hersteller-Verbraucher-Schema innerhalb der Grenzen der Kapazität des Ringpuffers implementieren. Ein Hardware-Watchdog kann Kernels wecken, die auf den Ringpuffer warten.

[0597] Die folgenden Codesequenzen veranschaulichen eine Atomic-Operation zum Anhängen und Entfernen per Pop von Einträgen von einem Ringpuffer gemäß einer Ausführungsform der Erfindung:

a. Ringpuffer-Append

```
InterlockedAppend(in dest64, in RingSize, out AppendIndexOut)
atomically execute (
    if( ( (dest64.AppendIndex+1) % RingSize ) != (
    dest64.PopFrontIndex % RingSize))
    {
      AppendIndexOut = dest64.AppendIndex;
      ++dest64.AppendIndex;
    }
    else
    {
      AppendIndexOut = 0xffffffff; // Fehler, Ringpuffer voll
    }
)
```

b. Ringpuffer-PopFront

```
InterlockedPopFront( in dest64, in RingSize, out PopIndexOut)
atomically execute (
    if( ( (dest64.PopFrontIndex) % RingSize) ! = (
    dest64.AppendIndex % RingSize) )
    {
      PopIndexOut = dest64.PopFrontIndex;
    ++dest64. PopFrontIndex;
    }
    else
    {
      PopIndexOut = 0xffffffff; // Fehler, Ringpuffer leer

    }
)
```

c. Beispielhafte Anwendunasfälle

i. Initialisieren des Ringpuffers mit verfügbarer Anzahl von Einträgen durch Verwenden von InterlockedAppend

ii. Eine Anzahl von Threads läuft, und nimmt vorübergehend Einträge durch Verwenden von Interlocked-PopFront auf/weist diese zu

iii. Einträge werden unter Verwendung von InterlockedAppend in den Ringpuffer zurückgegeben

iv. Threads können entscheiden, nicht auf Einträge zu warten, und diesen Fall behandeln

Pseudocode für ein Multi-Hersteller-Sample und ein Multi-Verbraucher-Sample sind in den **Fig. 84-85** veranschaulicht.

[0598] Ein Hersteller-Pseudocode-Sample ist in **Fig. 84A** veranschaulicht. Für dieses Beispiel wird angenommen, dass der job_entry_ready_bufferauf nur Nullen initialisiert wird, und der job_entry_consumed_buffer auf nur Einsen initialisiert wird:

[0599] Ein Verbraucher-Pseudocode-Sample ist in **Fig. 84B** veranschaulicht. Für dieses Beispiel wird angenommen, dass der Job_entry_ready_buffer auf nur Nullen initialisiert wird, und der Job_entry_consumed_bufferauf nur Einsen initialisiert wird.

[0600] **Fig. 83A** veranschaulicht einen beispielhaften Ringpuffer, der gemäß einer Ausführungsform implementiert wird. Es ist eine Ringpuffer-Popback-Operation gezeigt, bei der Einträge N, N+1 usw. per Pop entfernt und in Ringpuffereinträgen 0, 1 usw. gespeichert werden. **Fig. 83B** veranschaulicht ein 64-Bit-Zielregister 8211, das den Append(Anhängen)-Indexwert 8212 und den Popfront-Indexwert 8213 speichert, gemäß der folgenden Codesequenz:

```
InterlockedPopBack( in dest64, in RingSize, out PopIndexOut )
atomically execute (


 if( ( (dest64.PopFrontIndex) % RingSize ) != (
 dest64.AppendIndex % RingSize) )
 {
   PopIndexOut = dest64.AppendIndex;
   --dest64.AppendIndex;
 }
 else
 {
    PopIndexOut = 0xffffffff; // Fehler, Ringpuffer leer
 }       )
```

V. Atomic-Multiplikationsoperationen

[0601] Eine Ausführungsform eines Multiplizieren-Atomic spezifiziert ein Ziel und einen Typwert. Ein Multiplizieren-Atomic kann beispielsweise die folgende Form annehmen:

InterlockedMultiply(destination, type value)

[0602] In einer Ausführungsform multipliziert die Multiplikationsoperation atomar einen Wert eines spezifizierten Datentyps mit dem Wert in dem Ziel, der derselbe Datentyp oder ein unterschiedlicher Datentyp sein kann.

[0603] Multiplizieren-Atomic-Werte können beispielhaft und nicht einschränkend 4-Bit-, 8-Bit-, 16-Bit , 32-Bit- und 64-Bit-Ganzzahlen und 16-Bit-, 32-Bit- und 64-Bit-Gleitkommawerte sein. Die Werte können vorzeichenbehaftet oder vorzeichenlos sein. Darüber hinaus kann eine Anzahl paralleler Multiplikationsoperationen basierend auf der kleinsten Datenelementgröße durchgeführt werden. Die Gleitkomma-Multiplikationsschaltungsanordnung kann zum Beispiel dazu ausgelegt sein, eine einzelne 32-Bit-Gleitkomma-Multiplikation oder duale 16-Bit-Gleitkomma-Multiplikationen durchzuführen. Formate, wie etwa Bfloat16 oder TensorFloat16, können verwendet werden, um die parallelen Multiplikationen effizient durchzuführen. In ähnlicher Weise kann ein Ganzzahl-Multiplikator dazu geeignet sein, eine einzelne 32-Bit-Multiplikation, duale 16-Bit-Multiplikationen, vier 8-Bit-Multiplikationen oder acht 4-Bit-Multiplikationen durchzuführen. Verschiedene andere Typen von Datenformaten und parallelen Operationen können verwendet werden, während die zugrundeliegenden Prinzipien der Erfindung weiterhin erfüllt werden, einschließlich zum Beispiel 2 × FP16, float2, 4 × FP16, 11_11_10FP und 2 × 11_11_10FP.

[0604] Diese Atomics können für eine Vielzahl von Zwecken verwendet werden, einschließlich Operationen für maschinelles Lernen, Weighted Blended Order Independent Transparence (OIT) (gewichtete, gemischte reihenfolgeunabhängige Transparenz) oder Opacity Shadow Maps (Opazität-Schatten-Abbildungen).

EINRICHTUNG UND VERFAHREN FÜR GRAFIKPROZESSOR-VERWALTETE GEKACHELTE RESSOURCEN

[0605] Eine Ausführungsform der Erfindung verbessert die Effizienz, mit der ein durch den Benutzer geschriebenes GPU-Programm Daten, die in einem Puffer oder einer Textur gespeichert sind, cachen und wiederverwenden kann. Diese Ausführungsform stellt auch eine logische Repräsentation großer prozedural berechneter Ressourcen bereit, die gleichzeitig physisch in den GPU-Speicher passen können oder nicht.

[0606] In einer Ausführungsform der Erfindung wird eine neue gekachelte Ressource durch die GPU definiert und verwaltet, hierin als eine GPU-verwaltete gekachelte Ressource oder ein GPU-verwalteter Puffer bezeichnet. In einer Implementierung enthält der Puffer oder die andere gekachelte Speicherungsressource bis zu N Speicherblöcke mit fester Größe. Unterschiedliche GPU-Architekturen können eine unterschiedliche maximale Anzahl von Blöcken (N) unterstützen.

[0607] In einer Ausführungsform wird die GPU-verwaltete gekachelte Ressource verwendet, um Daten effizient zwischen Shadern gemeinsam zu nutzen, d. h., wobei ein Shader als ein „Hersteller“ für einen oder mehrere „Verbraucher“-Shader agiert. Zum Beispiel kann der Hersteller-Shader prozedural aktualisierten Inhalt erzeugen, den der Verbraucher-Shader verwenden kann, ohne eine Interaktion mit der CPU einzubeziehen. Als ein anderes Beispiel müssen bei Raytracing-Implementierungen möglicherweise verschiedene Formen einer Skinning-Animation beim Traversieren aktualisiert werden. Ein Shader kann bei einem kleinen Teil des Mesh Skinning durchführen, wodurch Ergebnisse in der gekachelten Ressource ohne CPU-Intervention gespeichert werden. Während andere Strahlen denselben Abschnitt verfolgen, können sie lokal von der gekachelten Ressource auf die Daten zugreifen, ohne auf den Hauptspeicher zuzugreifen.

[0608] **Fig. 85A** veranschaulicht eine Ausführungsform einer Architektur zum Implementieren von GPU-verwalteten gekachelten Ressourcen 8531. Ein Grafikprozessor 8521 beinhaltet einen Scheduler 8510 zum Planen von Shadern 8511A-B auf dem Satz von Ausführungseinheiten 4001. Die Ausführung der Shader erfordert einen Zugriff auf die gekachelten Ressourcen 8531, die durch einen Ressourcenmanager 8512 verwaltet werden. In dem unten bereitgestellten Beispiel wird ein Shader 8511A als ein „Hersteller“ designiert, der seine Ergebnisse in der gekachelten Ressource 8531 speichert, und der andere Shader 8511B ist ein „Verbraucher“, der die durch den Hersteller-Shader 8511A erzeugten Ergebnisse verwendet. Infolgedessen muss der Hersteller-Shader 8511A Schreibzugriff in die gekachelte Ressource 8521 haben, und der Verbraucher-Shader 8511B muss Lesezugriff auf die gekachelte Ressource 8531 haben. Es ist jedoch anzumerken, dass eine Hersteller/Verbraucher-Architektur nicht erforderlich ist, um die zugrundeliegenden Prinzipien der Erfindung zu erfüllen.

[0609] In einer Implementierung umfasst die gekachelte Ressource 8531 einen On-Chip-Kachelspeicher oder -Kachelpuffer, der kachelgroße Blöcke 0-(N-1) von Daten speichert. Die „Kachel“-Größe kann basierend auf der Architektur des Grafikprozessors 8521 und der Konfiguration der Grafikverarbeitungs-Pipeline variabel sein. In einer Ausführungsform ist die Grafikverarbeitungs-Pipeline dazu ausgelegt, durch Verwenden der gekachelten Ressource 8531 kachelbasiertes verzögertes Rendern, kachelbasiertes Immediate-Modus-Rendern und/oder eine andere Form von kachelbasierter Grafikverarbeitung durchzuführen.

[0610] In einer Ausführungsform fordert eine Ausführungseinheit (EU) 4001 oder eine andere Verarbeitungseinheit einen Block durch Verwenden eines Hash-Werts oder einer anderen Form von ID 8501 (z. B. eines 64-Bit-Hash in einer Ausführungsform) an. Ein Ressourcenmanager 8512 bestimmt, ob der Block innerhalb der gekachelten Ressource 8531 existiert, die N Blöcke mit fester Größe umfasst. Falls kein derartiger Block gefunden wird, räumt der Puffermanager 8510 den am längsten nicht verwendeten (Least Recently Used - LRU) Block, oder wählt einen unbenutzten Block aus, falls ein solcher existiert. Die Antwort 8502 identifiziert den zugewiesenen Block, den der Puffermanager 8510 mit dem gegebenen Hash-Wert als „verwendet“ markiert. In einer Implementierung wird auch ein Flag zurückgegeben, das angibt, dass der Block neu ist. Ein am längsten nicht verwendeter Block, der ersetzt wird, verliert den alten Inhalt, den er gespeichert hat. Falls der Block bereits vorhanden ist, wird ein Flag zurückgegeben, das angibt, dass der Block bereits vorhanden ist, und er wird dennoch zurückgegeben.

[0611] Obwohl als eine Komponente innerhalb des Grafikprozessors 8521 veranschaulicht, kann die gekachelte Ressource 8531 innerhalb eines Speichers außerhalb des Grafikprozessors 8521, wie etwa eines Systemspeichers oder eines Cache auf Systemebene, implementiert werden.

[0612] Es ist a priori bekannt, dass bestimmte Klassen von Shadern 8511A-B, die auf den EUs 4001 einer GPU ausgeführt werden, einen Speicherblock benötigen. Zum Beispiel können diese Shader immer in den Spuren einer Welle ausgeführt werden. In einer Ausführungsform konstruiert der Scheduler 8510, der die Ausführung dieser Shader 8511A-B plant, eine(n) 64-Bit-ID/-Hash von systemerzeugten Werten. Zum Beispiel verwendet eine Ausführungsform in dem Kontext des Raytracing die InstanceID und die GeometryID, um einen eindeutigen 64-Bit-Hash zu konstruieren. Jedoch kann eine Vielzahl anderer systemerzeugter Variablen verwendet werden.

[0613] In dieser Ausführungsform prüft der Scheduler 8510 über den Ressourcenmanager 8512, ob bereits ein Block der gekachelten Ressource 8531 für den 64-Bit-Hash zugewiesen ist. Falls ja, wird der Shader 8511A-B unter der Annahme ausgeführt, dass der Block bereits gecachte Daten enthält, und dass diese durch den Shader verbraucht werden können, und der Shader wird auf den EUs 4001 geplant. Der Ressourcenmanager 8512 sperrt den Speicherblock gegen Wiederverwenden, solange der Shader, der die in diesem Block gesperrten gecachten Daten verwendet, ausgeführt wird. Während der Shader durch eine oder mehrere EUs 4001 ausgeführt wird, aktualisiert er den Block in der gekachelten Ressource 8531 durch Verwenden der Block-ID 8501, und empfängt für bestimmte Operationen Antworten 8502 von dem Ressourcenmanager 8512.

[0614] In einer Ausführungsform, falls der Scheduler 8510 anfänglich feststellt, dass es keinen Block mit dem gegebenen 64-Bit-Hash gibt, lokalisiert der Ressourcenmanager 8512 einen unbenutzten Block, oder verwendet den am längsten unbenutzten Block (oder einen anderen Block), der bereits zugewiesen hat, und gegenwärtig nicht verwendet wird. Falls er einen solchen Block nicht lokalisieren kann, kann er eine Ausführung des Shaders aufschieben, bis ein solcher Block verfügbar wird. Wenn ein solcher verfügbar ist, sperrt der gekachelte Ressourcenmanager 8512 den gekachelten Ressourcenblock gegen Wiederverwendung, solange der Shader ausgeführt wird, und plant den Shader. Ein Flag kann an den Shader weitergegeben werden, um anzugeben, dass der Block leer ist, und dass der Shader ihn verwenden kann, um Daten zu erzeugen und zu speichern. Nach dem Schreiben von Daten in den gekachelten Ressourcenblock kann der Shader die Ausführung fortsetzen, als wäre der gekachelte Ressourcenblock mit seinen Daten bereits verfügbar gewesen.

[0615] Zurückkehrend zu dem Beispiel von Verbraucher/Hersteller oben kann ein Hersteller-Shader 8511A geplant werden, um einen neuen Block oder eine neue Kachel der prozeduralen Ressource 8531 zu erzeugen, falls der angeforderte Hash in dem Pool nicht gültig ist. Ein solcher angeforderter Hash kann durch einen oder mehrere Verbraucher-Shader 8511B erzeugt werden, die der Ressourcenmanager 8512 blockieren würde, bis ihre Anforderung gefüllt ist.

[0616] In einer Ausführungsform werden gekachelte Ressourcenblöcke zu einer Solid-State-Vorrichtung 8515 oder einem anderen Hochgeschwindigkeits-Speicherungsmedium geräumt. Die SSD 8515 oder eine andere Speicherungsvorrichtung kann lokal auf demselben Substrat und/oder derselben Karte wie der Grafikprozessor 8521 integriert sein, und kann dazu ausgelegt sein, gekachelte Ressourcenblöcke und andere Daten während interner Kontextwechsel des Grafikprozessors 8521 abzuspeichern.

[0617] Ein Verfahren gemäß einer Ausführungsform ist in **Fig. 85B** veranschaulicht. Das Verfahren kann im Kontext der oben beschriebenen Architekturen implementiert werden, ist aber nicht auf irgendeine spezielle Architektur beschränkt.

[0618] Bei 8551 evaluiert der Scheduler den nächsten Shader, der zur Ausführung geplant werden soll, und bestimmt bei 8552 eine Hash-ID, die verwendet werden soll, um den gekachelten Ressourcenblock zu identifizieren (z. B. durch Verwenden einer oder mehrerer der hierin beschriebenen Techniken). Bei 8553 fragt der Scheduler den gekachelten Ressourcenmanager mit der Hash-ID ab.

[0619] Falls ein Block für diese Hash-ID bereits zugewiesen ist, bei 8554 bestimmt, dann sperrt der gekachelte Ressourcenmanager bei 8555 den gekachelten Ressourcenblock, und der Shader verwendet bei 8556 den gekachelten Ressourcenblock während der Ausführung. Der gekachelte Ressourcenblock kann anschließend entsperrt werden, wenn der Shader abgeschlossen ist, es sei denn, er wird mit einer Hash-ID

eines Verbraucher-Shaders gesperrt, der die Daten anfordern wird, nachdem der aktuelle (Hersteller-) Shader abgeschlossen ist. In jedem Fall kehrt der Prozess zu 8551 zum Planen des nächsten Shaders zurück.

[0620] Falls bei 8554 kein gekachelter Ressourcenblock mit der Hash-ID identifiziert wird, dann weist der gekachelte Ressourcenmanager der Hash-ID einen gekachelten Ressourcenblock zu, und kann dem Shader ein Flag zuweisen, das angibt, dass er diesen gekachelten Ressourcenblock verwenden kann. Wie erwähnt, kann der gekachelte Ressourcenmanager existierende Daten von einem gekachelten Ressourcenblock räumen, um den gekachelten Ressourcenblock dem aktuellen Shader zuzuweisen. Der gekachelte Ressourcenblock wird bei 855 gesperrt, und der Shader verwendet bei 8556 den gekachelten Ressourcenblock während der Ausführung.

[0621] Der GPU-verwaltete gekachelte Puffer 8531 kann auf eine Vielzahl von Weisen verwendet werden. Zum Beispiel möchte eine SIMD-Welle von Spuren in denselben Überschneidungs-Shader-Rahmen eintreten, der durch einen bindungslosen Thread-Dispatcher (unten beschrieben) gebündelt wird. Bevor der Überschneidungs-Shader ausgeführt wird, fordert die Hardware einen Block von dem Puffermanager 8510 an.

[0622] Der 64-Bit-Hash kann auf unterschiedliche Weisen erzeugt werden. Zum Beispiel ist in einer Ausführungsform der 64-Bit-Hash die InstanzID der aktuellen Strahltraversierungsinstanz, kombiniert mit dem Frame-Zähler. Falls der Block neu ist, kann die Hardware einen Benutzer-Berechnungs-Shader starten, der innerhalb der Spuren der Welle läuft, der dann den Block füllt (z. B. mit Dreiecken, die Skinning unterzogen wurden). Falls der Block alt ist, dann wird der Shader möglicherweise nicht gestartet. Dann wird ein Überschneidungs-Shader ausgeführt, der mit dem Zeiger auf den Block versehen wird. Der Überschneidungs-Shader kann dann Strahl/Dreieck-Überschneidungen durchführen und/oder eine Unterstützung kann für eine Hardware-Anweisung für die Strahl/Dreieck-Überschneidungen bereitgestellt werden (wie hierin beschrieben). Alternativ kann der Block so entworfen sein, dass er nur Dreiecke enthält. In diesem Fall iteriert die Hardware über diese Dreiecke (ohne eine BVH über diese zu erstellen), und kann zum Beispiel Nächstgelegener-Treffer-Shader aktualisieren, oder in Beliebiger-Treffer-Shader aufrufen. Verschiedene andere Verwendungsfälle können sich die GPU-verwaltete gekachelte Ressource 8531, wie oben beschrieben, zunutze machen.

EINRICHTUNG UND VERFAHREN ZUM EFFIZIENTEN LAZY-BVH-BUILD

[0623] Komplexe dynamische Szenen sind für Echtzeit-Raytracing-Implementierungen herausfordernd. Prozedurale Oberflächen, Skinning-Animationen usw. erfordern Aktualisierungen von Triangulations- und Beschleunigungsstrukturen in jedem Frame, noch bevor der erste Strahl gestartet wird.

[0624] Lazy-Builds evaluieren Szenenelemente „on-demand“ (nach Bedarf), wie durch Strahltraversierung angesteuert. Das Rendern eines Frames startet mit einer groben Beschleunigungsstruktur, wie einem Szenengraphen oder Hierarchien des vorherigen Frames, und baut dann progressiv die neu benötigten Beschleunigungsstrukturen für die Objekte auf, die während der Traversierung durch Strahlen getroffen werden. Nicht sichtbare Objekte können effektiv von dem Konstruktionsprozess ausgeschlossen werden. Diese Techniken werden jedoch nicht einfach mit aktuellen Systemen und APIs implementiert, da die Programmierbarkeit höherer Ebene (d. h. pro Objekt), die für das Berechnen der Instanzsichtbarkeit wesentlich ist, nicht unterstützt wird.

[0625] Eine Ausführungsform der Erfindung unterstützt einen Multipass-Lazy-Build (MPLB) zum Echtzeit-Raytracing, der diese Probleme mit einem erweiterten Programmiermodell löst. Es ermöglicht, dass die Traversierung auf Instanzebene während jedes Strahlen-Dispatch verfolgt wird, und baut selektiv Beschleunigungsstrukturen unterster Ebene (Bottom Level Acceleration Structures - BLAS) nur für die potenziell sichtbare Geometrie zur Renderzeit auf. Ähnlich zu manchen adaptiven Sampling-Techniken kann MPLB, wie hierin beschrieben, mehrere Strahlen-Dispatches über denselben Satz von Pixeln erfordern, um Strahlen erneut zu zuvor nicht aufgebauten Teilen der Szene zu starten, aber bestimmte Ausführungsformen der Erfindung beinhalten Techniken zum Minimieren dieses Overhead, wie etwa die Annahme von Frame-zu-Frame-Kohärenz und gerasterter Primärsichtbarkeit. Diese Techniken können eine erhebliche Reduzierung der Build-Komplexität im Vergleich zu einmaligen Builders mit nur einer marginalen Zunahme der durchschnittlichen Traversierungskosten bereitstellen.

[0626] **Fig. 86A** veranschaulicht eine Ausführungsform eines On-Demand-(oder „lazy“)-Builders 8607 zum Durchführen von Lazy-Build-Operationen, wie hierin beschrieben. Zusätzlich beinhaltet diese Ausführungsform Traversierungssuspensionsschaltungsanordnung/-logik 8620 zum Suspendieren der Strahltraversie-

rung. Die Strahlentraversierungssuspensionsschaltungsanordnung/-logik 8620 kann in Hardware, Software oder einer beliebigen Kombination davon implementiert sein. Eine Strahlstapelspeicherung 8605 speichert suspendierte Strahlstapel 8610, wenn die Traversierung suspendiert wird (wie hierin ausführlicher beschrieben). Zusätzlich startet ein GPU-seitige Befehls-Scheduling Lazy-Build-Aufgaben und Strahlfortsetzungen auf den Ausführungseinheiten 4001 ohne Überwachung durch die CPU. Traversierungs-Atomics werden auch verwendet, um den Shader-Overhead zu reduzieren.

Traversierungssuspension bei fehlendem Antreffen der Beschleunigungsstruktur unterster Ebene (BLAS)

[0627] In einer Implementierung werden fehlende Instanzen (z. B. fehlende Beschleunigungsstrukturen der untersten Ebene der BVH 8005) durch Verwenden eines Programmiermodells mit einer Traversierungs-Shader-Erweiterung programmatisch markiert, sodass sie in einem separaten Durchlauf identifiziert und aktualisiert werden können. Dann wird entweder eine unvollständige Traversierung durchgeführt, oder die Traversierung wird abgebrochen.

[0628] Um die finalen Pixel zu rendern, muss der Primär-Shader des entsprechenden Pixels möglicherweise neu gestartet werden, was zu mehreren wiederholten Traversier- und Shader-Ausführungsoperationen führt. In einer Ausführungsform sichert die Traversierungssuspensionslogik 8620 den gesamten Strahlenkontext 8610 (Strahlstapel, Fortsetzungen usw.) in den Off-Chip-Speicher 8605, wenn die Traversierung suspendiert wird. In einer Ausführungsform ist diese Traversierungssuspension eine intrinsische Funktion, die durch den Treiber (z. B. SuspendTraversal()) verwaltet wird; die der Erfindung zugrundeliegenden Prinzipien sind jedoch nicht auf diese Implementierung beschränkt. Zusätzlich plant eine neue DispatchRay()-Variante auf der Host-Seite, die durch die CPU 3199 ausgeführt wird, die suspendierten Strahlstapel von dem Strahlkontext 8610 neu, um eine Traversierungs-Shader-Ausführung fortzusetzen.

GPU-seitiges Befehls-Scheduling für Build und Dispatch

[0629] Ein weiterer erheblicher Overhead aktueller Lazy-Build-Implementierungen ist die kontinuierliche Anforderung des Zurücklesens der CPU 3199 und des bedingten Scheduling des BVH-Builders 8007 und des Strahl-Dispatching auf der GPU 2505. Um die Effizienz zu verbessern, führt in einer Implementierung die BVH-Verarbeitungsschaltungsanordnung/-logik 8004 den BVH-Build asynchron mit der Strahltraversierung 8003 aus. Nach Abschluss der Build-Aufgaben führt die Raytracing-Engine 8000 das Strahl-Dispatch aus, um die suspendierten Strahlstapel von dem Strahlkontext 8610 fortzusetzen.

Traversierungs-Atomics zur Reduzierung von Traversierungs-Shader-Overhead

[0630] Ein Problem mit aktuellen Implementierungen ist es, dass, falls eine Instanz fehlt (unaufgebaut ist), mehrere Strahlen sie traversieren können, und sie für den Lazy-Builder 8607 markieren können, um ihn zu aktualisieren. Eine einfache Aufgabe, die durch nur einen Traversierungs-Shader-Aufruf ausgeführt werden könnte, wird durch hunderte oder mehr Aufrufe wiederholt. Der Traversierungs-Shader ist nicht ressourcenintensiv, er hat jedoch einen erheblichen Overhead zum Starten, Durchführen von Eingabe/Ausgabe-Funktionen und Speichern von Ergebnissen.

[0631] In einer Ausführungsform der Erfindung können unaufgebaute Instanz-Leaves als „Atomic“-Knoten markiert werden. Atomic-Knoten können von nur einem Strahl auf einmal traversiert werden. Ein Atomic-Knoten wird gesperrt, sobald ein Strahl ihn traversiert, und an dem Ende der Traversierungs-Shader-Ausführung wird es entsperrt. In einer Ausführungsform setzt der Traversierungs-Shader den Status eines Knotens auf „ungültig“, was verhindert, dass Strahlen in ihn eintreten, selbst nachdem die Sperre aufgehoben wurde. Dies ermöglicht es der Traversierungs-Hardware, entweder den Knoten zu überspringen, oder die Traversierung des Strahls zu suspendieren, ohne einen neuen Traversierungs-Shader auszuführen.

[0632] In einer Ausführungsform werden für Atomic-Knoten anstelle regulärer Atomic-Semantik bestimmte Mutex-/Zustandssemantiken verwendet. Falls zum Beispiel die Traversierungsschaltungsanordnung/-logik 8003 einen Strahl zu einem Proxy-Knoten traversiert, versucht sie, den Knoten zu sperren. Falls dies scheitert, da der Knoten bereits gesperrt ist, führt sie automatisch „suspendRay“ aus, ohne zu der EU 4001 zurückzukehren. Falls das Sperren erfolgreich ausgeführt wird, verarbeitet die Traversierungsschaltungsanordnung/-logik 8003 den Proxy-Knoten.

[0633] Eine Ausführungsform der Erfindung arbeitet gemäß dem in **Fig. 86B** gezeigten Verarbeitungsfluss. Als Übersicht baut der On-Demand-Builder 8607 Beschleunigungsstrukturen über Geometrieinstanzen 8660 auf, von denen bestimmt wird, dass sie potenziell sichtbar sind. Die potenziell sichtbaren Instanzen 8660 werden durch einen Pre-Builder 8655 basierend auf primären Sichtbarkeitsdaten vom G-Puffer 8650 und Sichtbarkeitshistoriendaten 8651, die Sichtbarkeit in dem vorherigen Frame angeben, erzeugt. Die potenziell sichtbaren Instanzen 8660 können auch basierend auf einer sichtbaren Abbildung 8675 der Beschleunigungsstruktur unterster Ebene (BLAS) bestimmt werden, die die Knoten unterster Ebene der Beschleunigungsstruktur angibt, die sichtbare Primitive beinhalten. In einer Ausführungsform wird die sichtbare BLAS-Abbildung 8675 als Reaktion auf Traversierungsoperationen, die durch die Traversierungslogik 8670 durchgeführt werden, die dedizierte Traversierungsschaltungsanordnungen und/oder auf den Ausführungseinheiten des Grafikprozessors ausgeführte Traversierungs-Shader beinhalten kann, laufend aktualisiert.

[0634] Der On-Demand-Builder 8607 erzeugt diejenigen Abschnitte der Beschleunigungsstruktur, die mit den potenziell sichtbaren Instanzen 8660 assoziiert sind. Ein Strahlerzeugungs-Shader 8678 erzeugt selektiv Strahlen basierend auf diesen Abschnitten der Beschleunigungsstruktur, die die Traversierungseinheit 8670 durch die Beschleunigungsstrukturabschnitte traversiert. Die Traversierungseinheit 8670 benachrichtigt den On-Demand-Builder 8670 über zusätzliche Beschleunigungsstrukturknoten, die er zum Traversieren benötigt, und aktualisiert die BLAS-Pixelmasken 8677, die durch den Strahlerzeugungs-Shader 8678 (der z. B. nur Strahlen für unmaskierte Pixel erzeugt) und die sichtbare BLAS-Abbildung 8675 verwendet werden.

[0635] Somit baut der On-Demand-Builder 8706 selektiv Beschleunigungsstrukturen der untersten Ebene über den potenziell sichtbaren Instanzen 8660 auf, und die Instanzsichtbarkeit wird während der Strahltraversierung 8670 aktualisiert. Im Gegensatz zu den vorherigen Implementierungen arbeiten die Ausführungsformen der Erfindung in mehreren Durchläufen, um kompliziertes Strahl-Scheduling zu vermeiden. Die Idee ist analog zu neueren Texturraumschattierungsansätzen, bei denen eine sichtbarkeitsgesteuerte Markierung von Texeln verwendet wird, um redundante Schattierung vor dem endgültigen Rendern zu vermeiden.

[0636] Im Betrieb werden zuerst die BLASes für leere Instanzen aufgebaut, die im vorherigen Durchlauf als potenziell sichtbar markiert wurden. In dem zweiten Durchlauf schießt der Strahlerzeugungs-Shader 8678 die Strahlen selektiv zu den nicht fertigen Pixeln zurück, wobei ein Traversierungs-Shader verwendet wird, um entweder mehr potenziell sichtbare leere Instanzen aufzuzeichnen oder das Pixel abzuschließen. Die Anzahl unvollständiger Pixel nimmt nach jeder Iteration ab, bis es keine Strahlen mehr gibt, die eine leere Instanz traversiert haben.

[0637] Eine Ausführungsform der Erfindung führt ein hybrides Rendern durch Verwenden des GPU-Rasterisierers und der Raytracing-Hardware gemeinsam durch. Dies liegt daran, dass, wenn der G-Puffer 8650 erstellt wird, die Primärsichtbarkeit aller Instanzen in der Szene einfach erhalten wird. Daher nutzt der Pre-Builder 8655 in diesen Ausführungsformen hybrides Rendern durch effizientes Konstruieren der anfänglichen Beschleunigungsstruktur durch Verwenden dieser Daten. Vor der ersten Iteration werden potenziell sichtbare Instanzen 8660 in dieser Pre-Build-Heuristik markiert (wie unten diskutiert).

[0638] Die Codesequenz unten ist eine abstrahierte High Level Shader-Sprache (HLSL), die eine Ausführungsform des beschriebenen Traversierungs-Shaders mit einigen intrinsischen Funktionen und Benutzerfunktionen beschreibt:

```
RWStructuredBuffer<vblas> visibleBlasMap[] : register(u0, space0);
RWStructuredBuffer<pmask> pixelMasks[] : register(u0, spacel);

[shader("traversal")]
void myVisibilityShader(in RayPayload rp) {
  uint2 index = DispatchRaysIndex();
  uint2 size = DispatchRaysDimensions();

  UpdateVisibility(visibleBlasMap, InstanceID(), true);

  // Control BLAS traversal with updating pixel mask
  RaytracingAccelerationStructure myAccStructure;
```

```
    bool isInstanceEmpty = IsEmptyInstance();
    if (isInstanceEmpty) {
      UpdateMask(pixelMasks, index.y*size.x + index.x, false);
      rp.trav_valid = false;
      skipTraversal ();
    }
    else if (!isInstanceEmpty && !rp.trav_valid)
      skipTraversal ();
    else {
      myAccStructure = FetchBLAS(InstanceID());
      RayDesc transformedRay = {...};
      // Set the next level instance and hit shader table offset
      SetInstance(myAccStructure, transformedRay, hitShaderOffset);
    }
  }
```

[0639] Die SkipTraversal()-Intrisik wird definiert, um die aktuelle Instanz zu ignorieren, und die Traversierung in der Beschleunigungsstruktur höherer Ebene fortzusetzen. Wie erwähnt, wird die sichtbare Abbildung 8675 der Beschleunigungsstruktur unterster Ebene (BLAS) verwendet, um Instanzsichtbarkeit aufzuzeichnen, die üblicherweise in Beschleunigungsstruktur-Buildern und Traversierungs-Shadern verwendet wird. Wie in **Fig. 86C** gezeigt, enthält eine Ausführungsform der sichtbaren BLAS-Abbildung 8675 ein Flag 8676, das mit jeder BLAS-ID 8674 assoziiert ist, das die BLAS-Sichtbarkeit angibt, auf die die Instanz verweist, und zwei Flags, Built_Full und Built_Empty, die angeben, ob die BLAS bereits aufgebaut wurde. Zusätzlich wird ein boolesches Flag, trav_valid, zu den Strahlnutzdaten hinzugefügt, um den Traversierungsstatus zu verfolgen, der verwendet werden kann, um zu prüfen, ob der Strahl bisher auf eine leere Instanz getroffen ist.

[0640] In einer Ausführungsform wird die Sichtbarkeit in dem Traversierungs-Shader konservativ aktualisiert, da alle traversierten Instanzen potenziell für den aktuellen Strahl sichtbar sind. Daher ist es die erste Aufgabe, das Sichtbarkeits-Flag als Wahr für die entsprechende BLAS der aktuellen Instanz zu setzen. Sie setzt auch das Flag der Sichtbarkeitshistorie (vis_history) als Wahr, um es in dem nächsten Frame wiederzuverwenden(Zeile 9 der obigen Codesequenz). Als Nächstes wird das Traversierungsziel basierend auf dem Status der aktuellen Instanz (leer oder voll) und dem Strahlstatus (d. h. dem trav_valid-Wert) bestimmt. Dies wird in drei Zustände 8690-8692 klassifiziert, wie in **Fig. 86D** gezeigt.

[0641] Für eine leere Instanz 8690 wird die entsprechende Pixelmaske zurückgesetzt (Zeile 15), um Strahlen beim nächsten Durchlauf erneut abzuschießen. Die aktuelle Traversierung wird dann durch Setzen des trav_valid-Flags in den Strahlnutzdaten ungültig gemacht (Zeile 16). Schließlich wird die TLAS-Traversierung fortgesetzt, indem SkipTraversal() aufgerufen wird.

[0642] Für den Fall 8691 der vollen Instanz und der ungültigen Traversierung hat die aktuelle Instanz eine aufgebaute BLAS, aber der Strahl hat bisher eine leere Instanz getroffen (d. h. trav_valid ist Falsch). Da der Strahl schließlich wieder auf das aktuelle Pixel geschossen wird, kann die BLAS-Traversierung übersprungen werden (Zeile 20).

[0643] Für eine volle Instanz und gültige Traversierung 8692 ruft der Traversierungs-Shader, da der Strahl die Beschleunigungsstruktur normal ohne leere Instanzen durchlief, die BLAS der aktuellen Instanz ab, und setzt die Traversierung fort. Falls der Strahl bis zu dem Ende der Traversierung Gültigkeit aufrechterhält, wird der Strahl normalerweise den Nächstgelegener-Treffer-Shader oder den Fehltreffer-Shader aufrufen und ausführen.

[0644] Andernfalls geben diese Shader die Steuerung zurück, ohne ihren Code auszuführen und den aktuellen Durchlauf abzuschließen, was die Overheads von Hardware-Strahltraversierung und Starten von Shadern für Sekundärstrahlen verhindert. In dem nächsten Durchlauf werden die Strahlen wieder nur auf das Pixel mit der Maske „Falsch“ geschossen, und eine gültige Traversierung für diese Pixel wird versucht.

[0645] Für die Beschleunigungsstruktur-Building-Operation werden die BLASes der Instanzen aufgebaut, oder leere Instanzen werden erzeugt, abhängig von dem Sichtbarkeits-Flag der Sichtbarkeitsbitmaske. Die potenziell sichtbare Instanz konstruiert normalerweise die BLAS (BUILD_FULL), und die nicht sichtbare Instanz berechnet nur den Begrenzungsrahmen der Geometrie, und packt ihn in den Leaf-Knoten der TLAS (BUILD_EMPTY). Es wird auch auf die anderen zwei Flags verwiesen, die angeben, ob eine BUILD_FULL-

oder BUILD_EMPTY-Aktion bereits für das aktuelle Objekt in dem vorherigen Durchlauf durchgeführt wurde. Durch Überprüfen dieser Flags können doppelte Handlungen für dasselbe Objekt in den unterschiedlichen Iterationen der Build-Traversierungs-Schleife vermieden werden.

[0646] Sobald der BLAS-Build-Prozess für die Objekte abgeschlossen ist, wird die endgültige Beschleunigungsstruktur konstruiert, indem die TLAS über diese BLASes aufgebaut werden. Die TLAS wird nur in dem ersten Durchlauf wieder aufgebaut, und in dem Rest der Durchläufe neu angepasst, da die Begrenzungsrahmen aller Objekte bereits in dem ersten Durchlauf eingerichtet sein könnten.

[0647] Wie oben beschrieben, führt eine Ausführungsform der Erfindung mehrere Durchläufe durch, wodurch sie mitunter redundant Strahlen für dasselbe Pixel abschießen lässt. Der aktuelle Durchlauf sollte nämlich die ungültige Traversierung in dem vorherigen Durchlauf wettmachen. Dies kann zu redundanten Hardware-Strahltraversierungen und Shader-Aufrufen führen. Eine Ausführungsform beschränkt diesen Overhead der Traversierungskosten jedoch nur auf die Pixel, die einer ungültigen Traversierung entsprechen, indem eine Pixelmaske angewendet wird.

[0648] Außerdem werden unterschiedliche Techniken verwendet, um potenziell sichtbare BLASes zu identifizieren (und sie aufzubauen), noch bevor der erste Strahl traversiert wird (z. B. durch den Pre-Builder 8655). Durch Verwenden des G-Puffers 8650 können direkt sichtbare Instanzen markiert werden, die wahrscheinlich durch Primärstrahlen traversiert werden. Darüber hinaus wird angenommen, dass es eine erhebliche Menge an Frame-zu-Frame-Kohärenz gibt; somit werden die BLASes von Instanzen, die in dem vorherigen Frame traversiert wurden, auch voraufgebaut. Die Kombination dieser beiden Techniken reduziert die Anzahl der Build-Traversierungs-Iterationen stark.

EINRICHTUNG UND VERFAHREN FÜR EINE MATERIAL-CULLING-MASKE

[0649] Existierende Raytracing-APIs verwenden eine 8-Bit-Cull-Maske, um die Strahltraversierung für bestimmte Geometrieinstanzen zu überspringen. Dies dient zum Beispiel dazu, spezifische Objekte am Schattenwerfen zu hindern, oder Objekte vor Reflexionen zu verbergen. Dieses Merkmal ermöglicht es, unterschiedliche Teilsätze der Geometrie innerhalb einer einzelnen Beschleunigungsstruktur zu repräsentieren, im Gegensatz zu dem Aufbau separater Beschleunigungsstrukturen für jeden Teilsatz. Die Biteinstellungen in der 8-Bit-Maske können verwendet werden, um die Traversierungsleistungsfähigkeit und den Ressourcen-Overhead zum Aufrechterhalten mehrerer Beschleunigungsstrukturen auszugleichen. Falls zum Beispiel ein Bit in der Maske auf 0 gesetzt ist, kann die entsprechende Instanz ignoriert werden.

[0650] Render-Engines können mehrere Geometrieinstanzen mit einem Asset assoziieren, und jede Geometrieinstanz kann mehrere Materialien enthalten. Aktuelle Raytracing-APIs ermöglichen jedoch nur die Spezifikation der Culling-Maske bei der Granularität einer Instanz. Dies bedeutet, dass Assets, die unterschiedliche Culling-Masken auf unterschiedlichen Materialien aufweisen, kein Standard-Culling verwenden können. Als eine Ausweichlösung verwenden aktuelle Implementierungen Beliebiger-Treffer-Shader, um Überschneidungen zu ignorieren, was aufwändig und kompliziert ist.

[0651] Wie in **Fig. 87** veranschaulicht, legt eine Ausführungsform der Erfindung diese Maskierungssteuerungen auf einer Pro-Material-Basis frei. Insbesondere beinhaltet eine Implementierung eine materialbasierte N-Bit-Cull-Maske 8701 zum Überspringen einer Strahltraversierung für Abschnitte von Geometrieinstanzen, die mit bestimmten Materialien assoziiert sind. In einer Ausführungsform wird eine materialbasierte 8-Bit-Cull-Maske verwendet, aber die zugrundeliegenden Prinzipien der Erfindung sind nicht auf diese Implementierung beschränkt. Im Gegensatz zu existierenden Implementierungen wird die materialbasierte Cull-Maske 8701 freigelegt, und kann durch die Traversierungsschaltungsanordnung/-logik 8003 zum Beispiel auf einer Pro-Material-Basis sowie auf einer Pro-Instanz-Basis genutzt werden.

[0652] In einer spezifischen Implementierung wird die N-Bit-Cull-Maske 8701 innerhalb einer Treffergruppe 8700 gespeichert, was ein Pro-Material-Culling mit fester Funktion bereitstellt, und die Notwendigkeit aufwändiger Beliebiger-Treffer-Shader-Ausweichlösungen verringert. Eine „Treffergruppe“ 8700, wie hierin verwendet, ist ein API-Objekt, das einen Satz von Shadern enthält, die zum Verarbeiten von Strahlen verwendet werden, die ein gegebenes Objekt in der Szene treffen. Der Satz von Shadern kann zum Beispiel einen Nächstgelegener-Treffer-Shader, einen Beliebiger-Treffer-Shader und (für prozedurale Geometrie) einen Überschneidungs-Shader beinhalten. In einer Implementierung wird die materialbasierte Cull-Maske 8701 mit der Treffergruppe 8700 als ein zusätzliches Datenelement assoziiert.

[0653] Um die Cull-Maske 8701 mit der Treffergruppe 8700 zu assoziieren, kann die Cull-Maske 8701 innerhalb der 32-Byte-Shader-Aufzeichnung gespeichert werden, den die API für die zu verwendende Implementierung bereitstellt (z. B. identifiziert über eine Aufzeichnungs-ID, wie hierin beschrieben). Es ist jedoch anzumerken, dass die zugrundeliegenden Prinzipien der Erfindung nicht auf jegliche bestimmte Technik zum Assoziieren einer Cull-Maske mit einer Treffergruppe beschränkt sind.

[0654] In einer Ausführungsform sondert die Traversierungs-/Überschneidungsschaltungsanordnung 8003 potenzielle Treffer basierend auf der materialbasierten Cull-Maske 8701 direkt aus. Zum Beispiel kann ein Maskenwert von 0 angeben, dass Instanzen mit einem entsprechenden Material ausgesondert werden sollten. Alternativ oder zusätzlich kann dieses Verhalten emuliert werden, indem Beliebiger-Treffer-Shader innerhalb des Treibers injiziert werden.

GEOMETRISCHER BILDBESCHLEUNIGER UND VERFAHREN

[0655] Ein Geometriebild ist eine Abbildung eines dreidimensionalen (3D) Dreiecks-Mesh auf eine zweidimensionale (2D) Domäne. Insbesondere kann ein Geometriebild Geometrie als ein 2D-Array quantisierter Punkte repräsentieren. Entsprechende Bilddaten, wie etwa Farben und Normale, können durch Verwenden derselben impliziten Oberflächenparametrisierung auch in 2D-Arrays gespeichert werden. Das durch das 2D-Array repräsentierte 2D-Dreiecks-Mesh wird durch ein regelmäßiges Gitter von Vertex-Positionen mit impliziter Konnektivität definiert.

[0656] In einer Ausführungsform der Erfindung wird ein Geometriebild durch Abbilden eines 3D-Dreiecks-Mesh in eine 2D-Ebene gebildet, was in einer implizierten Dreieckskonnektivität resultiert, die durch ein regelmäßiges Gitter von Vertex-Positionen definiert ist. Das resultierende 2D-Geometriebild kann auf verschiedene Weisen innerhalb der Grafik-Pipeline verarbeitet werden, einschließlich Downsampling und Upsampling durch Verwenden von Mipmaps.

[0657] Wie in **Fig. 88** veranschaulicht, führt eine Ausführungsform der Erfindung ein Raytracing durch, indem eine Quadtree-Struktur 8850 über die Geometriebilddomäne erzeugt wird, wobei jeder Quadtree-Knoten 8800, 8810-8813 einen achsenausgerichteten Begrenzungsrahmen (AABB) über den Vertex-Positionen des 2D-Dreiecks-Mesh 8820 speichert. Wie veranschaulicht, speichert jeder Knoten 8800, 8810 bis 8813 die Minimal- und Maximalkoordinaten des assoziierten AABB, der eines oder mehrere der Dreiecke und/oder Vertices enthält. Dadurch ergibt sich eine äußerst regelmäßige und sehr einfach zu berechnende Struktur.

[0658] Sobald die AABBs über das 2D-Dreiecks-Mesh konstruiert sind, können Raytracing-Operationen durch Verwenden der AABBs durchgeführt werden, wie hierin mit Bezug auf die verschiedenen Ausführungsformen der Erfindung beschrieben. Zum Beispiel können Traversierungsoperationen durchgeführt werden, um zu bestimmen, dass ein Strahl einen der Knoten 8810-8813 der untersten Ebene der BVH traversiert. Der Strahl kann dann auf Überschneidungen mit dem 2D-Mesh geprüft und Trefferergebnisse (falls vorhanden) wie hierin beschrieben erzeugt und verarbeitet werden (z. B. gemäß einem Material, das mit dem 2D-Dreiecks-Mesh assoziiert ist).

[0659] Wie veranschaulicht, ist die Speicherungs-/Komprimierungslogik 8850 in einer Ausführungsform dazu ausgelegt, die AABBs als Doppelbildpyramiden 8855 zu komprimieren und/oder zu speichern, wobei eine die Minimalwerte speichert und eine die Maximalwerte speichert. In dieser Ausführungsform können unterschiedliche Komprimierungsschemen, die für Geometriebilder entwickelt wurden, verwendet werden, um die Minimal- und Maximalbildpyramiden zu komprimieren.

[0660] Die Quadtree-Strukturen 8850, 8860 bis 8861, die oben mit Bezug auf **Fig. 88** beschrieben sind, können durch den BVH-Builder 8007 erzeugt werden. Alternativ dazu können die Quadtree-Strukturen durch einen unterschiedlichen Satz von Schaltungsanordnungen und/oder Logik erzeugt werden.

EINRICHTUNG UND VERFAHREN ZUR RAHMEN-RAHMEN-PRÜFUNG UND BESCHLEUNIGTEN KOLLISIONSDETEKTION zum RAYTRACING

[0661] **Fig. 89A-B** veranschaulicht eine Raytracing-Architektur gemäß einer Ausführungsform der Erfindung. Mehrere Ausführungseinheiten 8910 führen Shader und anderen Programmcode in Bezug auf Raytracing-Operationen aus. Eine „Traceray“-Funktion, die auf einer der Ausführungseinheiten (EUs) 8910 ausgeführt wird, löst einen Strahlenzustandsinitialisierer 8920 aus, um den Zustand zu initialisieren, der erforderlich ist, um einen aktuellen Strahl (über eine Strahl-ID/einen Strahl-Deskriptor identifiziert) durch eine Hüllkörperhie-

rarchie (BVH) zu verfolgen (z. B. in einem Stapel 5121 in einem Speicherpuffer 8918 oder einer anderen Datenstruktur in dem lokalen Speicher oder Systemspeicher 3198 gespeichert).

[0662] In einer Ausführungsform, falls die Traceray-Funktion einen Strahl identifiziert, für den eine vorherige Traversierungsoperation teilweise abgeschlossen wurde, dann verwendet der Zustandsinitialisierer 8920 die eindeutige Strahl-ID, um die assoziierten Raytracing-Daten 4902 und/oder Stapel 5121 aus einem oder mehreren Puffern 8918 in den Speicher 3198 zu laden. Wie erwähnt, kann der Speicher 3198 ein On-Chip-/lokaler Speicher oder Cache und/oder eine Speichervorrichtung auf Systemebene sein.

[0663] Wie mit Bezug auf andere Ausführungsformen besprochen, kann ein Verfolgungsarray 5249 aufrechterhalten werden, um den Traversierungsfortschritt für jeden Strahl zu speichern. Falls der aktuelle Strahl teilweise eine BVH traversiert hat, dann kann der Zustandsinitialisierer 8920 das Verfolgungsarray 5249 verwenden, um die BVH-Ebene/den BVH-Knoten zu bestimmen, bei dem neu gestartet werden soll.

[0664] Eine Traversierungs- und Raybox-Prüfeinheit 8930 traversiert den Strahl durch die BVH. Wenn ein Primitiv innerhalb eines Leaf-Knotens der BVH identifiziert wurde, prüft ein Instanz-/Viereck-Überschneidungsprüfer 8940 den Strahl auf eine Überschneidung mit dem Primitiv (z. B. ein oder mehrere Primitiv-Vierecke), wobei eine assoziierte Strahl-/Shader-Aufzeichnung von einem Raytracing-Cache 8960 wiederabgerufen wird, der innerhalb der Cache-Hierarchie des Grafikprozessors (hier mit einem L1-Cache 8970 gekoppelt gezeigt) integriert ist. Der Instanz-/Viereck-Überschneidungsprüfer 8940 wird hierin manchmal einfach als eine Überschneidungseinheit (z. B. Überschneidungseinheit 5103 in **Fig. 51**) bezeichnet.

[0665] Die Strahl-/Shader-Aufzeichnung wird an einen Thread-Dispatcher 8950 bereitgestellt, der neue Threads an die Ausführungseinheiten 8910 zumindest teilweise durch Verwenden der hierin beschriebenen bindungslosen Thread-Dispatching-Techniken versendet. In einer Ausführungsform beinhaltet die Strahl/Rahmen-Traversierungseinheit 8930 die oben beschriebene Traversierungs-/Stapelverfolgungslogik 5248, die den Traversierungsfortschritt für jeden Strahl innerhalb des Verfolgungsarrays 5249 verfolgt und speichert.

[0666] Eine Klasse von Problemen beim Rendern kann auf Prüfrahmenkollisionen mit anderen Hüllkörpern oder Rahmen abgebildet werden (z. B. aufgrund von Überlappung). Solche Rahmen-Abfragen können verwendet werden, um Geometrie innerhalb eines Abfragebegrenzungsrahmens für verschiedene Anwendungen zu enumerieren. Zum Beispiel können Rahmenabfragen verwendet werden, um Photonen während der Photonenabbildung zu sammeln, alle Lichtquellen zu enumerieren, die einen Abfragepunkt (oder einen Abfragebereich) beeinflussen können, und/oder nach dem Oberflächenpunkt, der einem Abfragepunkt am nächsten liegt, zu suchen. In einer Ausführungsform arbeiten die Rahmenabfragen auf derselben BVH-Struktur wie die Strahlabfragen; somit kann der Benutzer Strahlen durch eine Szene verfolgen, und Rahmenabfragen an derselben Szene durchführen.

[0667] In einer Ausführungsform der Erfindung werden Rahmenabfragen ähnlich wie Strahlabfragen in Bezug auf Raytracing-Hardware/-Software behandelt, wobei die Strahl/Rahmen-Traversierungseinheit 8930 eine Traversierung durch Verwenden von Rahmen/Rahmen-Operationen anstelle von Strahl/Rahmen-Operationen durchführt. In einer Ausführungsform kann die Traversierungseinheit 8930 denselben Satz von Merkmalen für Rahmen/Rahmen-Operationen verwenden, wie sie für Strahl/Rahmen-Operationen verwendet werden, einschließlich, jedoch nicht beschränkt auf Bewegungsunschärfe, Masken, Flags, Nächstgelegener-Treffer-Shader, Beliebiger-Treffer-Shader, Fehltreffer-Shader und Traversierungs-Shader. Eine Ausführungsform der Erfindung fügt zu jeder Raytracing-Nachricht oder -Anweisung (z. B. TraceRay, wie hierin beschrieben) ein Bit hinzu, um anzugeben, dass die Nachricht/Anweisung mit einer BoxQuery-Operation assoziiert ist. In einer Implementierung wird BoxQuery sowohl in synchronen als auch in asynchronen Raytracing-Modi (z. B. durch Verwenden von Standard-Dispatch- bzw. bindungslosen Thread-Dispatch-Operationen) aktiviert.

[0668] In einer Ausführungsform, sobald sie über das Bit auf den BoxQuery-Modus eingestellt ist, interpretiert die Raytracing-Hardware/-Software (z. B. die Traversierungseinheit 8930, Instanz-/Viereck-Überschneidungsprüfer 8940 usw.) die Daten, die mit der Raytracing-Nachricht/-Anweisung assoziiert sind, als Rahmendaten (z. B. min-/max-Werte in drei Dimensionen). In einer Ausführungsform werden Traversierungsbeschleunigungsstrukturen erzeugt und aufrechterhalten, wie zuvor beschrieben, aber ein Rahmen wird anstelle eines Strahls für jede primäre Stapel-ID initialisiert.

[0669] In einer Ausführungsform wird Hardware-Instanziierung nicht für Rahmenabfragen durchgeführt. Instanziierung kann jedoch in Software durch Verwenden von Traversierungs-Shadern emuliert werden. Wenn somit ein Instanzknoten während einer Rahmenabfrage erreicht wird, kann die Hardware den Instanz-

knoten als einen prozeduralen Knoten verarbeiten. Da der Header beider Strukturen derselbe ist, bedeutet dies, dass die Hardware den in dem Header des Instanzknotens gespeicherten Shader aufrufen wird, der dann die Punktabfrage innerhalb der Instanz fortsetzen kann.

[0670] In einer Ausführungsform wird ein Strahl-Flag gesetzt, um anzugeben, dass der Instanz-/Viereck-Überschneidungsprüfer 8940 den ersten Treffer akzeptieren und die Suche beenden wird (z. B. ACCEPT_FIRST_HIT AND_END_SEARCH-Flag). Wenn dieses Strahl-Flag nicht gesetzt ist, werden die geschnittenen Children von vorne nach hinten gemäß ihrem Abstand zu dem Abfragerahmen eingegeben, ähnlich wie die Strahlabfragen. Beim Suchen nach der Geometrie, die einem Punkt am nächsten liegt, verbessert diese Traversierungsreihenfolge die Leistungsfähigkeit erheblich, wie dies bei Strahlabfragen der Fall ist.

[0671] Eine Ausführungsform der Erfindung filtert Falsch-Positiv-Treffer mit Beliebiger-Treffer-Shadern aus. Obwohl Hardware zum Beispiel möglicherweise keine genaue Rahmen-/Dreiecksprüfung auf Leaf-Ebene durchführt, wird sie konservativ alle Dreiecke eines getroffenen Leaf-Knotens melden. Ferner kann Hardware, wenn der Suchrahmen durch einen Beliebiger-Treffer-Shader geschrumpft wird, Primitive eines per Pop entfernten Leaf-Knotens als einen Treffer zurückgeben, obwohl der Leaf-Knoten-Rahmen möglicherweise nicht mehr den geschrumpften Abfragerahmen überlappt.

[0672] Wie in **Fig. 89A** veranschaulicht, kann eine Rahmenabfrage durch die Ausführungseinheit (EU) 8910, die eine Nachricht/einen Befehl an die Hardware (d. h. TraceRay) sendet, ausgegeben werden. Die Verarbeitung fährt dann wie oben beschrieben fort, d. h. durch den Zustandsinitialisierer 8920, die Strahl/Rahmen-Traversierungslogik 8930, den Instanz-/Viereck-Überschneidungsprüfer 8940 und den bindungslosen Thread-Dispatcher 8950.

[0673] In einer Ausführungsform verwendet die Rahmenabfrage das MemRay-Datenlayout, wie es für Strahlabfragen verwendet wird, durch Speichern der Untergrenzen des Abfragerahmens in derselben Position wie der Strahlursprung, der Obergrenzen in derselben Position wie die Strahlrichtung, und eines Abfrageradius in den Fernwert.

```
struct MemBox
{
 // 32 Bytes (Semantik geändert)
  Vec3f lower;      // die Untergrenzen des Abfragerahmens
  Vec3f upper;      // die Obergrenzen des Abfragerahmens
  float unused;
  float radius;      // zusätzliche Erweiterung des Abfragerahmens (L0-
 Norm)

  // 32 Bytes (identisch mit Standard-MemRay)
 };
```

[0674] Durch Verwenden dieses MemBox-Layouts verwendet die Hardware den Rahmen [unterer -Radius, oberer +Radius], um die Abfrage durchzuführen. Daher werden die gespeicherten Grenzen in jeder Dimension um einen Radius in L0-Norm erweitert. Dieser Abfrageradius kann nützlich sein, um die Suchfläche leicht zu schrumpfen, z. B. für Suchen nach nächstgelegenen Punkten.

[0675] Da das MemBox-Layout nur die Strahlursprungs-, Strahlrichtungs- und T_{far}-Mitglieder des MemRay-Layouts wiederverwendet, braucht die Datenverwaltung in Hardware für Strahlabfragen nicht geändert zu werden. Stattdessen werden die Daten in der internen Speicherung (z. B. dem Raytracing-Cache 8960 und L1-Cache 8970) wie die Strahldaten gespeichert, und werden nur für Rahmen/Rahmen-Prüfungen unterschiedlich interpretiert.

[0676] In einer Ausführungsform werden die folgenden Operationen durch die Strahl-/Zustands-Initialisierungseinheit 8920 und die Strahl/Rahmen-Traversierungseinheit 8930 durchgeführt. Das zusätzliche Bit „BoxQueryEnable“ von der TraceRay-Nachricht wird in dem Zustandsinitialisierer 8920 in eine Pipeline eingeordnet (was seine Kompaktierung über Nachrichten hinweg beeinflusst), wodurch jeder Strahl/Rahmen-Traversierungseinheit 8930 eine Angabe der BoxQueryEnable-Einstellung bereitgestellt wird.

[0677] Die Strahl/Rahmen-Traversierungseinheit 8930 speichert „BoxQueryEnable“ mit jedem Strahl, wobei dieses Bit als ein Tag mit der anfänglichen Strahlenlastanforderung gesendet wird. Wenn die angeforderten Strahldaten von der Speicherschnittstelle zurückgegeben werden, wird bei eingestelltem BoxQueryEnable eine reziproke Berechnung umgangen, und wird stattdessen eine unterschiedliche Konfiguration für alle Komponenten in dem RayStore geladen (d. h. gemäß einem Rahmen anstelle eines Strahls).

[0678] Die Strahl/Rahmen-Traversierungseinheit 8930 ordnet das BoxQueryEnable Bit in eine Pipeline an die darunterliegende Prüflogik ein. In einer Ausführungsform wird der Raybox-Datenpfad gemäß den folgenden Konfigurationseinstellungen modifiziert. Falls BoxQueryEnable == 1, wird die Ebene des Rahmens nicht geändert, da sie sich basierend auf dem Vorzeichen der x-, y- und z-Komponenten der Strahlrichtung ändert. Für den Strahl durchgeführte Prüfungen, die für das Raybox unnötig sind, werden umgangen. Zum Beispiel wird angenommen, dass der abfragende Rahmen keine INF oder NANs hat, sodass diese Prüfungen in dem Datenpfad umgangen werden.

[0679] In einer Ausführungsform wird vor der Verarbeitung durch die Trefferbestimmungslogik eine weitere Additionsoperation ausgeführt, um den Wert unterer +Radius (im Wesentlichen der t-Wert von dem Treffer) und oberer -Radius zu bestimmen. Zusätzlich berechnet sie beim Treffen eines „Instanzknotens“ (in einer Hardware-Instanziierungsimplementierung) keinerlei Transformation, sondern startet stattdessen einen Überschneidungs-Shader durch Verwenden einer Shader-ID in dem Instanzknoten.

[0680] In einer Ausführungsform führt, wenn BoxQueryEnable eingestellt ist, die Strahl/Rahmen-Traversierungseinheit 8930 den NULL-Shader-Nachschlag für einen Beliebiger-Treffer-Shader nicht durch. Zusätzlich ruft die Strahl/Rahmen-Traversierungseinheit 8930, wenn BoxQueryEnable gesetzt ist, wenn ein gültiger Knoten von dem Typ QUAD, MESHLET ist, einen Überschneidungs-Shader ebenso auf, wie er nach dem Aktualisieren der potenziellen Trefferinformationen in Speicher einen ANY-HIT-SHADER (Beliebiger-Treffer-Shader) aufrufen würde.

[0681] In einer Ausführungsform ist ein separater Satz der verschiedenen in **Fig. 89A** veranschaulichten Komponenten in jeder Mehrkerngruppe 3100A (z. B. innerhalb der Raytracing-Kerne 3150) bereitgestellt. In dieser Implementierung kann jede Mehrkerngruppe 3100A parallel an einem unterschiedlichen Satz von Strahldaten und/oder Rahmendaten arbeiten, um Traversierungs- und Überschneidungsoperationen durchzuführen, wie hierin beschrieben.

EINRICHTUNG UND VERFAHREN FÜR MESHLET-KOMPRIMIERUNG UND -DEKOMPRIMIERUNG FÜR RAYTRACING

[0682] Wie oben beschrieben, ist ein „Meshlet“ ein Teilsatz eines Mesh, das durch Geometriepartitionierung erzeugt wird, die eine gewisse Anzahl von Vertices (z. B. 16, 32, 64, 256, usw.) basierend auf der Anzahl assoziierter Attribute beinhaltet. Meshlets können so ausgelegt sein, dass sie so viele Vertices wie möglich gemeinsam nutzen, um eine Vertex-Wiederverwendung während des Renderns zu ermöglichen. Diese Partitionierung kann vorberechnet werden, um eine Laufzeitverarbeitung zu vermeiden, oder kann jedes Mal, wenn ein Mesh gezeichnet wird, dynamisch zur Laufzeit durchgeführt werden.

[0683] Eine Ausführungsform der Erfindung führt eine Meshlet-Komprimierung aus, um die Speicherungsanforderungen für die Beschleunigungsstrukturen der untersten Ebene (BLAS) zu reduzieren. Diese Ausführungsform macht sich die Tatsache zunutze, dass ein Meshlet ein kleines Stück eines größeren Mesh mit ähnlichen Vertices repräsentiert, um eine effiziente Komprimierung innerhalb eines 128B-Datenblocks zu ermöglichen. Es sei jedoch angemerkt, dass die zugrundeliegenden Prinzipien der Erfindung nicht auf irgendeine spezielle Blockgröße beschränkt sind.

[0684] Meshlet-Komprimierung kann zu dem Zeitpunkt durchgeführt werden, zu dem die entsprechende Hüllkörperhierarchie (BVH) aufgebaut und an dem BVH-Verbrauchspunkt dekomprimiert wird (z. B. durch den Raytracing-Hardware-Block). In bestimmten unten beschriebenen Ausführungsformen wird eine Meshlet-Dekomprimierung zwischen dem L1-Cache (manchmal „LSC-Einheit“) und dem Raytracing-Cache (manchmal „RTC-Einheit“) durchgeführt. Wie hierin beschrieben, ist der Raytracing-Cache ein lokaler Hochgeschwindigkeits-Cache, der durch die Strahltraversierungs-/Überschneidungs-Hardware verwendet wird.

[0685] In einer Ausführungsform wird die Meshlet-Komprimierung in Hardware beschleunigt. Falls zum Beispiel der Ausführungseinheits(EU)-Pfad eine Dekomprimierung unterstützt (um z. B. potenziell eine Traver-

sierungs-Shader-Ausführung zu unterstützen), kann eine Meshlet-Dekomprimierung in den gemeinsamen Pfad aus dem L1-Cache integriert werden.

[0686] In einer Ausführungsform wird eine Nachricht verwendet, um eine Meshlet-Komprimierung auf 128B-Blöcke in Speicher zu initiieren. Zum Beispiel kann eine 4 X 64B-Nachrichteneingabe zu einer 128B-Blockausgabe an den Shader komprimiert werden. In dieser Implementierung wird ein zusätzlicher Knotentyp in der BVH hinzugefügt, um eine Assoziation mit einem komprimierten Meshlet anzugeben.

[0687] Fig. 89B veranschaulicht eine bestimmte Implementierung zur Meshlet-Komprimierung, die einen Meshlet-Komprimierungsblock (RTMC) 9030 und einen Meshlet-Dekomprimierungsblock (RTMD) 9090 beinhaltet, die innerhalb des Raytracing-Clusters integriert sind. Die Meshlet-Komprimierung 9030 wird aufgerufen, wenn eine neue Nachricht von einer Ausführungseinheit 8910, die einen Shader ausführt, zu dem Raytracing-Cluster (z. B. innerhalb eines Raytracing-Kerns 3150) übertragen wird. In einer Ausführungsform beinhaltet die Nachricht vier 64B-Phasen und eine 128B-Schreibadresse. Die Nachricht von der EU 8910 weist den Meshlet-Komprimierungsblock 9030 an, wo die Vertices und zugehörigen Meshlet-Daten in dem lokalen Speicher 3198 (und/oder Systemspeicher, abhängig von der Implementierung) platziert werden sollen. Der Meshlet-Komprimierungsblock 9030 führt dann eine Meshlet-Komprimierung durch, wie hierin beschrieben. Die komprimierten Meshlet-Daten können dann über die Speicherschnittstelle 9095 in dem lokalen Speicher 3198 und/oder dem Raytracing-Cache 8960 gespeichert werden, und es kann durch den Instanz-/Viereck-Überschneidungsprüfer 8940 und/oder einen Traversierungs-/Überschneidungs-Shader auf sie zugegriffen werden.

[0688] In **Fig. 89B** kann der Meshlet-Sammel- und -Dekomprimierungsblock 9090 die komprimierten Daten für ein Meshlet sammeln und die Daten in mehrere 64B-Blöcke dekomprimieren. In einer Implementierung werden nur dekomprimierte Meshlet-Daten innerhalb des L1-Caches 8970 gespeichert. In einer Ausführungsform wird die Meshlet-Dekomprimierung aktiviert, während die BVH-Knotendaten basierend auf dem Knotentyp (z. B. Leaf-Knoten, komprimiert) und der Primitiv-ID abgerufen werden. Der Traversierungs-Shader kann auch durch Verwenden derselben Semantik wie der Rest der Raytracing-Implementierung auf das komprimierte Meshlet zugreifen.

[0689] In einer Ausführungsform nimmt der Meshlet-Komprimierungsblock 9030 ein Array von Eingabedreiecken von einer EU 8910 an und erzeugt eine komprimierte 128B-Meshlet-Leaf-Struktur. Ein Paar aufeinanderfolgender Dreiecke in dieser Struktur bildet ein Viereck. In einer Implementierung beinhaltet die EU-Nachricht bis zu 14 Vertices und Dreiecke, wie in der Codesequenz unten angegeben. Das komprimierte Meshlet wird über die Speicherschnittstelle 9095 an der in der Nachricht bereitgestellten Adresse in den Speicher geschrieben.

[0690] In einer Ausführungsform berechnet der Shader das Bit-Budget für den Satz von Meshlets, und daher wird die Adresse so bereitgestellt, dass eine Grundflächenkomprimierung möglich ist. Diese Nachrichten werden nur für komprimierbare Meshlets initiiert.

```
struct CompressMeshletMsg {
```

```
  uint64_t     address;          // Header: 128B ausgerichtete
Zieladresse für das Meshlet

  float        vert_x[14];       // bis zu 14 Vertex-Koordinaten
  uint32_t     vert_x_bits;      // max Vertex-Bits
  uint32_t     numPrims;         // Anzahl von Dreiecken (für Vierecke
immer geradzahlig)
float        vert_y[14];
  uint32_t     vert_y_bits;      // max Vertex-Bits
  uint32_t     numIdx;           // Anzahl der Indizes

  float        vert_z[14];
  uint32_t     vert_z_bits;      // max Vertex-Bits
  uint32_t     numPrimIDBits;

  int32_t      primID[14];       // primIDs
  PrimLeafDesc primLeafDesc;

  Struct {
    int8_t idx_x;
    int8_t idx_y;
    int8_t idx_z;
    int8_t last;                // 1, falls das Dreieck in dem Leaf das
letzte ist, andernfalls 0
  } index[14];                  // Vertex-Indizes

  int32_t pad0;
  int32_t pad1;
}
```

[0691] In einer Ausführungsform dekomprimiert der Meshlet-Dekomprimierungsblock 9090 zwei aufeinanderfolgende Vierecke (128B) von einem 128B-Meshlet und speichert die dekomprimierten Daten in dem L1-Cache 8970. Die Tags in dem L1-Cache 8970 verfolgen den Index jedes dekomprimierten Vierecks (einschließlich des Dreiecksindexes) und die Meshlet-Adresse. Der Raytracing-Cache 8960 sowie eine EU 8910 können ein dekomprimiertes 64B-Viereck von dem L1-Cache 8970 abrufen. In einer Ausführungsform ruft eine EU 8910 ein dekomprimiertes Viereck durch Ausgeben einer MeshletQuadFetch-Nachricht an den L1-Cache 8960 ab, wie unten gezeigt. Separate Nachrichten können zum Abrufen der ersten 32 Bytes und der letzten 32 Bytes des Vierecks ausgegeben werden.

[0692] Shader können auf Dreiecks-Vertices von der Viereckstruktur zugreifen, wie unten gezeigt. In einer Ausführungsform werden die „if“-Aussagen durch „sel“-Anweisungen ersetzt.

```
// Unter der Annahme, dass Vertex i eine durch den Compiler
bestimmte Konstante ist
float3 getVertexi(Quad& q, int triID, int vertexID) {
  if (triID == 0)
    return quad.vi;
  else if (i == j0)
    return quad.v0;
  else if (i == j1)
    return quad.v1;
  else if (i == j2)
    return quad.v2;
}
```

[0693] In einer Ausführungsform kann der Raytracing-Cache 8960 ein dekomprimiertes Viereck direkt von der Bank des L1-Cache 8970 abrufen, indem die Meshlet-Adresse und der Viereckindex bereitgestellt werden.

```
GetQuadData {
  uint1_t  msb; // MS 32B oder LS 32B
  Uint4_t  triangle_idx; // Index des Dreiecks innerhalb des Meshlet.
Für Vierecke immer geradzahlig.
  uint64_t meshlet_addr;
}
```

Meshlet-Komprimierungsarozess

[0694] Nach dem Zuweisen von Bits für einen festen Overhead, wie etwa geometrische Eigenschaften (z. B. Flags und Masken), werden Daten des Meshlet zu dem komprimierten Block hinzugefügt, während das verbleibende Bit-Budget basierend auf Deltas an (pos.x, pos.y, pos.z) im Vergleich zu (base.x, base.y, base.z), wobei die Basiswerte die Position des ersten Vertex in der Liste umfassen, berechnet wird. In ähnlicher Weise können auch Prim-ID-Deltas berechnet werden. Da das Delta mit dem ersten Vertex verglichen wird, ist es kostengünstiger, mit niedriger Latenz zu dekomprimieren. Die Basisposition und primIDs sind Teil des konstanten Overhead in der Datenstruktur, zusammen mit der Breite der Delta-Bits. Für verbleibende Vertices eines geradzahligen Dreiecks werden Positions-Deltas und Prim-ID-Deltas auf unterschiedlichen 64B-Blöcken gespeichert, um sie parallel zu packen.

[0695] Durch Verwenden dieser Techniken verbraucht die BVH-Build-Operation beim Herausschreiben der komprimierten Daten über die Speicherschnittstelle 9095 eine geringere Bandbreite in den Speicher. Außerdem ermöglicht in einer Ausführungsform das Speichern des komprimierten Meshlets in dem L3-Cache die Speicherung von mehr BVH-Daten mit der gleichen L3-Cache-Größe. In einer Arbeitsimplementierung werden mehr als 50 % Meshlets 2:1 komprimiert. Während eine BVH mit komprimierten Meshlets verwendet wird, resultiert eine Bandbreiteneinsparung an dem Speicher in Leistungseinsparungen.

[0696] Wie oben beschrieben, ist bindungsloser Thread-Dispatch (BTD) eine Weise zum Lösen des SIMD-Divergenzproblems für Raytracing in Implementierungen, die gemeinsam genutzten lokalen Speicher (SLM) oder Speicherbarrieren nicht unterstützen. Ausführungsformen der Erfindung beinhalten Unterstützung für generalisiertes BTD, das verwendet werden kann, um SIMD-Divergenz für verschiedene Berechnungsmodelle anzusprechen. In einer Ausführungsform kann ein beliebiger Berechnungs-Dispatch mit einer Thread-Gruppenbarriere und SLM einen bindungslosen Child-Thread spawnen, und alle Threads können über BTD umgruppiert und versendet werden, um die Effizienz zu verbessern. In einer Implementierung ist ein bindungsloser Child-Thread jeweils pro Parent erlaubt, und den Ursprungs-Threads ist es erlaubt, ihren SLM-Raum mit den bindungslosen Child-Threads gemeinsam zu nutzen. Sowohl SLM als auch Barrieren werden nur freigegeben, wenn schlussendlich konvergierte Parents enden (d. h. EOTs durchführen). Eine bestimmte Ausführungsform ermöglicht eine Verstärkung innerhalb eines aufrufbaren Modus, der Baumtraversierungsfälle mit mehr als einem gespawnten Child ermöglicht.

[0697] **Fig. 90** veranschaulicht grafisch einen anfänglichen Satz von Threads 9000, die synchron durch die SIMD-Pipeline verarbeitet werden können. Die Threads 9000 können zum Beispiel synchron als eine Arbeitsgruppe versendet und ausgeführt werden. In dieser Ausführungsform kann der anfängliche Satz synchroner Threads 9000 jedoch mehrere divergierende Spawn-Threads 9001 erzeugen, die andere Spawn-Threads 9011 innerhalb der hierin beschriebenen asynchronen Raytracing-Architekturen erzeugen können. Schließlich kehren konvergierende Spawn-Threads 9021 zu dem ursprünglichen Satz von Threads 9000 zurück, der dann die synchrone Ausführung fortsetzen kann, wobei der Kontext nach Bedarf gemäß dem Verfolgungsarray 5249 wiederhergestellt wird.

[0698] In einer Ausführungsform unterstützt eine bindungslose Thread-Dispatch(BTD)-Funktion SIMD16- und SIMD32-Modi, variable Mehrzweckregister(GPR)-Verwendung, gemeinsam genutzten Speicher (SLM) und BTD-Barrieren durch Fortbestehen durch die Wiederaufnahme des Parent-Threads nach Ausführung und Abschluss (nachdivergierendes und dann konvergierendes Spawn). Eine Ausführungsform der Erfindung beinhaltet eine Hardware-verwaltete Implementierung zum Wiederaufnehmen der Parent-Threads, und eine Software-verwaltete Dereferenzierung der SLM- und Barriereressourcen.

[0699] In einer Ausführungsform der Erfindung haben die folgenden Begriffe die folgenden Bedeutungen:

[0700] Aufrufbarer Modus: Threads, die durch bindungsloses Thread-Dispatch erzeugt werden, befinden sich in dem „aufrufbaren Modus“. Diese Threads können auf den vererbten gemeinsam genutzten lokalen Speicherplatz zugreifen, und optional einen Thread pro Thread in dem aufrufbaren Modus spawnen. In diesem Modus haben Threads keinen Zugriff auf die Barriere auf Arbeitsgruppenebene.

[0701] Arbeitsgruppen(WG)-Modus: Wenn Threads auf dieselbe Weise mit konstituierenden SIMD-Spuren ausgeführt werden, wie sie vom Standard-Thread-Dispatch versendet werden, werden sie als in dem Arbeitsgruppenmodus befindlich definiert. In diesem Modus haben Threads Zugriff auf Barrieren auf Arbeitsgruppenebene, sowie auf gemeinsam genutzten lokalen Speicher. In einer Ausführungsform wird der Thread-Dispatch als Reaktion auf einen „Berechnungs-Walker“-Befehl initiiert, der einen Nur-Berechnungs-Kontext initiiert.

[0702] Gewöhnlicher Spawn: Auch als reguläre gespawnte Threads 9011 (**Fig. 90**) bezeichnet, wird gewöhnlicher Spawn immer dann initiiert, wenn ein aufrufbares Element ein anderes aufruft. Solche gespawnten Threads werden in dem aufrufbaren Modus in Betracht gezogen.

[0703] Divergierender Spawn: Wie in **Fig. 90** gezeigt, werden divergierende Spawn-Threads 9001 ausgelöst, wenn ein Thread von dem Arbeitsgruppenmodus in den aufrufbaren Modus übergeht. Argumente eines divergierenden Spawn sind die SIMD-Breite und Festfunktions-Thread-ID (FFTID), die untergruppeneinheitlich sind.

[0704] Konvergierender Spawn: Konvergierende Spawn-Threads 9021 werden ausgeführt, wenn ein Thread vom abrufbaren Modus zurück in den Arbeitsgruppenmodus übergeht. Argumente eines konvergierenden Spawn sind eine Pro-Spur-FFTID und eine Maske, die angibt, ob der Stapel der Spur leer ist oder nicht. Diese Maske muss dynamisch berechnet werden, indem der Wert des Pro-Spur-Stapelzeigers an der Rückgabestelle geprüft wird. Der Compiler muss diese Maske berechnen, da sich diese aufrufbaren Threads

rekursiv gegenseitig aufrufen können. Spuren in einem konvergierenden Spawn, bei denen das Konvergenzbit nicht gesetzt ist, verhalten sich wie gewöhnliche Spawns.

[0705] Das bindungslose Thread-Dispatch löst das SIMD-Divergenzproblem für Raytracing in einigen Implementierungen, die keine gemeinsam genutzten lokalen Speicher- oder Barriereoperationen ermöglichen. Zusätzlich wird in einer Ausführungsform der Erfindung BTD verwendet, um SIMD-Divergenz durch Verwenden einer Vielzahl von Berechnungsmodellen anzusprechen. Insbesondere kann ein jegliches Berechnungs-Dispatch mit einer Thread-Gruppenbarriere und gemeinsam genutztem lokalen Speicher bindungslose Child-Threads (z. B. jeweils ein Child-Thread pro Parent) spawnen, und alle selben Threads können durch BTD für eine bessere Effizienz umgruppiert und versendet werden. Diese Ausführungsform ermöglicht es, dass die Ursprungs-Threads ihren gemeinsamen lokalen Speicherplatz mit ihren Child-Threads gemeinsam nutzen. Die gemeinsam genutzten lokalen Speicherzuweisungen und -barrieren werden erst freigegeben, wenn schlussendlich konvergierte Parents enden (wie durch End-of-Thread (EOT)-Indikatoren angegeben). Eine Ausführungsform der Erfindung stellt auch eine Verstärkung innerhalb eines aufrufbaren Modus bereit, was Baumtraversierungsfälle ermöglicht, bei denen mehr als ein Child gespawnt wird.

[0706] Obwohl nicht darauf beschränkt, wird eine Ausführungsform der Erfindung auf einem System implementiert, bei dem keine Unterstützung zur Verstärkung durch eine jegliche SIMD-Spur bereitgestellt wird (d. h. nur eine einzelne ausstehende SIMD-Spur in der Form eines divergierten oder konvergierten Spawn-Threads ermöglicht wird). Zusätzlich werden in einer Implementierung die 32b von (FFTID, BARRIER_ID, SLM_ID) beim Dispatch eines Threads an den BTD-fähigen Dispatcher 8950 gesendet. In einer Ausführungsform werden alle diese Räume freigegeben, bevor die Threads gestartet und diese Informationen an den bindungslosen Thread-Dispatcher 8950 gesendet werden. In einer Implementierung ist jeweils nur ein einzelner Kontext aktiv. Daher kann ein Rogue-Kernel selbst nach der Manipulation der FFTID nicht auf den Adressraum des anderen Kontextes zugreifen.

[0707] In einer Ausführungsform werden, falls Stapel-ID-Zuweisung aktiviert ist, gemeinsam genutzter lokaler Speicher und Barrieren nicht mehr dereferenziert, wenn ein Thread endet. Stattdessen werden sie nur dereferenziert, falls alle assoziierten Stapel-IDs freigegeben wurden, wenn der Thread endet. Eine Ausführungsform verhindert Festfunktions-Thread-ID(FFTID)-Leckagen, indem sichergestellt wird, dass Stapel-IDs ordnungsgemäß freigegeben werden.

[0708] In einer Ausführungsform werden Barrierenachrichten derart spezifiziert, dass sie explizit eine Barriere-ID von dem sendenden Thread nehmen. Dies ist notwendig, um eine Barrieren-/SLM-Nutzung nach einem bindungslosen Thread-Dispatch-Aufruf zu ermöglichen.

[0709] **Fig. 91** veranschaulicht eine Ausführungsform einer Architektur zum Durchführen von bindungslosem Thread-Dispatching und Thread-/Arbeitsgruppen-Präemption, wie hierin beschrieben. Die Ausführungseinheiten (EU) 8910 dieser Ausführungsform unterstützen direkte Manipulation der Thread-Ausführungsmaske 9150-9153, und jede BTD-Spawn-Nachricht unterstützt FFTID-Referenzzählung zum erneuten Spawnen eines Parent-Threads nach Abschluss des konvergierenden Spawn 9021. Somit unterstützt die hierin beschriebene Raytracing-Schaltungsanordnung zusätzliche Nachrichtenvarianten für BTD-Spawn- und TraceRay-Nachrichten. In einer Ausführungsform erhält der BTD-fähige Dispatcher 8950 eine Pro-FFTID-Zählung (wie durch Thread-Dispatch zugewiesen) ursprünglicher SIMD-Spuren auf divergierenden Spawn-Threads 9001 aufrecht, und zählt für konvergierende Spawn-Threads 9021 herunter, um die Wiederaufnahme der Parent-Threads 9000 zu starten.

[0710] Verschiedene Ereignisse können während der Ausführung gezählt werden, einschließlich, jedoch nicht beschränkt auf reguläre Spawn-Ausführungen 9011; divergierende Spawn-Ausführungen 9001; konvergierende Spawn-Ereignisse 9021; eines FFTID-Zählers, der eine minimale Schwelle erreicht (z. B. 0); und für (FFTID, BARRIER_ID, SLM_ID) ausgeführte Lasten.

[0711] In einer Ausführungsform werden gemeinsam genutzter lokaler Speicher (SLM) und Barrierezuordnung mit BTD-fähigen Threads (d. h. zum Berücksichtigen von ThreadGroup-Semantiken) ermöglicht. Der BTD-fähige Thread-Dispatcher 8950 entkoppelt die FFTID-Freigabe und die Barriere-ID-Freigabe von den End-of-Thread(EOT)-Indikationen (z. B. über spezifische Nachrichten).

[0712] Um in einer Ausführungsform aufrufbare Shader von Berechnungs-Threads zu unterstützen, wird ein treiberverwalteter Puffer 9170 verwendet, um Arbeitsgruppeninformationen über die bindungslosen Thread-

Dispatches hinweg zu speichern. In einer bestimmten Implementierung beinhaltet der treiberverwaltete Puffer 9170 mehrere Einträge, wobei jeder Eintrag mit einer anderen FFTID assoziiert ist.

[0713] In einer Ausführungsform werden innerhalb des Zustandsinitialisierers 8920 zwei Bits zugeordnet, um den Pipeline-Spawn-Typ anzugeben, der für Nachrichtenkompaktierung berücksichtigt wird. Zum Divergieren von Nachrichten berücksichtigt der Zustandsinitialisierer 8920 auch die FFTID von der Nachricht und Pipelines mit jeder SIMD-Spur zu dem Strahl/Rahmen-Traversierungsblock 8930 oder dem bindungslosen Thread-Dispatcher 8950. Zum konvergierenden Spawn 9021 gibt es eine FFTID für jede SIMD-Spur in der Nachricht und eine Pipeline-FFTID mit jeder SIMD-Spur für die Strahl/Rahmen-Traversierungseinheit 8930 oder den bindungslosen Thread-Dispatcher 8950. In einer Ausführungsform ordnet die Strahl/Rahmen-Traversierungseinheit 8930 auch den Spawn-Typ, einschließlich konvergierenden Spawn 9021, in eine Pipeline ein. Insbesondere speichert und ordnet in einer Ausführungsform die Strahl/Rahmen-Traversierungseinheit 8930 die FFTID mit jedem strahlkonvergierenden Spawn 9021 für TraceRay-Nachrichten in eine Pipeline ein.

[0714] In einer Ausführungsform weist der Thread-Dispatcher 8950 eine dedizierte Schnittstelle auf, um die folgende Datenstruktur in Vorbereitung auf das Dispatching eines neuen Threads mit dem gesetzten bindungslosen Thread-Dispatch-Aktivierungsbit bereitzustellen:

```
Struct tsl_sts_inf { // nicht verzögerbare Schnittstelle
      Logic[8] FFTID;
      Logic[8] BARRIER_ID;
      Logic[8] SLM_ID;
      Logic[8] count_valid_simd_lanes;
}
```

[0715] Der bindungslose Thread-Dispatcher 8950 verarbeitet auch die End-of-Thread(EOT)-Nachricht mit drei zusätzlichen Bits: Release_FFTID, Release_BARRIER_ID, Release_SLM_ID. Wie erwähnt, gibt die End-of-Thread(EOT)-Nachricht nicht notwendigerweise alle mit den IDs assoziierten Zuordnungen frei/dereferenziert diese, sondern nur die mit einem gesetzten Freigabebit. Ein typischer Anwendungsfall ist, wenn ein divergierendes Spawn 9001 initiiert wird, der Spawn-Thread eine EOT-Nachricht erzeugt, aber das Freigabebit nicht gesetzt ist. Seine Fortsetzung nach dem konvergierenden Spawn 9021 erzeugt eine andere EOT-Nachricht, diesmal jedoch mit dem gesetzten Freigabebit. Erst in diesem Stadium werden alle Pro-Thread-Ressourcen zurückgeführt.

[0716] In einer Ausführungsform implementiert der bindungslose Thread-Dispatcher 8950 eine neue Schnittstelle, um die FFTID, die BARRIER_ID, die SLM_ID und die Spurzählung zu laden. Er speichert alle diese Informationen in eine FFTID-adressierbare Speicherung 9121, die eine bestimmte Anzahl von Einträgen tief ist (max_fftid, 144 Einträge tief in einer Ausführungsform). In einer Implementierung verwendet der BTD-fähige Dispatcher 8950 als Reaktion auf einen jeglichen regulären Spawn 9011 oder divergierenden Spawn 9001 diese Identifikationsinformationen für jede SIMD-Spur, führt Abfragen an die FFTID-adressierbare Speicherung 9121 auf einer Pro-FFTID-Basis durch und speichert die Thread-Daten in dem Sortierpuffer, wie oben beschrieben (siehe z. B. inhaltsadressierbarer Speicher 4201 in **Fig. 42**). Dies resultiert in der Speicherung einer zusätzlichen Datenmenge (z. B. 24 Bits) in dem Sortierpuffer 4201 pro SIMD-Spur.

[0717] Beim Empfangen einer konvergierenden Spawn-Nachricht wird für jede SIMD-Spur von dem Zustandsinitialisierer 8920 oder dem Strahl/Rahmen-Traversierungsblock 8930 zu dem bindungslosen Thread-Dispatcher 8950 die Pro-FFTID-Zählung dekrementiert. Wenn der FFTID-Zähler eines gegebenen Parents null wird, wird der gesamte Thread mit ursprünglichen Ausführungsmasken 9150-9153 mit einer Fortsetzungs-Shader-Aufzeichnung 4201 geplant, der durch die konvergierende Spawn-Nachricht in der Sortierschaltungsanordnung 4008 bereitgestellt wird.

[0718] Unterschiedliche Ausführungsformen der Erfindung können gemäß unterschiedlichen Konfigurationen arbeiten. In einer Ausführungsform müssen zum Beispiel alle divergierenden Spawns 9001, die durch einen Thread durchgeführt werden, übereinstimmende SIMD-Breiten haben. Zusätzlich darf in einer Ausführungsform eine SIMD-Spur keinen konvergierenden Spawn 9021 durchführen, wobei das Convergence-Mask-Bit innerhalb der relevanten Ausführungsmaske 9150-9153 gesetzt ist, es sei denn, dass ein früherer Thread einen divergierenden Spawn mit derselben FFTID durchgeführt hat. Wird ein divergierender Spawn 9001 mit einer gegebenen Stapel-ID durchgeführt, muss ein konvergierender Spawn 9021 vor dem nächsten divergierenden Spawn auftreten.

[0719] Falls eine SIMD-Spur in einem Thread einen divergierenden Spawn durchführt, dann müssen alle Spuren letztendlich einen divergierenden Spawn durchführen. Ein Thread, der einen divergierenden Spawn durchgeführt hat, kann keine Barriere ausführen, oder ein Deadlock wird auftreten. Diese Einschränkung ist notwendig, um Spawns innerhalb eines divergenten Steuerflusses zu ermöglichen. Die Parent-Untergruppe kann nicht erneut gespawnt werden, bis alle Spuren divergiert und wieder konvergiert sind.

[0720] Ein Thread muss schließlich nach dem Durchführen eines jeglichen Spawn enden, um Fortschritt zu gewährleisten. Falls mehrere Spawns vor der Thread-Beendigung durchgeführt werden, kann Deadlock auftreten. In einer bestimmten Ausführungsform werden die folgenden Invarianten befolgt, obwohl die zugrundeliegenden Prinzipien der Erfindung nicht darauf beschränkt sind:

- Alle divergierenden Spawns, die durch einen Thread durchgeführt werden, müssen übereinstimmende SIMD-Breiten aufweisen.
- Eine SIMD-Spur darf keinen konvergierenden Spawn mit dem innerhalb der relevanten Ausführungsmaske 9150-9153 gesetzten ConvergenceMask-Bit durchführen, es sei denn, dass ein früherer Thread einen divergierenden Spawn mit derselben FFTID durchgeführt hat.
- Falls ein divergierender Spawn mit einer gegebenen Stapel-ID durchgeführt wird, muss ein konvergierender Spawn vor dem nächsten divergierenden Spawn auftreten.
- Falls eine SIMD-Spur in einem Thread einen divergierenden Spawn durchführt, dann müssen alle Spuren letztendlich einen divergierenden Spawn durchführen. Ein Thread, der einen divergierenden Spawn durchgeführt hat, kann keine Barriere ausführen, oder ein Deadlock wird auftreten. Diese Einschränkungen ermöglicht Spawns innerhalb eines divergenten Steuerflusses. Die Parent-Untergruppe kann nicht erneut gespawnt werden, bis alle Spuren divergiert und wieder konvergiert sind.
- Ein Thread muss schließlich nach dem Ausführen eines jeglichen Spawn enden, um Fortschritt zu gewährleisten. Falls mehrere Spawns vor der Thread-Beendigung durchgeführt werden, kann Deadlock auftreten.

[0721] In einer Ausführungsform beinhaltet der BTD-fähige Dispatcher 8950 Thread-Präemptionslogik 9120, um die Ausführung bestimmter Typen von Arbeitslasten/Threads zu präemptieren, um Ressourcen zum Ausführen anderer Typen von Arbeitslasten/Threads freizugeben. Zum Beispiel können die verschiedenen hierin beschriebenen Ausführungsformen sowohl Berechnungsarbeitslasten als auch Grafikarbeitslasten (einschließlich Raytracing-Arbeitslasten) ausführen, die mit unterschiedlichen Prioritäten ausgeführt werden können und/oder unterschiedliche Latenzanforderungen haben können. Um die Anforderungen jeder Arbeitslast/jedes Threads anzusprechen, suspendiert eine Ausführungsform der Erfindung Strahltraversierungsoperationen, um Ausführungsressourcen für eine Arbeitslast/einen Thread mit höherer Priorität oder eine Arbeitslast/einen Thread, die/der ansonsten spezifizierte Latenzanforderungen nicht erfüllen wird, freizugeben.

[0722] Wie oben mit Bezug auf die **Fig. 52A-B** beschrieben, reduziert eine Ausführungsform die Speicherungsanforderungen für die Traversierung durch Verwenden eines kurzen Stapels 5203-5204, um eine begrenzte Anzahl von BVH-Knoten während Traversierungsoperationen zu speichern. Diese Techniken können durch die Ausführungsform in **Fig. 91** verwendet werden, wobei die Strahl/Rahmen-Traversierungseinheit 8930 Einträge effizient zu dem kurzen Stapel 5203-5204 pushes und von diesem per Pop entfernt, um zu gewährleisten, dass die erforderlichen BVH-Knoten 5290-5291 verfügbar sind. Zusätzlich dazu aktualisiert der Traversierungs-/Stapelverfolger 5248, während Traversierungsoperationen durchgeführt werden, die Verfolgungsdatenstruktur, die hierin als das Verfolgungsarray 5249 bezeichnet wird, sowie die relevanten Stapel 5203-5204 und Raytracing-Daten 4902. Durch Verwenden dieser Techniken kann die Traversierungsschaltungsanordnung/-logik 8930, wenn das Traversieren eines Strahls pausiert und neu gestartet wird, die Verfolgungsdatenstruktur 5249 konsultieren und auf die relevanten Stapel 5203-5204 und die Raytracing-Daten 4902 zugreifen, um Traversierungsoperationen für diesen Strahl an demselben Ort innerhalb der BVH, an der sie aufgehört hat, zu beginnen.

[0723] In einer Ausführungsform bestimmt die Thread-Präemptionslogik 9120, wann ein Satz von Traversierungs-Threads (oder anderen Thread-Typen) präemptiert werden soll, wie hierin beschrieben, (um z. B. Ressourcen für eine Arbeitslast/einen Thread mit höherer Priorität freizugeben), und benachrichtigt die Strahl/-Rahmen-Traversierungseinheit 8930, sodass sie die Verarbeitung eines der aktuellen Threads pausieren kann, um Ressourcen zur Verarbeitung des Threads mit höherer Priorität freizugeben. In einer Ausführungsform wird die „Benachrichtigung“ einfach durch Versenden von Anweisungen für einen neuen Thread durchgeführt, bevor die Traversierung auf einem alten Thread abgeschlossen ist.

[0724] Somit beinhaltet eine Ausführungsform der Erfindung Hardware-Unterstützung sowohl für synchrones Raytracing, das in Arbeitsgruppenmodus arbeitet (d. h., bei der alle Threads einer Arbeitsgruppe synchron ausgeführt werden), als auch für asynchrones Raytracing, das bindungslosen Thread-Dispatch verwendet, wie hierin beschrieben. Diese Techniken verbessern drastisch die Leistungsfähigkeit im Vergleich zu aktuellen Systemen, die erfordern, dass alle Threads in einer Arbeitsgruppe abgeschlossen werden, bevor eine Präemption durchgeführt wird. Im Gegensatz dazu können die hierin beschriebenen Ausführungsformen eine Präemption auf Stapelebene und Thread-Ebene durch genaues Verfolgen einer Traversierungsoperation, Speichern nur der Daten, die zum Neustart erforderlich sind, und Verwenden kurzer Stapel, wenn geeignet, durchführen. Diese Techniken sind zumindest teilweise möglich, weil die Raytracing-Beschleunigungs-Hardware- und Ausführungseinheiten 8910 über eine persistente Speicherstruktur 3198 kommunizieren, die auf der Pro-Strahl-Ebene und der Pro-BVH-Ebene verwaltet wird.

[0725] Wenn eine TraceRay-Nachricht wie oben beschrieben erzeugt wird, und es eine Präemptions-Forderung gibt, kann die Strahltraversierungsoperation in verschiedenen Stufen präemptiert werden, einschließlich (1) noch nicht gestartet, (2) teilweise abgeschlossen und präemptiert, (3) Traversierung abgeschlossen ohne bindungslosen Thread-Dispatch und (4) Traversierung abgeschlossen, aber mit einem bindungslosen Thread-Dispatch. Falls die Traversierung noch nicht gestartet ist, werden keine zusätzlichen Daten von dem Verfolgungsarray 5249 benötigt, wenn die Raytrace-Nachricht wieder aufgenommen wird. Falls die Traversierung teilweise abgeschlossen wurde, liest der Traversierungs-/Stapelverfolger 5248 das Verfolgungsarray 5249, um zu bestimmen, wo die Traversierung wieder aufgenommen werden soll, durch Verwenden der Raytracing-Daten 4902 und Stapel 5121 nach Bedarf. Er kann das Verfolgungsarray 5249 durch Verwenden der eindeutigen ID, die jedem Strahl zugewiesen ist, abfragen.

[0726] Falls die Traversierung abgeschlossen war, und es keinen bindungslosen Thread-Dispatch gab, kann ein bindungsloser Thread-Dispatch durch Verwenden jeglicher Trefferinformationen, die in dem Verfolgungsarray 5249 (und/oder anderen Datenstrukturen 4902, 5121) gespeichert sind, geplant werden. Falls die Traversierung abgeschlossen ist und es einen bindungslosen Thread-Dispatch gab, dann wird der bindungslose Thread wiederhergestellt und die Ausführung wird wieder aufgenommen, bis sie abgeschlossen ist.

[0727] In einer Ausführungsform beinhaltet das Verfolgungsarray 5249 einen Eintrag für jede eindeutige Strahl-ID für Strahlen im Flug, und jeder Eintrag kann eine der Ausführungsmasken 9150-9153 für einen entsprechenden Thread beinhalten. Alternativ können die Ausführungsmasken 9150-9153 in einer separaten Datenstruktur gespeichert werden. In jeder Implementierung kann jeder Eintrag in dem Verfolgungsarray 5249 einen 1-Bit-Wert aufweisen oder mit diesem assoziiert sein, um anzugeben, ob der entsprechende Strahl erneut ausgesendet werden muss, wenn die Strahl/Rahmen-Traversierungseinheit 8930 den Betrieb nach einer Präemption wieder aufnimmt. In einer Implementierung wird dieser 1-Bit-Wert innerhalb einer Thread-Gruppe (d. h. einer Arbeitsgruppe) verwaltet. Dieses Bit kann zu Beginn der Strahltraversierung auf 1 gesetzt werden, und kann wieder auf 0 zurückgesetzt werden, wenn die Strahltraversierung abgeschlossen ist.

[0728] Die hierin beschriebenen Techniken ermöglichen, dass Traversierungs-Threads, die mit der Strahltraversierung assoziiert sind, durch andere Threads (z. B. Berechnungs-Threads) präemptiert werden, ohne darauf zu warten, dass der Traversierungs-Thread und/oder die gesamte Arbeitsgruppe abgeschlossen ist, wodurch die Leistungsfähigkeit, die mit Threads mit hoher Priorität und/oder niedriger Latenz assoziiert ist, verbessert wird. Außerdem kann aufgrund der hierin beschriebenen Techniken zum Verfolgen des Traversierungsfortschritts der Traversierungs-Thread dort neu gestartet werden, wo er aufgehört hat, was signifikante Verarbeitungszyklen und Ressourcennutzung bewahrt. Zusätzlich ermöglichen die oben beschriebenen Ausführungsformen, dass ein Arbeitsgruppen-Thread einen bindungslosen Thread spawnt und Mechanismen für eine Rekonvergenz bereitstellt, um in den ursprünglichen SIMD-Architekturzustand zurück zu gelangen. Diese Techniken verbessern effektiv die Leistungsfähigkeit für Raytracing- und Berechnungs-Threads um eine Größenordnung.

EINRICHTUNG UND VERFAHREN ZUM DATENPARALLELEN RAYTRACING

[0729] Bei wissenschaftlicher Visualisierung (aber auch in Filmen und anderen Domänen) wachsen Datensätze zunehmend auf Größen an, die nicht durch einen einzelnen Knoten verarbeitet werden können. Für Offline-Algorithmen (hauptsächlich in Filmen) wird dies häufig durch Paging, Caching und Out-Of-Core-Techniken gehandhabt; wenn aber eine interaktive Einstellung erforderlich ist (z. B. Visualisierung für Öl und Gas, wissenschaftliche Visualisierung in einer Großdaten-/HPC-Umgebung, interaktive Filminhaltsvorschauen usw.) ist dies nicht mehr möglich. In diesem Fall ist es absolut erforderlich, eine Form von datenparallelem

Rendern zu verwenden, wobei die Daten über mehrere unterschiedliche Knoten hinweg partitioniert werden, sodass sämtliche Daten über alle Knoten hinweg gespeichert werden können, und wobei diese Knoten beim Rendern des erforderlichen Bildes zusammenwirken.

[0730] Die Ausführungsformen der Erfindung beinhalten eine Einrichtung und ein Verfahren zum Reduzieren der Bandbreite für das Transferieren von Strahlen und/oder Volumenblöcken in dem Kontext eines datenverteilten Raytracing über mehrere Berechnungsknoten hinweg. **Fig. 92** veranschaulicht zum Beispiel Raytracing-Cluster 9200, das mehrere Raytracing-Knoten 9210-9213 umfasst, die Raytracing-Operationen parallel durchführen, wobei potenziell die Ergebnisse auf einem der Knoten kombiniert werden. In der veranschaulichten Architektur sind die Raytracing-Knoten 9210-9213 über ein Gateway 9220 kommunikativ mit einer clientseitigen Raytracing-Anwendung 9230 gekoppelt.

[0731] Unten wird in der Beschreibung angenommen, dass mehrere Knoten 9210-9213 die Raytracing-Daten gemeinsam halten. Jeder solche Knoten 9210-9213 kann ein(e) oder mehrere CPUs, GPUs, FPGAs usw. enthalten, und die Berechnungen können entweder auf einzelnen oder einer Kombination dieser Ressourcen durchgeführt werden. In einer Ausführungsform kommunizieren die Berechnungsknoten 9210-9213 miteinander durch eine Form von Netzwerk 9215, wie etwa Infiniband, OmniPath oder NVLink, um einige zu nennen. Die Daten können über die Speicher dieser Knoten 9210-9213 hinweg partitioniert werden, entweder da die Anwendung, die den Renderer verwendet, die Daten selbst partitioniert hat (wie dies für viele In-situ-Algorithmen oder parallele Middleware, wie etwa ParaView, Visite usw., der Fall ist), oder da der Renderer diese Partitionierung erzeugt hat.

[0732] Um ein paralleles Rendern in einer solchen Umgebung auszuführen, gibt es eine Vielzahl von algorithmischen Auswahlen: Bei Compositing-basierten Ansätzen rendert jeder Knoten ein Bild seiner lokalen Daten und kombiniert diese Teilergebnisse durch Verwenden von Tiefen- und/oder Alpha-Compositing. Ansätze zum Weiterleiten (oder Cachen) von Daten berechnen Operationen eines gegebenen Strahls (oder Pixels, Pfads usw.) auf einem gegebenen Knoten, detektieren, wann immer dieser Strahl/dieses Pixel/dieser Pfad Daten benötigt, die sich auf einem anderen Knoten befinden, und rufen diese Daten auf Anfrage ab. Strahlweiterleitungsbasierte Ansätze leiten Daten nicht an die Strahlen weiter, die diese benötigen, und senden stattdessen die Strahlen dahin, wo die Daten sind: wenn ein Knoten detektiert, dass ein Strahl eine Verarbeitung mit Daten eines anderen Knotens benötigt, sendet er diesen Strahl an den Knoten, der diese Daten besitzt.

[0733] Unter diesen Auswahlen ist das Compositing am einfachsten und am weitesten verbreitet; es ist jedoch nur für relativ einfache Rendereffekte anwendbar, und kann nicht einfach für Effekte, wie etwa Schatten, Reflexionen, Umgebungsverdeckung, globale Beleuchtung, volumetrische Streuung, volumetrische Schatten usw., verwendet werden. Solche Effekte, die häufiger durch Benutzer benötigt werden, erfordern eine Art von Raytracing, bei dem das datenparallele Rendern entweder Daten zu den Strahlen abruft, oder die Strahlen zu den Daten sendet. Beide Ansätze wurden schon vorher verwendet, und ihre Einschränkungen werden gut verstanden. Insbesondere leiden beide Ansätze unter hohen Bandbreitenanforderungen, entweder durch Senden von bis zu Milliarden von Strahlen (zur Strahlweiterleitung) oder dadurch, dass sie jeden Knoten 9210-9213 bis zu viele Gigabytes an Daten (zur Datenweiterleitung) abrufen lassen, oder beides (wenn eine Kombination beider verwendet wird).

[0734] Obwohl die Netzwerkbandbreite dramatisch ansteigt, steigen auch Datengröße und/oder Strahlzählung, was bedeutet, dass diese Bandbreite in der Praxis sehr schnell der limitierende Faktor für die Leistungsfähigkeit ist. Tatsächlich ist es oft der einzige Grund, dass interaktive Leistungsfähigkeit nicht erreicht werden kann, außer bei sehr vereinfachenden Einstellungen (wie etwa Nur-Primärstrahl-Rendern, wobei auch Compositing verwendet werden könnte).

[0735] Eine Ausführungsform der Erfindung konzentriert sich auf den Kerngedanken, dass in der Praxis sehr große Teile der Daten für einen gegebenen Frame oft tatsächlich nicht ins Gewicht fallen. Zum Beispiel verwendet der Benutzer beim Volumen-Rendern häufig eine „Übertragungsfunktion“, um bestimmte Bereiche der Daten hervorzuheben, wobei weniger interessante Datensätze auf vollständig transparent eingestellt werden. Eindeutig würde ein Strahl, der nur „uninteressante“ Daten traversieren würde, diese Daten nicht abrufen (oder an diese Daten gesendet werden) müssen, und die jeweilige Bandbreite kann eingespart werden. In ähnlicher Weise muss er für oberflächenbasiertes Raytracing, falls ein Strahl durch einen Raumbereich läuft, der einem anderen Knoten gehört, dort aber tatsächlich keinerlei Dreiecke überschneidet, tatsächlich nicht mit Dreiecken dieses anderen Knotens interagieren.

[0736] Eine Ausführungsform erweitert die Konzepte von „Leerraumüberspringen“ und „Hüllkörper“ von einzelnen Knoten zu datenparallelem Rendern in der Form der Verwendung sogenannter „Proxys“ 9230-9233 für die Daten eines Knotens. Insbesondere berechnet jeder Knoten einen Proxy 9230-9233 mit sehr geringer Speichergrundfläche seiner eigenen Daten, sodass dieser Proxy die Fähigkeit bereitstellt, diese Daten entweder zu approximieren oder konservativ zu binden. Alle Knoten 9210-9213 tauschen dann ihre Proxys 9230-9233 aus, sodass jeder Knoten die Proxys jedes anderen Knotens hat. Zum Beispiel beinhalten die Proxys 9230, die auf dem Knoten 9210 gespeichert sind, Proxy-Daten von den Knoten 9211-9213. Wenn ein Knoten einen Strahl durch einen räumlichen Bereich verfolgen muss, der einem anderen Knoten gehört, verfolgt er zuerst diesen Strahl durch seine eigene Kopie des Proxys dieses Knotens. Falls dieser Proxy garantiert, dass keine aussagekräftige Interaktion auftritt, kann er das Senden dieses Strahls/das Abrufen dieser Daten überspringen, wobei die dafür erforderliche Bandbreite eingespart wird.

[0737] **Fig. 93** veranschaulicht zusätzliche Einzelheiten eines Raytracing-Knotens 9210 gemäß einer Ausführungsform der Erfindung. Volumenunterteilungsmodul 9265 unterteilt ein Volumen in mehrere Partitionen, von denen jede durch einen unterschiedlichen Knoten verarbeitet wird. Ein Arbeitsdatensatz 9360 umfasst die Daten für die Partition, die durch den Knoten 9210 verarbeitet werden sollen. Ein Proxy-Erzeugungsmodul 9250 erzeugt ein Proxy 9340 basierend auf dem Arbeitsdatensatz 9360. Der Proxy 9340 wird zu jedem der anderen Raytracing-Knoten 9211-9213 übertragen, die den Proxy verwenden, um nicht benötigte Daten wie hierin beschrieben auszusortieren. In ähnlicher Weise werden Proxys 9341-9343, die jeweils auf den Knoten 9211-9213 erzeugt werden, zu dem Knoten 9210 übertragen. Eine Raytracing-Engine 9315 führt Raytracing-Operationen durch Verwenden sowohl des lokal gespeicherten Arbeitsdatensatzes 9360 als auch der Proxys 9341-9343 durch, die durch jeden der miteinander verbundenen Knoten 9211-9213 bereitgestellt werden.

[0738] **Fig. 94** veranschaulicht ein Beispiel, bei dem in dem Kontext des Volumen-Renderns ein gegebener Volumendatensatz 9400 zu groß ist, um auf einem Knoten gerendert zu werden, sodass er in mehrere Blöcke 9401-9404 (in diesem Fall einen 2x2-Satz) partitioniert wird. Wie in **Fig. 95** veranschaulicht, kann dieses logisch partitionierte Volumen dann über unterschiedliche Knoten 9210-9213 hinweg verteilt werden, wobei jeder einen Teil des Volumens beibehält.

[0739] Traditionell muss ein Knoten jedes Mal, wenn er einen Strahl senden will, der durch räumliche Bereiche anderer Knoten verläuft, diesen Strahl entweder an diese Knoten senden, oder Daten dieser Knoten abrufen. In **Fig. 96** verfolgt der Knoten 9210 zum Beispiel einen Strahl, der durch den Raum läuft, der den Knoten 9211-9213 gehört.

[0740] Wie in **Fig. 97** veranschaulicht, berechnet in einer Ausführungsform jeder Knoten 9210-9213 jeweils einen lokalen Proxy 9240-9243 für seinen Teil der Daten 9401-9404, wobei der Proxy eine beliebige Art von Objekt ist, das (erheblich) kleiner ist, aber das Approximieren oder konservative Begrenzen dieser Daten dieses Knotens ermöglicht. In einer Ausführungsform berechnet jeder Knoten zum Beispiel ein sogenanntes „Makrozellengitter“; ein Gitter mit niedrigerer Auflösung, bei dem jede Zelle einem Bereich von Zellen in dem Eingabevolumen entspricht, und bei dem jede solche Zelle zum Beispiel den minimalen und maximalen Skalarwert in diesem Bereich speichert (in dem Kontext des Einzelknotenrenderns wird dies üblicherweise für „Raumüberspringen“ verwendet). In dem veranschaulichten Beispiel berechnet jeder Knoten 9210-9213 einen solchen Proxy 9240-9243 für seinen Teil der Daten. In einer Ausführungsform tauschen dann alle Knoten ihre jeweiligen Proxys aus, bis jeder Knoten alle Proxys für jeden Knoten aufweist, wie in **Fig. 98** veranschaulicht.

[0741] Sind für eine gegebene Übertragungsfunktionseinstellung nur einige der Datenwerte tatsächlich interessant (in dem Sinne, dass sie nicht vollständig transparent sind), so kann dies konservativ in dem Proxy detektiert werden (ebenso wie es herkömmliches Einzelknoten-Raumüberspringen macht). Dies ist in **Fig. 99** als Bereiche 9940-9943 veranschaulicht.

[0742] Da zusätzlich jeder Knoten Proxys jedes anderen Knotens aufweist, kann jeder Knoten auch konservativ begrenzen, welche Bereiche der anderen Knoten interessant sind, basierend auf den Proxys, die er für diese Knoten aufweist, wie in **Fig. 100** gezeigt. Falls der Knoten 9210 einen Strahl verfolgen muss, der Datenbereiche der Knoten 9211-9213 überspannt, dann kann der Strahl auf den Proxy projiziert und dort traversiert werden, wie durch den gepunkteten Pfeil angegeben. Dies gibt an, dass, obwohl der Strahl durch den Raum läuft, der den Knoten 9210-9212 gehört, nur der Knoten 9212 tatsächlich jegliche interessante Bereiche enthält, sodass dieser Strahl an den Knoten 9212, wie in **Fig. 100** durch den durchgezogenen Pfeil angegeben, ohne Verarbeitung auf den Knoten 9210 oder Senden an den Knoten 9211 weitergeleitet werden kann

(oder, in einem Caching-Kontext, können Daten nur von dem Knoten 9210 anstatt von sowohl 9211 als auch 9212 abgerufen werden).

[0743] Ein Verfahren gemäß einer Ausführungsform der Erfindung ist in **Fig. 101** veranschaulicht. Das Verfahren kann im Kontext der oben beschriebenen Architekturen implementiert werden, ist aber nicht auf irgendeine spezielle Verarbeitungs- oder Systemarchitektur beschränkt.

[0744] Bei 10101 wird ein Volumen logisch in mehrere Partitionen (N) unterteilt, und bei 10102 werden Daten, die mit den N Partitionen assoziiert sind, auf N unterschiedliche Knoten verteilt (z. B. eine Partition pro Knoten in einer Ausführungsform). Bei 10103 berechnet jeder Knoten ein Proxy für seine jeweilige Partition, und sendet das Proxy an die anderen Knoten. Bei 10104 werden Traversierungs-/Überschneidungsoperationen für einen aktuellen Strahl oder eine aktuelle Gruppe von Strahlen (z. B. ein Strahlenbündel) durch Verwenden der Proxys durchgeführt, wobei bestimmte Bereiche innerhalb der Proxys, die für die Operationen nicht relevant sind, potenziell ignoriert werden. Wie erwähnt, sind für eine gegebene Übertragungsfunktionseinstellung nur einige der Datenwerte tatsächlich interessant (z. B. weil sie nicht vollständig transparent sind). Dies kann konservativ in dem Proxy detektiert werden, wie dies mit dem Einzelknoten-Raumüberspringen geschieht. Falls der Strahl mit dem Proxy interagiert, bestimmt bei 10105, werden bei 10106 der Strahl bzw. die Strahlen zu dem Knoten gesendet, der mit dem Proxy assoziiert ist, oder Daten werden von dem Knoten wieder abgerufen. Der nächste Strahl oder die nächste Gruppe von Strahlen wird dann bei 10107 ausgewählt.

[0745] Natürlich hängen die hierin beschriebenen Ausführungsformen von den tatsächlichen Daten, der Strahlverteilung, der Übertragungsfunktion usw. ab. Es ist jedoch nicht ungewöhnlich, dass Daten sehr ähnlich dem oben gegebenen Beispiel aussehen. Offensichtlich würden die Proxys für Datensätze/Übertragungsfunktionen, die zu einem deutlich weniger „dünnbesetzten“ nachklassifizierten Datensatz führen (d. h. Datensatz, nachdem die Übertragungsfunktion angewendet wurde), nicht viel helfen. In diesem Fall würden die Strahlen jedoch wahrscheinlich sehr schnell enden und müssen daher nicht sehr oft zwischen Knoten gesendet werden und erzeugen keine übermäßige Bandbreite. Im Wesentlichen sind die größten Problemfälle jene, bei denen viele der Daten spärlich sind und bei denen Strahlen über viele Knoten laufen, und dies sind genau die Fälle, bei denen die hierin beschriebenen Techniken am effektivsten sind.

EINRICHTUNG UND VERFAHREN zum RAYTRACING MIT SHADER-AUFRUF-GRAPHENANALYSE

[0746] Während Raytracing im Wesentlichen eine „Blackbox“ hinsichtlich der Art und Weise ist, wie Shader versendet und ausgeführt werden, beeinflusst das Shader-Ausführungsmuster direkt die Debuggbarkeit und die Leistungsfähigkeit der Raytracing-Engine.

[0747] Bestimmte Raytracing-Implementierungen setzen Beschränkungen für die Shader-Aufzeichnungsausführungen, die in einen einzigen SIMD-Thread gebündelt werden können. Zum Beispiel können in der BTD-Einheit (BTD: Bindless Thread Dispatch - bindungsloser Thread-Dispatch), die durch den Abtretungsempfänger der vorliegenden Anmeldung entwickelt wird, gewisse Raytracing-Hardware-Shader-Aufzeichnungen nur innerhalb des Umfangs desselben Dual-Sub-Slice (DSS) gebündelt werden (siehe z. B. **Fig. 52A** und assoziierter Text). Falls darüber hinaus zwei Shader-Aufzeichnung-Dispatches über die Zeit verteilt sind, werden sie aufgrund von begrenztem Pufferraum möglicherweise nicht miteinander gebündelt. Diese zwei Einschränkungen können dazu führen, dass einem ausführungskritischen Pfad, wie etwa einem Sub-Slice, mehr Arbeit als der andere zugewiesen wird, und können auch zu Hardware-Threads mit niedriger SIMD-Spurnutzung führen.

[0748] Ein „Sub-Slice“ oder „Slice“, auf das hierin Bezug genommen wird, umfasst einen definierten Teilsatz einer Raytracing-Engine. Ein Sub-Slice kann alle der Raytracing-Hardware-Ressourcen beinhalten, die erforderlich sind, um unabhängig Traversierung und Überschneidung an einer spezifizierten Anzahl von Strahlen durchzuführen. **Fig. 52A** veranschaulicht zum Beispiel duale Strahlbänke 5201-5202 bzw. duale entsprechende Stapel 2403-2404. In einer Ausführungsform umfasst jede Strahlbank 5201-5202 mehrere Einträge zum Speichern einer entsprechenden Vielzahl von eingehenden Strahlen 5206, die aus dem Speicher geladen werden. Die entsprechenden Stapel 5203 bzw. 5204 umfassen ausgewählte BVH-Knotendaten 5290-5291, die aus dem Speicher gelesen und lokal zur Strahlverarbeitung gespeichert werden.

[0749] **Fig. 102** veranschaulicht ein Beispiel mit Shader-Aufzeichnungen 1-4, die innerhalb von zwei Sub-Slices, 1 und 2, versendet werden. In diesem Beispiel können die zwei Instanzen der Shader-Aufzeichnung 4 10201, die für die Strahlen 1 und 2 innerhalb des Sub-Slice 1 versendet werden, nicht gruppiert werden, da

die Shader-Aufrufe um zu viel Zeit getrennt sind. Die zwei Instanzen der Shader-Aufzeichnung 3 10202 können nicht gruppiert werden, da sie sich in unterschiedlichen Sub-Slices befinden. Außerdem ist, wie durch 10203 angegeben, die Arbeit, die über die Sub-Slices versendet wird, ungleichmäßig.

[0750] Bestehende Techniken, wie etwa die Verwendung von Hardware-Metriken und binären Shader-Instrumentierungsansätzen, stellen nicht den erforderlichen Detailgrad bereit, um ein Debugging der Raytracing-Leistungsfähigkeit zu ermöglichen. Zum Beispiel können diese Techniken nicht verwendet werden, um zu bestimmen, warum bestimmte Shader-Aufrufe im Laufe der Zeit verteilt wurden, den Aufrufstapel der Shader-Aufrufe, die den großen kritischen Pfad erzeugt haben, und die Primär- oder Sekundärstrahlen, die geändert/entfernt werden können, um das Problem anzusprechen.

[0751] Ausführungsformen der Erfindung beinhalten Techniken zum Erzeugen eines Raytracing-Shader-Aufruf-Graphen, der Beziehungen zwischen ausgeführten Shader-Aufzeichnungen aufbaut und bindet sie an Strahlen, in deren Umfang sie ausgeführt wurden. Diese Ausführungsformen automatisieren auch eine Engpassanalyse für Raytracing-Shader-Scheduling. Dies ermöglicht im Gegenzug einem Anwendungsentwickler, eine gewichtete Entscheidung darüber zu treffen, wie die Strahlverteilung zu ändern ist oder welche Shader zu optimieren sind.

[0752] Eine Ausführungsform der Erfindung beinhaltet die in **Fig. 103** veranschaulichten Betriebsphasen. Das veranschaulichte Verfahren kann auf den verschiedenen hierin beschriebenen Systemarchitekturen durchgeführt werden, ist aber nicht auf irgendeine spezielle Architektur beschränkt.

[0753] Die erste Phase 10301 führt binäre Instrumentierung von Raytracing-Shadern und Trace-Datensammlung durch. Die Instrumentierung ist aufgrund der Tatsache kompliziert, dass Raytracing-Shader durch bestimmte Compiler (z. B. wie etwa dem Intel Graphics Compiler (IGC)) in mehrere Chunks aufgeteilt werden können und dass die Verfolgung von DispatchRaysIndex eine ressourcenintensive Operation ist. Folglich wird in einer Ausführungsform Instrumentierungscode an spezifischen Punkten hinzugefügt, sodass das Ausführungsprofil rekonstruiert werden kann und Leistungsfähigkeitsdaten gemessen werden können, aber gleichzeitig der eingeführte Overhead minimiert wird. Spezifische Einzelheiten für eine Ausführungsform werden nachstehend bereitgestellt.

[0754] In der nächsten Phase 10302 wird ein Aufruf-Graph rekonstruiert. In der Rekonstruktionsphase 10302 werden, nachdem Shader ausgeführt wurden, die gesammelten Daten verarbeitet und in Leistungsfähigkeitsdaten umgewandelt (z. B. Anweisungsblocktimings, SIMD-Belegungswerte usw.). Eine Assoziation dieser Daten wird mit konkreten Shader-Aufzeichnungen konstruiert und die Shader-Aufzeichnungen werden auf einen Ursprungs-Primärstrahl abgebildet.

[0755] In Phase 10303 wird der rekonstruierte Aufruf-Graph auf den Shader-Quellcode abgebildet. Da der Compiler zu Optimierungszwecken einen Raytracing-Shader in einen anderen Inline bringen kann, gibt es möglicherweise keine 1-zu-1-Abbildung zwischen Binär- und Quellcode. Somit wird in einer Ausführungsform die Abbildung zwischen Binärcodebereichen auf den ursprünglichen Quellcode vom Compiler abgerufen, der dazu ausgelegt sein kann, diese Informationen während der Shader-Kompilierung zu akkumulieren. Unter Verwendung dieser Informationen werden die Tracing-Punkte auf den Quellcode abgebildet, wodurch ermöglicht wird, dass die Blöcke in dem Shader-Aufruf-Graphen entsprechenden Codeabschnitten zur weiteren Analyse zugewiesen werden.

[0756] In Phase 10304 werden Leistungsfähigkeitsengpässe und Möglichkeiten zur Verbesserung identifiziert. Eine Ausführungsform verwendet den konstruierten Shader-Aufruf-Graphen, um ineffiziente Ausführungsmuster zu lokalisieren, die eine effiziente Shader-Aufzeichnungsgruppierung des Thread-Dispatcher (z. B. BTD) verhindern. Diese ineffizienten Ausführungsmuster können zum Beispiel Identifizieren von Instanzen, bei denen dieselbe Shader-Aufzeichnung über mehrere duale Sub-Slices verteilt ist, Instanzen, bei denen Ausführungen für eine Shader-Aufzeichnung zu sehr über die Zeit verteilt sind und daher eine niedrige SIMD-Belegung erzeugen, und Instanzen von ausführungskritischen Pfaden beinhalten.

[0757] In der Endphase 10305 werden Optimierungsempfehlungen erzeugt. Basierend auf der Art des Problems kann zum Beispiel ein Optimierungsschritt vorgeschlagen werden, wie etwa Minimieren einer Anzahl von Sekundärstrahlen für einen spezifischen Primärstrahl, Ändern eines Raycasting-Musters, um eine Trefferlokalität in Bezug auf einen speziellen Shader-Aufzeichnungsaufruf zu erhöhen, und Optimieren eines bestimmten Shaders, der in einen kritischen Pfad fällt.

[0758] Unter Bezugnahme auf **Fig. 104** werden in einer Implementierung die verschiedenen Phasen durch eine Raytracing-Optimierungsplattform 10400 durchgeführt, die als ein Werkzeug zum Optimieren von Raytracing-Shadern und assoziiertem Programmcode implementiert werden kann. Die Plattform 10400 kann in Software, Hardware oder einer beliebigen Kombination davon implementiert sein. Zum Beispiel beinhaltet die Raytracing-Optimierungsplattform 10400 Logik 10411-10415 zum Implementieren der entsprechenden Phasen 10301-10305, die mit Bezug auf **Fig. 103** beschrieben sind.

[0759] Binär-Shader 10401 werden als Eingabe an eine Binärinstrumentierungs-Engine 10411 geliefert, die die erste Phase 10301, binäre Instrumentierung von Raytracing-Shadern und Trace-Datensammlung durchführt. Die resultierenden instrumentierten Shader werden ausgeführt 10402 und die Ausführungsdaten werden gesammelt, um durch die Aufruf-Graphen-Rekonstruktionslogik 10412 zum Rekonstruieren des Aufruf-Graphen verwendet zu werden (d. h. Phase 10302).

[0760] Quellcode-zu-Binärcode-Bereichsabbildungen werden von dem Compiler 10404 bereitgestellt, der dazu ausgelegt ist, diese Informationen während der Shader-Kompilierung zu akkumulieren. Unter Verwendung dieser Informationen bildet die Shader-Quellabbildungslogik 10413 die Tracing-Punkte auf den Quellcode ab (Phase 10303), wodurch ermöglicht wird, dass die Blöcke in dem Shader-Aufruf-Graphen entsprechenden Codeabschnitten zur weiteren Analyse zugewiesen werden.

[0761] Ein Effizienzanalysemodul 10414 analysiert die Ergebnisse, um Leistungsfähigkeitsengpässe und Möglichkeiten zur Verbesserung zu identifizieren (Phase 10304). Eine Optimierungslogik 10415 erzeugt dann empfohlene Optimierungshandlungen (Phase 10305).

[0762] Die empfohlenen Optimierungshandlungen 10305 können einem Endbenutzer in Form einer Liste von Empfehlungen bereitgestellt werden. Alternativ oder zusätzlich können bestimmte Typen von Optimierungshandlungen 10305 bei Benutzeranfrage automatisch durchgeführt werden. Zum Beispiel kann die Raytracing-Optimierungsplattform 10400 eine Option bereitstellen, um eine oder mehrere der detektierten Ineffizienzen in dem Programmcode automatisch zu beheben, und dies bei Bestätigung durch den Benutzer tun.

[0763] Die folgenden zusätzlichen Einzelheiten sind für eine bestimmte Ausführungsform der Erfindung bereitgestellt. Es sei jedoch darauf hingewiesen, dass viele dieser spezifischen Einzelheiten nicht erforderlich sind, um die der Erfindung zugrundeliegenden Prinzipien zu erfüllen.

[0764] In einer Ausführungsform ist eine Shader-ID ein Wert, der einen Shader eindeutig identifiziert, wie etwa ein Hash-Wert, der durch den Compiler erzeugt wird. Der Shader-Aufzeichnungsindex ist ein Index zu einer Shader-Aufzeichnung in einer entsprechenden Fehltreffer-, Treffergruppen- oder Aufrufbarer-Shader-Tabelle. Die Shader-Aufzeichnung selbst kann den Shader und eine lokale Root-Signatur beinhalten.

[0765] In einer Ausführungsform haben die folgenden zusätzlichen Begriffe die folgenden Bedeutungen:

- Ein „Dispatch-Strahlenindex“ umfasst die x- und y-Werte für die Breiten- und Höhenabmessungen des Strahl-Dispatch-Gitters.
- Eine „TID“ ist eine Thread-Kennung, die einen Thread auf dem Grafikprozessor eindeutig identifiziert.
- Eine „SIMD-Spur-Dispatch-Maske“ ist eine Maske, die spezifiziert, welche SIMD-Spuren für einen bestimmten Shader-Aufruf aktiv sind.
- „RTDispatchGlobals“ sind globale Argumente für alle Raytracing-Shader-Aufrufe.
- „LocalArg“ bezieht sich auf lokale Argumente für einen bestimmten Shader-Aufruf und wird ihm als ein Zeiger bereitgestellt.
- „Stapel-ID“ bezieht sich auf den Aufrufstapel von Shader-Aufrufen, die von demselben „Primär"-Strahl stammen (z. B. der TraceRay-Intrinsik in dem RayGen-Shader). Der Aufrufstapel kann über die Sendeanweisung, wobei das „Stapel-ID-Freigabe“-Flag gesetzt ist, zu einem Pool freier Stapel-IDs bewegt werden.
- „EOT“ bezieht sich auf das Ende einer Thread-Nachricht.

1. Binäre Instrumentierung von Raytracing-Shadern & Trace-Sammlung

[0766] In der ersten Phase 10301 (binäre Instrumentierung von Shadern und Trace-Sammlung) fügt die Binärinstrumentierungslogik der Raytracing-Optimierungsplattform 10400 Binärinstrumentierungs-Trace-

Punkte hinzu, die unterschiedliche Daten sampeln. Raytracing-Shader werden üblicherweise durch den Compiler in mehrere Chunks aufgeteilt, wie etwa eine einzige Primär-Shader-Aufzeichnung mit mehreren Fortsetzungs-Shader-Aufzeichnungen. Diese Chunks können auf verschiedenen Ausführungseinheiten (EUs) und Hardware-Threads, aber innerhalb desselben Sub-Slice (z. B. eines DSS in einigen Implementierungen) ausgeführt werden. Dies erschwert das Verfolgen davon, welcher Benutzer-Shader oder welche Shader-Aufzeichnung tatsächlich bei einer spezifischen EU und zu einer gegebenen Zeit ausgeführt wird. Um dieses Problem zu lösen, wird jeder binäre Chunk an jedem „Eintrittspunkt“ und vor jeder entsprechenden EOT-Nachricht instrumentiert. In einer Ausführungsform wird ein Tracing an ShaderID und LocalArgPtr durchgeführt (was auf der Nachverarbeitungsstufe in ShaderRecordIndex umgewandelt werden).

[0767] Fig. 105 veranschaulicht ein Beispiel, bei dem ein Raytracing-Shader 10511 durch den Compiler 10404 in Chunks 10515 aufgeteilt wird. Die Binär-Shader-Instrumentierungs- und Trace-Sammel-Logik 10301 fügt dann den Chunks 10515 Instrumentierungscode hinzu. Wie bei 10502 angegeben, wird Trace-Instrumentierungscode an jedem Eintrittspunkt (z. B. bei 10521A-C) und vor jedem EOT (z. B. bei 10522) hinzugefügt.

[0768] Leistungsfähigkeitscharakteristiken sind zum Klassifizieren eines konkreten Engpasstyps wichtig. Das bindungslose Thread-Dispatching kann einige Spuren maskieren, wenn es nicht in der Lage ist, ausreichend ähnliche Shader-Aufzeichnungen zu akkumulieren. In einer Ausführungsform wird, um dieses Problem zu analysieren, die SIMD-Spur-Dispatch-Maske zu Beginn jeder Primär-Shader-Aufzeichnung gesampelt, die in einen Pro-Shader-SIMD-Spur-Belegungswert (z. B. einen Prozentsatz belegter Spuren) umgewandelt wird. Zusätzlich dazu werden Zeitstempel an jedem Primär-/Fortsetzungs-Shader-Aufzeichnung-„Eintrittspunkt“ und vor jeder EOT-Nachricht gesammelt, um die Ausführungslatenz für jeden Shader zu charakterisieren.

[0769] Um alle Shader-Aufzeichnungsausführungen in Cluster pro Primärstrahl zu organisieren und ihre Aufrufreihenfolge zu verfolgen, verfolgt eine Ausführungsform der Raytracing-Optimierungsplattform 10400 DispatchRaysIndex, bei dem es sich um eine Funktion handelt, die Berechnungen sowohl mit statischen Daten vom Compiler 10404 als auch mit dynamischen Daten, die zur Laufzeit verfolgt werden (z. B. Ausführungsdaten in **Fig. 104**), erfordert. Zusätzlich führt eine Ausführungsform der Raytracing-Optimierungsplattform 10400 diese Berechnungen pro Spur durch, da jede SIMD-Spur einen Shader von einem vollständig anderen Strahl verarbeiten kann:

DispatchRaysIndex[lane] = F(static-data, dynamic_data[lane])

[0770] Um den assoziierten Overhead zu vermeiden, verfolgt eine Ausführungsform der Raytracing-Optimierungsplattform 10400 nur die Stapel-ID und Sub-Slice-ID (z. B. die DSS-ID oder SS-ID) zu Beginn jeder Primär- und Fortsetzungs-Shader-Aufzeichnung, und verfolgt DispatchRaysIndex nur vor der „StackID Free“-Nachricht. Das Paar von [Stapel-ID, SS-ID] ist ein niedriger Overhead zum Verfolgen, da sie aus Registern gelesen werden, und identifiziert die Kette von Shader-Ausführungen eindeutig, die durch den Primärstrahlwurf ausgelöst werden, bis die „StackID Free“-Nachricht gesendet wird.

[0771] Diese Technik ist in **Fig. 106** gezeigt, die veranschaulicht, dass Stapel-ID und DSS-ID verfolgt und die Ergebnisse in einem Profilbildungspuffer 10601 zu Beginn der Shader-Aufzeichnungen 1 und 2 gespeichert werden. Nur als Reaktion auf eine Änderung des StackIDRelease-Indikators (der angibt, dass die Stapel-ID freigegeben ist) werden Trace-Daten für DispatchRaysIndex gesammelt und im Profilbildungspuffer 10601 gespeichert. Folglich verfolgt diese Ausführungsform der Raytracing-Optimierungsplattform 10400 einen einzigen DispatchRaysIndex, der ausreicht, um die gesamte Ausführungskette auf einen bestimmten Strahl abzubilden.

[0772] Unter Umständen, unter denen die Stapel-ID oder DispatchRayIndex noch nicht dem Strahl zugewiesen wurde, kann die Bestimmung von DispatchRayIndex innerhalb des Strahlenerzeugungs(RayGen)-Shaders direkt durchgeführt werden. In einer Ausführungsform stellt der Compiler 10404 zum Beispiel eine Raytracing-Kachelgröße (z. B. Thread-Gruppe) an die Binärinstrumentierungs-Engine 10301 bereit, die dann X_{Index}, Y_{Index} innerhalb des RayGen-Shaders wie folgt berechnet (für eine 16x16-Kachel und SIMD8 = 32 Threads pro Kachel):

$$X_{Offset} = i + 8*(TID\&1),$$

wobei i = 0,1,2...7

$$Y_{Offset} = TID/2$$

$X_{Index} = GroupIDX*16 + X_{Offset}$

$Y_{Index} = GroupIDY*16 + Y_{Offset}$

[0773] Zusammenfassend beinhalten in einer Ausführungsform die Daten, die zur Analyse gesampelt werden, eines oder mehrere von Folgendem: ShaderID und LocalArgPtr, die SIMD-Spur-Dispatch-Maske, den Trace-Zeitstempel, die Stapel-ID und die Sub-Slice-ID, den DispatchRayIndex und die „StackID Free"-Nachricht. Diese gesampelten Werte werden dann wie hierin beschrieben verwendet, um ineffiziente Bedingungen zu identifizieren und zu beheben.

2. Aufruf-Graphen-Rekonstruktion

[0774] Die Aufruf-Graphen-Rekonstruktionslogik 10302 der Raytracing-Optimierungsplattform 10400 verarbeitet die von den instrumentierten Shadern 10402 gesammelten Daten, um die Shader-Aufrufreihenfolge zu rekonstruieren und sie auf Primärstrahlen abzubilden.

[0775] Zur Erläuterung wird das folgende beispielhafte Trace-Profil verwendet. Es sei jedoch darauf hingewiesen, dass die der Erfindung zugrunde liegenden Prinzipien nicht auf diesen spezifischen Datensatz beschränkt sind. Zur Vereinfachung ist die DSS-ID hier die gleiche, es ist aber wichtig, dass die Stapel-ID im Umfang der DSS eindeutig ist, sodass dies im Betrieb berücksichtigt werden sollte:

TABELLE E

Nr.	**Trace-Punkt-Typ**	**DispatchRaysIndex**	**Shader-Hash**	**Shader-Aufzeichnungs-index**	**Stapel-ID**	**DSS-ID**
1	Eintrittspunkt	-	A	SR0	1	0
2	EOT	-	A	SR0	1	0
3	Eintrittspunkt	-	B	SR1	1	0
4	EOT (StackIDFree)	(0, 0)	B	SR1	1	0
5	Eintrittspunkt	-	A	SR2	1	0
6	EOT (StackIDFree)	(1, 1)	A	SR2	1	0

[0776] Unter Bezugnahme auf die Daten in Tabelle E identifiziert die Kombination von Stapel-ID und DSS-ID einen Primärstrahl und alle assoziierten Sekundärstrahlen eindeutig. Weil Stapel-ID jedoch für zukünftige Strahlen freigegeben und wiederverwendet werden kann, müssen alle StackIDFree-Nachrichten in der Analyse berücksichtigt werden. Der DispatchRaysIndex-Wert definiert einen konkreten Primärstrahl, der ausgeführt wurde. Die Shader-Hash- und Shader-Aufzeichnungs-Indexwerte definieren den Shader-Code bzw. die Shader-Aufzeichnung, die für einen bestimmten Strahl ausgeführt werden.

[0777] Damit gesagt kann das obige Trace-Beispiel in die folgenden Aussagen zerlegt werden:

- Es gibt zwei Primärstrahlen, die durch Einträge 1-2 und 5-6 repräsentiert sind.
- Der erste Primärstrahl (1-2) spawnt einen Sekundärstrahl, der durch Einträge 3-4 repräsentiert wird.
- Beide Primärstrahlen führen denselben Shader A aus, jedoch im Umfang unterschiedlicher Shader-Aufzeichnungen - SRC bzw. SR2.
- Der Sekundärstrahl führt den Shader B im Umfang der Shader-Aufzeichnung SR1 aus.

[0778] Dies erzeugt im Gegenzug den in **Fig. 107** gezeigten Ausführungsfluss , der zwei Primärstrahlen (TraceRay (0, 0) und TraceRay (1, 1)) veranschaulicht, die anfänglich durch die Shader-Aufzeichnung 0 10700 und die Shader-Aufzeichnung 2 10702 verarbeitet werden, die mit Shader A assoziiert sind. Die Shader-Aufzeichnung 1 10701, die mit Shader B assoziiert ist, verarbeitet den Sekundärstrahl, der durch den ersten Primärstrahl (TraceRay (0, 0)) erzeugt wird.

3. Abbildung auf Shader-Quellcode

[0779] Wie erwähnt ist dies, da der Compiler für Optimierungszwecke einen Raytracing-Shader in einen anderen Inline bringen kann, keine 1-zu-1-Abbildung zwischen Binär- und Quellcodes. Um die Abbildung durchzuführen, ruft somit die Shader-Quellabbildungslogik von dem Compiler 10404 die Quellzeilenabbildung zwischen Binärcodebereichen auf ursprünglichen Quellcode ab. In einer Implementierung ist der Compiler 10404 dazu ausgelegt, diese Daten während der Shader-Kompilierung zu akkumulieren. Unter Verwendung dieser Informationen bildet die Shader-Quellabbildungslogik 10413 die Tracing-Punkte auf den Quellcode ab. Dies ermöglicht, dass die Blöcke in dem Shader-Aufruf-Graphen entsprechenden Codeabschnitten sowie assoziierten Leistungsfähigkeitsdaten (z. B. Timings und SIMD-Spur-Nutzung) zur weiteren Analyse zugewiesen werden.

4-5. Identifizieren von Leistungsfähigkeitsengpässen und Empfehlen von Optimierungen

[0780] In einer Ausführungsform korreliert die Effizienzanalyselogik 10414 Tracing-Leistungsfähigkeitsdaten, GPU-Metriken und Shader-Ausführungsmuster, um unterschiedliche Klassen von Problemen zu identifizieren, die die Optimiererlogik 10415 verwendet, um Optimierungsempfehlungen zu geben.

[0781] Beispielhaft und nicht einschränkend können die verfolgten Zeitstempel und GPU-Metriken (Leistungsfähigkeitsdaten) angeben, dass manche Teile der Verarbeitungsressourcen (z. B. duale Sub-Slices (DSSs)) häufiger belegt sind als andere. Gleichzeitig kann der Shader-Aufruf-Graph eine signifikante Anzahl von endenden Primärstrahlen zeigen (z. B. über Fehltreffer-Shader-Aufrufe). Die Effizienzanalyselogik 10414 kann dies als ein Ungleichgewicht aufgrund einer frühzeitigen Strahlbeendigung identifizieren, und die Optimiererlogik 10415 kann das Reduzieren der Anzahl von Strahlen, die frühzeitig enden, über Fehltreffer-Shader vorschlagen.

[0782] Als ein anderes Beispiel können die verfolgten Zeitstempel und GPU-Metriken angeben, dass manche Teile der Verarbeitungsressourcen (z. B. duale Sub-Slices (DSSs)) häufiger belegt sind als andere, während der Shader-Aufruf-Graph eine Shader-Aufzeichnung zeigen kann, die einen kritischen Pfad erzeugt (z. B. eine mit der längsten Ausführungszeit oder mit Erzeugung der meisten Strahlen). Die Effizienzanalyselogik 10414 kann dies als ein Ungleichgewicht aufgrund von Shader-Aufzeichnungsabhängigkeiten identifizieren, und die Optimiererlogik 10415 kann das Optimieren des Shaders für diese Shader-Aufzeichnung vorschlagen oder versuchen, ihn aus dem kritischen Pfad zu entfernen, indem sie seine Parent-Shader-Aufrufe analysiert.

[0783] Alternativ dazu können die Leistungsfähigkeitsdaten eine niedrige SIMD-Belegung für eine spezifische Shader-Aufzeichnung angeben, während der Shader-Aufruf-Graph zeigt, dass einzelne Shader-Aufzeichnungsaufrufe auf einem DSS über Pfade mit einer erheblich unterschiedlichen Länge stammten (z. B. mit einer Differenz größer als eine Schwelle). Die Effizienzanalyselogik 10414 kann dieses Problem als Shader-Aufzeichnungsaufrufe identifizieren, die zu sehr über die Zeit verteilt sind, und die Optimiererlogik 10415 kann das Schneiden längerer Aufrufpfade vorschlagen, um die Aufruflokalität zeitlich zu erhöhen (z. B. durch Entfernen von Sekundärstrahlen).

[0784] Gleichermaßen können die Leistungsfähigkeitsdaten eine niedrige SIMD-Belegung für eine spezifische Shader-Aufzeichnung angeben, während der Shader-Aufruf-Graph zeigt, dass dieselbe Primär-Shader-Aufzeichnung in Strahlen aufgerufen wird, die über die dualen Sub-Slices verteilt sind. Die Effizienzanalyselogik 10414 kann dieses Problem als Shader-Aufzeichnungsaufrufe identifizieren, die über duale Sub-Slices verteilt sind, und die Optimiererlogik 10415 kann das Ändern des Strahlwurf-Musters vorschlagen, um die Lokalität relativ zu der Dispatch-Kachel (z. B. eine 32x32-Kachel) zu erhöhen.

[0785] Natürlich handelt es sich bei dem Obenstehenden lediglich um Beispiele. Die zugrundeliegenden Prinzipien der Erfindung können mit einer Vielzahl unterschiedlicher Arten von Leistungsfähigkeitsdaten und Shader-Aufruf-Graphen implementiert werden, was zu unterschiedlichen Analyseergebnissen und Optimierungen führt.

BEISPIELE

[0786] Das Folgende sind beispielhafte Implementierungen unterschiedlicher Ausführungsformen der Erfindung.

[0787] Beispiel 1. Ein Verfahren, das umfasst: Durchführen einer binären Instrumentierung von Raytracing-Shadern; Verfolgen der Ausführung der Raytracing-Shader, um Ausführungsmetriken zu erzeugen; Konstruieren eines Shader-Aufruf-Graphen basierend auf den Ausführungsmetriken; Abbilden des Shader-Aufruf-Graphen auf Shader-Quellcode, um eine Quellcodeabbildung zu erzeugen; Bestimmen von Ineffizienzen bei der Raytracing-Shader-Ausführung basierend auf der Quellcodeabbildung; und Identifizieren von Optimierungshandlungen basierend auf den Ineffizienzen.

[0788] Beispiel 2. Das Verfahren des Beispiels 1, wobei das Konstruieren des Shader-Aufruf-Graphen ferner umfasst: Assoziieren von Ausführungsmetriken mit Shader-Aufzeichnungen von Raytracing-Shadern.

[0789] Beispiel 3. Das Verfahren des Beispiels 2, wobei das Konstruieren des Shader-Aufruf-Graphen ferner umfasst: Abbilden der Shader-Aufzeichnungen auf Primärstrahlen.

[0790] Beispiel 4. Das Verfahren des Beispiels 1, das ferner umfasst: Umwandeln der Ausführungsmetriken in Leistungsfähigkeitsdaten; und Konstruieren des Shader-Aufruf-Graphen unter Verwendung der Leistungsfähigkeitsdaten.

[0791] Beispiel 5. Das Verfahren des Beispiels 1, wobei das Abbilden des Shader-Aufruf-Graphen auf den Shader-Quellcode ferner umfasst: Bestimmen einer Quellzeilenabbildung zwischen Binärcodebereichen auf den Quellcode; und Abbilden von Tracing-Punkten auf den Quellcode unter Verwendung der Quellzeilenabbildung.

[0792] Beispiel 6. Das Verfahren des Beispiels 1, wobei das Bestimmen von Ineffizienzen in der Raytracing-Shader-Ausführung Identifizieren einer ersten Verarbeitungsressourcenzuweisung umfasst, die stärker als eine zweite Verarbeitungsressourcenzuweisung belastet ist.

[0793] Beispiel 7. Das Verfahren des Beispiels 6, wobei die erste Verarbeitungsressourcenzuweisung ein erstes duales Sub-Slice (DSS) umfasst und die zweite Verarbeitungsressourcenzuweisung ein zweites DSS umfasst.

[0794] Beispiel 8. Das Verfahren des Beispiels 7, wobei als Reaktion darauf, dass der Shader-Aufruf-Graph eine Anzahl von endenden Primärstrahlen über einer Schwelle angibt, die bestimmten Ineffizienzen ein Ungleichgewicht aufgrund frühzeitiger Strahlbeendigung beinhalten und die Optimierungshandlungen Reduzieren der Anzahl von Strahlen, die frühzeitig enden, über Fehltreffer-Shader beinhalten.

[0795] Beispiel 9. Das Verfahren des Beispiels 1, wobei das Bestimmen von Ineffizienzen in der Raytracing-Shader-Ausführung eine niedrige SIMD-Belegung für eine spezifische Shader-Aufzeichnung umfasst und die Optimierungshandlungen Schneiden längerer Aufrufpfade beinhalten, um die Aufruflokalität zeitlich zu erhöhen, oder ein Ändern eines Strahlwurfmusters beinhalten, um die Lokalität relativ zu einer Dispatch-Kachel zu erhöhen.

[0796] Beispiel 10. Das Verfahren des Beispiels 9, wobei eine der Optimierungshandlungen basierend auf dem Analysieren des Shader-Aufruf-Graphen ausgewählt wird.

[0797] Beispiel 11. Eine Einrichtung, die umfasst: Eine Binärinstrumentierungs-Engine zum Durchführen einer binären Instrumentierung von Raytracing-Shadern und zum Verfolgen der Ausführung der Raytracing-Shader, um Ausführungsmetriken zu erzeugen; Aufruf-Graphen-Konstruktionslogik zum Konstruieren eines Shader-Aufruf-Graphen basierend auf den Ausführungsmetriken; Shader-Quellabbildungslogik zum Abbilden des Shader-Aufruf-Graphen auf Shader-Quellcode, um eine Quellcodeabbildung zu erzeugen; Effizienzanalyselogik zum Bestimmen von Ineffizienzen bei der Raytracing-Shader-Ausführung basierend auf der Quellcodeabbildung; und Optimierungslogik zum Identifizieren von Optimierungshandlungen basierend auf den Ineffizienzen.

[0798] Beispiel 12. Die Einrichtung des Beispiels 11, wobei zum Konstruieren des Shader-Aufruf-Graphen die Aufruf-Graphen-Konstruktionslogik ausgelegt ist zum Assoziieren von Ausführungsmetriken mit Shader-Aufzeichnungen von Raytracing-Shadern.

[0799] Beispiel 13. Die Einrichtung des Beispiels 12, wobei zum Konstruieren des Shader-Aufruf-Graphen die Aufruf-Graphen-Konstruktionslogik ausgelegt ist zum Abbilden der Shader-Aufzeichnungen auf Primärstrahlen.

[0800] Beispiel 14. Die Einrichtung des Beispiels 11, wobei die Binärinstrumentierungs-Engine ausgelegt ist zum Umwandeln der Ausführungsmetriken in Leistungsfähigkeitsdaten und die Shader-Aufruf-Graphen-Konstruktionslogik ausgelegt ist zum Konstruieren des Aufruf-Graphen unter Verwendung der Leistungsfähigkeitsdaten.

[0801] Beispiel 15. Die Einrichtung des Beispiels 11, wobei zum Abbilden des Shader-Aufruf-Graphen auf Shader-Quellcode die Shader-Quellcodeabbildungslogik ausgelegt ist zum: Bestimmen einer Quellzeilenabbildung zwischen Binärcodebereichen auf den Quellcode; und Abbilden von Tracing-Punkten auf den Quellcode unter Verwendung der Quellzeilenabbildung.

[0802] Beispiel 16. Die Einrichtung des Beispiels 11, wobei zum Bestimmen von Ineffizienzen bei der Raytracing-Shader-Ausführung die Effizienzanalyselogik ausgelegt ist zum Identifizieren einer ersten Verarbeitungsressourcenzuweisung, die stärker als eine zweite Verarbeitungsressourcenzuweisung belastet ist.

[0803] Beispiel 17. Die Einrichtung des Beispiels 16, wobei die erste Verarbeitungsressourcenzuweisung ein erstes duales Sub-Slice (DSS) umfasst und die zweite Verarbeitungsressourcenzuweisung ein zweites DSS umfasst.

[0804] Beispiel 18. Die Einrichtung des Beispiels 17, wobei als Reaktion darauf, dass der Shader-Aufruf-Graph eine Anzahl von endenden Primärstrahlen über einer Schwelle angibt, die Effizienzanalyselogik ausgelegt ist zum Bestimmen eines Ungleichgewichts aufgrund einer frühzeitigen Strahlbeendigung und die Optimierungslogik ausgelegt ist zum Angeben einer Reduzierung der Anzahl von Strahlen, die frühzeitig enden, über Fehltreffer-Shader.

[0805] Beispiel 19. Die Einrichtung des Beispiels 11, wobei die Effizienzanalyselogik ausgelegt ist zum Bestimmen einer niedrigen SIMD-Belegung für eine spezifische Shader-Aufzeichnung und die Optimierungslogik ausgelegt ist zum Schneiden längerer Aufrufpfade, um die Aufruflokalität zeitlich zu erhöhen, oder Ändern eines Strahlwurfmusters, um die Lokalität relativ zu einer Dispatch-Kachel zu erhöhen.

[0806] Beispiel 20. Die Einrichtung des Beispiels 19, wobei die Optimierungslogik ausgelegt ist entweder zum Schneiden längerer Aufrufpfade, um die Aufruflokalität zeitlich zu erhöhen, oder zum Ändern eines Strahlwurfmusters, um die Lokalität relativ zu einer Dispatch-Kachel zu erhöhen, basierend auf dem Analysieren des Shader-Aufruf-Graphen.

[0807] Beispiel 21. Ein maschinenlesbares Medium mit darauf gespeichertem Programmcode, der bei Ausführung durch eine Maschine die Maschine veranlasst, die folgenden Operationen durchzuführen: Durchführen einer binären Instrumentierung von Raytracing-Shadern; Verfolgen der Ausführung der Raytracing-Shader, um Ausführungsmetriken zu erzeugen; Konstruieren eines Shader-Aufruf-Graphen basierend auf den Ausführungsmetriken; Abbilden des Shader-Aufruf-Graphen auf Shader-Quellcode, um eine Quellcodeabbildung zu erzeugen; Bestimmen von Ineffizienzen bei der Raytracing-Shader-Ausführung basierend auf der Quellcodeabbildung; und Identifizieren von Optimierungshandlungen basierend auf den Ineffizienzen.

[0808] Beispiel 22. Das maschinenlesbare Medium des Beispiels 21, wobei das Konstruieren des Shader-Aufruf-Graphen ferner umfasst: Assoziieren von Ausführungsmetriken mit Shader-Aufzeichnungen von Raytracing-Shadern.

[0809] Beispiel 23. Das maschinenlesbare Medium des Beispiels 22, wobei das Konstruieren des Shader-Aufruf-Graphen ferner umfasst: Abbilden der Shader-Aufzeichnungen auf Primärstrahlen.

[0810] Beispiel 24. Das maschinenlesbare Medium des Beispiels 21, ferner umfassend Programmcode zum Veranlassen, dass die Maschine die folgenden Operationen durchführt: Umwandeln der Ausführungsmetriken in Leistungsfähigkeitsdaten; und Konstruieren des Shader-Aufruf-Graphen unter Verwendung der Leistungsfähigkeitsdaten.

[0811] Beispiel 25. Das maschinenlesbare Medium des Beispiels 21, wobei das Abbilden des Shader-Aufruf-Graphen auf den Shader-Quellcode ferner umfasst: Bestimmen einer Quellzeilenabbildung zwischen Binärcodebereichen auf den Quellcode; und Abbilden von Tracing-Punkten auf den Quellcode unter Verwendung der Quellzeilenabbildung.

[0812] Beispiel 26. Das maschinenlesbare Medium des Beispiels 21, wobei das Bestimmen von Ineffizienzen in der Raytracing-Shader-Ausführung Identifizieren einer ersten Verarbeitungsressourcenzuweisung umfasst, die stärker als eine zweite Verarbeitungsressourcenzuweisung belastet ist.

[0813] Beispiel 27. Das maschinenlesbare Medium des Beispiels 26, wobei die erste Verarbeitungsressourcenzuweisung ein erstes duales Sub-Slice (DSS) umfasst und die zweite Verarbeitungsressourcenzuweisung ein zweites DSS umfasst.

[0814] Beispiel 28. Das maschinenlesbare Medium des Beispiels 27, wobei als Reaktion darauf, dass der Shader-Aufruf-Graph eine Anzahl von endenden Primärstrahlen über einer Schwelle angibt, die bestimmten Ineffizienzen ein Ungleichgewicht aufgrund frühzeitiger Strahlbeendigung beinhalten und die Optimierungshandlungen Reduzieren der Anzahl von Strahlen, die frühzeitig enden, über Fehltreffer-Shader beinhalten.

[0815] Beispiel 29. Das maschinenlesbare Medium des Beispiels 21, wobei das Bestimmen von Ineffizienzen in der Raytracing-Shader-Ausführung eine niedrige SIMD-Belegung für eine spezifische Shader-Aufzeichnung umfasst und die Optimierungshandlungen Schneiden längerer Aufrufpfade beinhalten, um die Aufruflokalität zeitlich zu erhöhen, oder ein Ändern eines Strahlwurfmusters beinhalten, um die Lokalität relativ zu einer Dispatch-Kachel zu erhöhen.

[0816] Beispiel 30. Das maschinenlesbare Medium des Beispiels 29, wobei eine der Optimierungshandlungen basierend auf dem Analysieren des Shader-Aufruf-Graphen ausgewählt wird.

[0817] Die Ausführungsformen der Erfindung ermöglichen eine Profilbildung verschiedener Shader-Ausführungsineffizienzen, die ansonsten nicht mit existierenden Profilbildungsverfahren aufgrund eindeutiger Raytracing-Shader-Ausführungsmodelle zu detektieren sind.

[0818] Ausführungsformen der Erfindung können verschiedene Schritte beinhalten, die oben beschrieben wurden. Die Schritte können als maschinenausführbare Anweisungen verkörpert werden, die verwendet werden können, um einen Mehrzweck- oder Spezialprozessor dazu zu veranlassen, die Schritte durchzuführen. Alternativ können diese Schritte durch spezifische Hardware-Komponenten, die festverdrahtete Logik zum Durchführen der Schritte enthalten, oder durch eine beliebige Kombination von programmierten Computerkomponenten und angepassten Hardware-Komponenten durchgeführt werden.

[0819] Wie hierin beschrieben, können sich Anweisungen auf spezifische Konfigurationen von Hardware beziehen, wie etwa anwendungsspezifische integrierte Schaltungen (ASICs), die dazu ausgelegt sind, bestimmte Operationen durchzuführen, oder die eine vorbestimmte Funktionalität aufweisen, oder Software-Anweisungen, die in einem Speicher gespeichert sind, der in einem nichtflüchtigen computerlesbaren Medium umgesetzt ist. Somit können die in den Figuren gezeigten Techniken durch Verwenden von Code und Daten implementiert werden, die auf einer oder mehreren elektronischen Vorrichtungen (z. B. einer Endstation, einem Netzwerkelement usw.) gespeichert sind und ausgeführt werden. Solche elektronischen Vorrichtungen speichern und kommunizieren (intern und/oder mit anderen elektronischen Vorrichtungen über ein Netzwerk) Code und Daten durch Verwenden von computermaschinenlesbaren Medien, wie etwa nichtflüchtigen computermaschinenlesbaren Speicherungsmedien (z. B. Magnetplatten; optischen Platten; Direktzugriffsspeicher; Nurlesespeicher; Flash-Speichervorrichtungen; Phasenwechselspeicher) und flüchtigen computermaschinenlesbaren Kommunikationsmedien (z. B. elektrischen, optischen, akustischen oder einer anderen Form propagierter Signale - wie etwa Trägerwellen, Infrarotsignalen, digitalen Signalen usw.).

[0820] Zusätzlich beinhalten solche elektronischen Vorrichtungen typischerweise einen Satz von einem oder mehreren Prozessoren, die mit einer oder mehreren anderen Komponenten, wie etwa einer oder mehreren Speicherungsvorrichtungen (nichtflüchtigen maschinenlesbaren Speicherungsmedien), Benutzereingabe-/-ausgabevorrichtungen (z. B. einer Tastatur, einem Touchscreen und/oder einer Anzeige) und Netzwerkverbindungen, gekoppelt sind. Die Kopplung des Satzes von Prozessoren und anderen Komponenten erfolgt typischerweise durch einen oder mehrere Busse und Brücken (auch als Bussteuerungen bezeichnet). Die Speicherungsvorrichtung und die Signale, die den Netzwerkverkehr führen, repräsentieren ein oder mehrere maschinenlesbare Speicherungsmedien bzw. maschinenlesbare Kommunikationsmedien. Somit speichert die Speicherungsvorrichtung einer gegebenen elektronischen Vorrichtung typischerweise Code und/oder Daten zur Ausführung auf dem Satz von einem oder mehreren Prozessoren dieser elektronischen Vorrichtung. Natürlich können ein oder mehrere Teile einer Ausführungsform der Erfindung durch Verwenden unterschiedlicher Kombinationen von Software, Firmware und/oder Hardware implementiert werden. Über diese gesamte ausführliche Beschreibung hinweg wurden zu Zwecken der Erläuterung zahlreiche spezifische Ein-

zelheiten dargelegt, um ein gründliches Verständnis der vorliegenden Erfindung bereitzustellen. Für den Fachmann ist jedoch offensichtlich, dass die Erfindung ohne manche dieser speziellen Details umgesetzt werden kann. In bestimmten Fällen wurden wohlbekannte Strukturen und Funktionen nicht in aufwändigem Detail beschrieben, um ein Verschleiern des Gegenstands der vorliegenden Erfindung zu vermeiden. Dementsprechend sollten der Schutzumfang und die Idee der Erfindung hinsichtlich der folgenden Ansprüche beurteilt werden.

Patentansprüche

1. Verfahren, umfassend:
Durchführen einer binären Instrumentierung von Raytracing-Shadern;
Verfolgen der Ausführung der Raytracing-Shader, um Ausführungsmetriken zu erzeugen;
Konstruieren eines Shader-Aufruf-Graphen basierend auf den Ausführungsmetriken;
Abbilden des Shader-Aufruf-Graphen auf Shader-Quellcode, um eine Quellcodeabbildung zu erzeugen;
Bestimmen von Ineffizienzen bei der Raytracing-Shader-Ausführung basierend auf der Quellcodeabbildung; und
Identifizieren von Optimierungshandlungen basierend auf den Ineffizienzen.

2. Verfahren nach Anspruch 1, wobei das Konstruieren des Shader-Aufruf-Graphen ferner umfasst:
Assoziieren von Ausführungsmetriken mit Shader-Aufzeichnungen von Raytracing-Shadern.

3. Verfahren nach Anspruch 2, wobei das Konstruieren des Shader-Aufruf-Graphen ferner umfasst:
Abbilden der Shader-Aufzeichnungen auf Primärstrahlen.

4. Verfahren nach einem der Ansprüche 1 bis 3, ferner umfassend:
Umwandeln der Ausführungsmetriken in Leistungsfähigkeitsdaten; und
Konstruieren des Shader-Aufruf-Graphen unter Verwendung der Leistungsfähigkeitsdaten.

5. Verfahren nach einem der Ansprüche 1 bis 4, wobei das Abbilden des Shader-Aufruf-Graphen auf den Shader-Quellcode ferner umfasst:
Bestimmen einer Quellzeilenabbildung zwischen Binärcodebereichen auf den Quellcode; und
Abbilden von Tracing-Punkten auf den Quellcode unter Verwendung der Quellzeilenabbildung.

6. Verfahren nach einem der Ansprüche 1 bis 5, wobei das Bestimmen von Ineffizienzen in der Raytracing-Shader-Ausführung Identifizieren einer ersten Verarbeitungsressourcenzuweisung umfasst, die stärker als eine zweite Verarbeitungsressourcenzuweisung belastet ist.

7. Verfahren nach Anspruch 6, wobei die erste Verarbeitungsressourcenzuweisung ein erstes duales Sub-Slice (DSS) umfasst und die zweite Verarbeitungsressourcenzuweisung ein zweites DSS umfasst.

8. Verfahren nach Anspruch 7, wobei als Reaktion darauf, dass der Shader-Aufruf-Graph eine Anzahl von endenden Primärstrahlen über einer Schwelle angibt, die bestimmten Ineffizienzen ein Ungleichgewicht aufgrund frühzeitiger Strahlbeendigung beinhalten und die Optimierungshandlungen Reduzieren der Anzahl von Strahlen, die frühzeitig enden, über Fehltreffer-Shader beinhalten.

9. Verfahren nach einem der Ansprüche 1 bis 8, wobei das Bestimmen von Ineffizienzen in der Raytracing-Shader-Ausführung eine niedrige SIMD-Belegung für eine spezifische Shader-Aufzeichnung umfasst und die Optimierungshandlungen Schneiden längerer Aufrufpfade beinhalten, um die Aufruflokalität zeitlich zu erhöhen, oder Ändern eines Strahlwurfmusters beinhalten, um die Lokalität relativ zu einer Dispatch-Kachel zu erhöhen.

10. Verfahren nach Anspruch 9, wobei eine der Optimierungshandlungen basierend auf dem Analysieren des Shader-Aufruf-Graphen ausgewählt wird.

11. Einrichtung, umfassend:
eine Binärinstrumentierungs-Engine, um eine binäre Instrumentierung von Raytracing-Shadern durchzuführen und die Ausführung der Raytracing-Shader zu verfolgen, um Ausführungsmetriken zu erzeugen;
Aufruf-Graphen-Konstruktionslogik zum Konstruieren eines Shader-Aufruf-Graphen basierend auf den Ausführungsmetriken;
Shader-Quellabbildungslogik zum Abbilden des Shader-Aufruf-Graphen auf einen Shader-Quellcode, um

eine Quellcodeabbildung zu erzeugen;
Effizienzanalyselogik zum Bestimmen von Ineffizienzen bei der Raytracing-Shader-Ausführung basierend auf der Quellcodeabbildung; und
Optimierungslogik zum Identifizieren von Optimierungshandlungen basierend auf den Ineffizienzen.

12. Einrichtung nach Anspruch 11, wobei zum Konstruieren des Shader-Aufruf-Graphen die Aufruf-Graphen-Konstruktionslogik ausgelegt ist zum Assoziieren von Ausführungsmetriken mit Shader-Aufzeichnungen von Raytracing-Shadern.

13. Einrichtung nach Anspruch 12, wobei zum Konstruieren des Shader-Aufruf-Graphen die Aufruf-Graphen-Konstruktionslogik ausgelegt ist zum Abbilden der Shader-Aufzeichnungen auf Primärstrahlen.

14. Einrichtung nach einem der Ansprüche 11 bis 13, wobei die Binärinstrumentierungs-Engine ausgelegt ist zum Umwandeln der Ausführungsmetriken in Leistungsfähigkeitsdaten und die Shader-Aufruf-Graphen-Konstruktionslogik ausgelegt ist zum Konstruieren des Aufruf-Graphen unter Verwendung der Leistungsfähigkeitsdaten.

15. Einrichtung nach einem der Ansprüche 11 bis 14, wobei zum Abbilden des Shader-Aufruf-Graphen auf Shader-Quellcode die Shader-Quellcodeabbildungslogik ausgelegt ist zum:
Bestimmen einer Quellzeilenabbildung zwischen Binärcodebereichen auf den Quellcode; und
Abbilden von Tracing-Punkten auf den Quellcode unter Verwendung der Quellzeilenabbildung.

Es folgen 131 Seiten Zeichnungen

Anhängende Zeichnungen

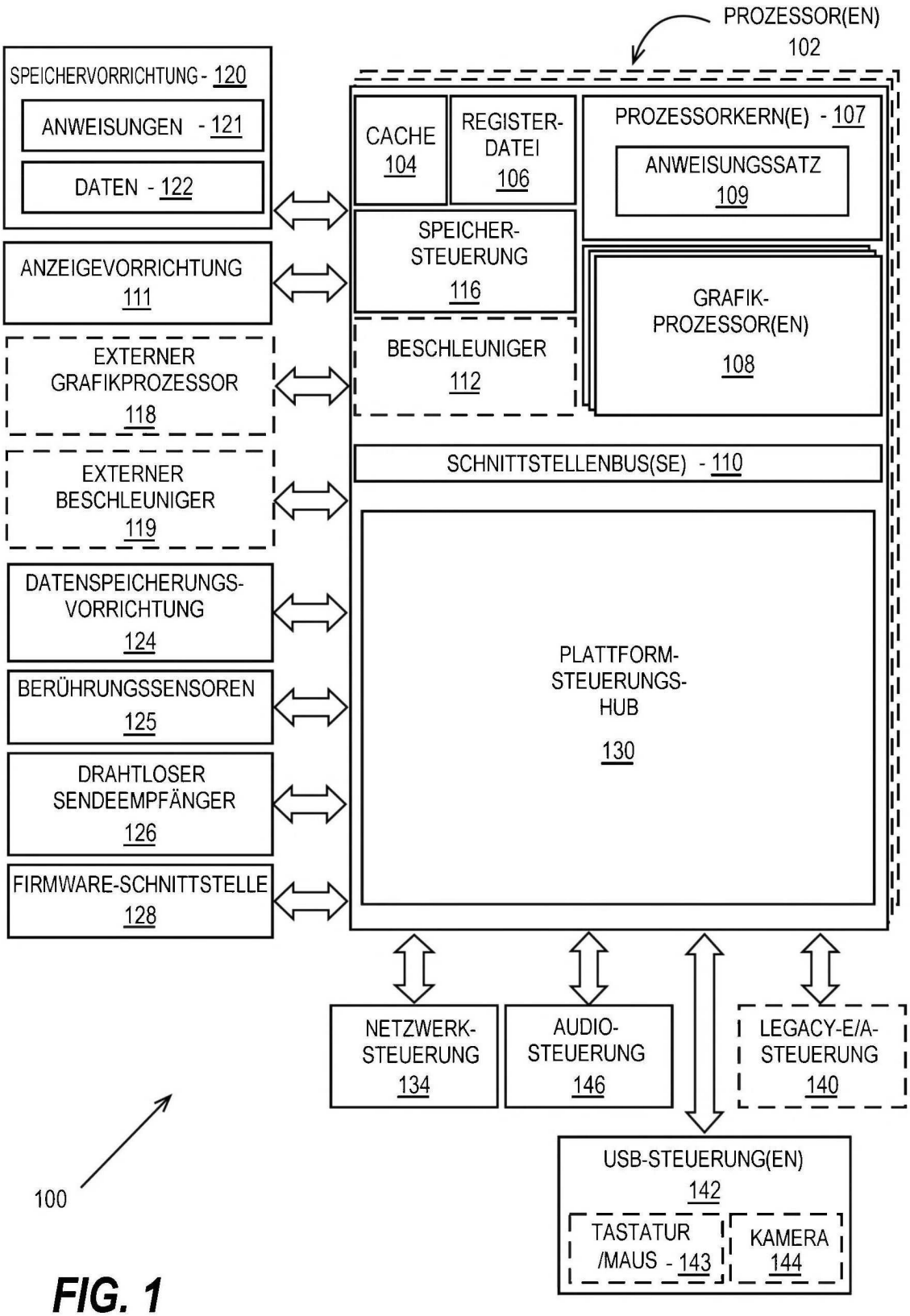


FIG. 1

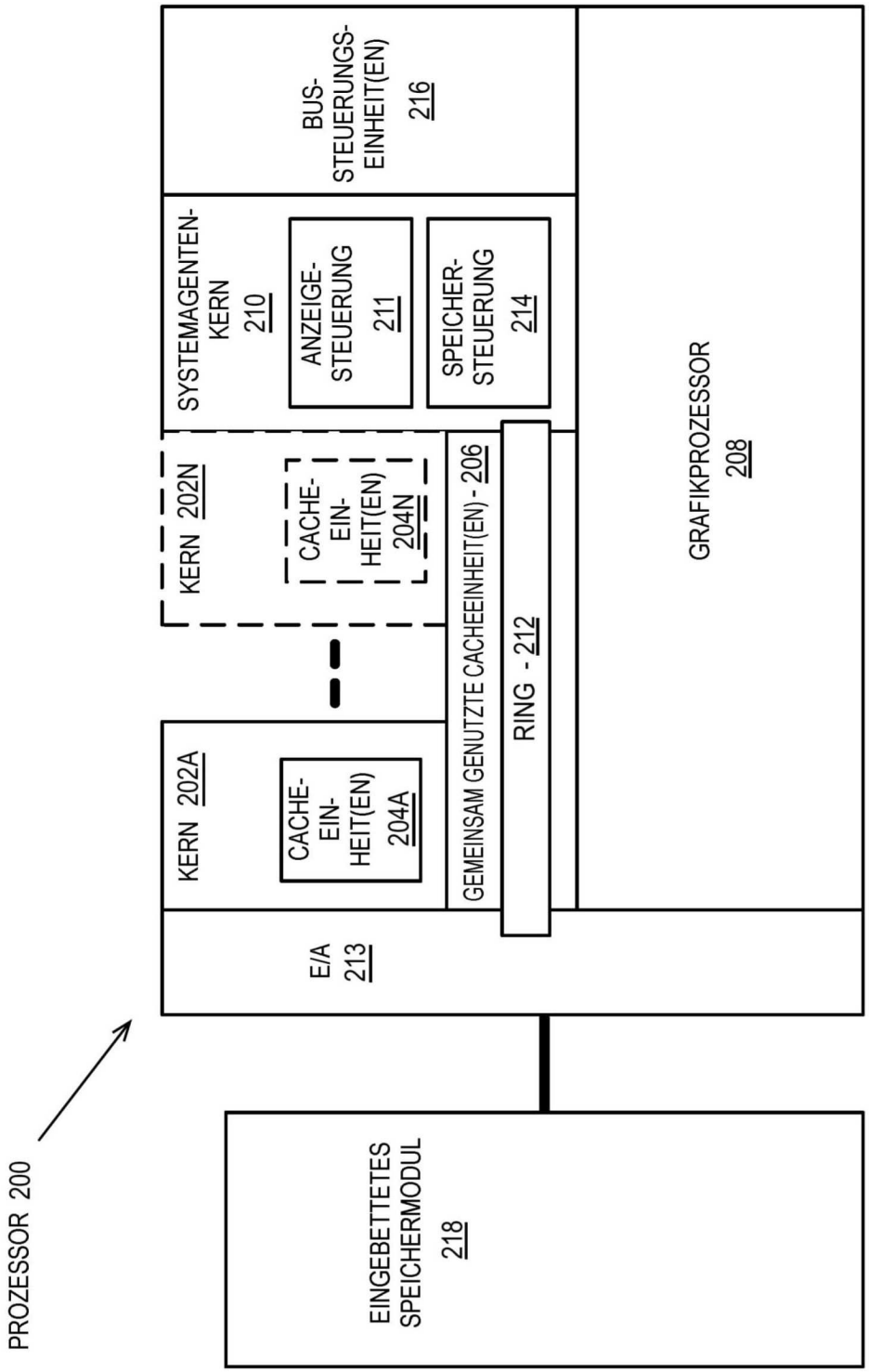

PROZESSOR 200
EINGEBETTETES SPEICHERMODUL 218
E/A 213
KERN 202A
CACHE-EIN-HEIT(EN) 204A
KERN 202N
CACHE-EIN-HEIT(EN) 204N
GEMEINSAM GENUTZTE CACHEEINHEIT(EN) - 206
RING - 212
GRAFIKPROZESSOR 208
SYSTEMAGENTEN-KERN 210
ANZEIGE-STEUERUNG 211
SPEICHER-STEUERUNG 214
BUS-STEUERUNGS-EINHEIT(EN) 216

FIG. 2A

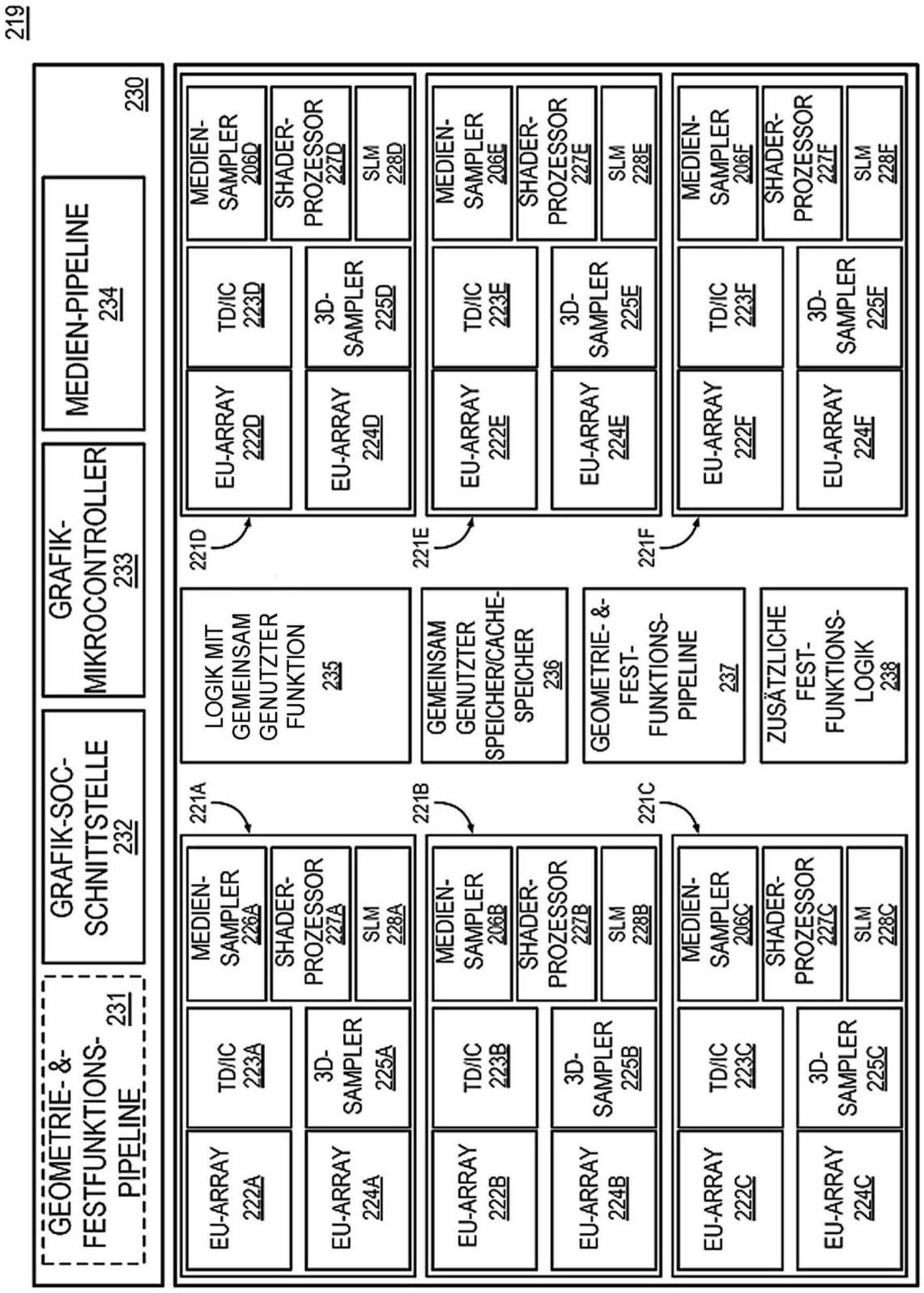

219
GEOMETRIE- & FESTFUNKTIONS-PIPELINE 231
GRAFIK-SOC-SCHNITTSTELLE 232
GRAFIK-MIKROCONTROLLER 233
MEDIEN-PIPELINE 234
230
LOGIK MIT GEMEINSAM GENUTZTER FUNKTION 235
GEMEINSAM GENUTZTER SPEICHER/CACHE-SPEICHER 236
GEOMETRIE- & FEST-FUNKTIONS-PIPELINE 237
ZUSÄTZLICHE FEST-FUNKTIONS-LOGIK 238
221A
221B
221C
221D
221E
221F
EU-ARRAY 222A
TD/IC 223A
3D-SAMPLER 225A
EU-ARRAY 224A
MEDIEN-SAMPLER 206A
SHADER-PROZESSOR 227A
SLM 228A
EU-ARRAY 222B
TD/IC 223B
3D-SAMPLER 225B
EU-ARRAY 224B
MEDIEN-SAMPLER 206B
SHADER-PROZESSOR 227B
SLM 228B
EU-ARRAY 222C
TD/IC 223C
3D-SAMPLER 225C
EU-ARRAY 224C
MEDIEN-SAMPLER 206C
SHADER-PROZESSOR 227C
SLM 228C
EU-ARRAY 222D
TD/IC 223D
3D-SAMPLER 225D
EU-ARRAY 224D
MEDIEN-SAMPLER 206D
SHADER-PROZESSOR 227D
SLM 228D
EU-ARRAY 222E
TD/IC 223E
3D-SAMPLER 225E
EU-ARRAY 224E
MEDIEN-SAMPLER 206E
SHADER-PROZESSOR 227E
SLM 228E
EU-ARRAY 222F
TD/IC 223F
3D-SAMPLER 225F
EU-ARRAY 224F
MEDIEN-SAMPLER 206F
SHADER-PROZESSOR 227F
SLM 228F

FIG. 2B

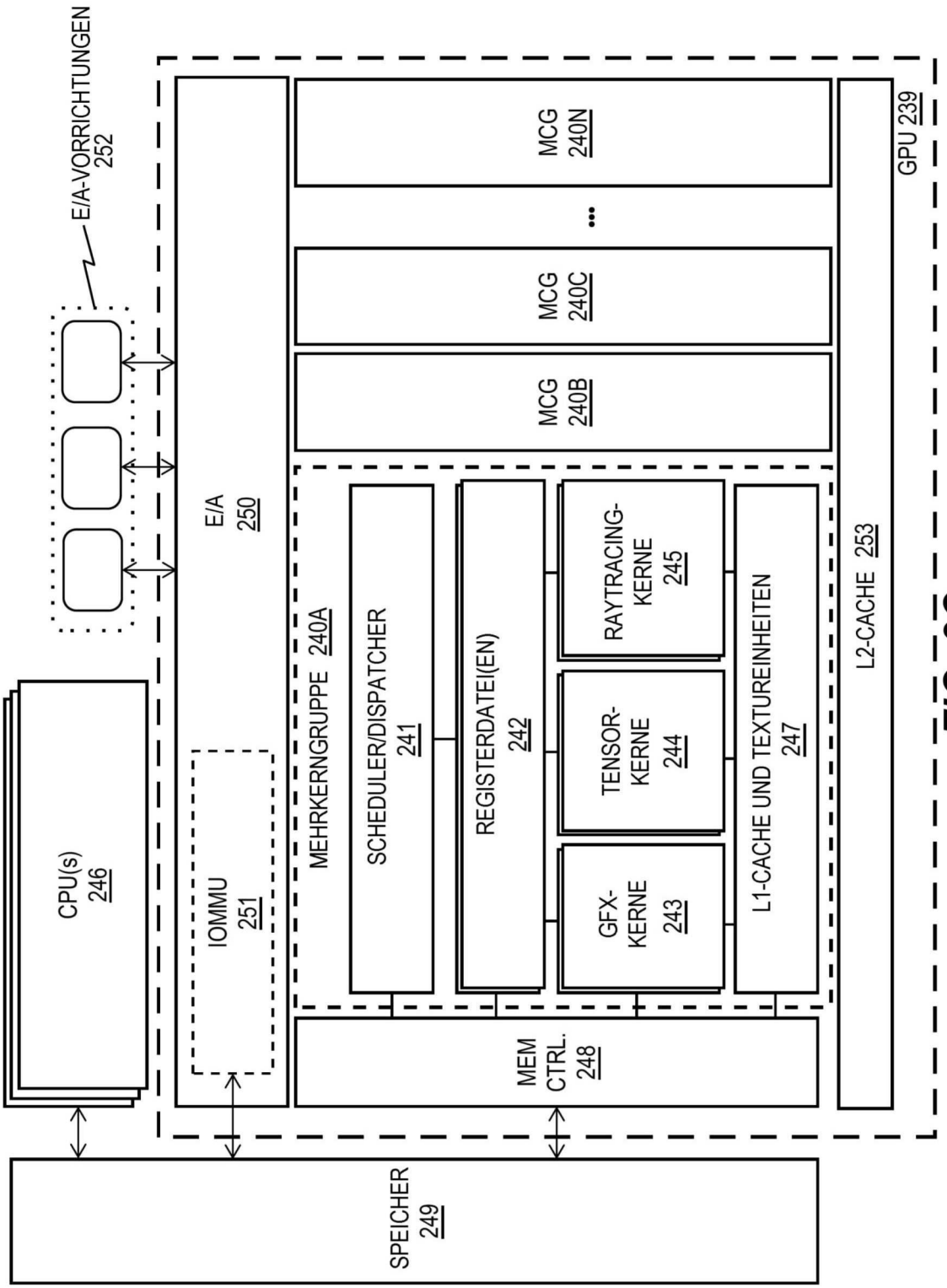

E/A-VORRICHTUNGEN 252
GPU 239
E/A 250
MCG 240N
MCG 240C
MCG 240B
MEHRKERNGRUPPE 240A
SCHEDULER/DISPATCHER 241
REGISTERDATEI(EN) 242
RAYTRACING-KERNE 245
TENSOR-KERNE 244
GFX-KERNE 243
L1-CACHE UND TEXTUREINHEITEN 247
L2-CACHE 253
MEM CTRL. 248
IOMMU 251
CPU(s) 246
SPEICHER 249

FIG. 2C

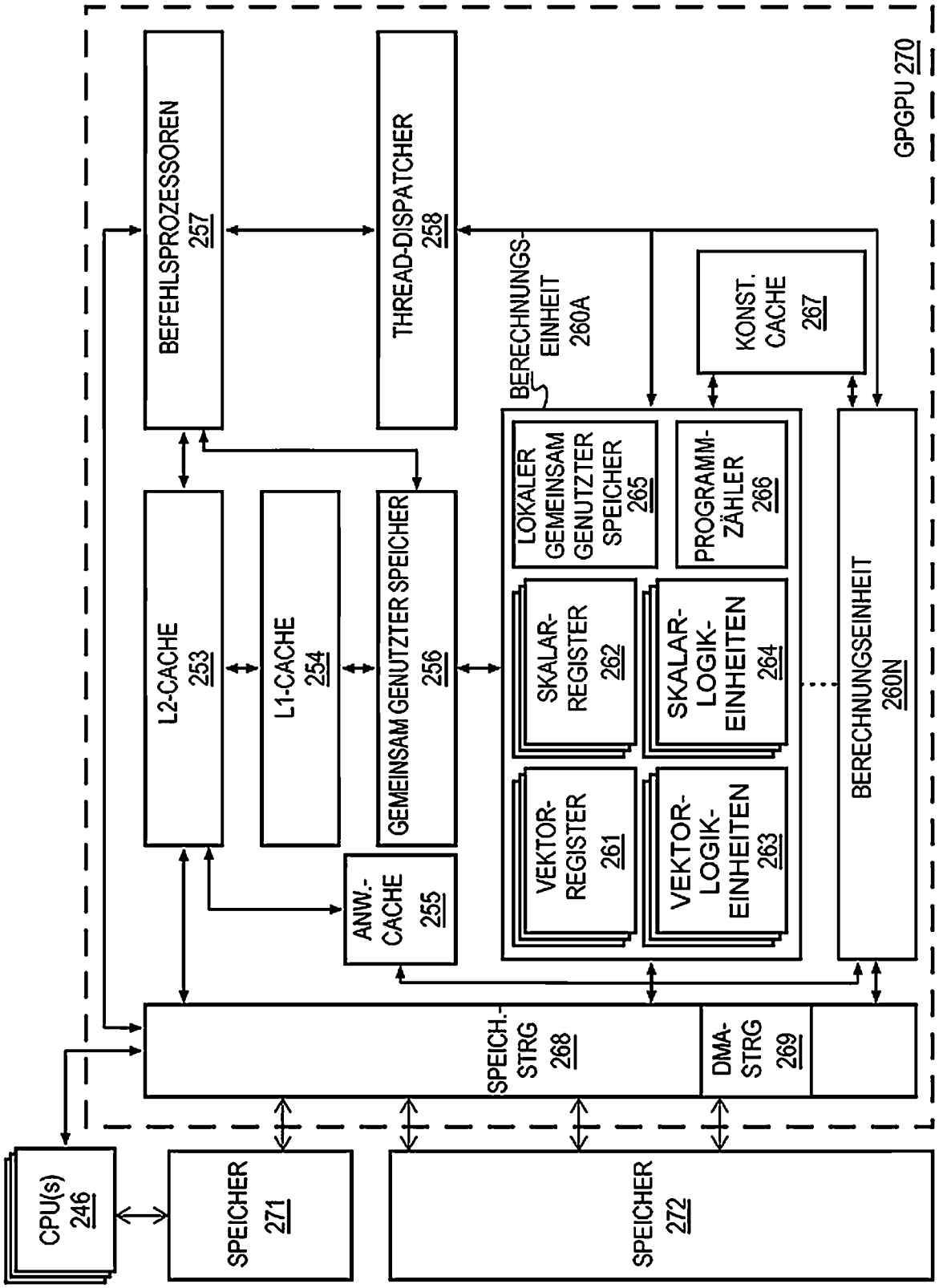

GPGPU 270
BEFEHLSPROZESSOREN 257
THREAD-DISPATCHER 258
BERECHNUNGS-EINHEIT 260A
KONST. CACHE 267
LOKALER GEMEINSAM GENUTZTER SPEICHER 265
PROGRAMM-ZÄHLER 266
L2-CACHE 253
L1-CACHE 254
GEMEINSAM GENUTZTER SPEICHER 256
SKALAR-REGISTER 262
SKALAR-LOGIK-EINHEITEN 264
VEKTOR-REGISTER 261
VEKTOR-LOGIK-EINHEITEN 263
BERECHNUNGSEINHEIT 260N
ANW.-CACHE 255
SPEICH.-STRG 268
DMA-STRG 269
CPU(s) 246
SPEICHER 271
SPEICHER 272

FIG. 2D

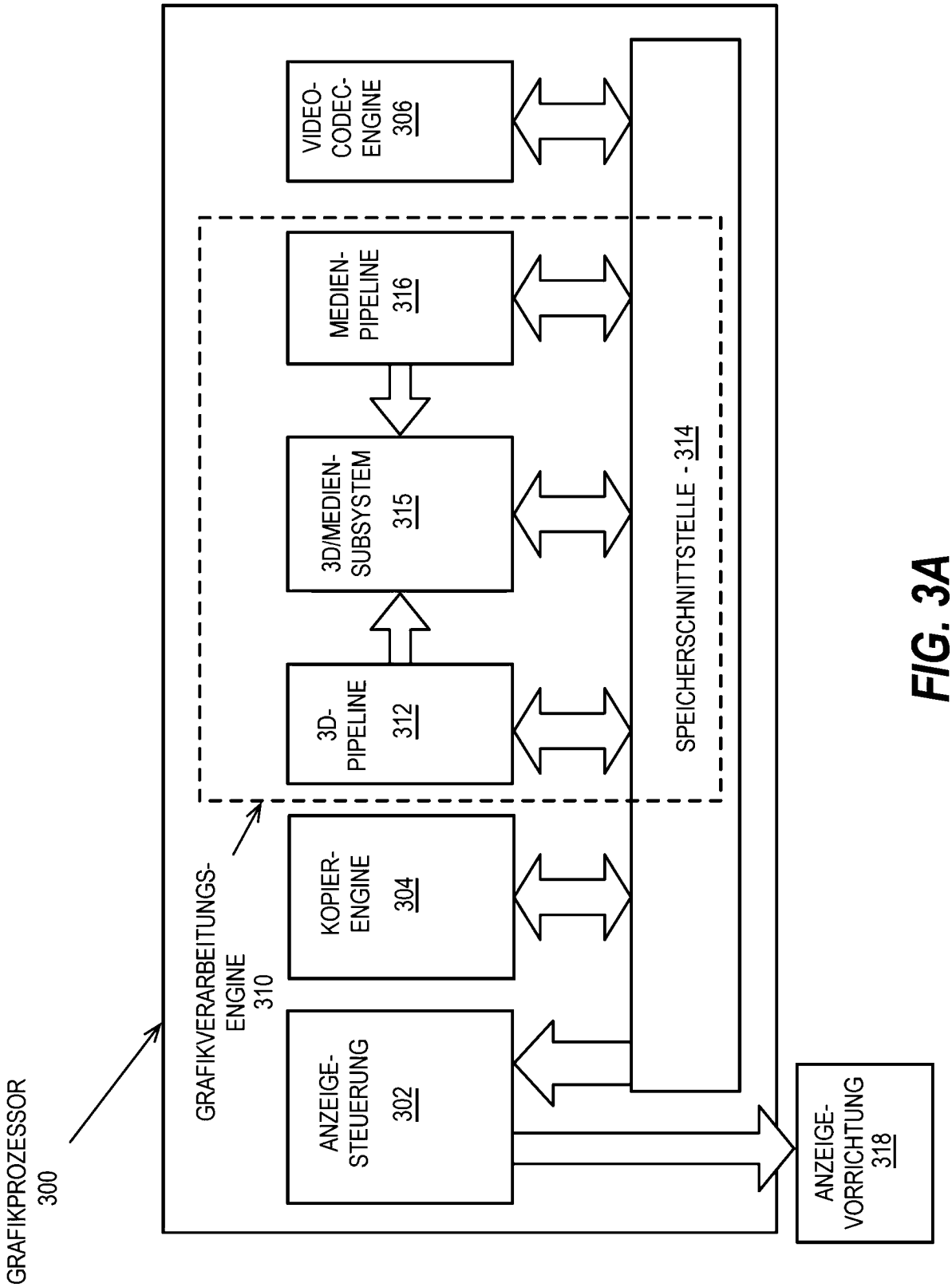

GRAFIKPROZESSOR
300
GRAFIKVERARBEITUNGS-
ENGINE
310
ANZEIGE-
STEUERUNG
302
KOPIER-
ENGINE
304
3D-
PIPELINE
312
3D/MEDIEN-
SUBSYSTEM
315
MEDIEN-
PIPELINE
316
VIDEO-
CODEC-
ENGINE
306
SPEICHERSCHNITTSTELLE - 314
ANZEIGE-
VORRICHTUNG
318

FIG. 3A

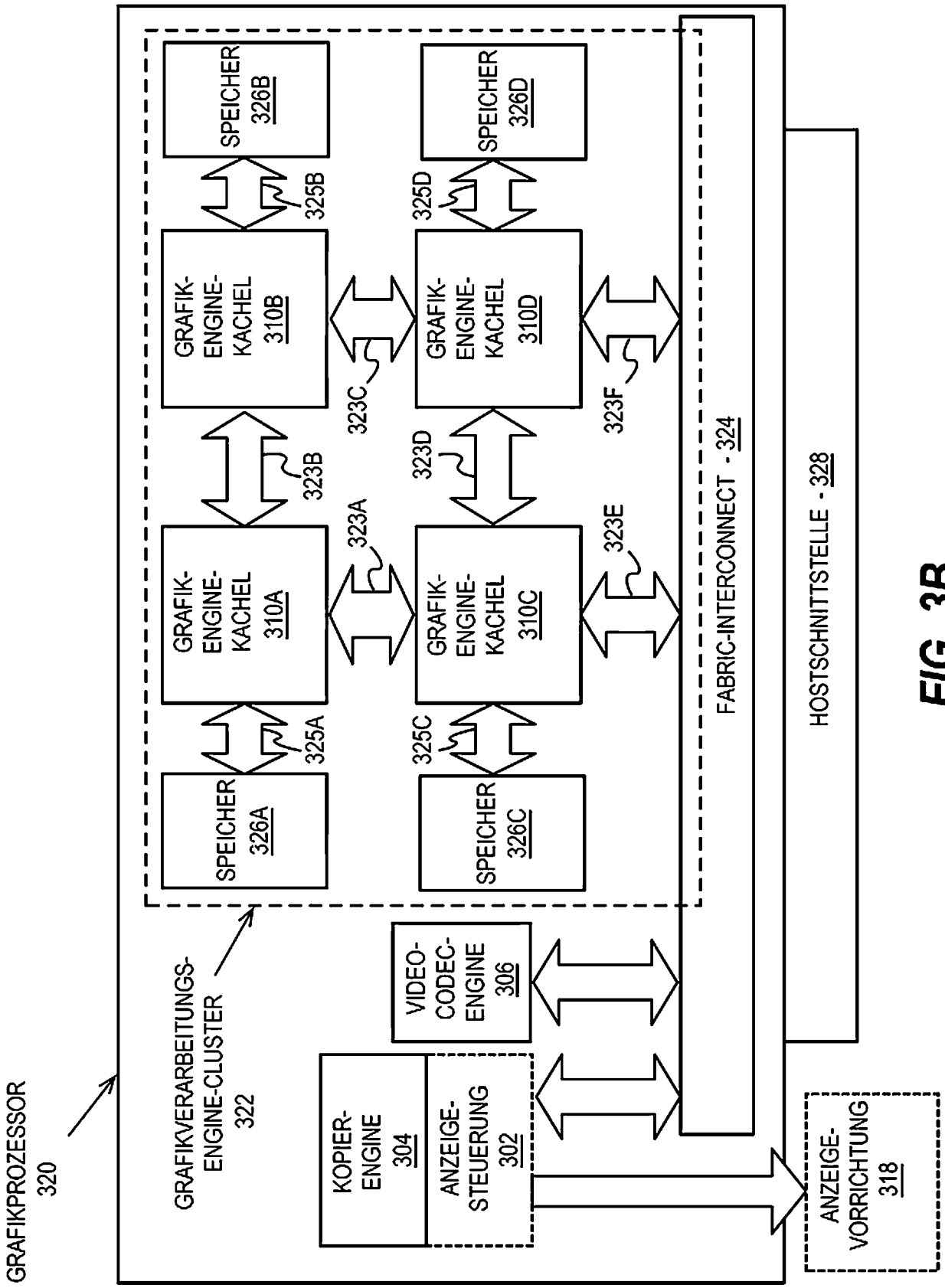

GRAFIKPROZESSOR 320
GRAFIKVERARBEITUNGS-ENGINE-CLUSTER 322
SPEICHER 326A
SPEICHER 326B
SPEICHER 326C
SPEICHER 326D
GRAFIK-ENGINE-KACHEL 310A
GRAFIK-ENGINE-KACHEL 310B
GRAFIK-ENGINE-KACHEL 310C
GRAFIK-ENGINE-KACHEL 310D
325A
325B
325C
325D
323A
323B
323C
323D
323E
323F
FABRIC-INTERCONNECT - 324
HOSTSCHNITTSTELLE - 328
VIDEO-CODEC-ENGINE 306
KOPIER-ENGINE 304
ANZEIGE-STEUERUNG 302
ANZEIGE-VORRICHTUNG 318

FIG. 3B

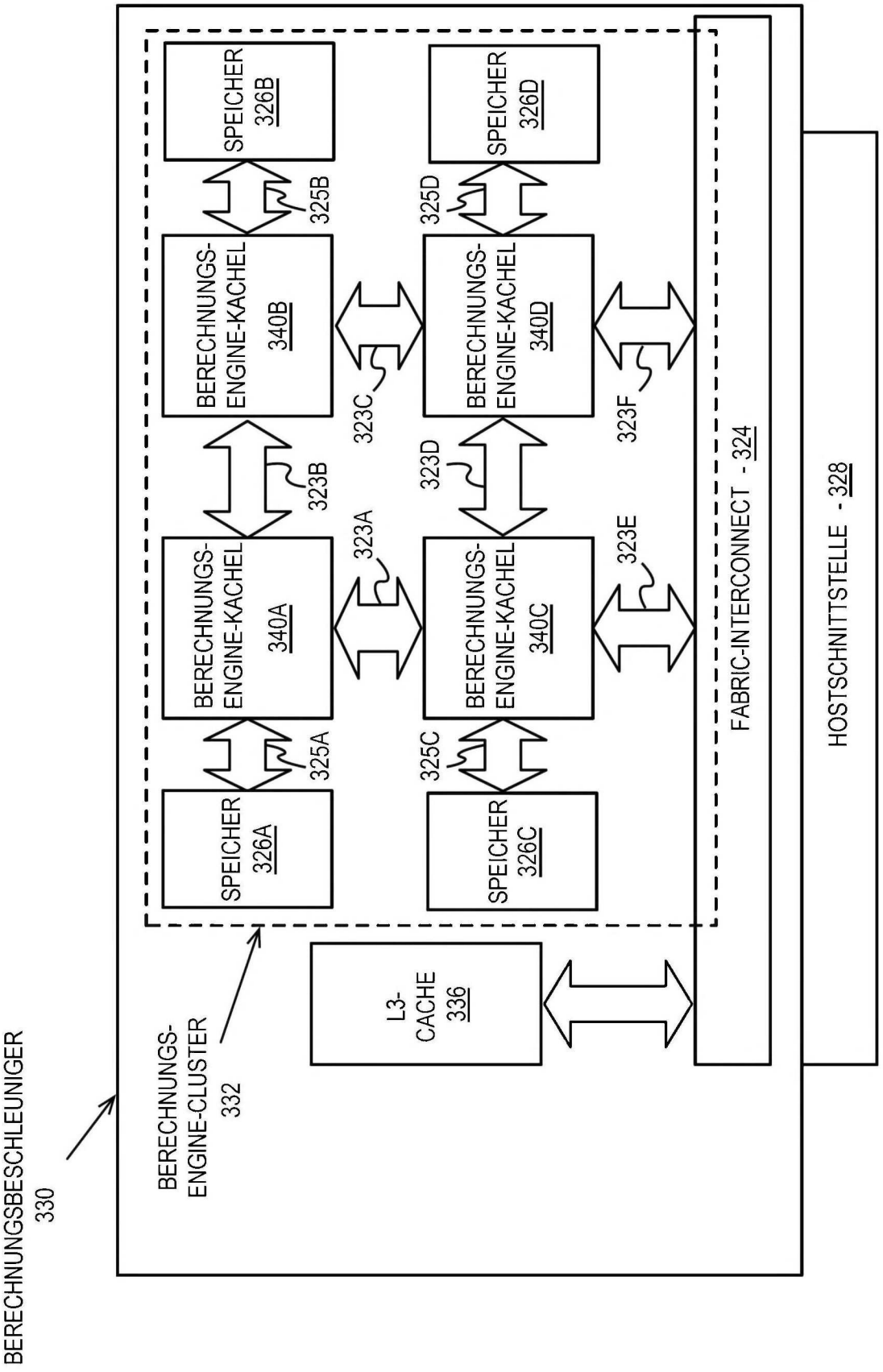

BERECHNUNGSBESCHLEUNIGER
330
BERECHNUNGS-
ENGINE-CLUSTER
332
SPEICHER
326A
SPEICHER
326B
SPEICHER
326C
SPEICHER
326D
BERECHNUNGS-
ENGINE-KACHEL
340A
BERECHNUNGS-
ENGINE-KACHEL
340B
BERECHNUNGS-
ENGINE-KACHEL
340C
BERECHNUNGS-
ENGINE-KACHEL
340D
L3-
CACHE
336
FABRIC-INTERCONNECT - 324
HOSTSCHNITTSTELLE - 328
325A
325B
325C
325D
323A
323B
323C
323D
323E
323F

FIG. 3C

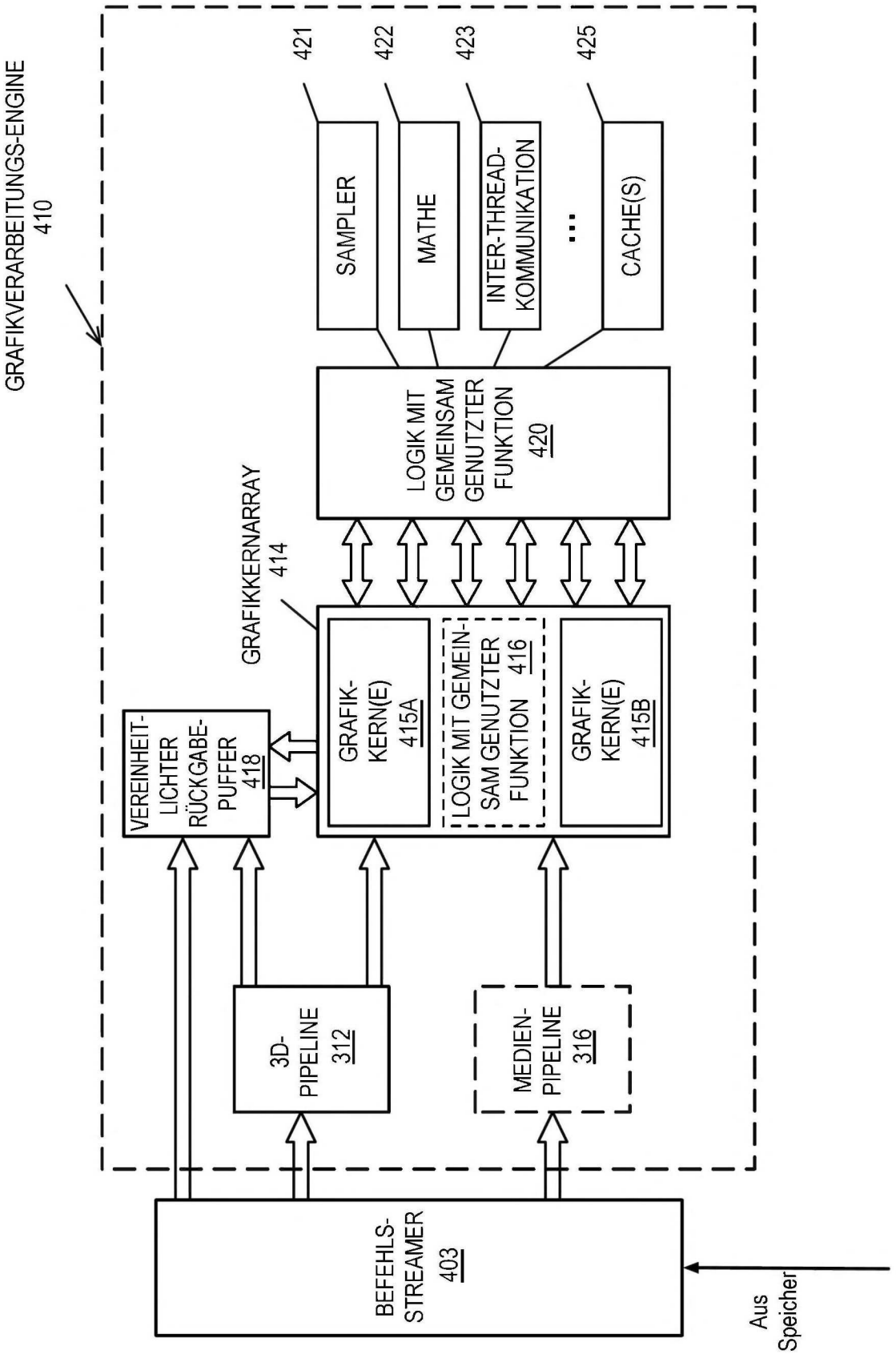

GRAFIKVERARBEITUNGS-ENGINE
410
421
422
423
425
SAMPLER
MATHE
INTER-THREAD-KOMMUNIKATION
...
CACHE(S)
LOGIK MIT GEMEINSAM GENUTZTER FUNKTION
420
GRAFIKKERNARRAY
414
VEREINHEITLICHTER RÜCKGABEPUFFER
418
GRAFIKKERN(E)
415A
LOGIK MIT GEMEINSAM GENUTZTER FUNKTION
416
GRAFIKKERN(E)
415B
3D-PIPELINE
312
MEDIEN-PIPELINE
316
BEFEHLS-STREAMER
403
Aus Speicher

FIG. 4

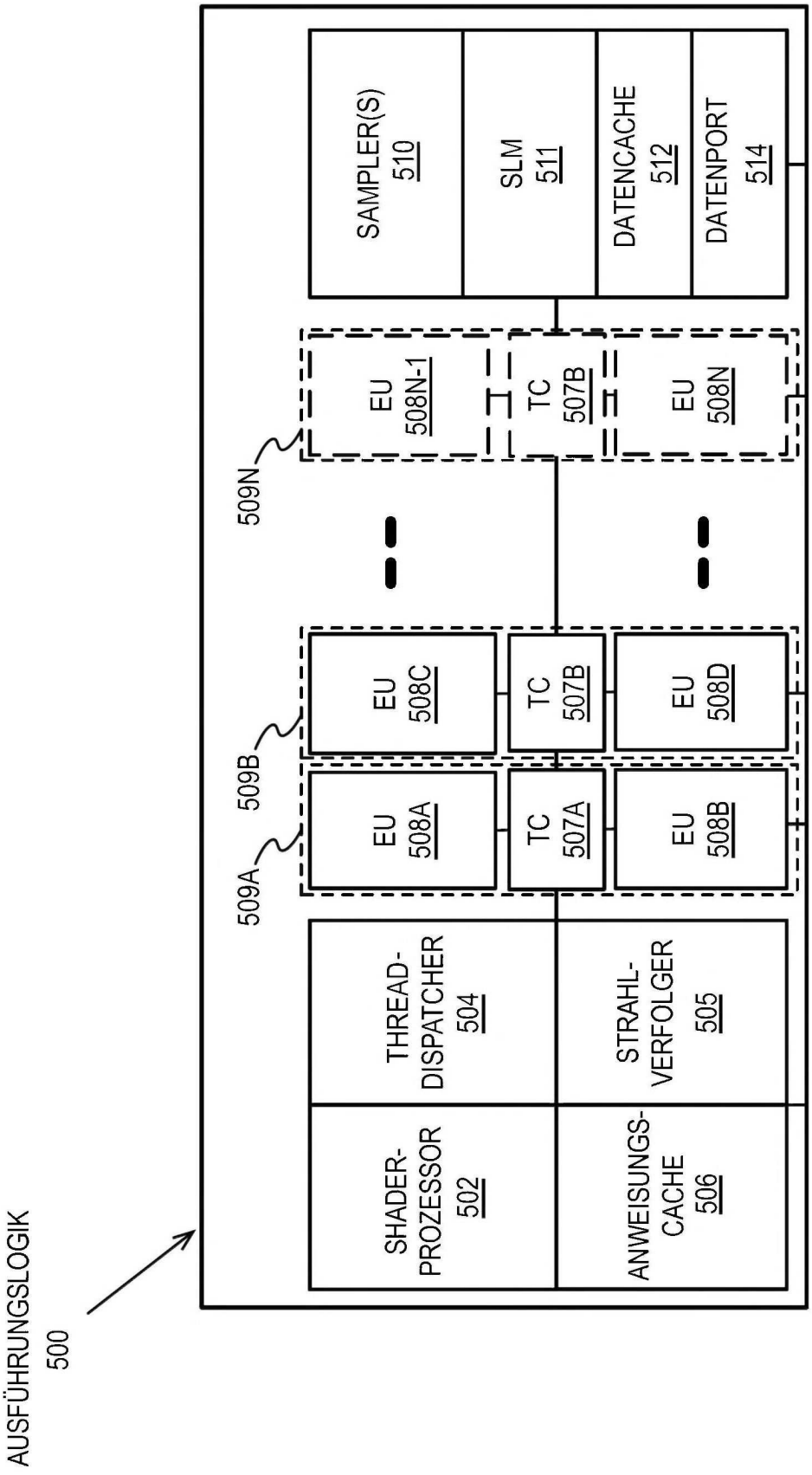

AUSFÜHRUNGSLOGIK
500
SHADER-PROZESSOR 502
THREAD-DISPATCHER 504
STRAHL-VERFOLGER 505
ANWEISUNGS-CACHE 506
509A
509B
509N
EU 508A
TC 507A
EU 508B
EU 508C
TC 507B
EU 508D
EU 508N-1
TC 507B
EU 508N
SAMPLER(S) 510
SLM 511
DATENCACHE 512
DATENPORT 514

FIG. 5A

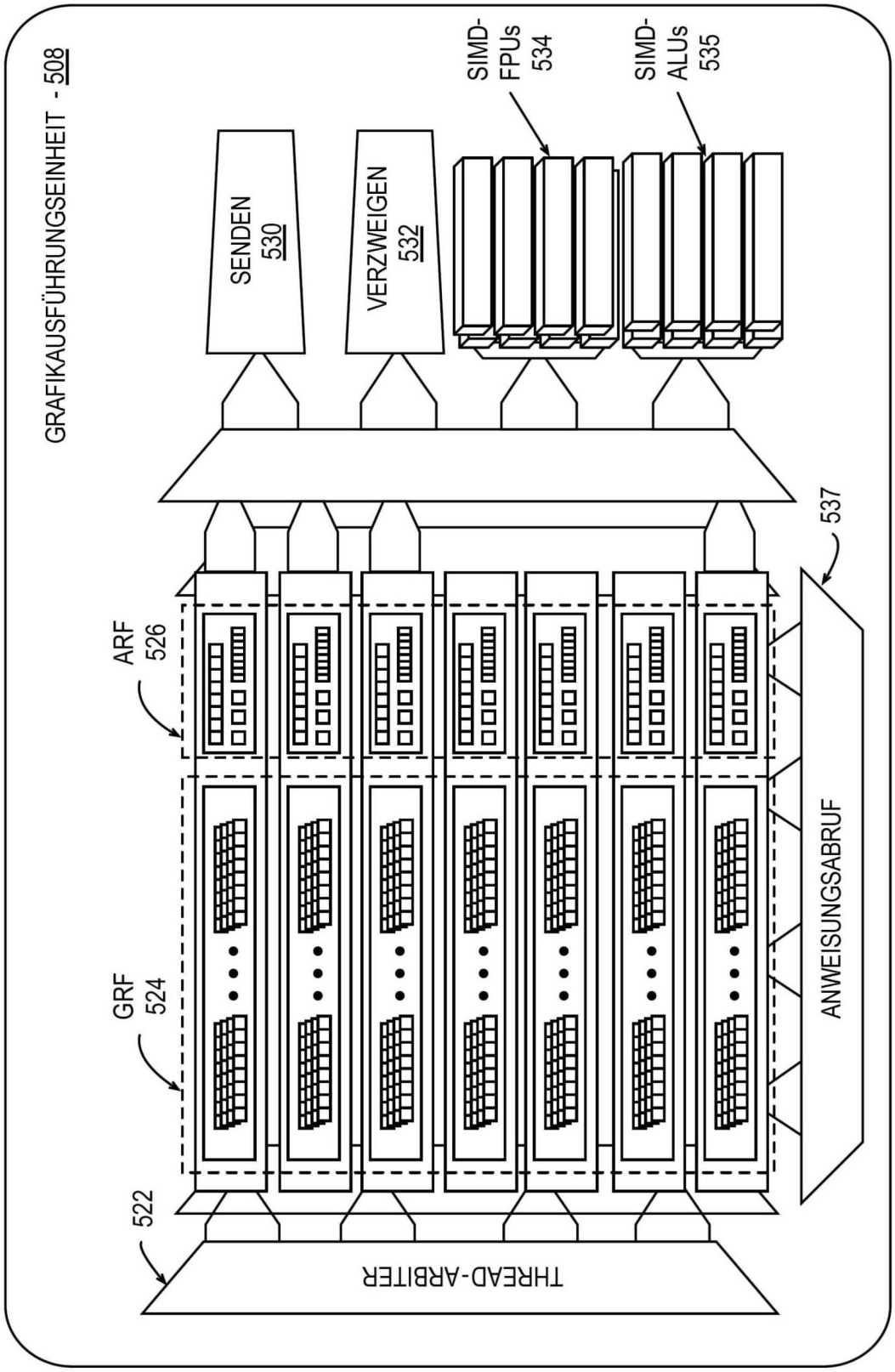

GRAFIKAUSFÜHRUNGSEINHEIT - 508
SENDEN 530
VERZWEIGEN 532
SIMD-FPUs 534
SIMD-ALUs 535
537
ARF 526
GRF 524
522
THREAD-ARBITER
ANWEISUNGSABRUF

FIG. 5B

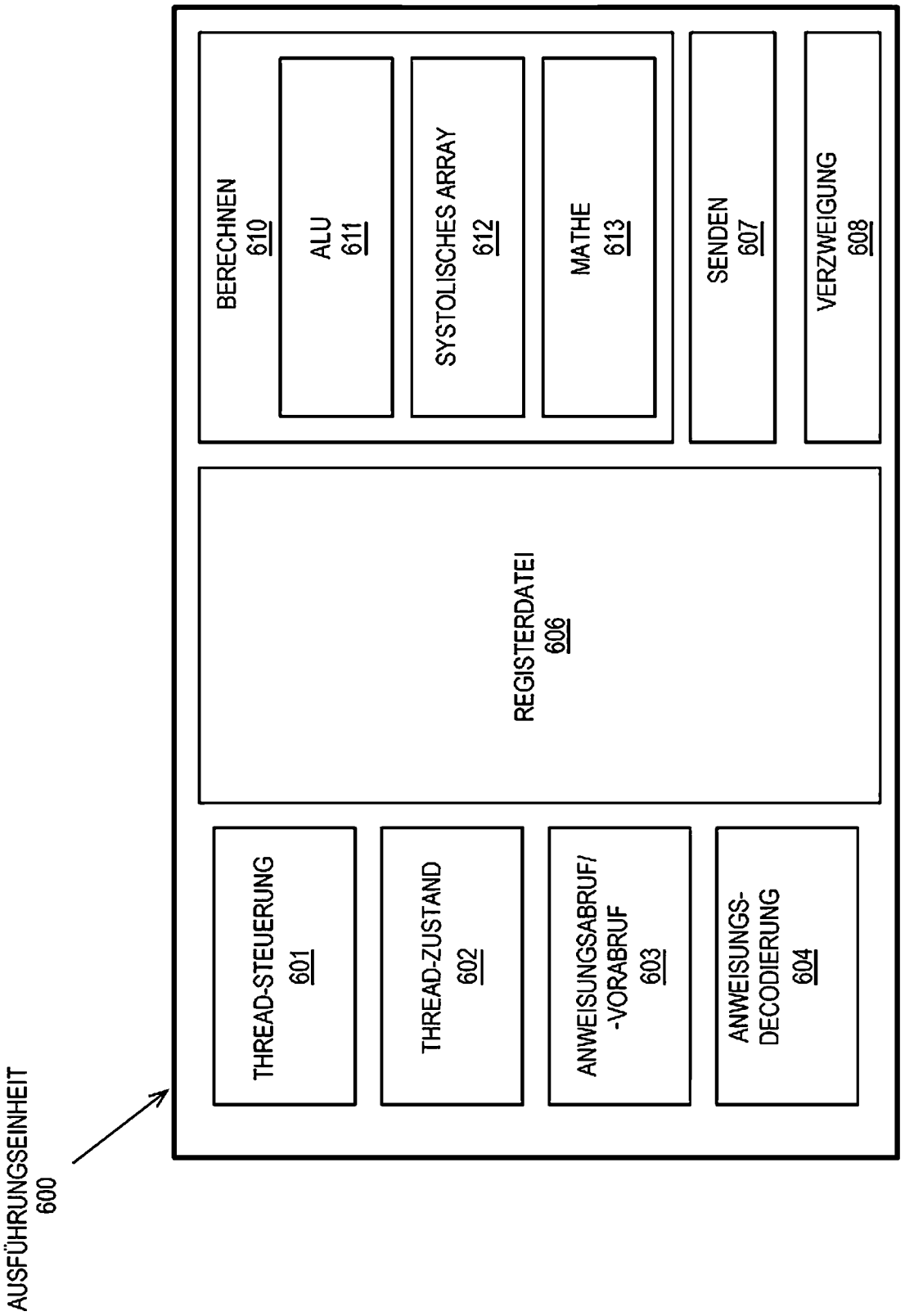

AUSFÜHRUNGSEINHEIT
600
THREAD-STEUERUNG
601
THREAD-ZUSTAND
602
ANWEISUNGSABRUF/
-VORABRUF
603
ANWEISUNGS-
DECODIERUNG
604
REGISTERDATEI
606
BERECHNEN
610
ALU
611
SYSTOLISCHES ARRAY
612
MATHE
613
SENDEN
607
VERZWEIGUNG
608

FIG. 6

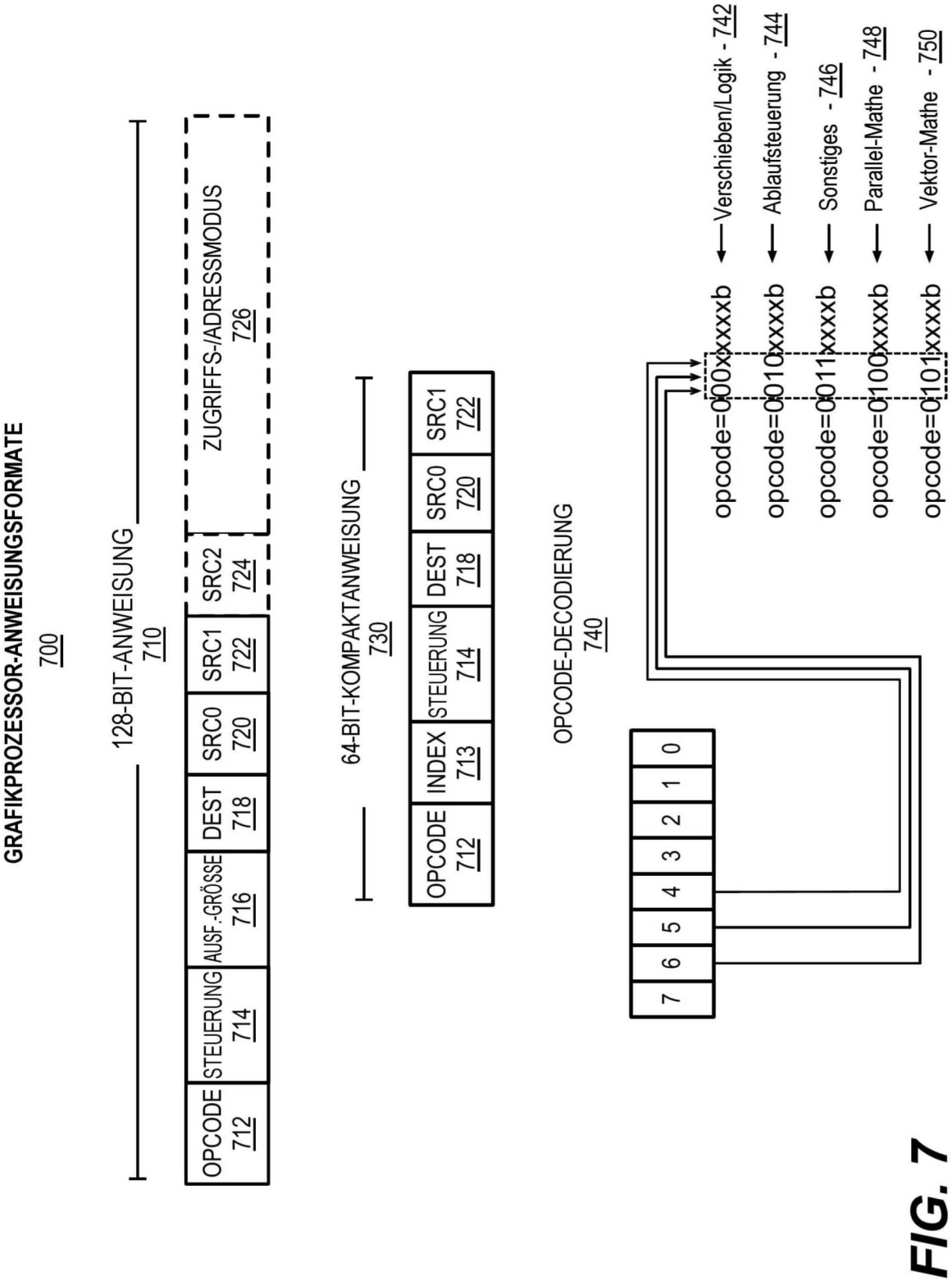

GRAFIKPROZESSOR-ANWEISUNGSFORMATE
700
128-BIT-ANWEISUNG
710
OPCODE 712
STEUERUNG 714
AUSF.-GRÖSSE 716
DEST 718
SRC0 720
SRC1 722
SRC2 724
ZUGRIFFS-/ADRESSMODUS 726
64-BIT-KOMPAKTANWEISUNG
730
OPCODE 712
INDEX 713
STEUERUNG 714
DEST 718
SRC0 720
SRC1 722
OPCODE-DECODIERUNG
740
7 6 5 4 3 2 1 0
opcode=000xxxxb
opcode=001xxxxb
opcode=010xxxxb
opcode=011xxxxb
opcode=100xxxxb
opcode=101xxxxb
Verschieben/Logik - 742
Ablaufsteuerung - 744
Sonstiges - 746
Parallel-Mathe - 748
Vektor-Mathe - 750

FIG. 7

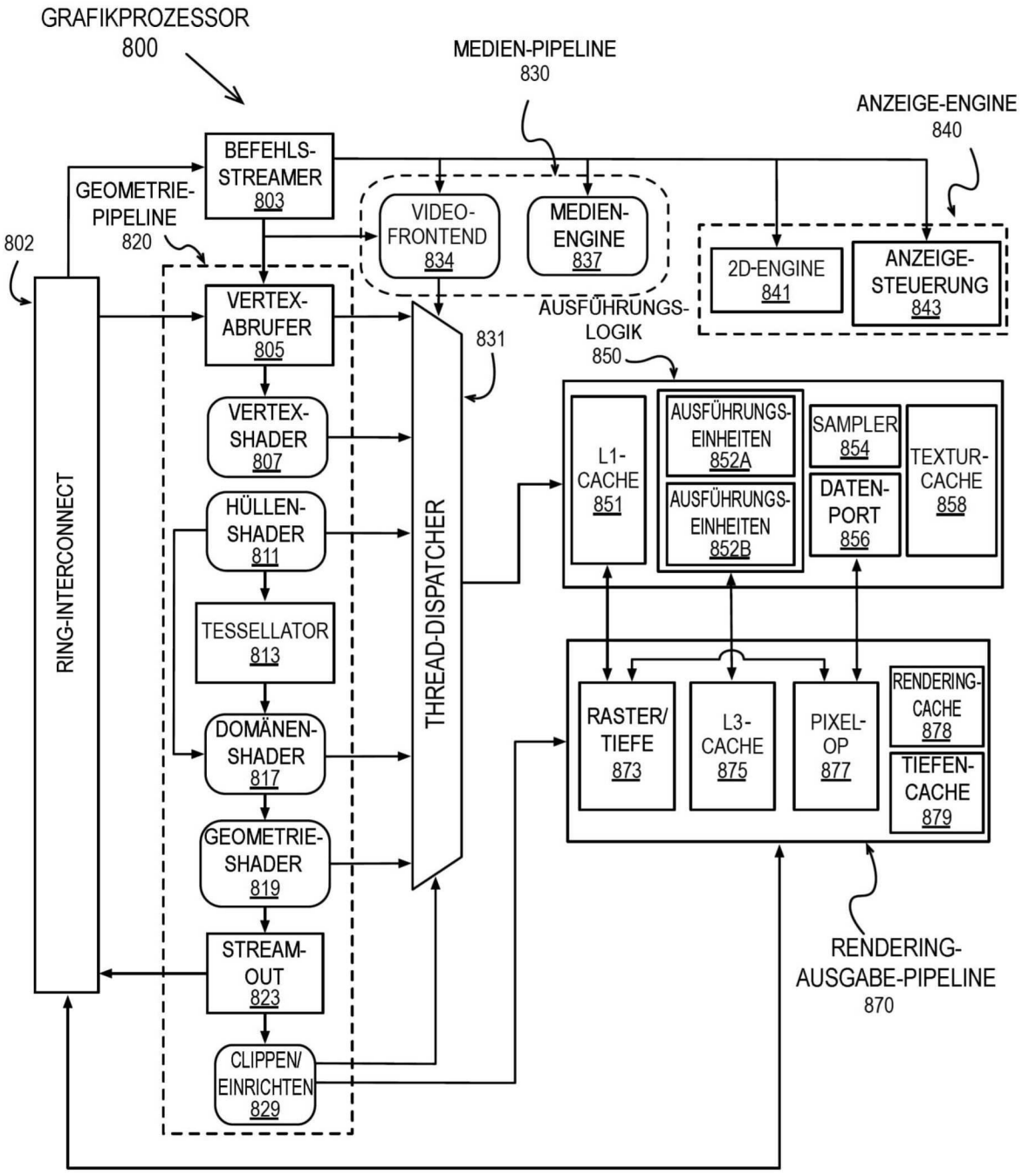

GRAFIKPROZESSOR
800
MEDIEN-PIPELINE
830
ANZEIGE-ENGINE
840
BEFEHLS-STREAMER
803
GEOMETRIE-PIPELINE
820
802
VIDEO-FRONTEND
834
MEDIEN-ENGINE
837
2D-ENGINE
841
ANZEIGE-STEUERUNG
843
AUSFÜHRUNGS-LOGIK
850
831
VERTEX-ABRUFER
805
VERTEX-SHADER
807
HÜLLEN-SHADER
811
TESSELLATOR
813
DOMÄNEN-SHADER
817
GEOMETRIE-SHADER
819
STREAM-OUT
823
CLIPPEN/ EINRICHTEN
829
RING-INTERCONNECT
THREAD-DISPATCHER
L1-CACHE
851
AUSFÜHRUNGS-EINHEITEN
852A
AUSFÜHRUNGS-EINHEITEN
852B
SAMPLER
854
DATEN-PORT
856
TEXTUR-CACHE
858
RASTER/ TIEFE
873
L3-CACHE
875
PIXEL-OP
877
RENDERING-CACHE
878
TIEFEN-CACHE
879
RENDERING-AUSGABE-PIPELINE
870

FIG. 8

FIG. 9A GRAFIKPROZESSOR-BEFEHLSFORMAT
900

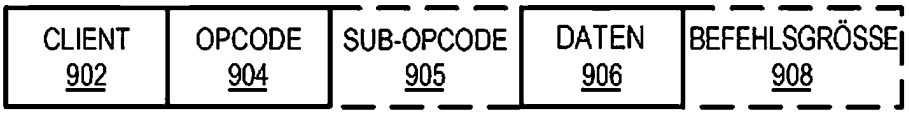
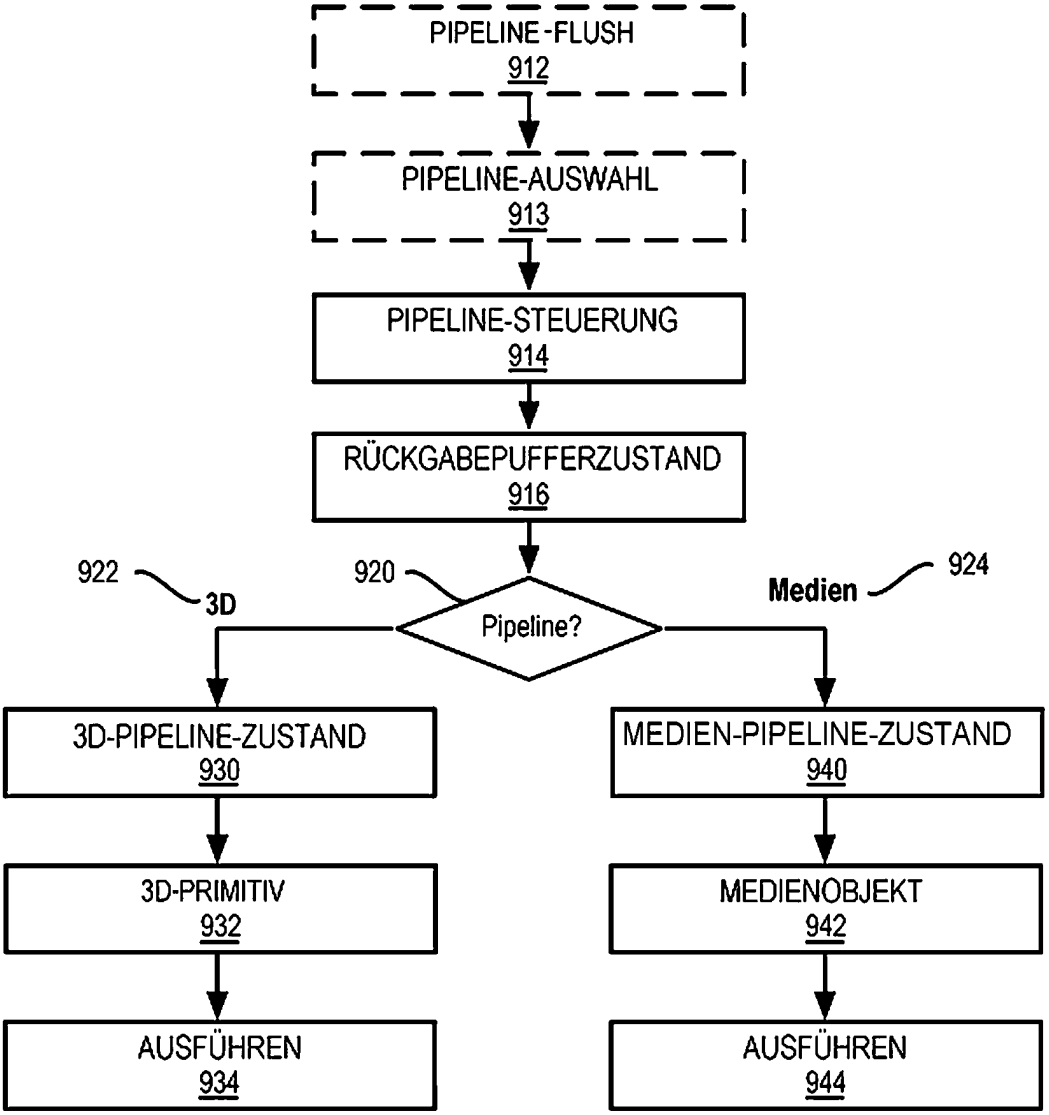


FIG. 9B GRAFIKPROZESSOR-BEFEHLSSEQUENZ
910



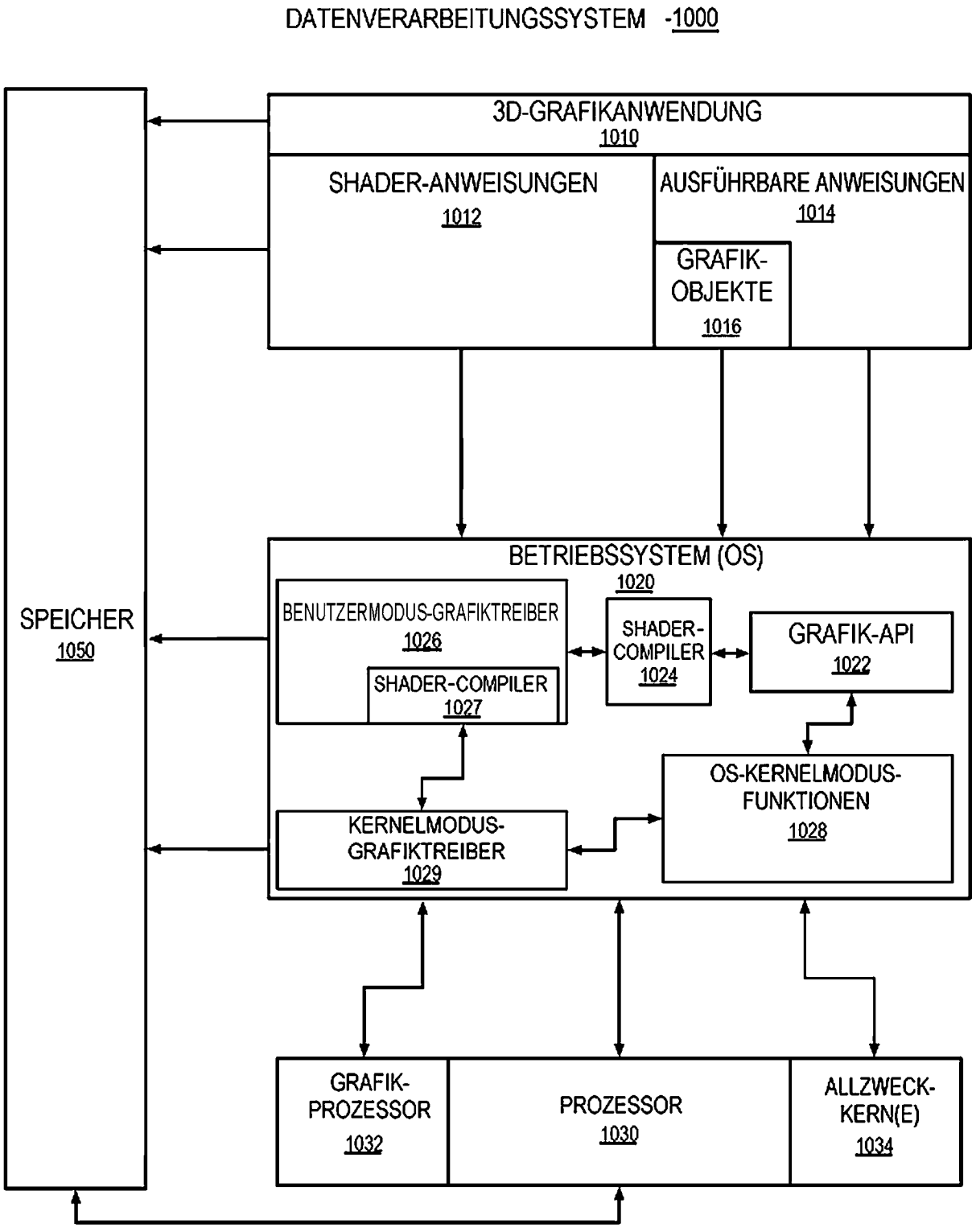

DATENVERARBEITUNGSSYSTEM -1000
3D-GRAFIKANWENDUNG
1010
SHADER-ANWEISUNGEN
1012
AUSFÜHRBARE ANWEISUNGEN
1014
GRAFIK-OBJEKTE
1016
SPEICHER
1050
BETRIEBSSYSTEM (OS)
1020
BENUTZERMODUS-GRAFIKTREIBER
1026
SHADER-COMPILER
1027
SHADER-COMPILER
1024
GRAFIK-API
1022
OS-KERNELMODUS-FUNKTIONEN
1028
KERNELMODUS-GRAFIKTREIBER
1029
GRAFIK-PROZESSOR
1032
PROZESSOR
1030
ALLZWECK-KERN(E)
1034

FIG. 10

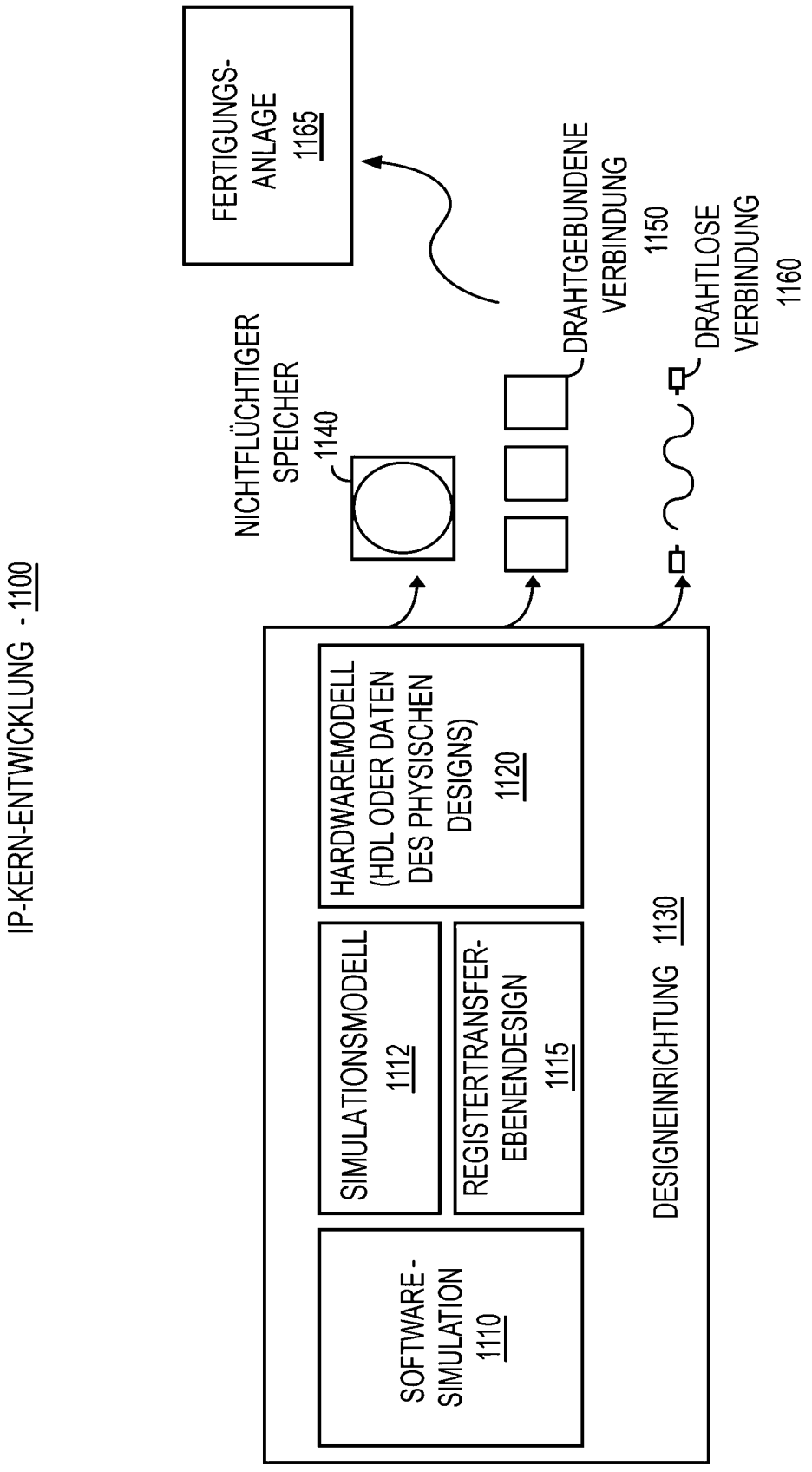

IP-KERN-ENTWICKLUNG - 1100
DESIGNEINRICHTUNG 1130
SOFTWARE-SIMULATION 1110
SIMULATIONSMODELL 1112
REGISTERTRANSFER-EBENENDESIGN 1115
HARDWAREMODELL (HDL ODER DATEN DES PHYSISCHEN DESIGNS) 1120
NICHTFLÜCHTIGER SPEICHER 1140
DRAHTGEBUNDENE VERBINDUNG 1150
DRAHTLOSE VERBINDUNG 1160
FERTIGUNGS-ANLAGE 1165

FIG. 11A

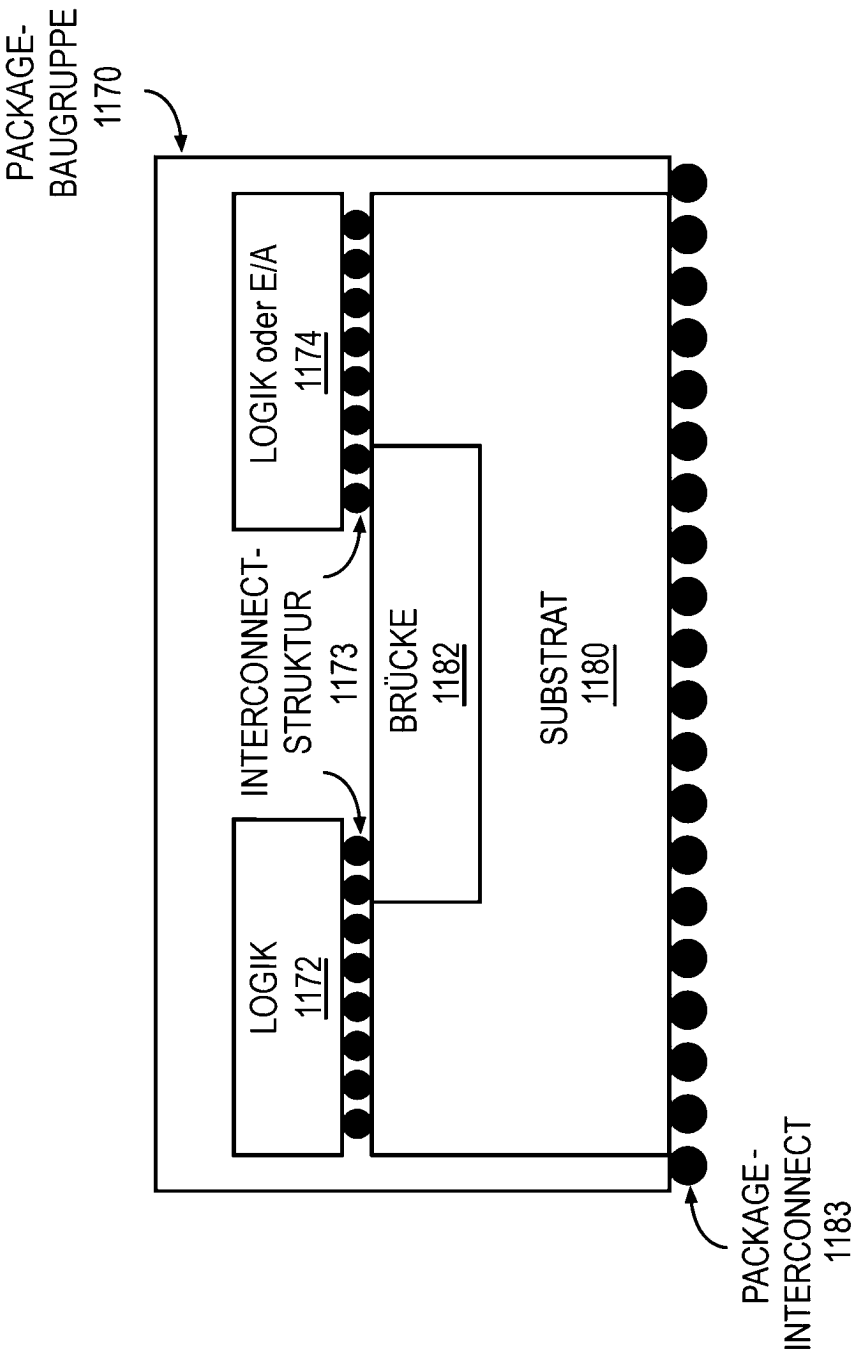

PACKAGE-BAUGRUPPE 1170
LOGIK oder E/A 1174
INTERCONNECT-STRUKTUR 1173
BRÜCKE 1182
SUBSTRAT 1180
LOGIK 1172
PACKAGE-INTERCONNECT 1183

FIG. 11B

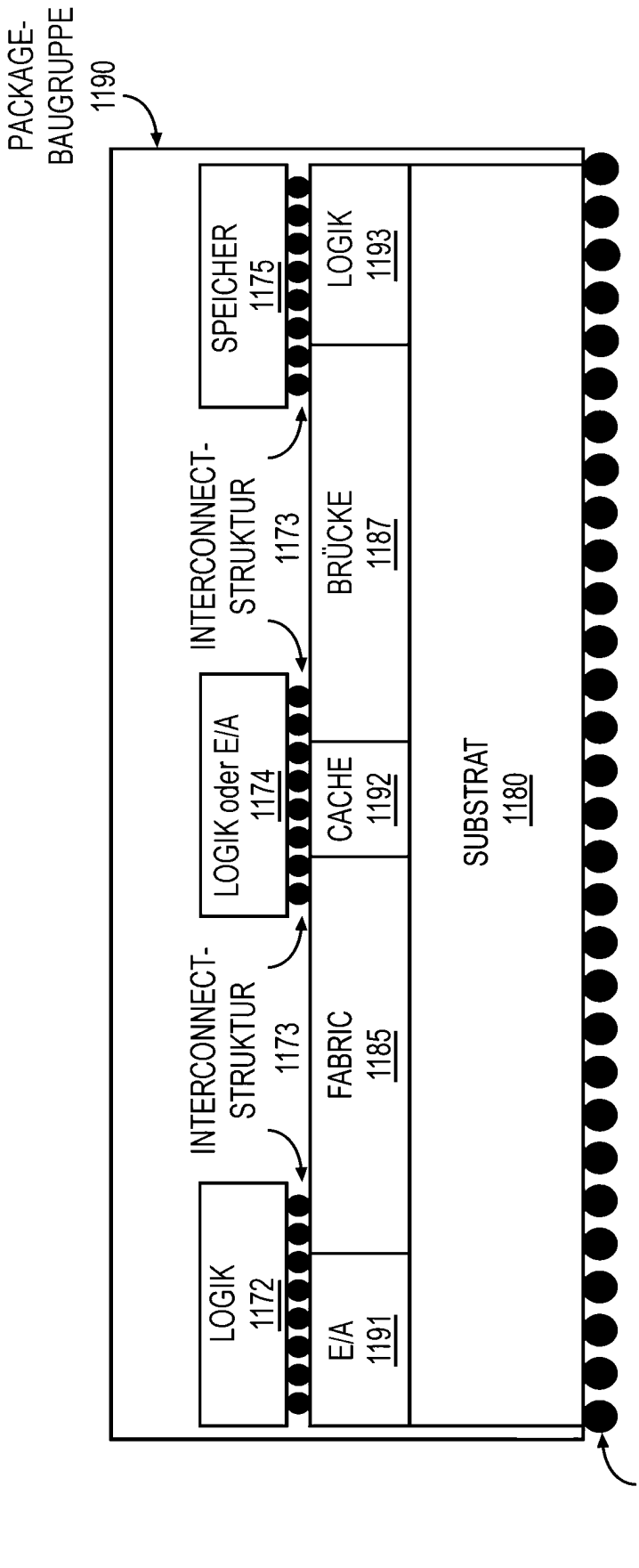

PACKAGE-BAUGRUPPE 1190
LOGIK 1172
E/A 1191
FABRIC 1185
CACHE 1192
BRÜCKE 1187
LOGIK 1193
SPEICHER 1175
LOGIK oder E/A 1174
INTERCONNECT-STRUKTUR 1173
INTERCONNECT-STRUKTUR 1173
SUBSTRAT 1180
PACKAGE-INTERCONNECT 1183

FIG. 11C

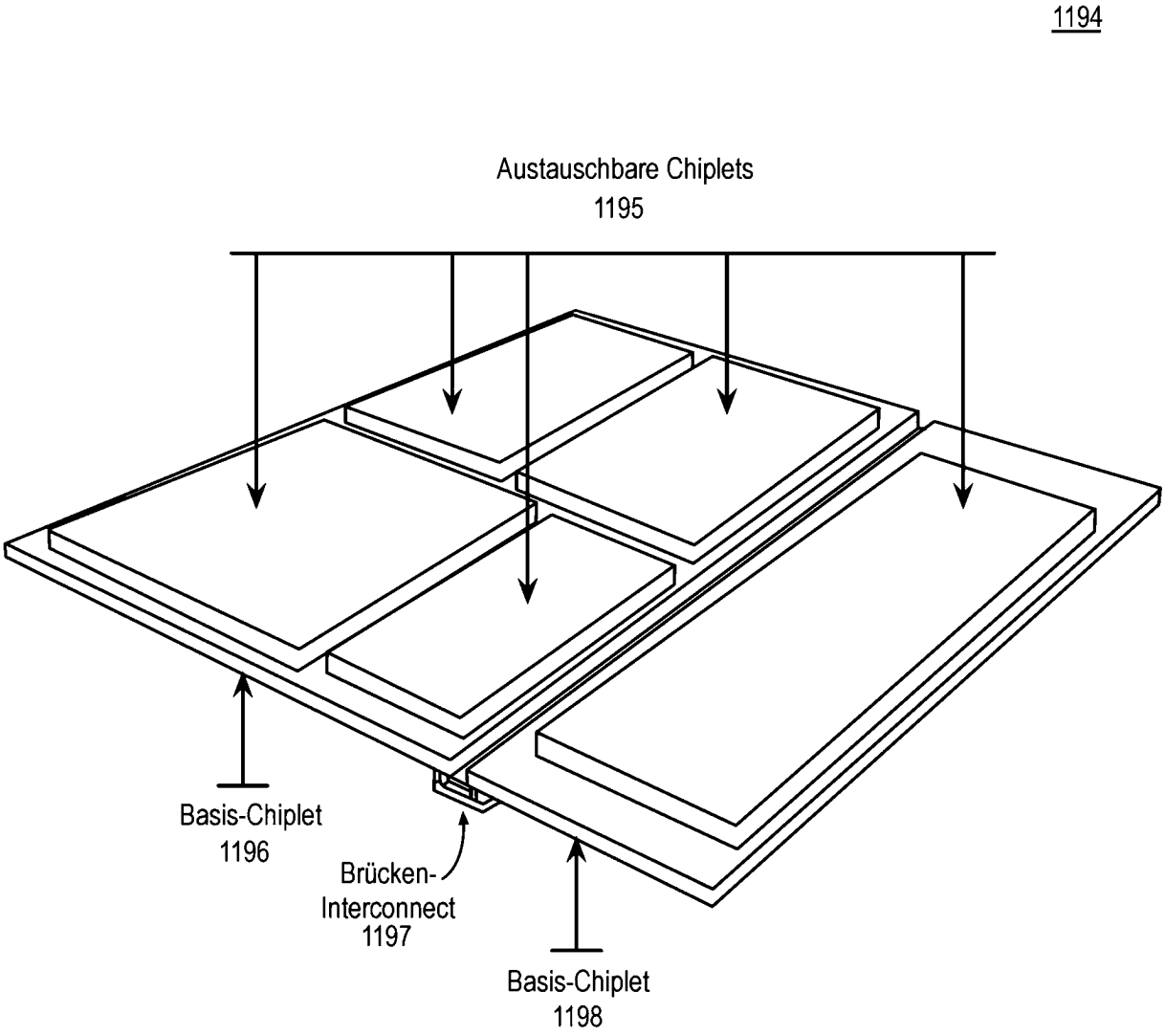

1194
Austauschbare Chiplets
1195
Basis-Chiplet
1196
Brücken-
Interconnect
1197
Basis-Chiplet
1198

FIG. 11D

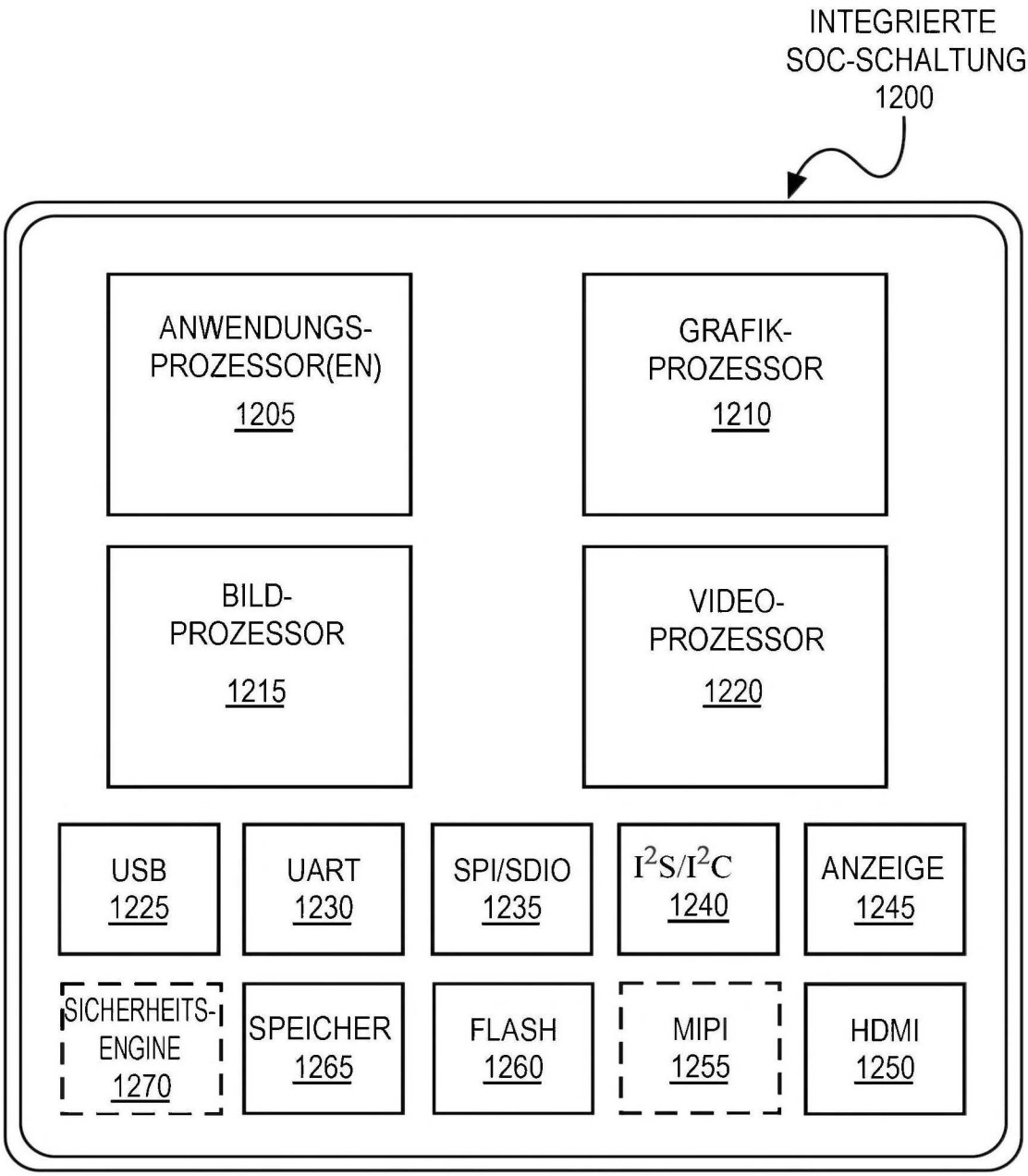

INTEGRIERTE SOC-SCHALTUNG 1200
ANWENDUNGS-PROZESSOR(EN) 1205
GRAFIK-PROZESSOR 1210
BILD-PROZESSOR 1215
VIDEO-PROZESSOR 1220
USB 1225
UART 1230
SPI/SDIO 1235
I2S/I2C 1240
ANZEIGE 1245
SICHERHEITS-ENGINE 1270
SPEICHER 1265
FLASH 1260
MIPI 1255
HDMI 1250

FIG. 12

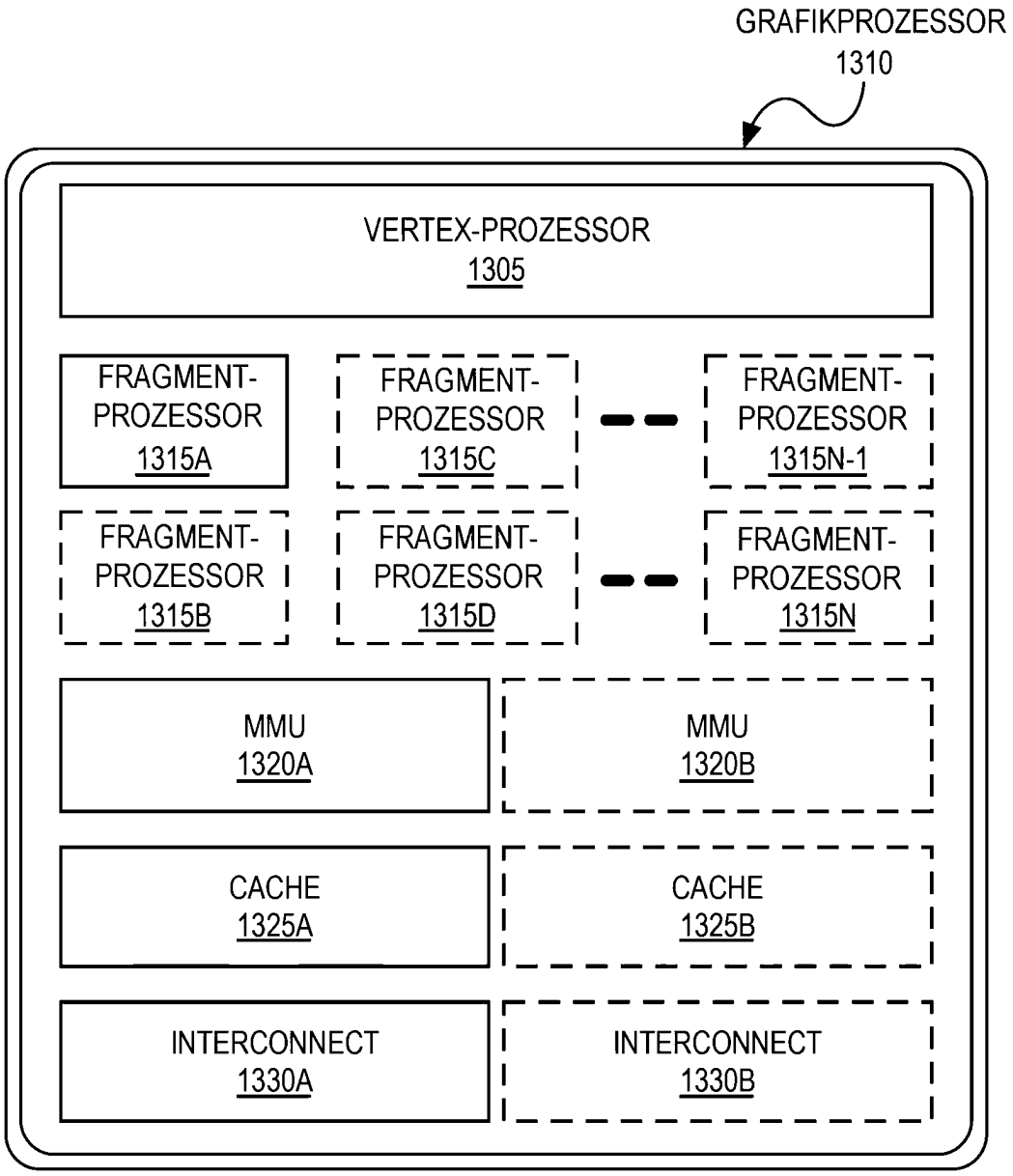

GRAFIKPROZESSOR
1310
VERTEX-PROZESSOR
1305
FRAGMENT-PROZESSOR
1315A
FRAGMENT-PROZESSOR
1315C
FRAGMENT-PROZESSOR
1315N-1
FRAGMENT-PROZESSOR
1315B
FRAGMENT-PROZESSOR
1315D
FRAGMENT-PROZESSOR
1315N
MMU
1320A
MMU
1320B
CACHE
1325A
CACHE
1325B
INTERCONNECT
1330A
INTERCONNECT
1330B

FIG. 13

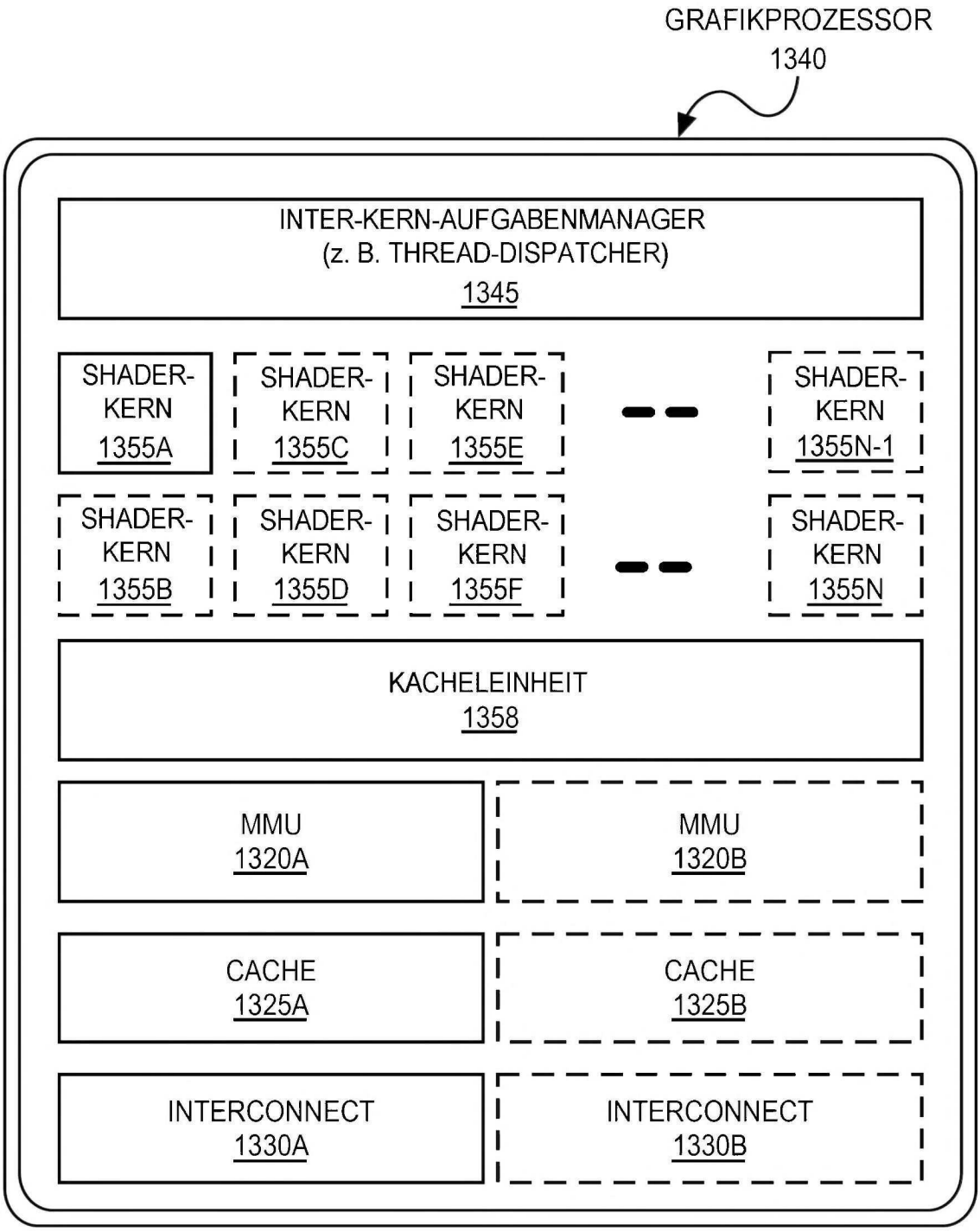

GRAFIKPROZESSOR
1340
INTER-KERN-AUFGABENMANAGER
(z. B. THREAD-DISPATCHER)
1345
SHADER-KERN 1355A
SHADER-KERN 1355C
SHADER-KERN 1355E
SHADER-KERN 1355N-1
SHADER-KERN 1355B
SHADER-KERN 1355D
SHADER-KERN 1355F
SHADER-KERN 1355N
KACHELEINHEIT
1358
MMU 1320A
MMU 1320B
CACHE 1325A
CACHE 1325B
INTERCONNECT 1330A
INTERCONNECT 1330B

FIG. 14

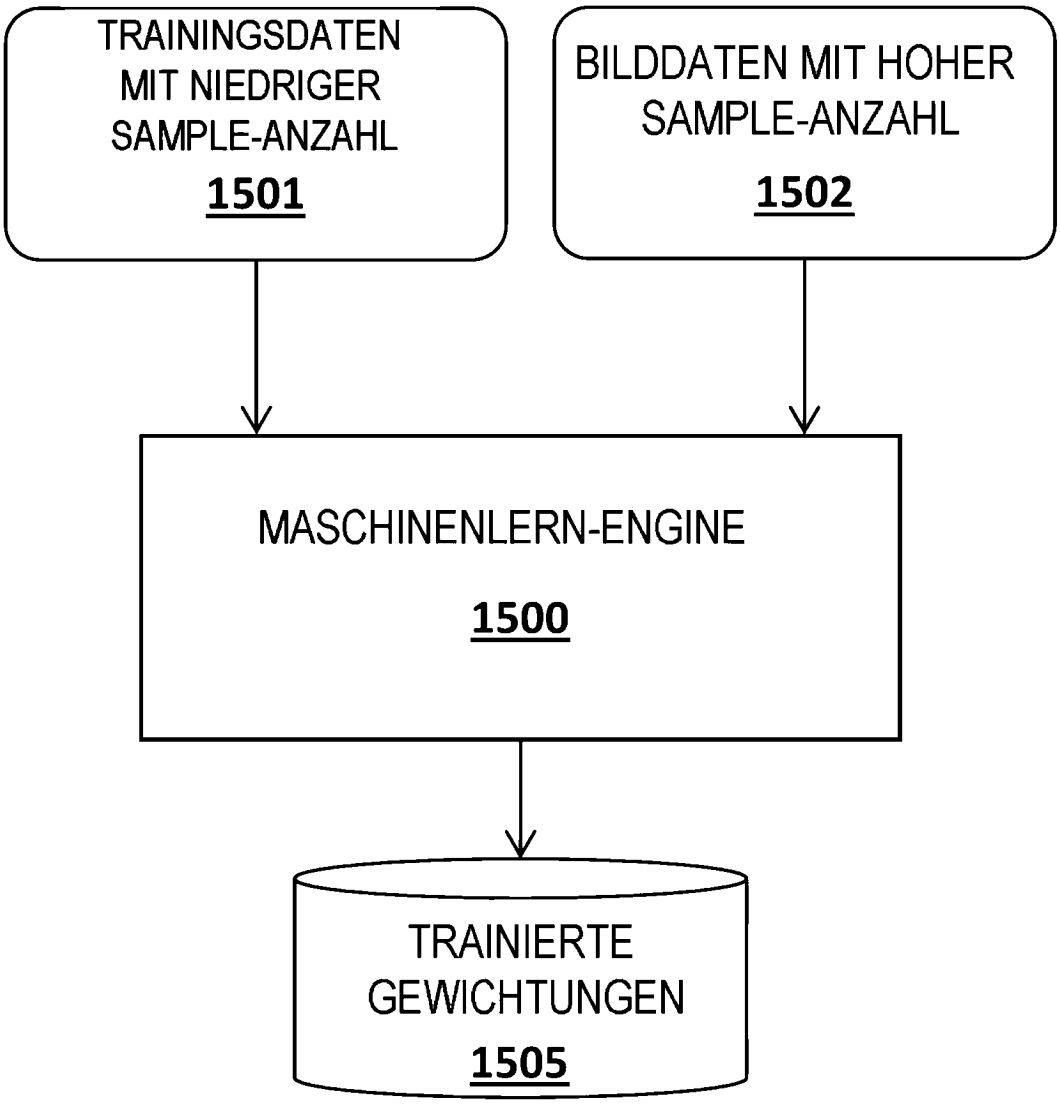

TRAININGSDATEN MIT NIEDRIGER SAMPLE-ANZAHL
1501
BILDDATEN MIT HOHER SAMPLE-ANZAHL
1502
MASCHINENLERN-ENGINE
1500
TRAINIERTE GEWICHTUNGEN
1505

FIG. 15

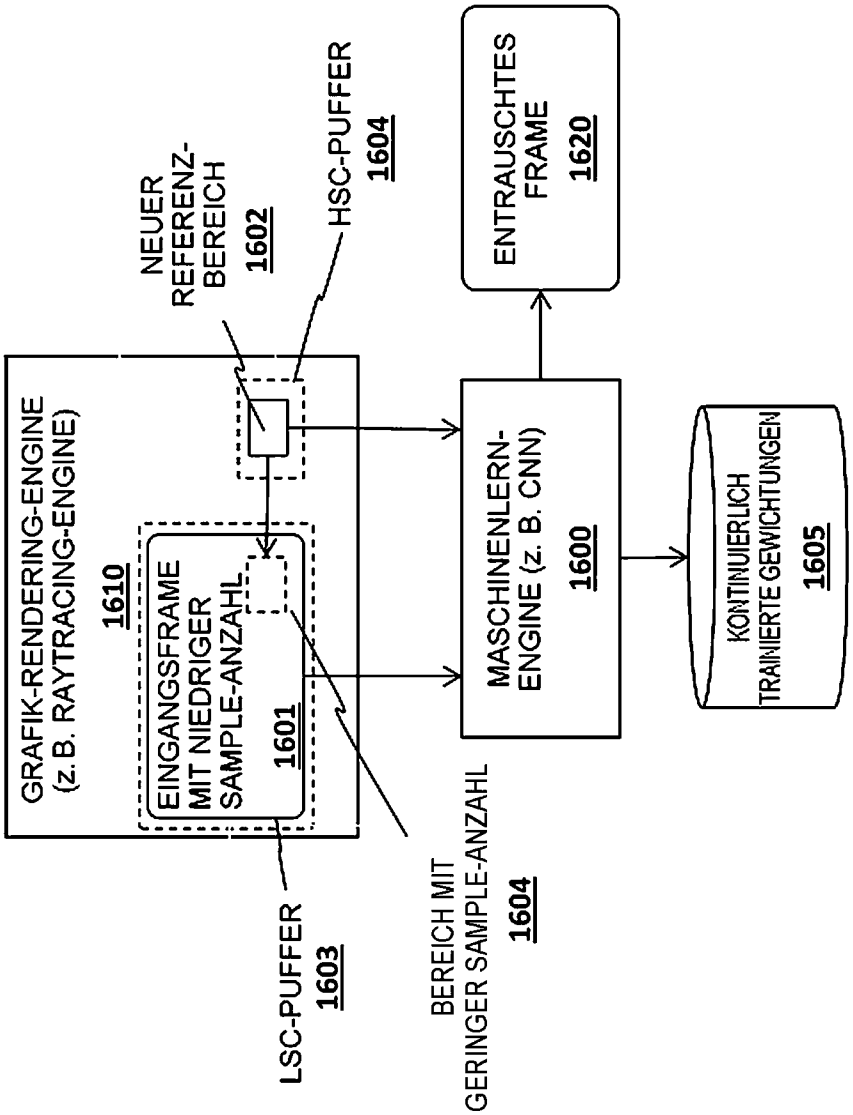

GRAFIK-RENDERING-ENGINE (z. B. RAYTRACING-ENGINE)
1610
EINGANGSFRAME MIT NIEDRIGER SAMPLE-ANZAHL
1601
NEUER REFERENZ-BEREICH
1602
HSC-PUFFER
1604
LSC-PUFFER
1603
BEREICH MIT GERINGER SAMPLE-ANZAHL
1604
MASCHINENLERN-ENGINE (z. B. CNN)
1600
ENTRAUSCHTES FRAME
1620
KONTINUIERLICH TRAINIERTE GEWICHTUNGEN
1605

FIG. 16

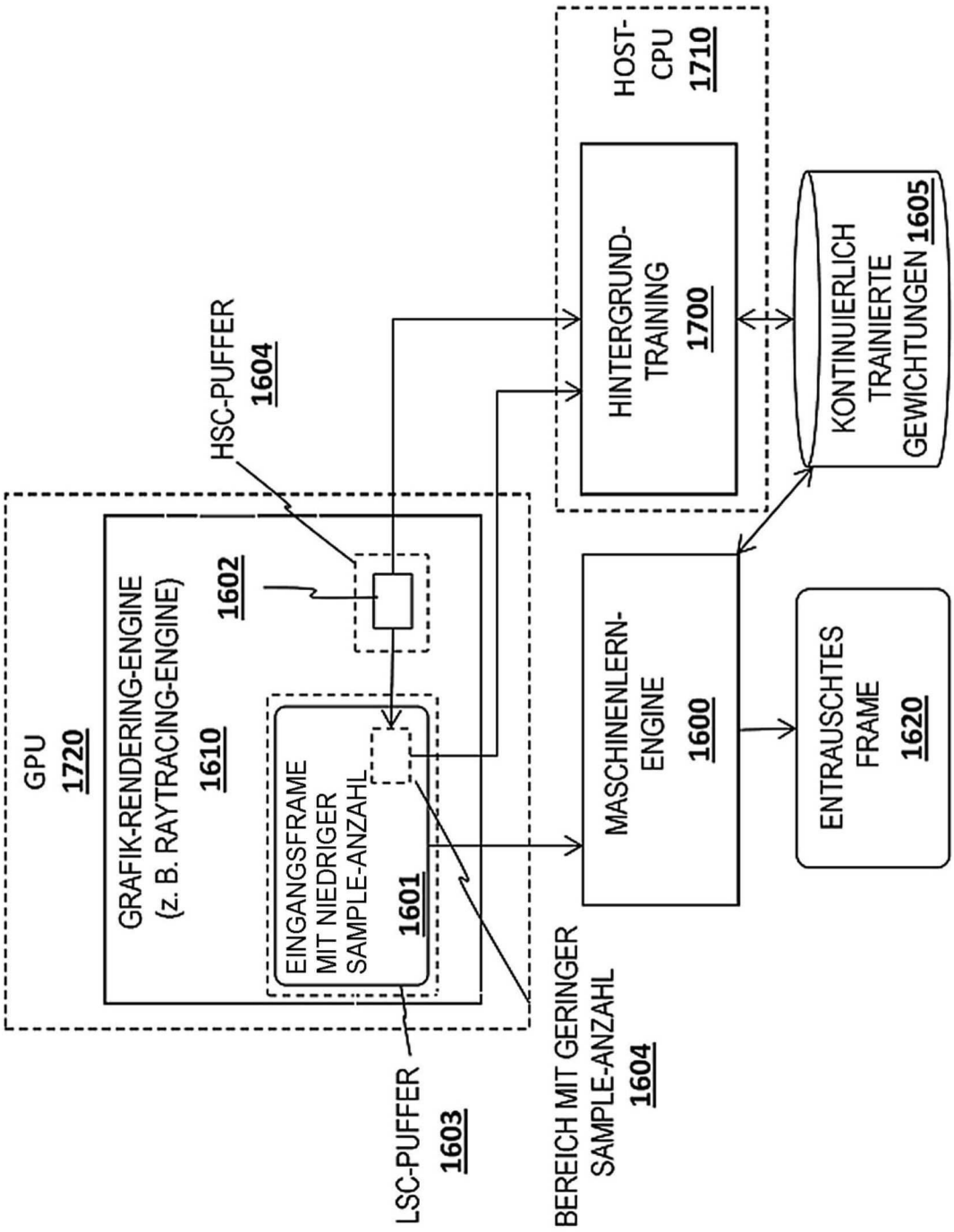

GPU
1720
GRAFIK-RENDERING-ENGINE
(z. B. RAYTRACING-ENGINE)
1610
1602
EINGANGSFRAME
MIT NIEDRIGER
SAMPLE-ANZAHL
1601
HSC-PUFFER
1604
LSC-PUFFER
1603
BEREICH MIT GERINGER
SAMPLE-ANZAHL
1604
HOST-
CPU
1710
HINTERGRUND-
TRAINING
1700
KONTINUIERLICH
TRAINIERTE
GEWICHTUNGEN 1605
MASCHINENLERN-
ENGINE
1600
ENTRAUSCHTES
FRAME
1620

FIG. 17

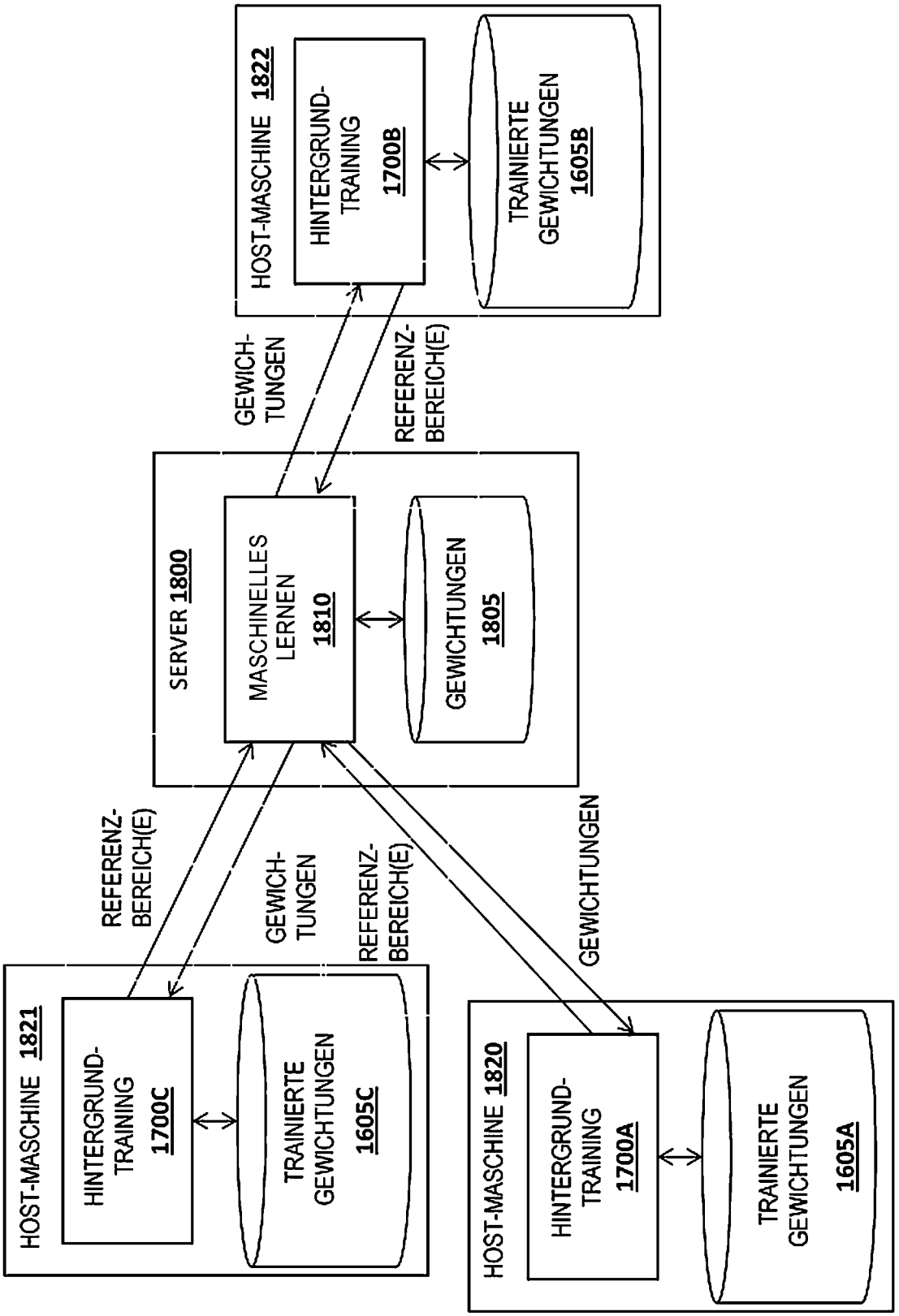

HOST-MASCHINE 1822
HINTERGRUND-TRAINING 1700B
TRAINIERTE GEWICHTUNGEN 1605B
GEWICH-TUNGEN
REFERENZ-BEREICH(E)
SERVER 1800
MASCHINELLES LERNEN 1810
GEWICHTUNGEN 1805
REFERENZ-BEREICH(E)
GEWICH-TUNGEN
REFERENZ-BEREICH(E)
GEWICHTUNGEN
HOST-MASCHINE 1821
HINTERGRUND-TRAINING 1700C
TRAINIERTE GEWICHTUNGEN 1605C
HOST-MASCHINE 1820
HINTERGRUND-TRAINING 1700A
TRAINIERTE GEWICHTUNGEN 1605A

FIG. 18A

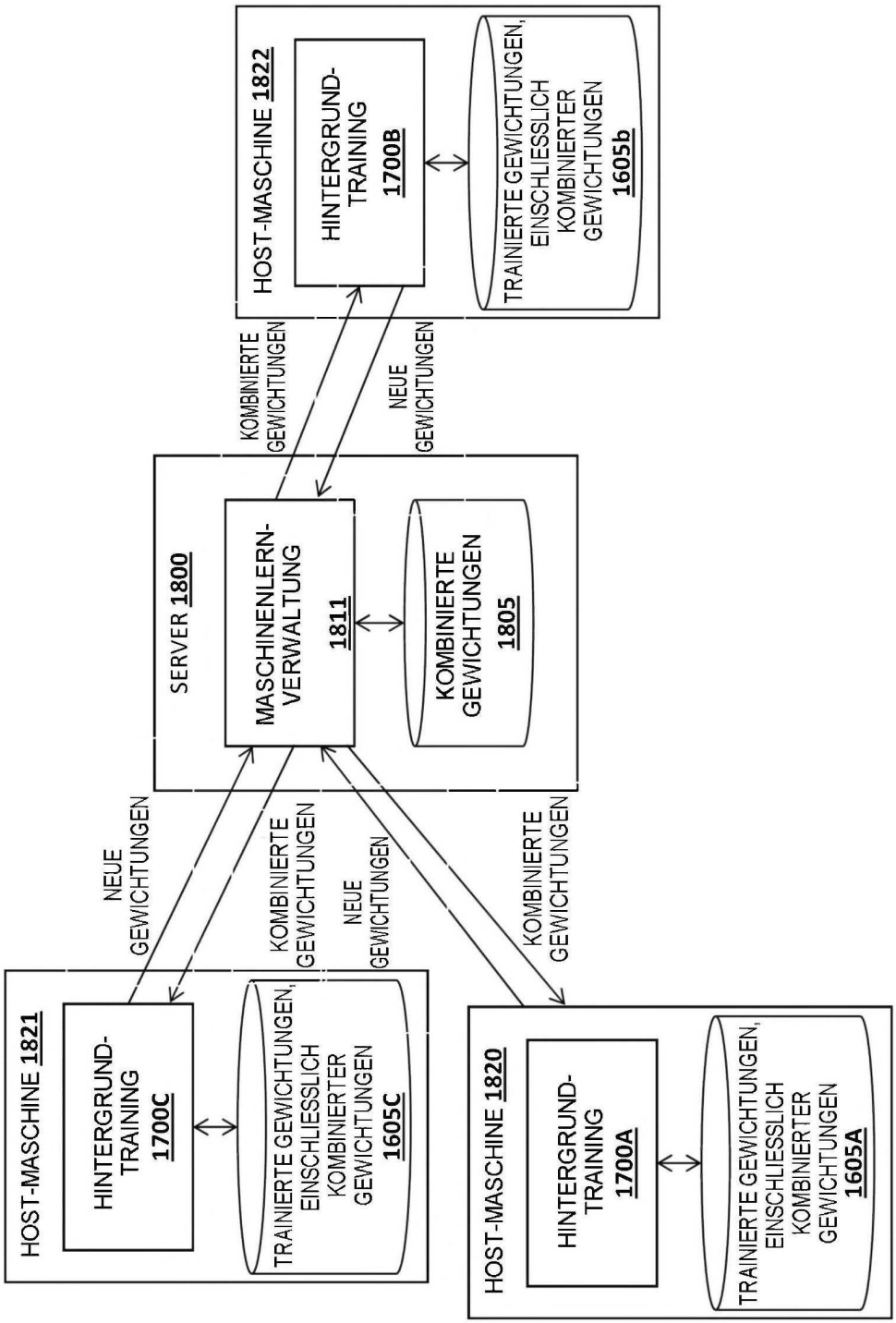

HOST-MASCHINE 1822
HINTERGRUND-TRAINING 1700B
TRAINIERTE GEWICHTUNGEN, EINSCHLIESSLICH KOMBINIERTER GEWICHTUNGEN 1605b
KOMBINIERTE GEWICHTUNGEN
NEUE GEWICHTUNGEN
SERVER 1800
MASCHINENLERN-VERWALTUNG 1811
KOMBINIERTE GEWICHTUNGEN 1805
NEUE GEWICHTUNGEN
KOMBINIERTE GEWICHTUNGEN
NEUE GEWICHTUNGEN
KOMBINIERTE GEWICHTUNGEN
HOST-MASCHINE 1821
HINTERGRUND-TRAINING 1700C
TRAINIERTE GEWICHTUNGEN, EINSCHLIESSLICH KOMBINIERTER GEWICHTUNGEN 1605C
HOST-MASCHINE 1820
HINTERGRUND-TRAINING 1700A
TRAINIERTE GEWICHTUNGEN, EINSCHLIESSLICH KOMBINIERTER GEWICHTUNGEN 1605A

FIG. 18B

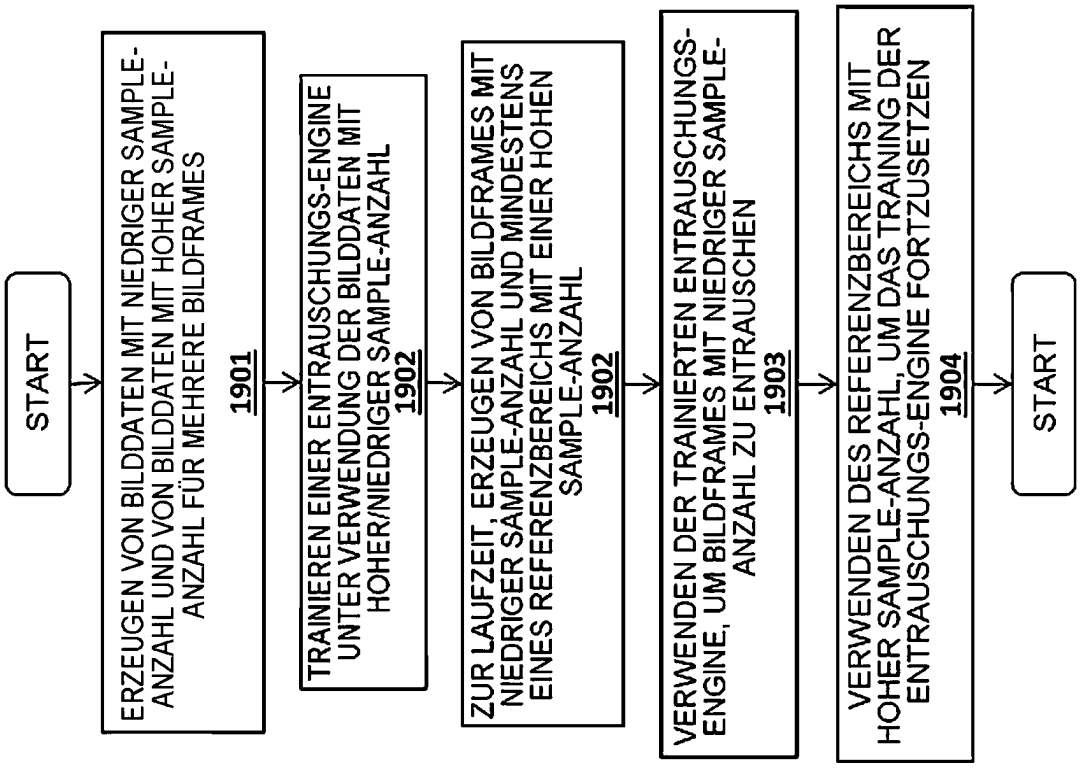

START
ERZEUGEN VON BILDDATEN MIT NIEDRIGER SAMPLE-ANZAHL UND VON BILDDATEN MIT HOHER SAMPLE-ANZAHL FÜR MEHRERE BILDFRAMES
1901
TRAINIEREN EINER ENTRAUSCHUNGS-ENGINE UNTER VERWENDUNG DER BILDDATEN MIT HOHER/NIEDRIGER SAMPLE-ANZAHL
1902
ZUR LAUFZEIT, ERZEUGEN VON BILDFRAMES MIT NIEDRIGER SAMPLE-ANZAHL UND MINDESTENS EINES REFERENZBEREICHS MIT EINER HOHEN SAMPLE-ANZAHL
1902
VERWENDEN DER TRAINIERTEN ENTRAUSCHUNGS-ENGINE, UM BILDFRAMES MIT NIEDRIGER SAMPLE-ANZAHL ZU ENTRAUSCHEN
1903
VERWENDEN DES REFERENZBEREICHS MIT HOHER SAMPLE-ANZAHL, UM DAS TRAINING DER ENTRAUSCHUNGS-ENGINE FORTZUSETZEN
1904
START

FIG. 19

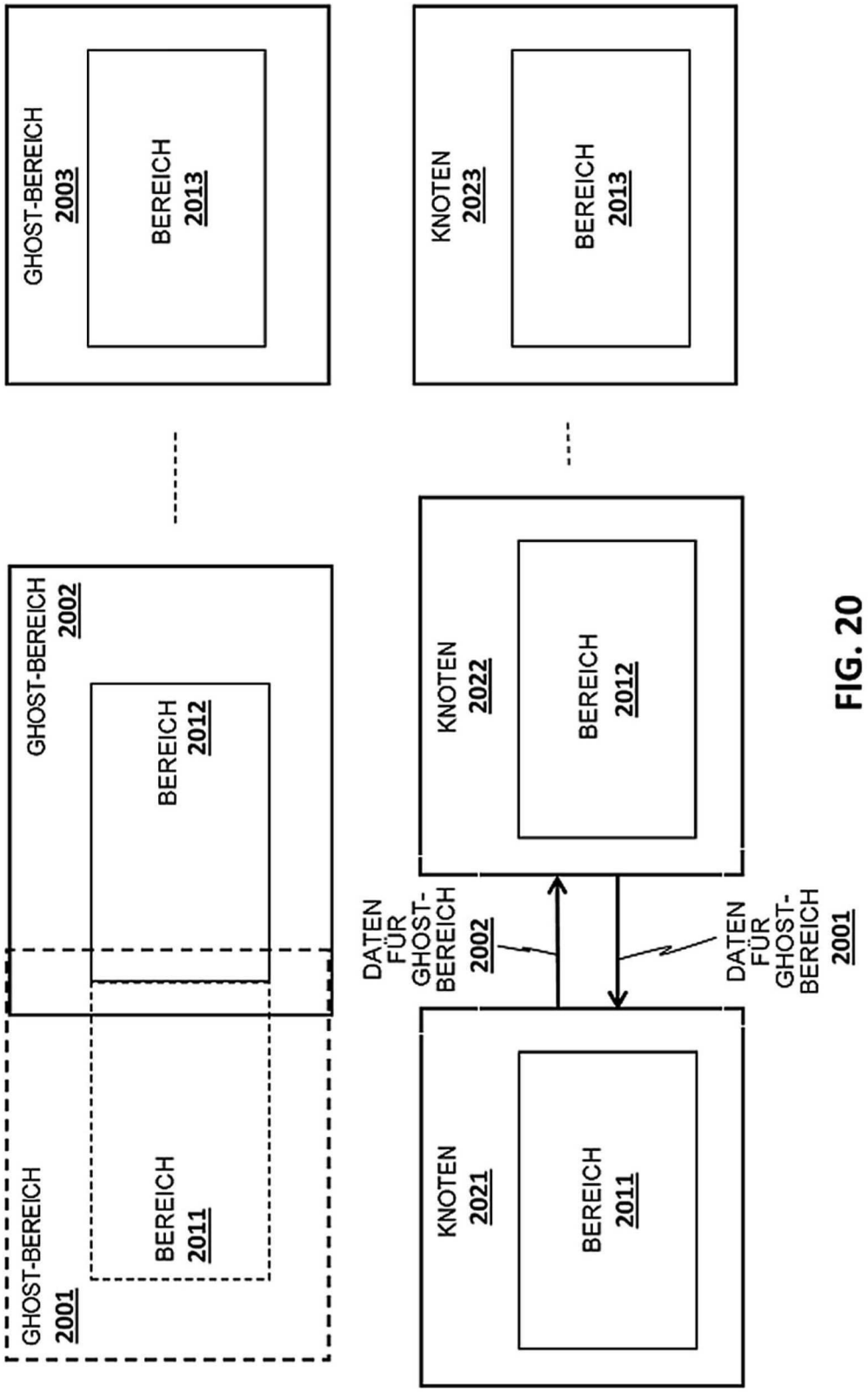

GHOST-BEREICH 2003
BEREICH 2013
KNOTEN 2023
BEREICH 2013
GHOST-BEREICH 2002
BEREICH 2012
KNOTEN 2022
BEREICH 2012
DATEN FÜR GHOST-BEREICH 2002
DATEN FÜR GHOST-BEREICH 2001
GHOST-BEREICH 2001
BEREICH 2011
KNOTEN 2021
BEREICH 2011

FIG. 20

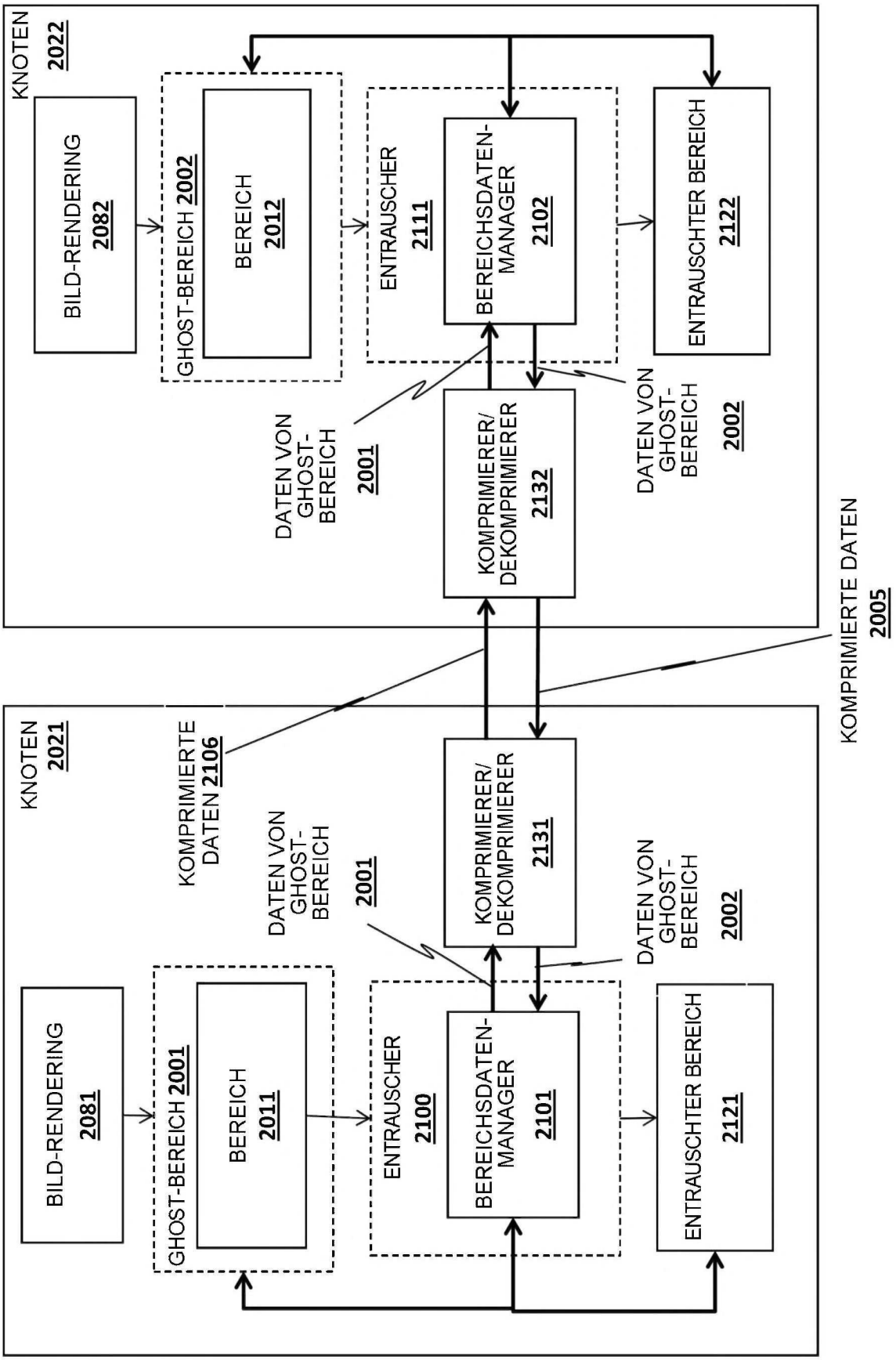

KNOTEN 2022
BILD-RENDERING 2082
GHOST-BEREICH 2002
BEREICH 2012
ENTRAUSCHER 2111
BEREICHSDATEN-MANAGER 2102
ENTRAUSCHTER BEREICH 2122
DATEN VON GHOST-BEREICH 2001
KOMPRIMIERER/ DEKOMPRIMIERER 2132
DATEN VON GHOST-BEREICH 2002
KOMPRIMIERTE DATEN 2005
KNOTEN 2021
KOMPRIMIERTE DATEN 2106
DATEN VON GHOST-BEREICH 2001
KOMPRIMIERER/ DEKOMPRIMIERER 2131
DATEN VON GHOST-BEREICH 2002
BILD-RENDERING 2081
GHOST-BEREICH 2001
BEREICH 2011
ENTRAUSCHER 2100
BEREICHSDATEN-MANAGER 2101
ENTRAUSCHTER BEREICH 2121

FIG. 21

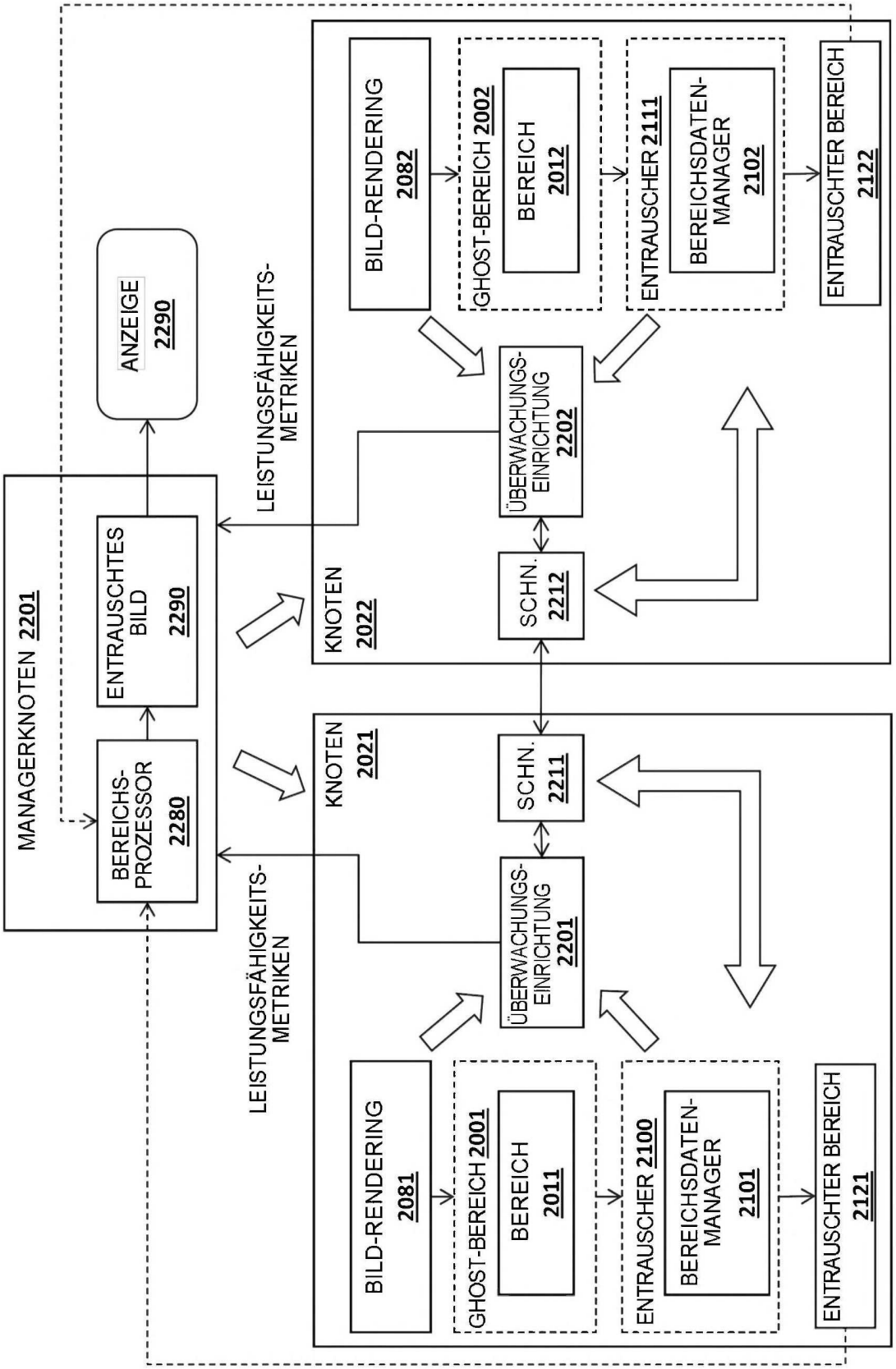

MANAGERKNOTEN 2201
BEREICHS-PROZESSOR 2280
ENTRAUSCHTES BILD 2290
ANZEIGE 2290
LEISTUNGSFÄHIGKEITS-METRIKEN
LEISTUNGSFÄHIGKEITS-METRIKEN
KNOTEN 2022
BILD-RENDERING 2082
GHOST-BEREICH 2002
BEREICH 2012
ENTRAUSCHER 2111
BEREICHSDATEN-MANAGER 2102
ENTRAUSCHTER BEREICH 2122
ÜBERWACHUNGS-EINRICHTUNG 2202
SCHN. 2212
KNOTEN 2021
SCHN. 2211
ÜBERWACHUNGS-EINRICHTUNG 2201
BILD-RENDERING 2081
GHOST-BEREICH 2001
BEREICH 2011
ENTRAUSCHER 2100
BEREICHSDATEN-MANAGER 2101
ENTRAUSCHTER BEREICH 2121

FIG. 22

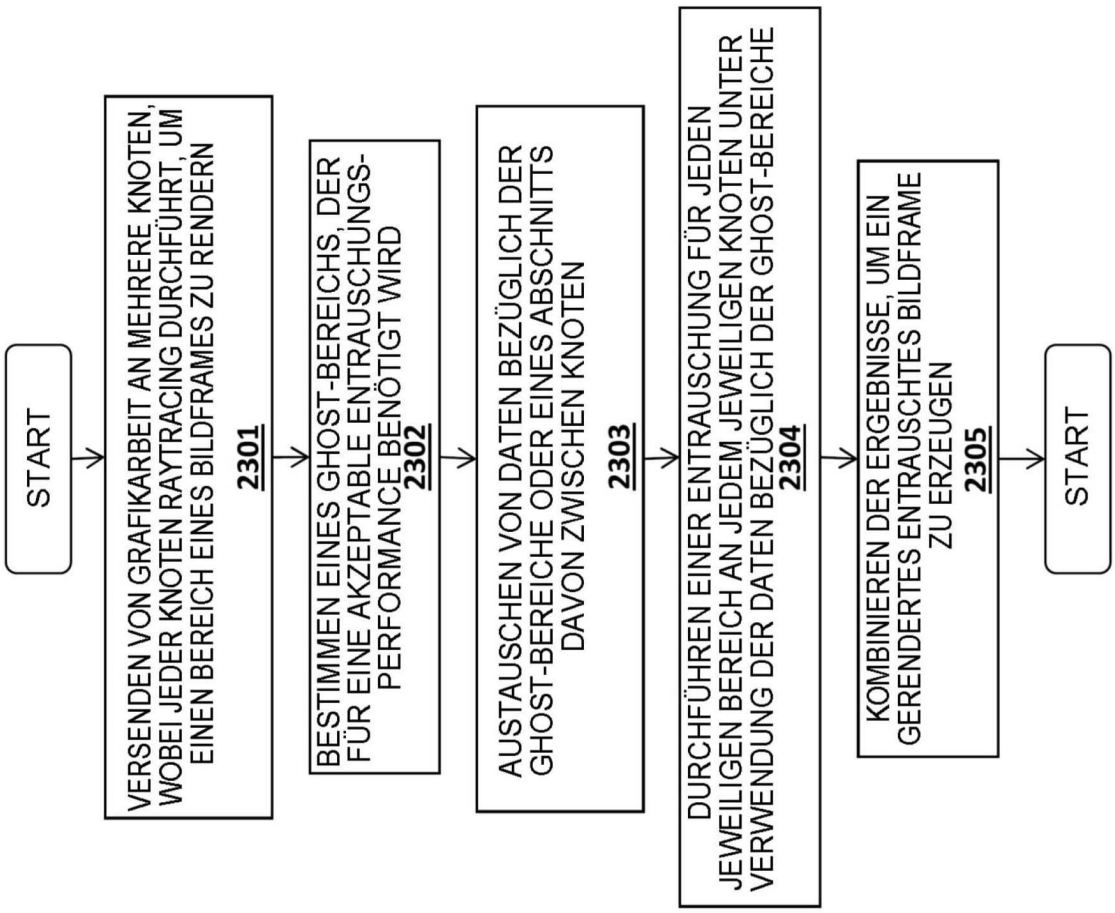

START
VERSENDEN VON GRAFIKARBEIT AN MEHRERE KNOTEN, WOBEI JEDER KNOTEN RAYTRACING DURCHFÜHRT, UM EINEN BEREICH EINES BILDFRAMES ZU RENDERN
2301
BESTIMMEN EINES GHOST-BEREICHS, DER FÜR EINE AKZEPTABLE ENTRAUSCHUNGS-PERFORMANCE BENÖTIGT WIRD
2302
AUSTAUSCHEN VON DATEN BEZÜGLICH DER GHOST-BEREICHE ODER EINES ABSCHNITTS DAVON ZWISCHEN KNOTEN
2303
DURCHFÜHREN EINER ENTRAUSCHUNG FÜR JEDEN JEWEILIGEN BEREICH AN JEDEM JEWEILIGEN KNOTEN UNTER VERWENDUNG DER DATEN BEZÜGLICH DER GHOST-BEREICHE
2304
KOMBINIEREN DER ERGEBNISSE, UM EIN GERENDERTES ENTRAUSCHTES BILDFRAME ZU ERZEUGEN
2305
START

FIG. 23

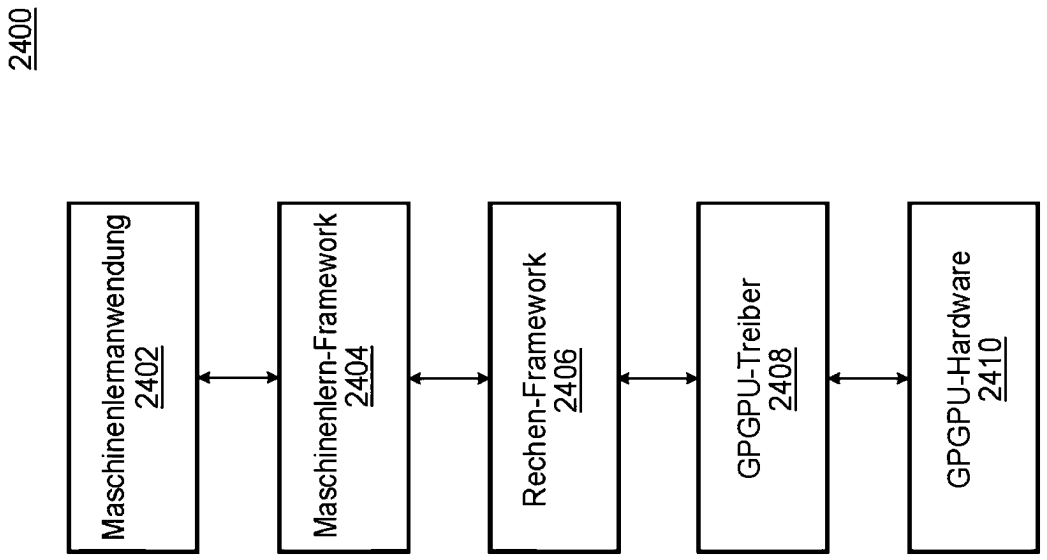

2400
Maschinenlernanwendung
2402
Maschinenlern-Framework
2404
Rechen-Framework
2406
GPGPU-Treiber
2408
GPGPU-Hardware
2410

FIG. 24

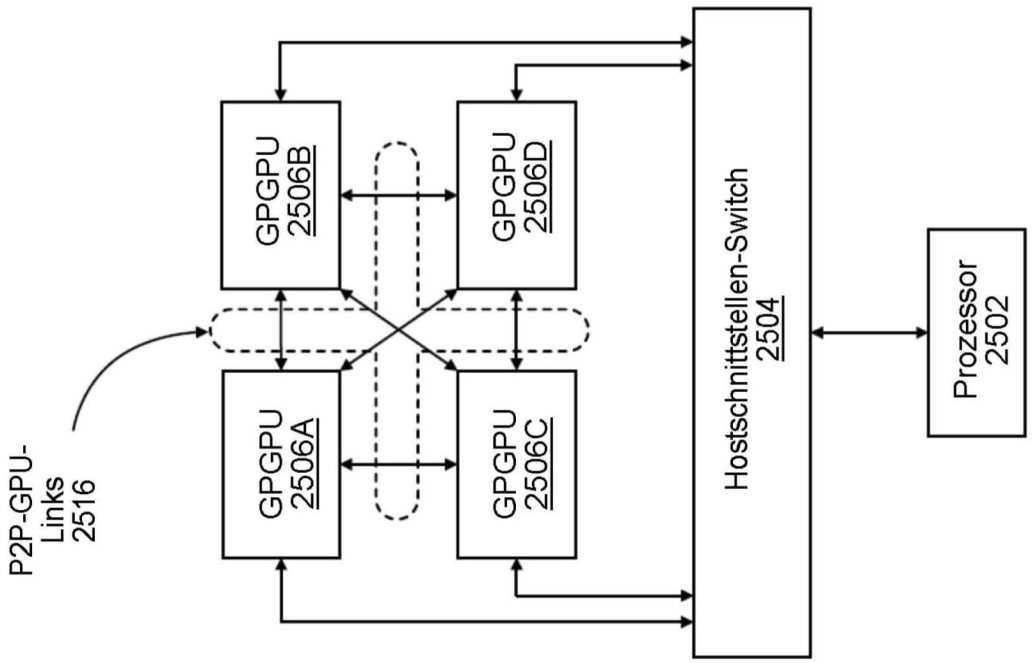

P2P-GPU-Links 2516
GPGPU 2506A
GPGPU 2506B
GPGPU 2506C
GPGPU 2506D
Hostschnittstellen-Switch 2504
Prozessor 2502

FIG. 25

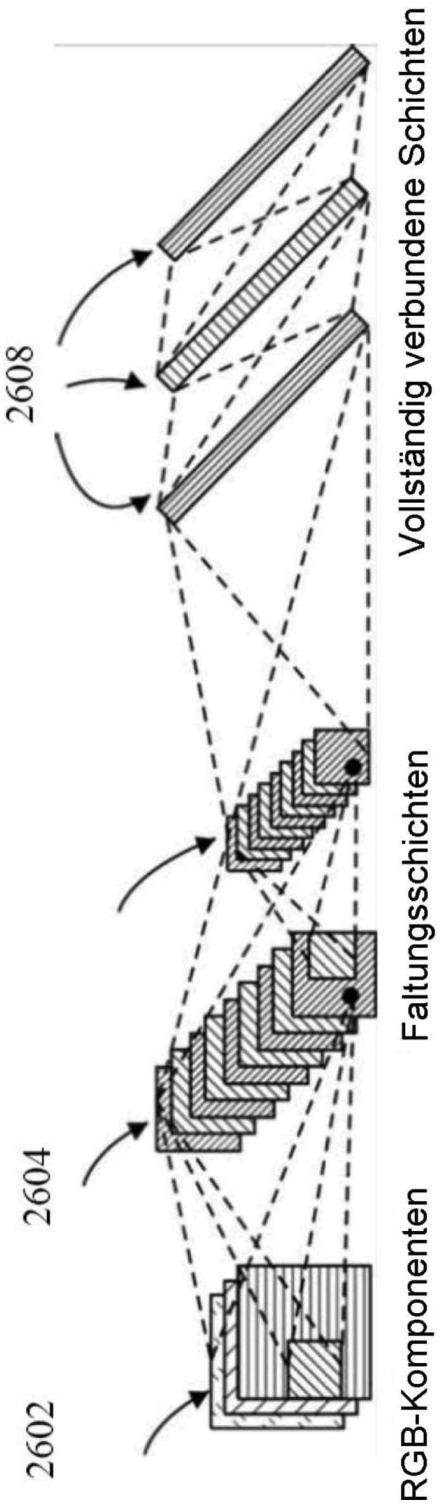

2608
2604
2602
Vollständig verbundene Schichten
Faltungsschichten
RGB-Komponenten

FIG. 26

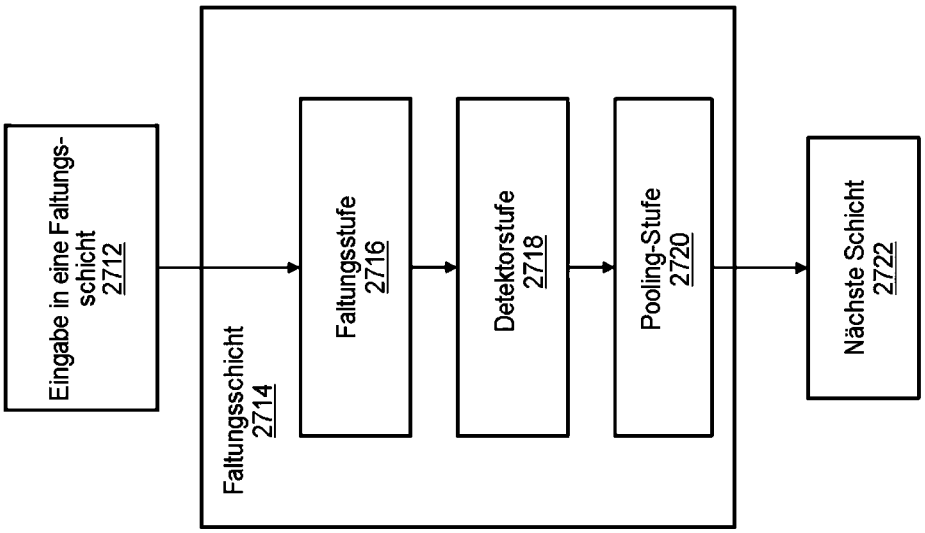

Eingabe in eine Faltungs-schicht 2712
Faltungsschicht 2714
Faltungsstufe 2716
Detektorstufe 2718
Pooling-Stufe 2720
Nächste Schicht 2722

FIG. 27

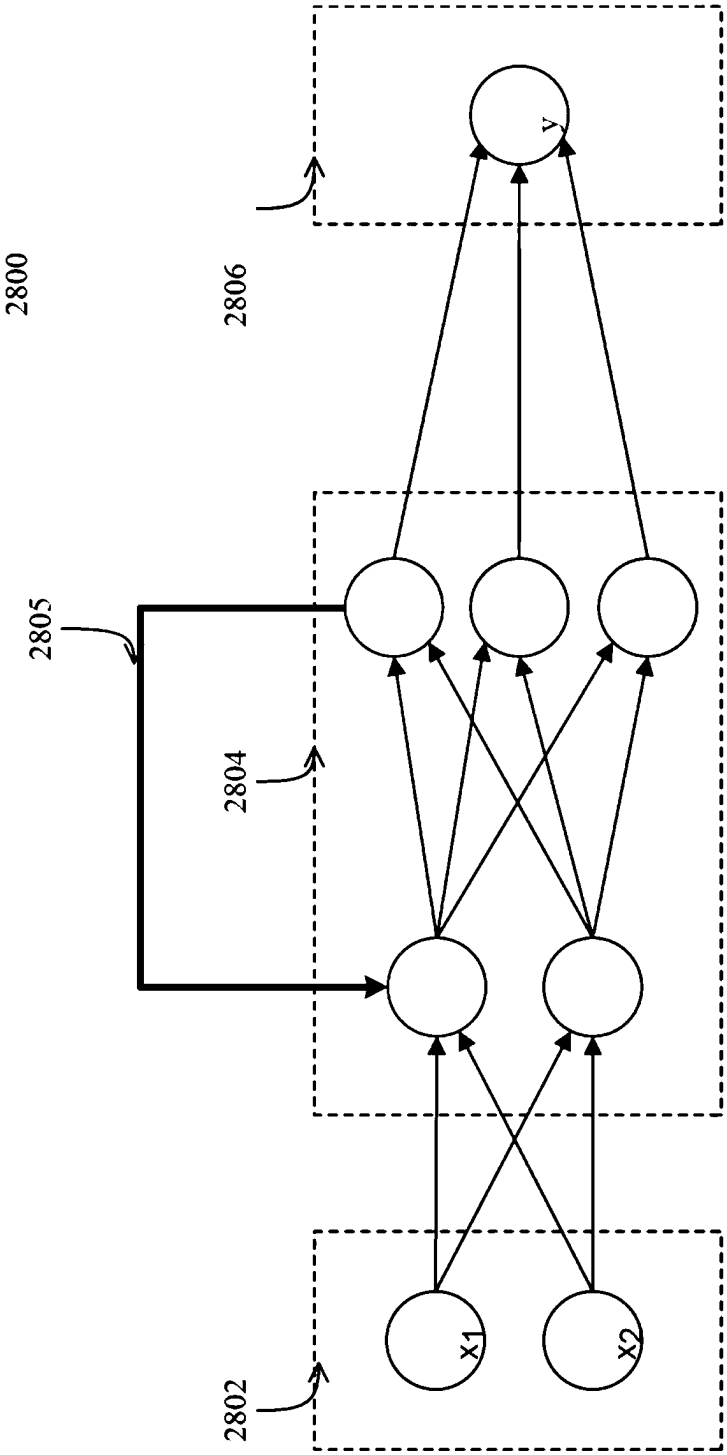


FIG. 28

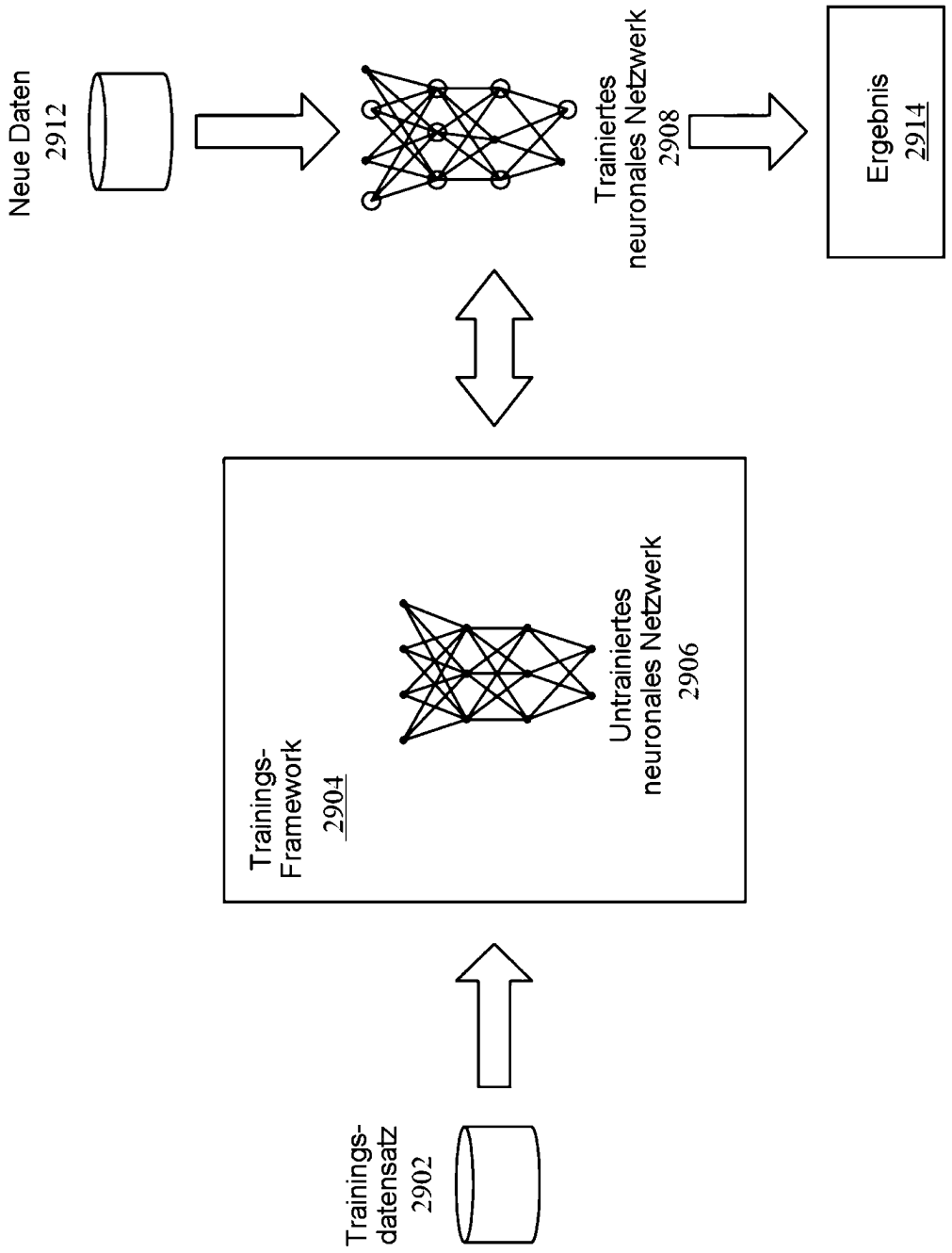

Neue Daten
2912
Trainiertes
neuronales Netzwerk
2908
Ergebnis
2914
Trainings-
Framework
2904
Untrainiertes
neuronales Netzwerk
2906
Trainings-
datensatz
2902

FIG. 29

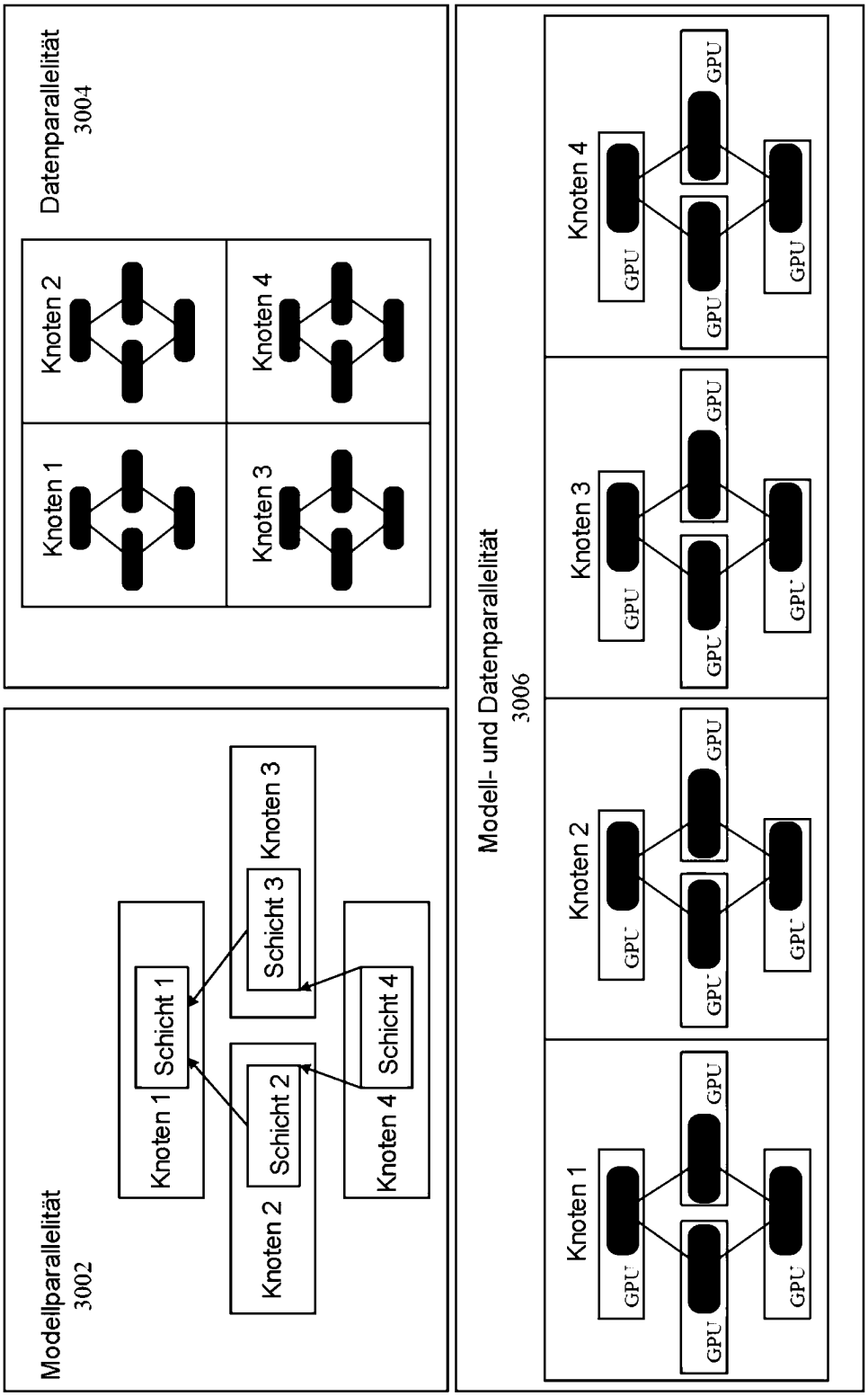

Modellparallelität
3002
Knoten 1
Schicht 1
Knoten 2
Schicht 2
Knoten 3
Schicht 3
Knoten 4
Schicht 4
Datenparallelität
3004
Knoten 1
Knoten 2
Knoten 3
Knoten 4
Modell- und Datenparallelität
3006
Knoten 1
Knoten 2
Knoten 3
Knoten 4
GPU

FIG. 30A

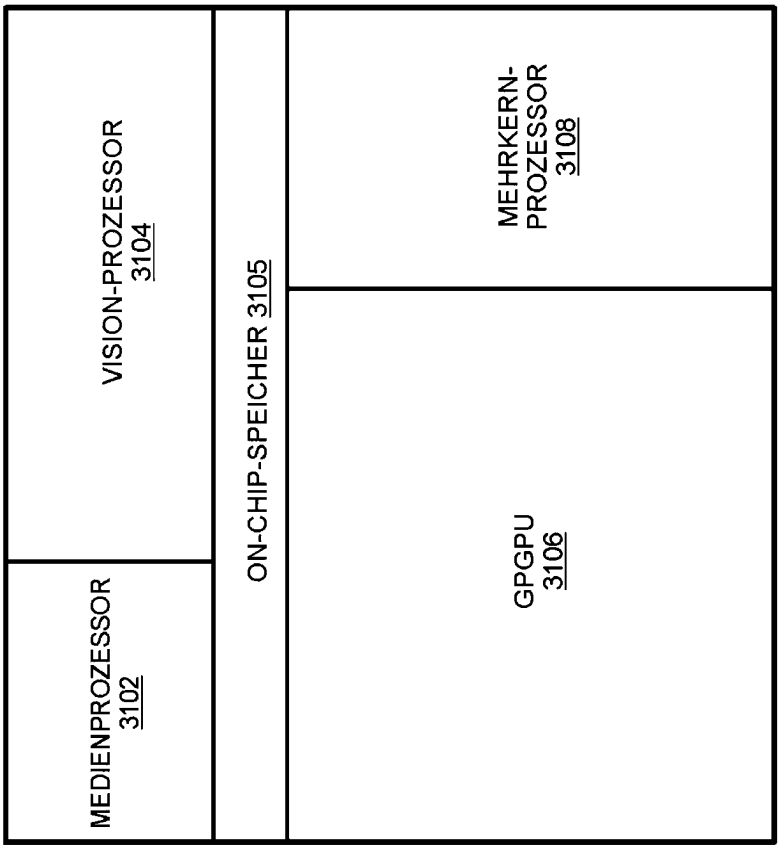

3100
MEDIENPROZESSOR 3102
VISION-PROZESSOR 3104
ON-CHIP-SPEICHER 3105
GPGPU 3106
MEHRKERN-PROZESSOR 3108

FIG. 30B

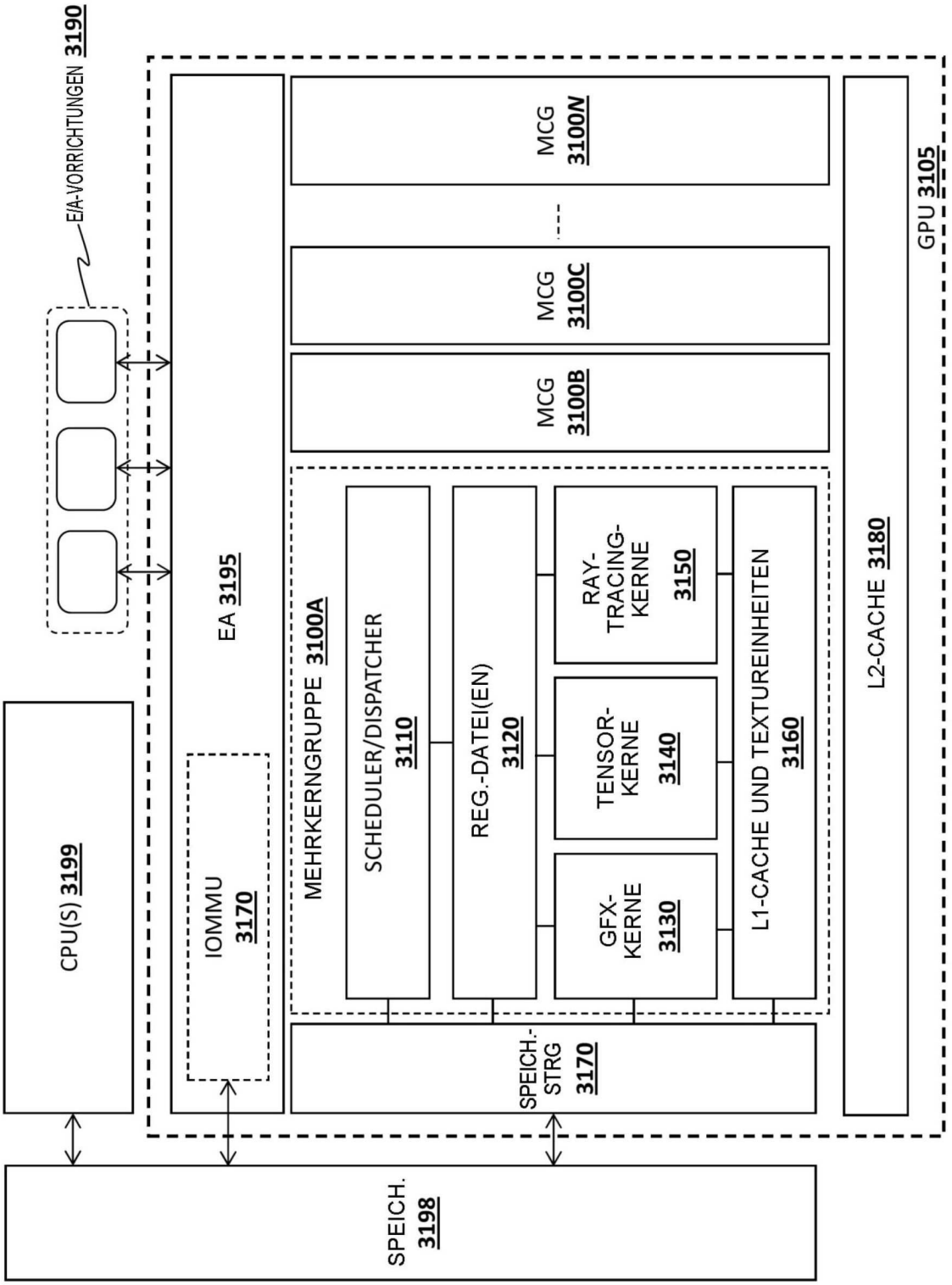

E/A-VORRICHTUNGEN 3190
CPU(S) 3199
IOMMU 3170
EA 3195
GPU 3105
MCG 3100N
MCG 3100C
MCG 3100B
MEHRKERNGRUPPE 3100A
SCHEDULER/DISPATCHER 3110
REG.-DATEI(EN) 3120
RAY-TRACING-KERNE 3150
TENSOR-KERNE 3140
GFX-KERNE 3130
L1-CACHE UND TEXTUREINHEITEN 3160
L2-CACHE 3180
SPEICH.-STRG 3170
SPEICH. 3198

FIG. 31

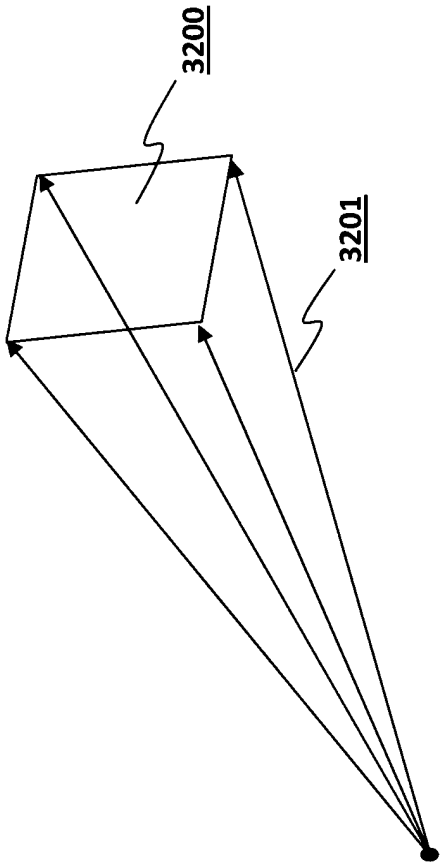


FIG. 32

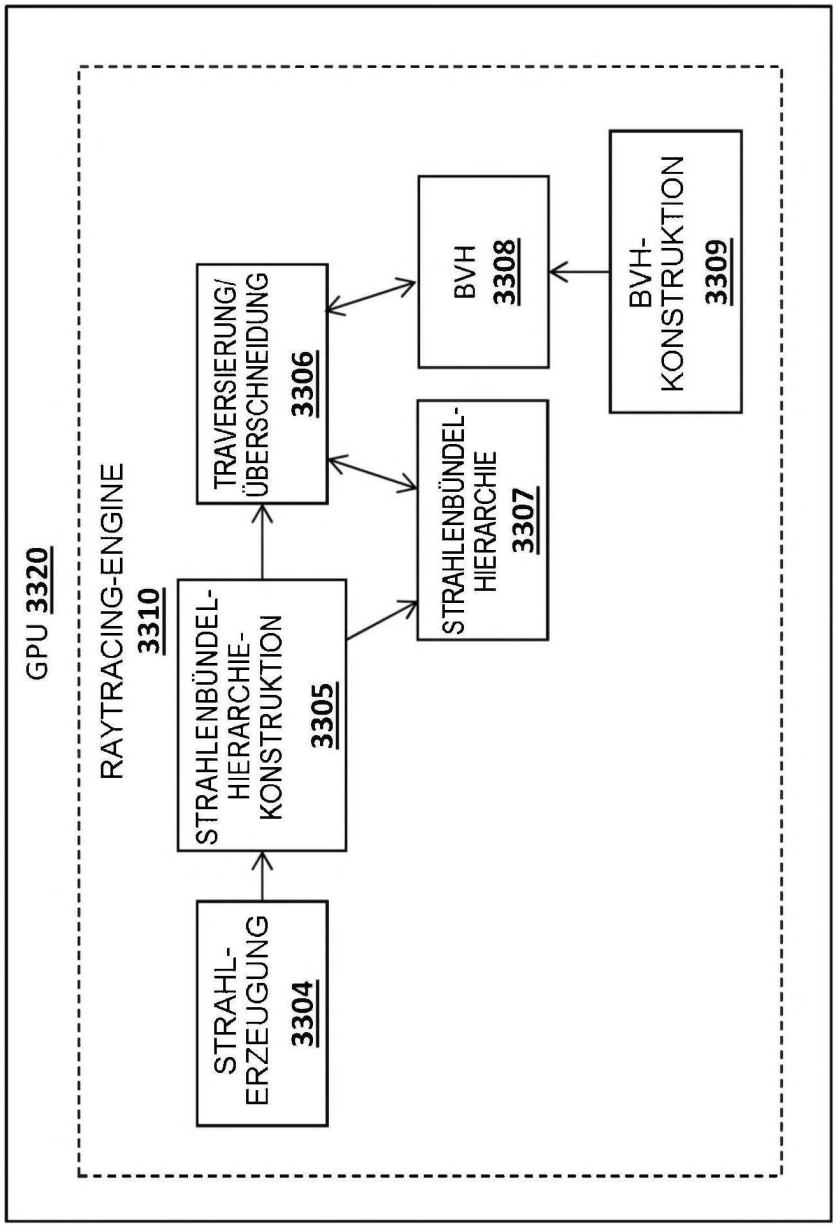

GPU 3320
RAYTRACING-ENGINE 3310
STRAHL-ERZEUGUNG 3304
STRAHLENBÜNDEL-HIERARCHIE-KONSTRUKTION 3305
TRAVERSIERUNG/ ÜBERSCHNEIDUNG 3306
STRAHLENBÜNDEL-HIERARCHIE 3307
BVH 3308
BVH-KONSTRUKTION 3309

FIG. 33

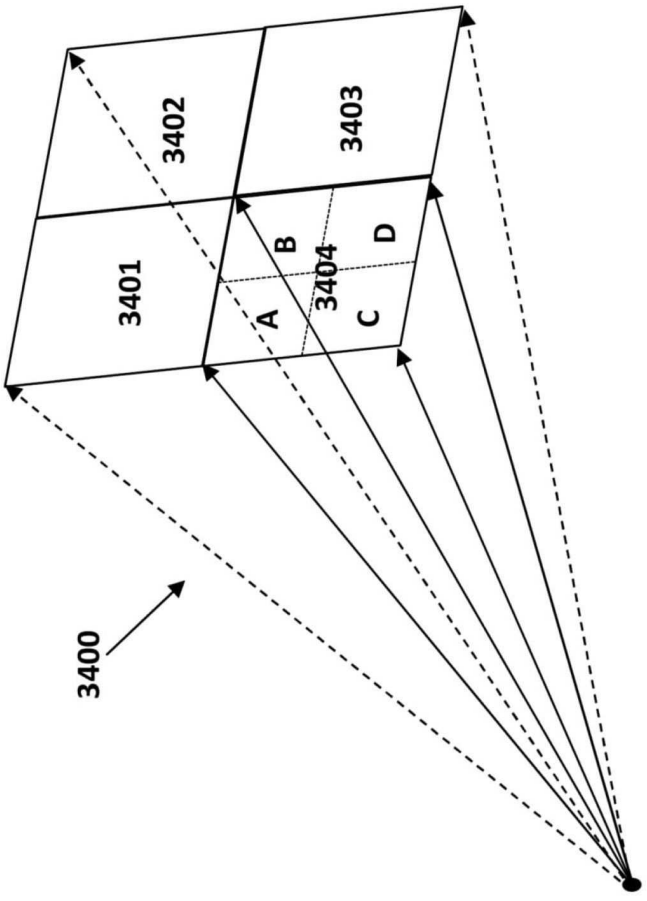


FIG. 34

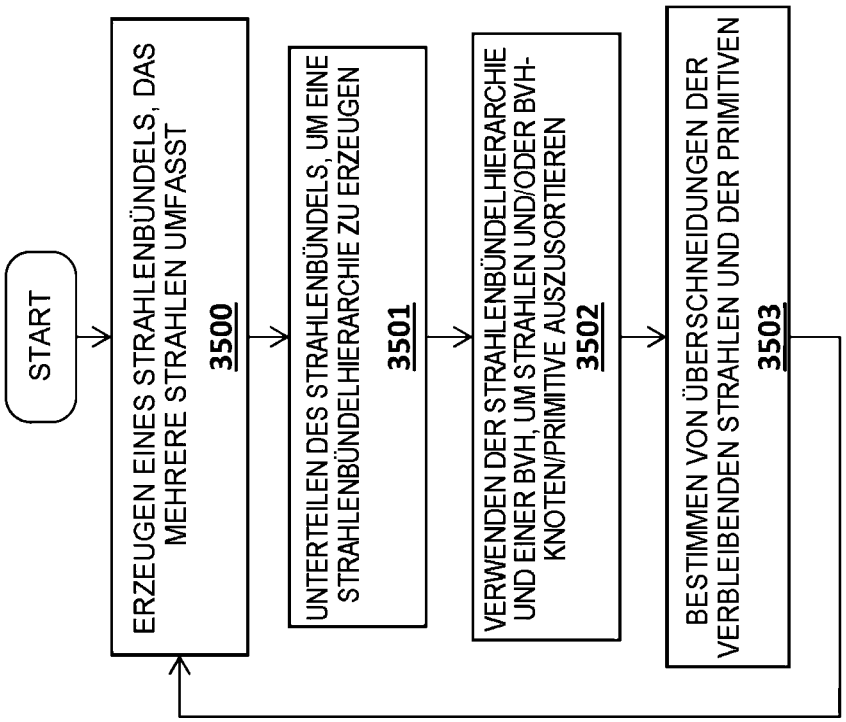

START
ERZEUGEN EINES STRAHLENBÜNDELS, DAS MEHRERE STRAHLEN UMFASST
3500
UNTERTEILEN DES STRAHLENBÜNDELS, UM EINE STRAHLENBÜNDELHIERARCHIE ZU ERZEUGEN
3501
VERWENDEN DER STRAHLENBÜNDELHIERARCHIE UND EINER BVH, UM STRAHLEN UND/ODER BVH-KNOTEN/PRIMITIVE AUSZUSORTIEREN
3502
BESTIMMEN VON ÜBERSCHNEIDUNGEN DER VERBLEIBENDEN STRAHLEN UND DER PRIMITIVEN
3503

FIG. 35

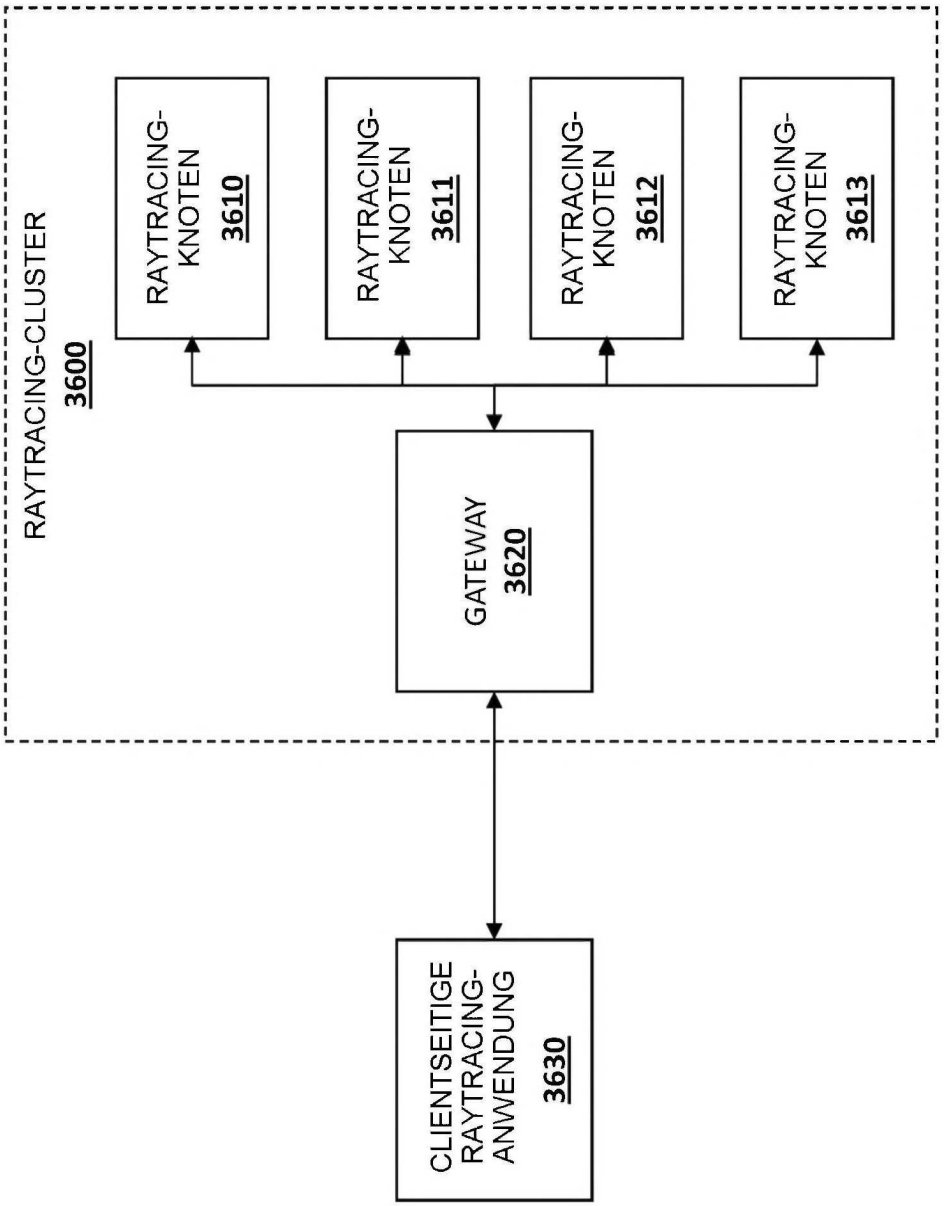

RAYTRACING-CLUSTER
3600
RAYTRACING-KNOTEN
3610
RAYTRACING-KNOTEN
3611
RAYTRACING-KNOTEN
3612
RAYTRACING-KNOTEN
3613
GATEWAY
3620
CLIENTSEITIGE RAYTRACING-ANWENDUNG
3630

FIG. 36

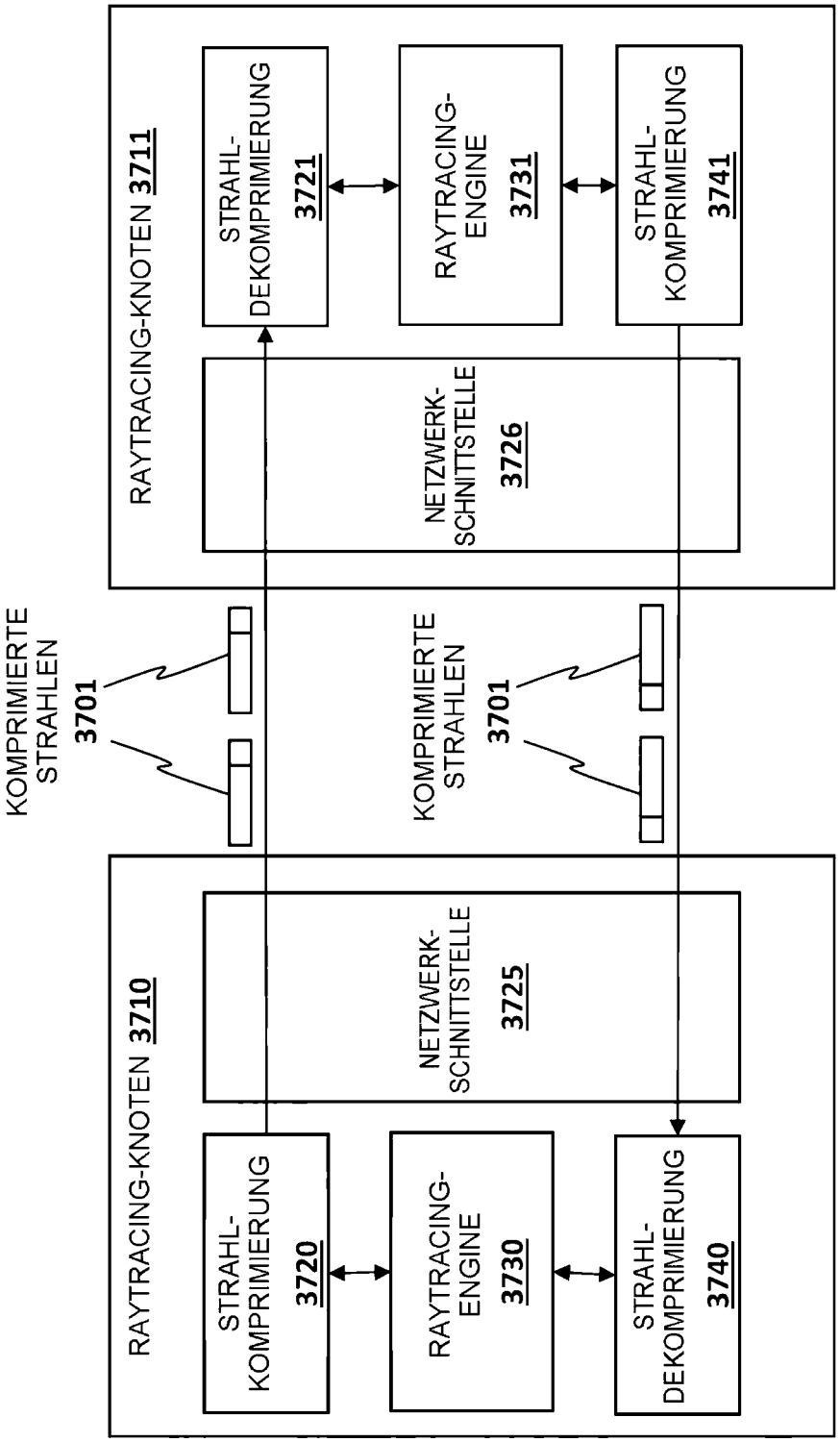

RAYTRACING-KNOTEN 3711
STRAHL-DEKOMPRIMIERUNG 3721
RAYTRACING-ENGINE 3731
STRAHL-KOMPRIMIERUNG 3741
NETZWERK-SCHNITTSTELLE 3726
KOMPRIMIERTE STRAHLEN 3701
KOMPRIMIERTE STRAHLEN 3701
RAYTRACING-KNOTEN 3710
NETZWERK-SCHNITTSTELLE 3725
STRAHL-KOMPRIMIERUNG 3720
RAYTRACING-ENGINE 3730
STRAHL-DEKOMPRIMIERUNG 3740

FIG. 37

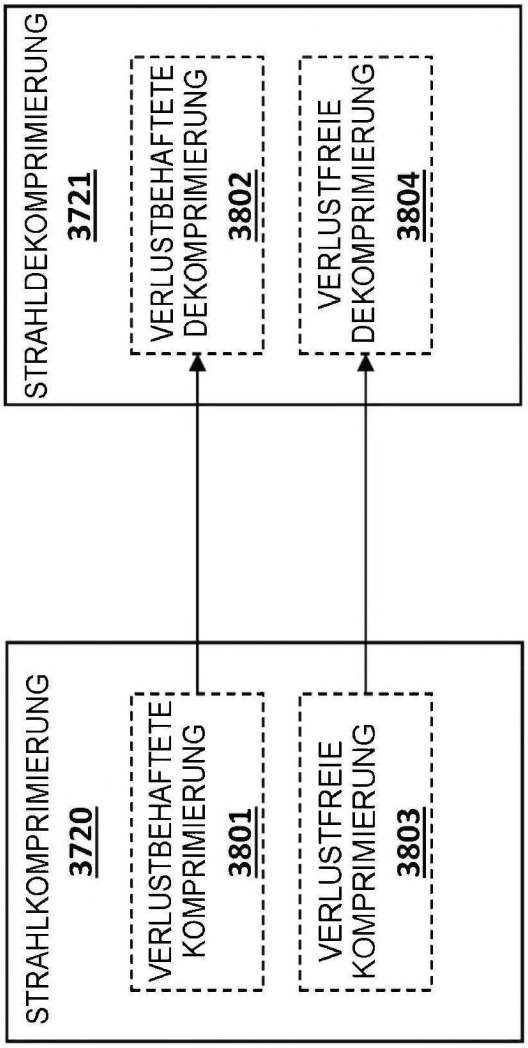

STRAHLDEKOMPRIMIERUNG
3721
VERLUSTBEHAFTETE DEKOMPRIMIERUNG
3802
VERLUSTFREIE DEKOMPRIMIERUNG
3804
STRAHLKOMPRIMIERUNG
3720
VERLUSTBEHAFTETE KOMPRIMIERUNG
3801
VERLUSTFREIE KOMPRIMIERUNG
3803

FIG. 38

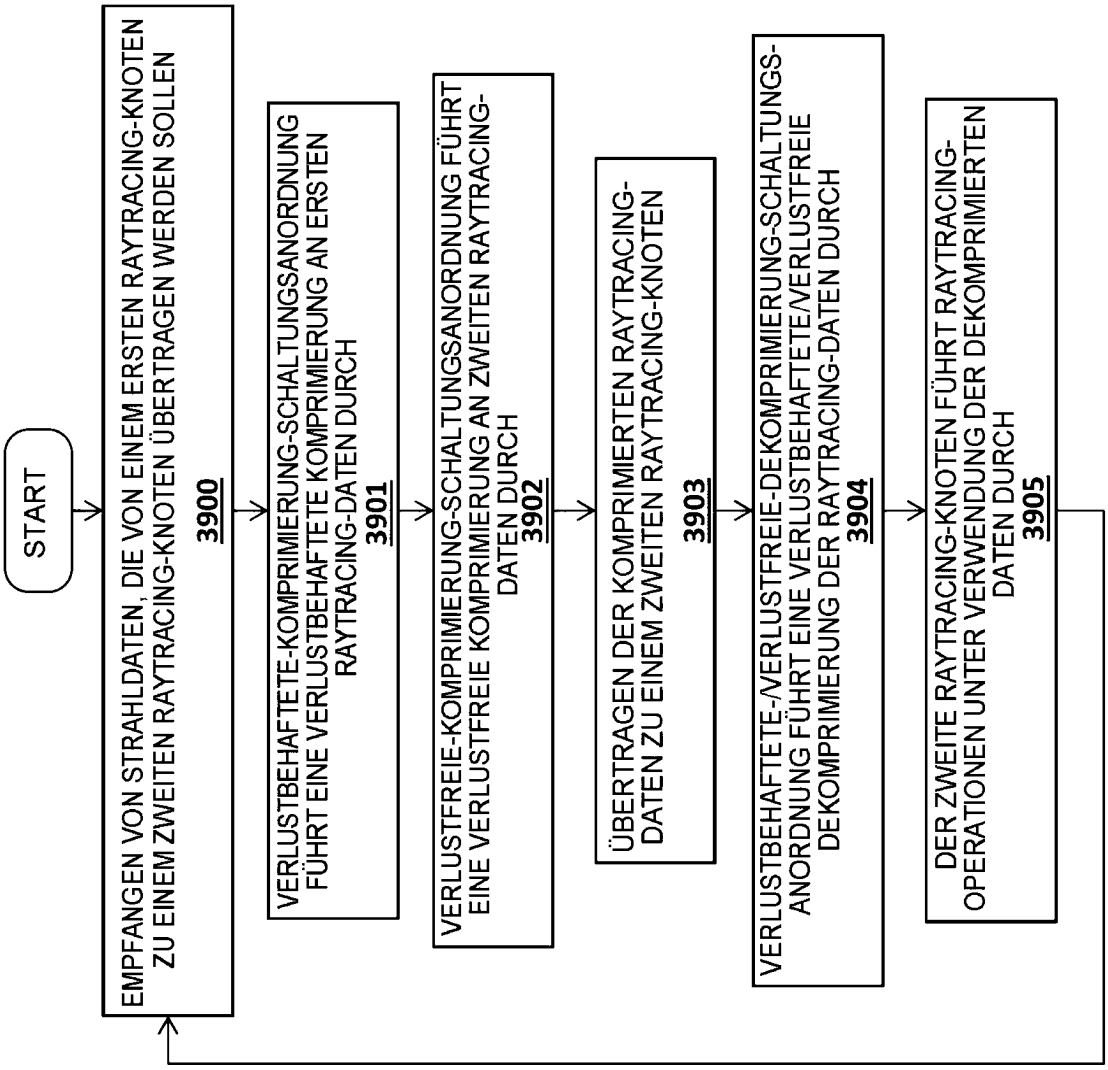

START
EMPFANGEN VON STRAHLDATEN, DIE VON EINEM ERSTEN RAYTRACING-KNOTEN ZU EINEM ZWEITEN RAYTRACING-KNOTEN ÜBERTRAGEN WERDEN SOLLEN
3900
VERLUSTBEHAFTETE-KOMPRIMIERUNG-SCHALTUNGSANORDNUNG FÜHRT EINE VERLUSTBEHAFTETE KOMPRIMIERUNG AN ERSTEN RAYTRACING-DATEN DURCH
3901
VERLUSTFREIE-KOMPRIMIERUNG-SCHALTUNGSANORDNUNG FÜHRT EINE VERLUSTFREIE KOMPRIMIERUNG AN ZWEITEN RAYTRACING-DATEN DURCH
3902
ÜBERTRAGEN DER KOMPRIMIERTEN RAYTRACING-DATEN ZU EINEM ZWEITEN RAYTRACING-KNOTEN
3903
VERLUSTBEHAFTETE-/VERLUSTFREIE-DEKOMPRIMIERUNG-SCHALTUNGS-ANORDNUNG FÜHRT EINE VERLUSTBEHAFTETE/VERLUSTFREIE DEKOMPRIMIERUNG DER RAYTRACING-DATEN DURCH
3904
DER ZWEITE RAYTRACING-KNOTEN FÜHRT RAYTRACING-OPERATIONEN UNTER VERWENDUNG DER DEKOMPRIMIERTEN DATEN DURCH
3905

FIG. 39

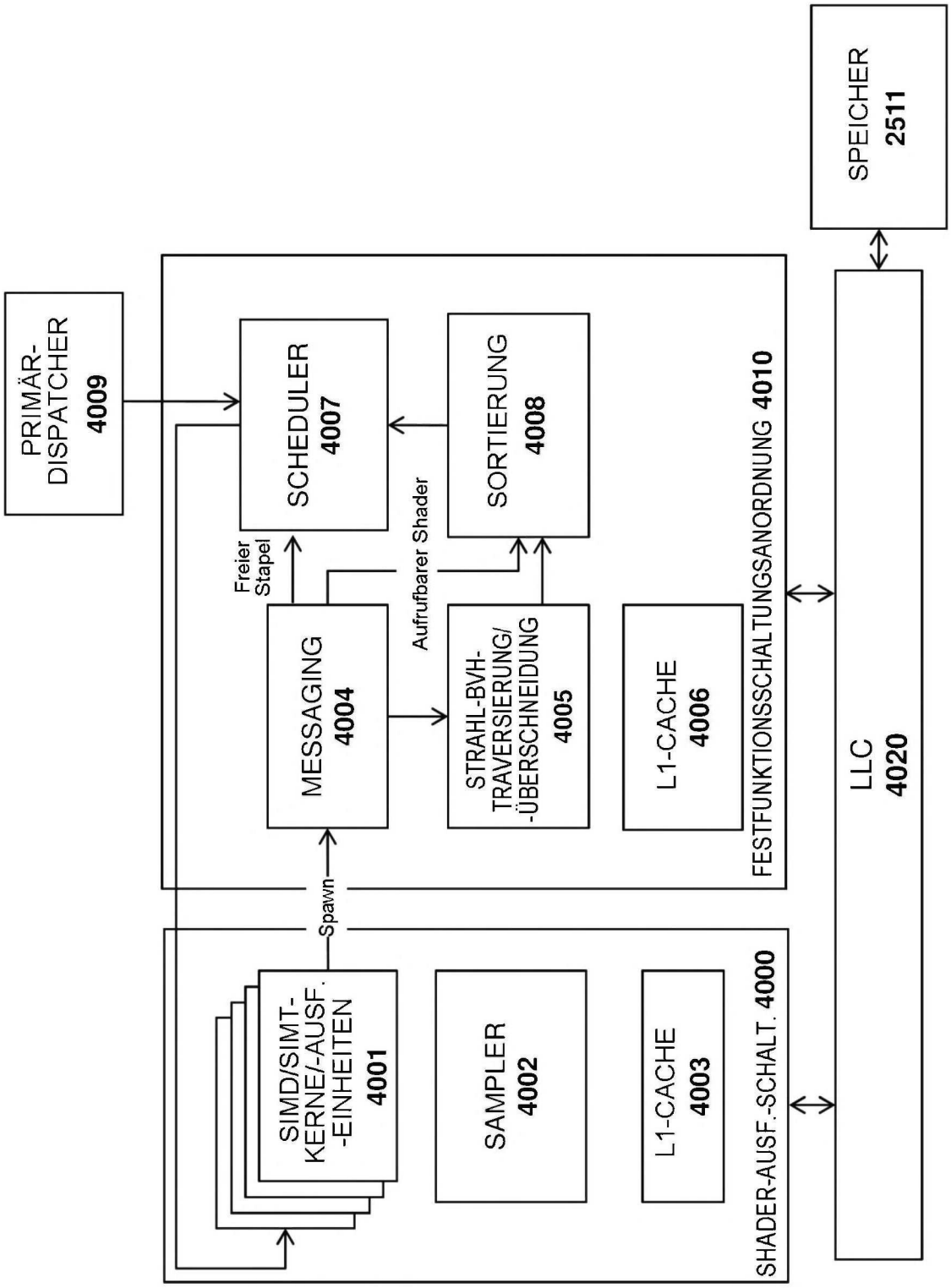

PRIMÄR-DISPATCHER 4009
SCHEDULER 4007
SORTIERUNG 4008
Freier Stapel
Aufrufbarer Shader
MESSAGING 4004
STRAHL-BVH-TRAVERSIERUNG/-ÜBERSCHNEIDUNG 4005
L1-CACHE 4006
FESTFUNKTIONSSCHALTUNGSANORDNUNG 4010
SPEICHER 2511
LLC 4020
Spawn
SIMD/SIMT-KERNE/-AUSF.-EINHEITEN 4001
SAMPLER 4002
L1-CACHE 4003
SHADER-AUSF.-SCHALT. 4000

FIG. 40

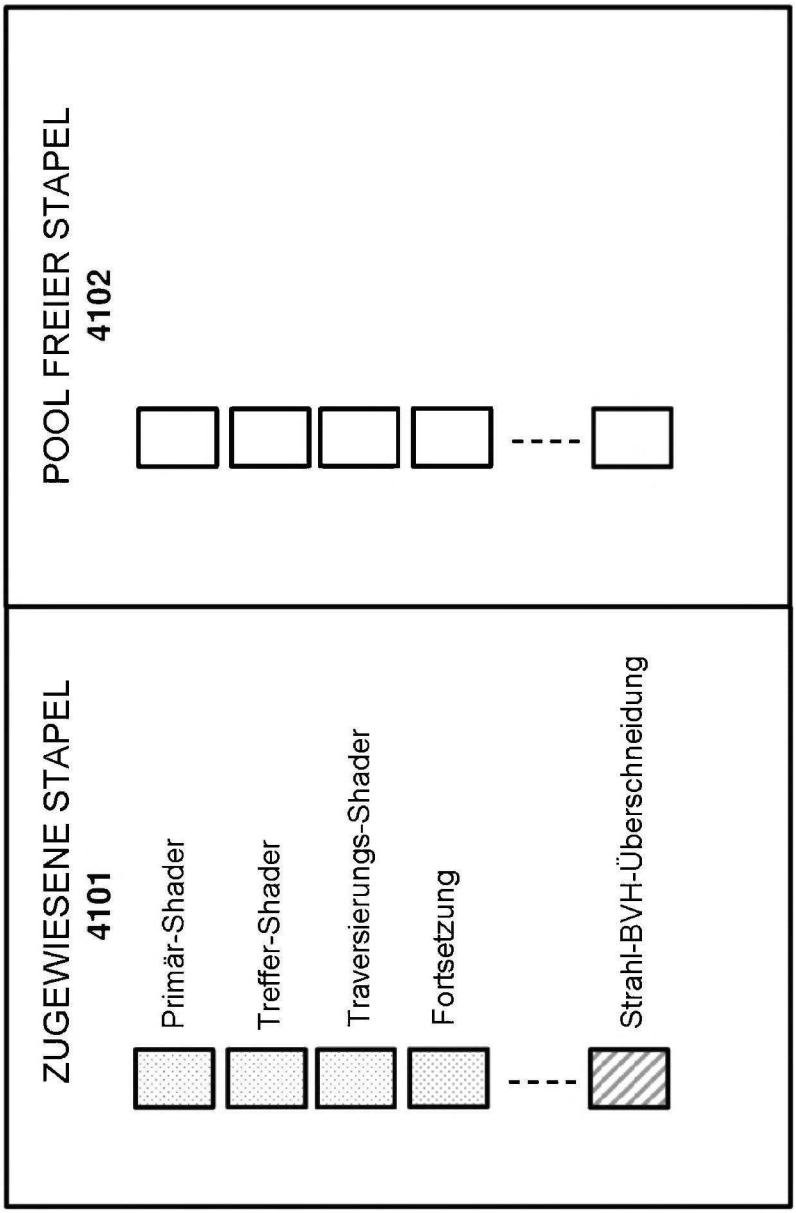

POOL FREIER STAPEL
4102
ZUGEWIESENE STAPEL
4101
Primär-Shader
Treffer-Shader
Traversierungs-Shader
Fortsetzung
Strahl-BVH-Überschneidung
AUFRUFSTAPELREFERENZEN

FIG. 41

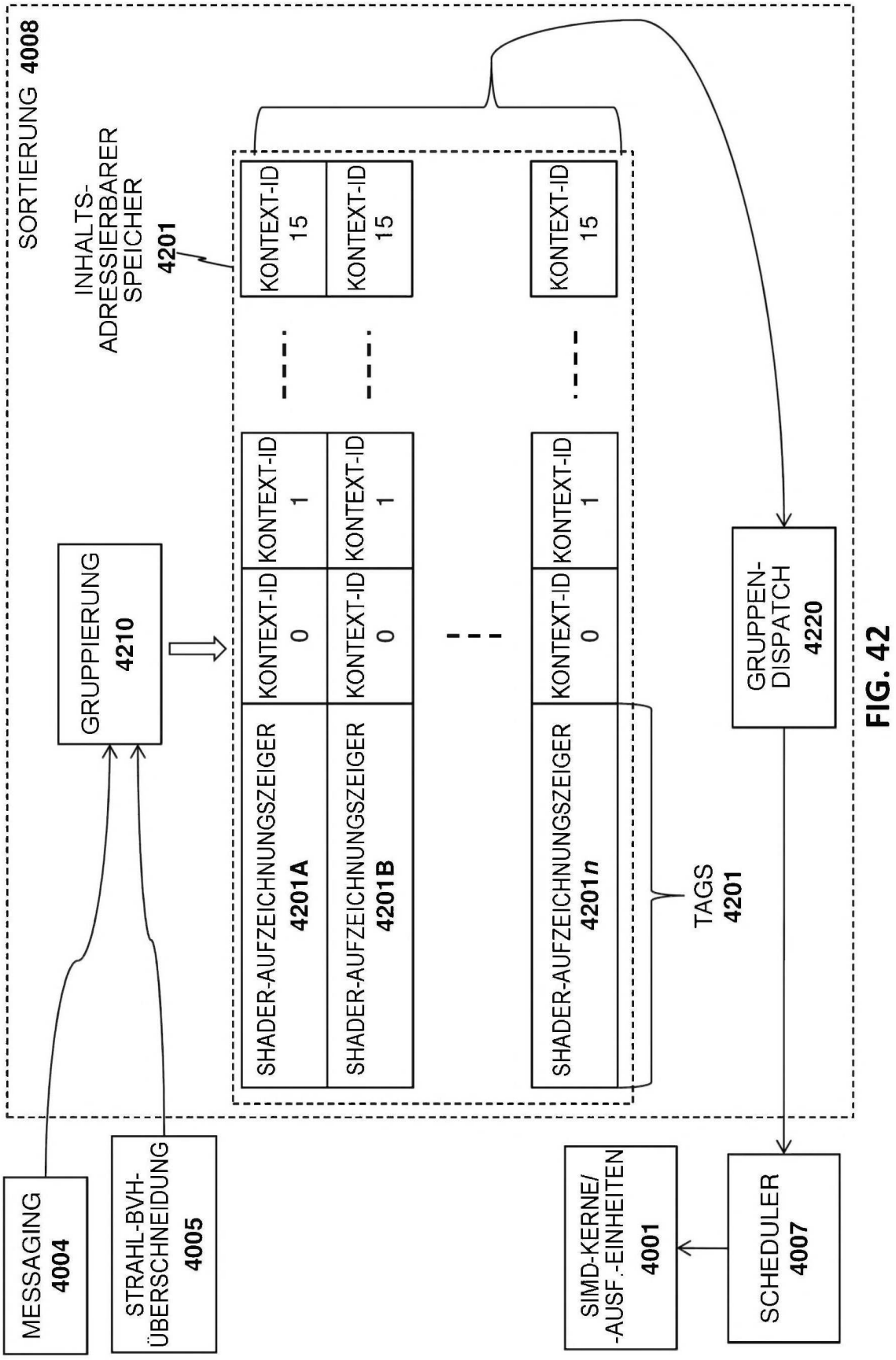

SORTIERUNG 4008
INHALTS-ADRESSIERBARER SPEICHER 4201
GRUPPIERUNG 4210
SHADER-AUFZEICHNUNGSZEIGER 4201A
SHADER-AUFZEICHNUNGSZEIGER 4201B
SHADER-AUFZEICHNUNGSZEIGER 4201n
KONTEXT-ID 0
KONTEXT-ID 1
KONTEXT-ID 15
TAGS 4201
GRUPPEN-DISPATCH 4220
MESSAGING 4004
STRAHL-BVH-ÜBERSCHNEIDUNG 4005
SIMD-KERNE/-AUSF.-EINHEITEN 4001
SCHEDULER 4007

FIG. 42

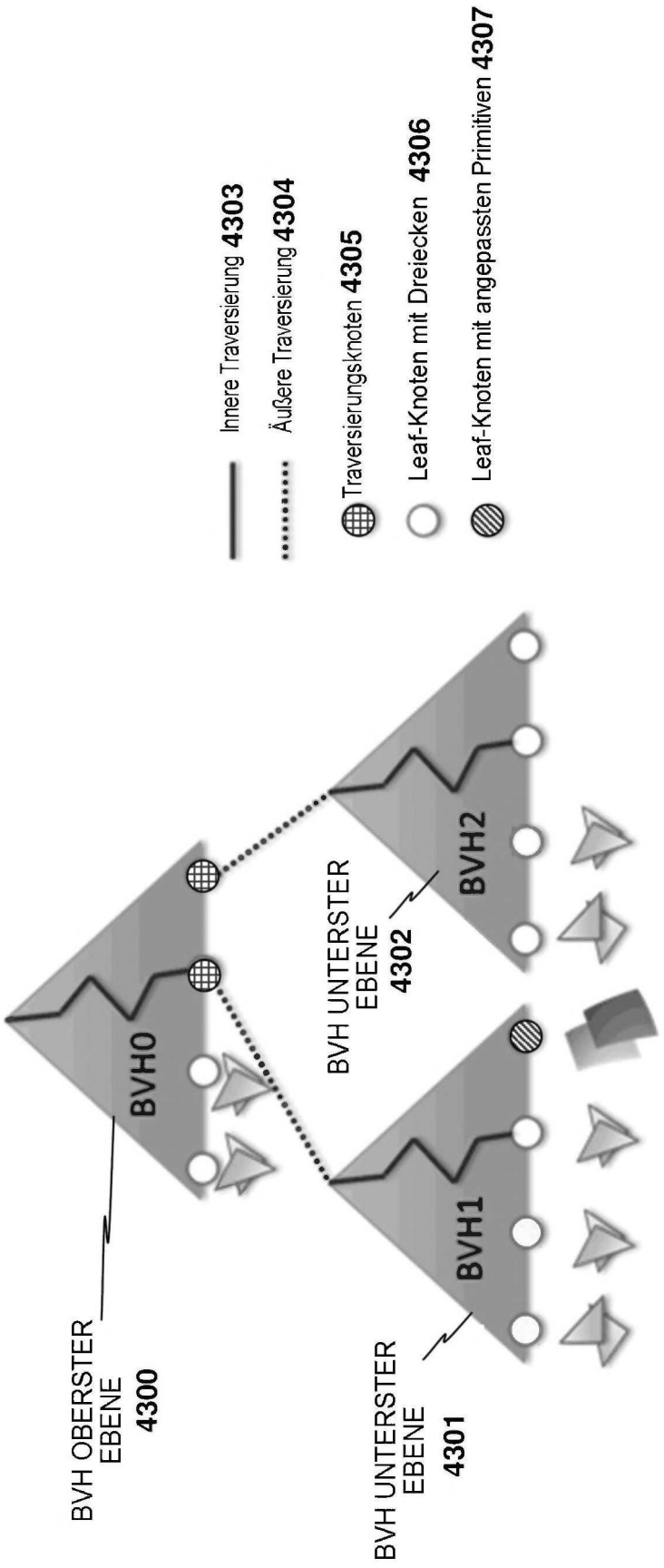

Innere Traversierung 4303
Äußere Traversierung 4304
Traversierungsknoten 4305
Leaf-Knoten mit Dreiecken 4306
Leaf-Knoten mit angepassten Primitiven 4307
BVH OBERSTER EBENE 4300
BVH UNTERSTER EBENE 4301
BVH UNTERSTER EBENE 4302
BVH0
BVH1
BVH2

FIG. 43

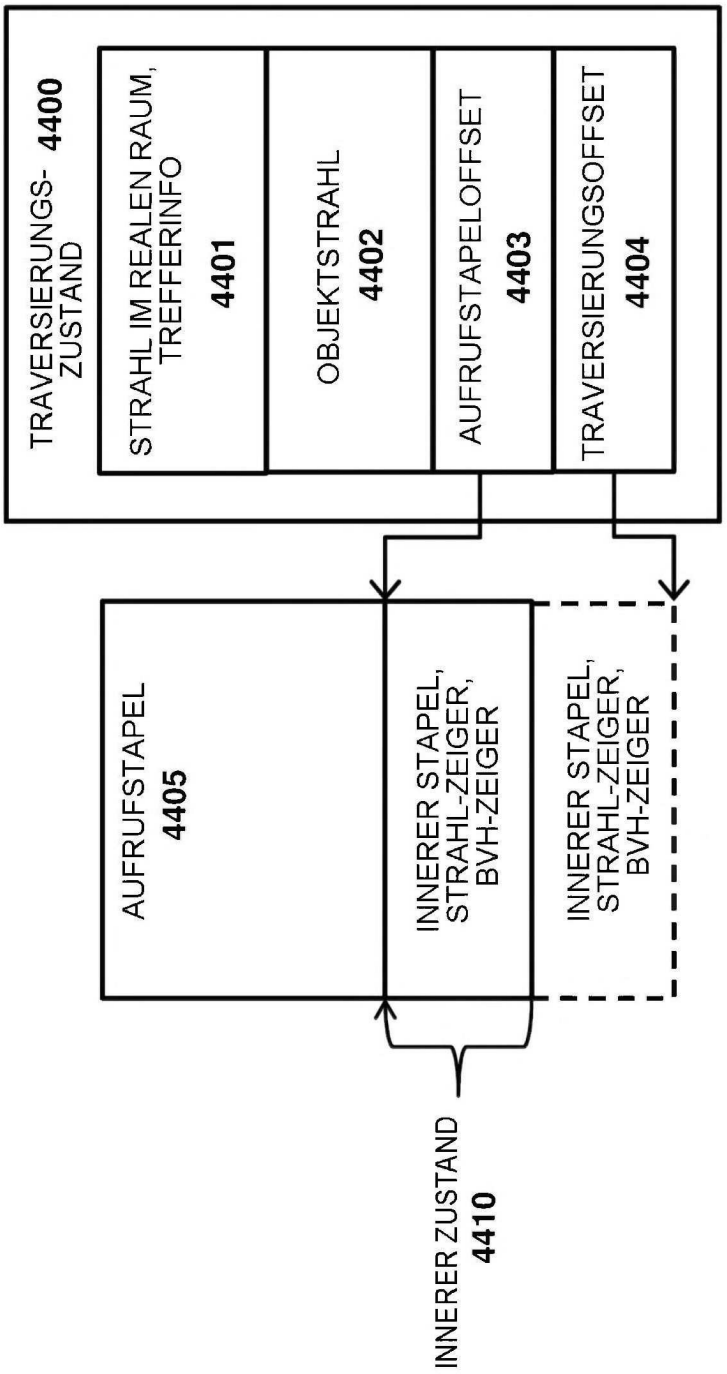

TRAVERSIERUNGS-ZUSTAND 4400
STRAHL IM REALEN RAUM, TREFFERINFO 4401
OBJEKTSTRAHL 4402
AUFRUFSTAPELOFFSET 4403
TRAVERSIERUNGSOFFSET 4404
AUFRUFSTAPEL 4405
INNERER STAPEL, STRAHL-ZEIGER, BVH-ZEIGER
INNERER STAPEL, STRAHL-ZEIGER, BVH-ZEIGER
INNERER ZUSTAND 4410

FIG. 44

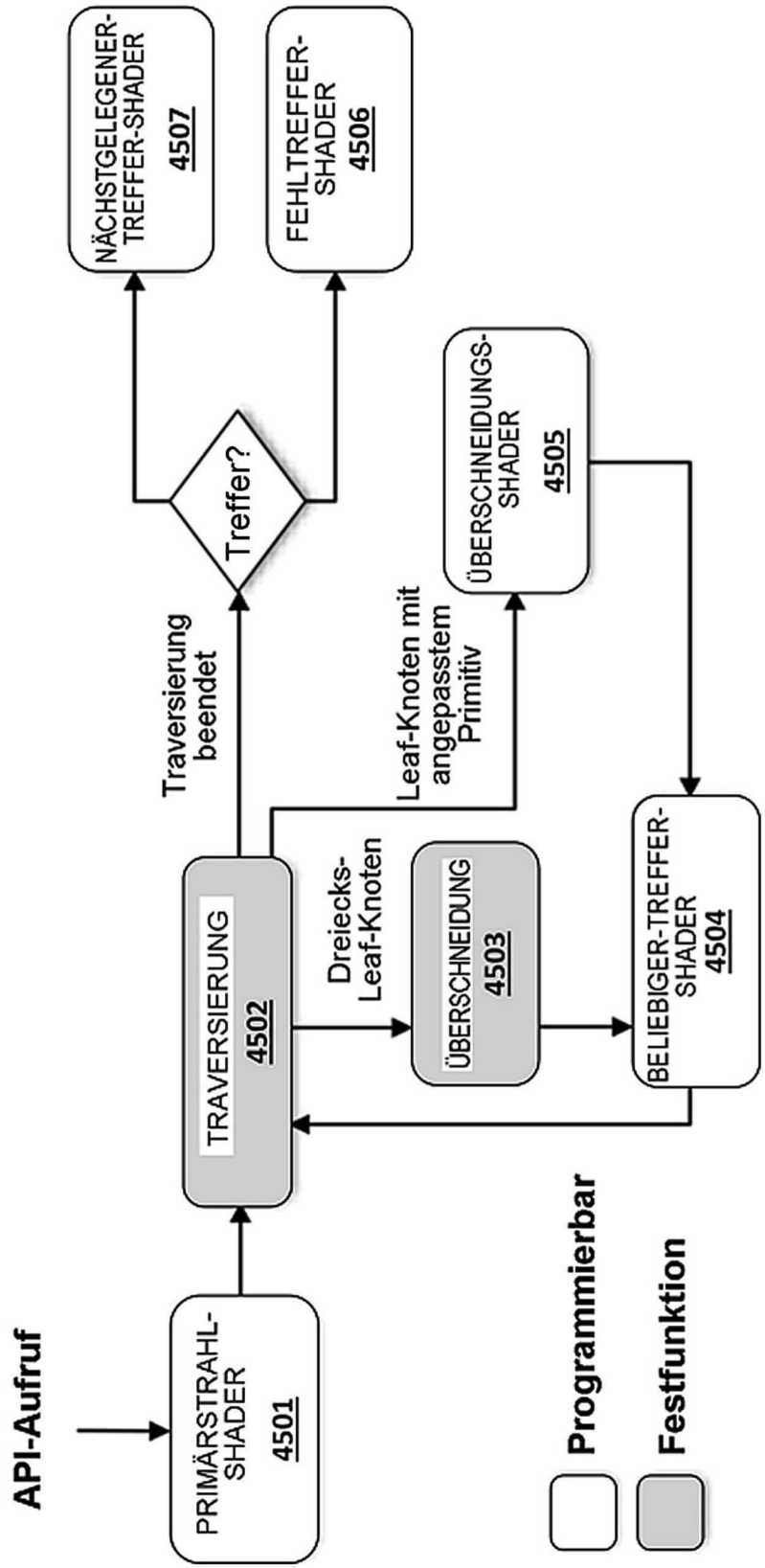

API-Aufruf
PRIMÄRSTRAHL-SHADER 4501
TRAVERSIERUNG 4502
Dreiecks-Leaf-Knoten
ÜBERSCHNEIDUNG 4503
BELIEBIGER-TREFFER-SHADER 4504
Leaf-Knoten mit angepasstem Primitiv
ÜBERSCHNEIDUNGS-SHADER 4505
Traversierung beendet
Treffer?
NÄCHSTGELEGENER-TREFFER-SHADER 4507
FEHLTREFFER-SHADER 4506
Programmierbar
Festfunktion

FIG. 45

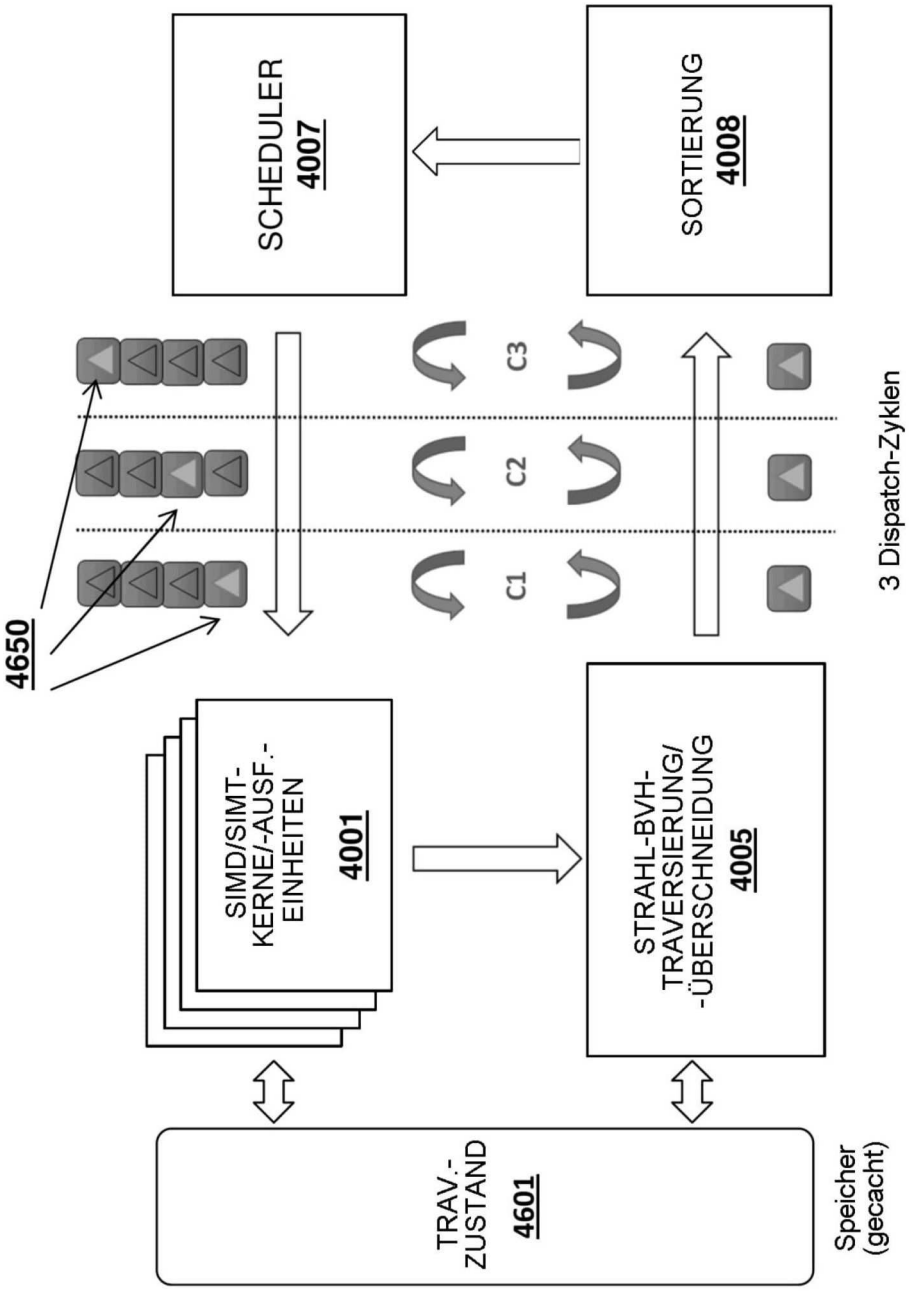


FIG. 46B

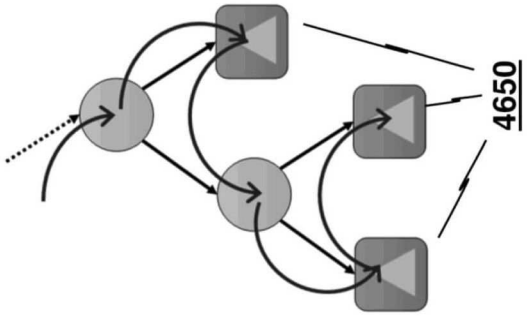


FIG. 46A

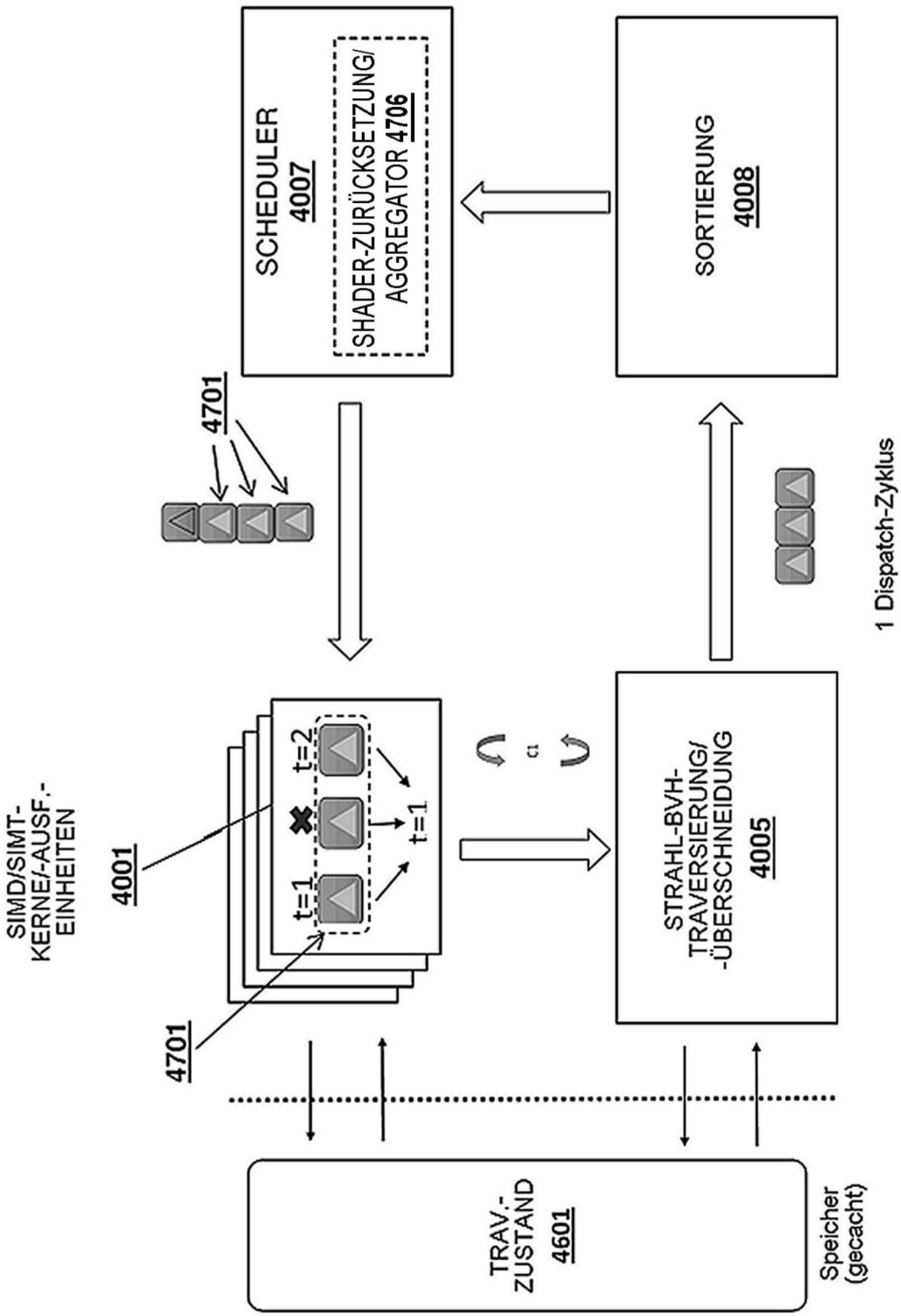

SIMD/SIMT-KERNE/-AUSF.-EINHEITEN
4001
4701
t=1
t=2
t=1
4701
SCHEDULER 4007
SHADER-ZURÜCKSETZUNG/ AGGREGATOR 4706
SORTIERUNG 4008
STRAHL-BVH-TRAVERSIERUNG/ -ÜBERSCHNEIDUNG 4005
TRAV.-ZUSTAND 4601
Speicher (gecacht)
1 Dispatch-Zyklus

FIG. 47

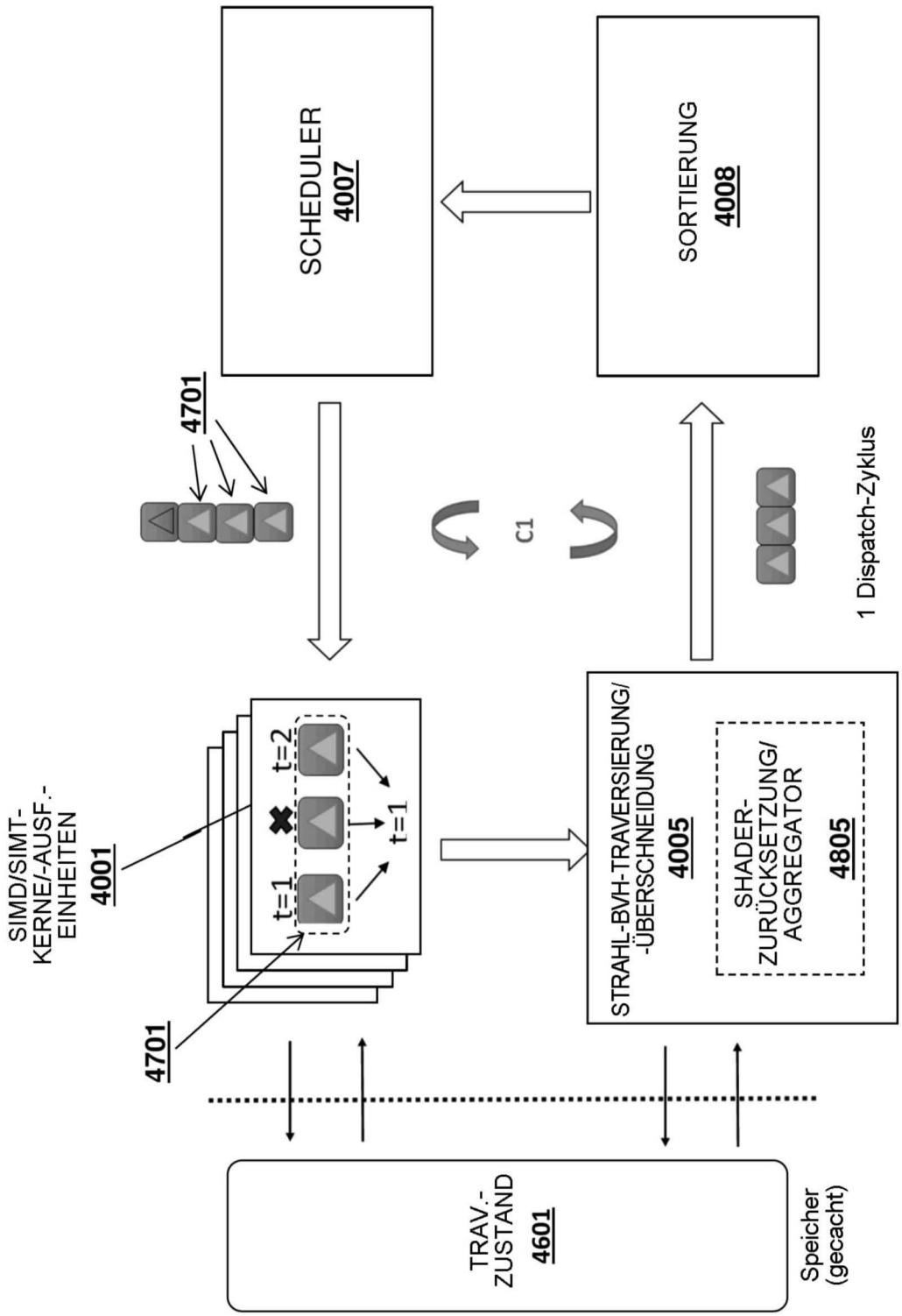

SCHEDULER
4007
SORTIERUNG
4008
4701
C1
1 Dispatch-Zyklus
SIMD/SIMT-KERNE/-AUSF.-EINHEITEN
4001
t=1
t=2
t=1
4701
STRAHL-BVH-TRAVERSIERUNG/-ÜBERSCHNEIDUNG
4005
SHADER-ZURÜCKSETZUNG/AGGREGATOR
4805
TRAV.-ZUSTAND
4601
Speicher (gecacht)

FIG. 48

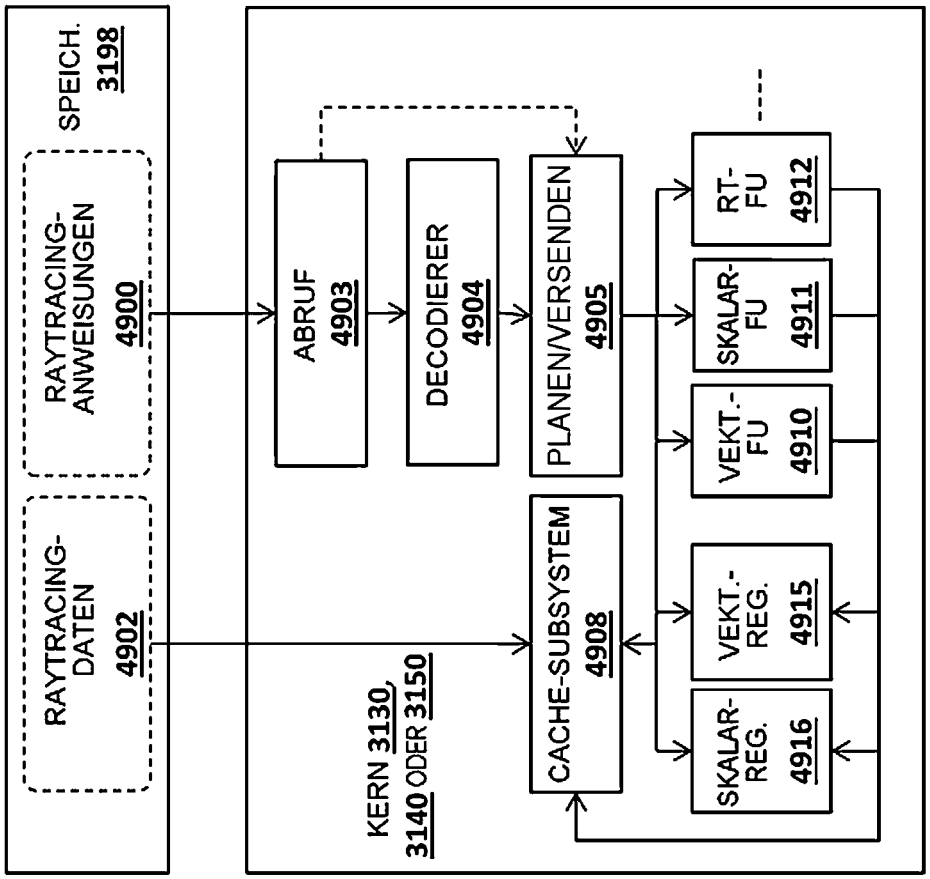

SPEICH. 3198
RAYTRACING-ANWEISUNGEN 4900
RAYTRACING-DATEN 4902
KERN 3130, 3140 ODER 3150
ABRUF 4903
DECODIERER 4904
PLANEN/VERSENDEN 4905
CACHE-SUBSYSTEM 4908
RT-FU 4912
SKALAR-FU 4911
VEKT.-FU 4910
VEKT.-REG. 4915
SKALAR-REG. 4916

FIG. 49

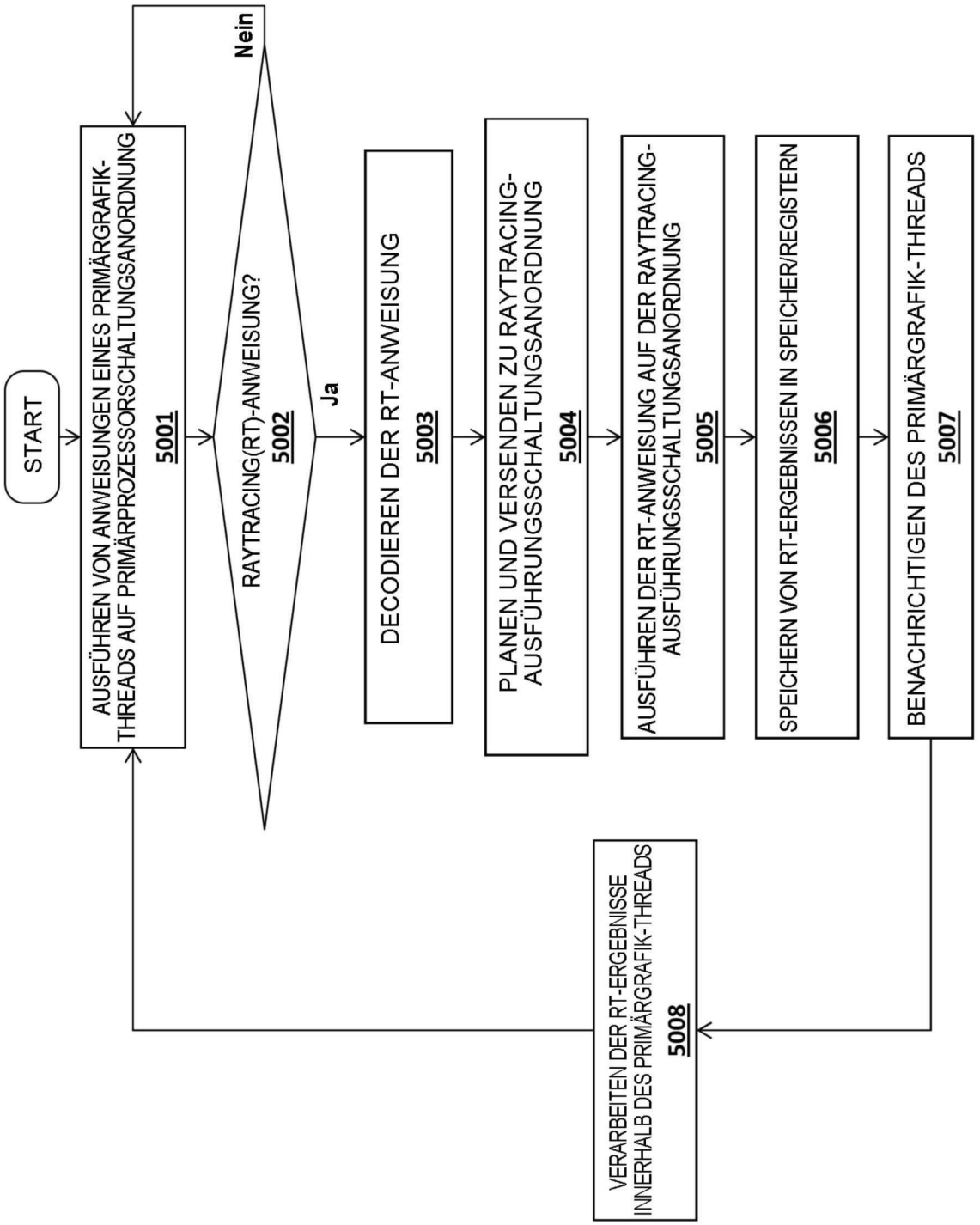

START
AUSFÜHREN VON ANWEISUNGEN EINES PRIMÄRGRAFIK-THREADS AUF PRIMÄRPROZESSORSCHALTUNGSANORDNUNG
5001
RAYTRACING(RT)-ANWEISUNG?
5002
Nein
Ja
DECODIEREN DER RT-ANWEISUNG
5003
PLANEN UND VERSENDEN ZU RAYTRACING-AUSFÜHRUNGSSCHALTUNGSANORDNUNG
5004
AUSFÜHREN DER RT-ANWEISUNG AUF DER RAYTRACING-AUSFÜHRUNGSSCHALTUNGSANORDNUNG
5005
SPEICHERN VON RT-ERGEBNISSEN IN SPEICHER/REGISTERN
5006
BENACHRICHTIGEN DES PRIMÄRGRAFIK-THREADS
5007
VERARBEITEN DER RT-ERGEBNISSE INNERHALB DES PRIMÄRGRAFIK-THREADS
5008

FIG. 50

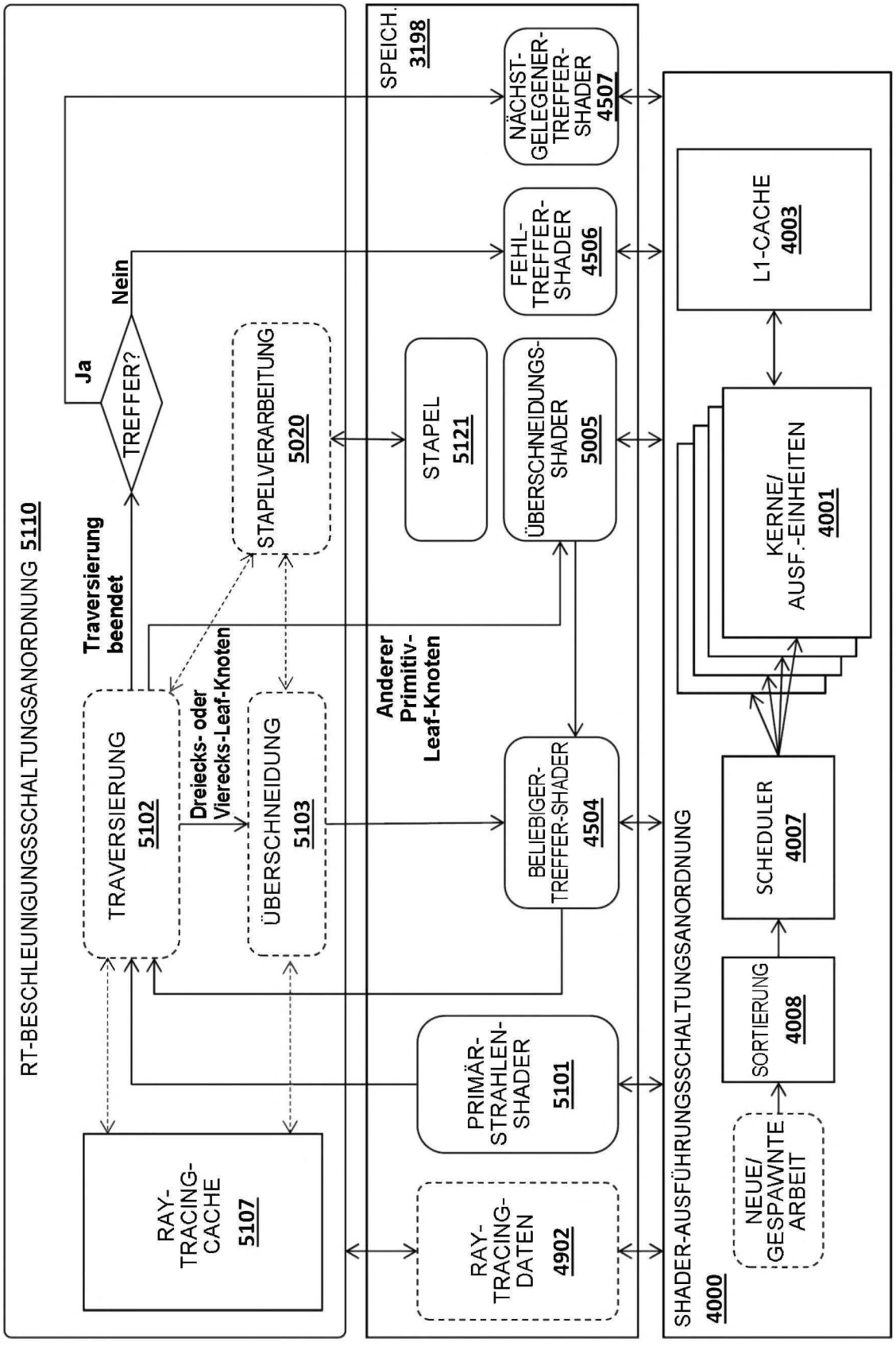

RT-BESCHLEUNIGUNGSSCHALTUNGSANORDNUNG 5110
SPEICH. 3198
SHADER-AUSFÜHRUNGSSCHALTUNGSANORDNUNG 4000
TRAVERSIERUNG 5102
ÜBERSCHNEIDUNG 5103
TREFFER?
Ja
Nein
Traversierung beendet
Dreiecks- oder Viereck-Leaf-Knoten
Anderer Primitiv-Leaf-Knoten
STAPELVERARBEITUNG 5020
STAPEL 5121
ÜBERSCHNEIDUNGS-SHADER 5005
NÄCHST-GELEGENER-TREFFER-SHADER 4507
FEHL-TREFFER-SHADER 4506
BELIEBIGER-TREFFER-SHADER 4504
PRIMÄR-STRAHLEN-SHADER 5101
RAY-TRACING-CACHE 5107
RAY-TRACING-DATEN 4902
L1-CACHE 4003
KERNE/ AUSF.-EINHEITEN 4001
SCHEDULER 4007
SORTIERUNG 4008
NEUE/ GESPAWNTE ARBEIT

FIG. 51

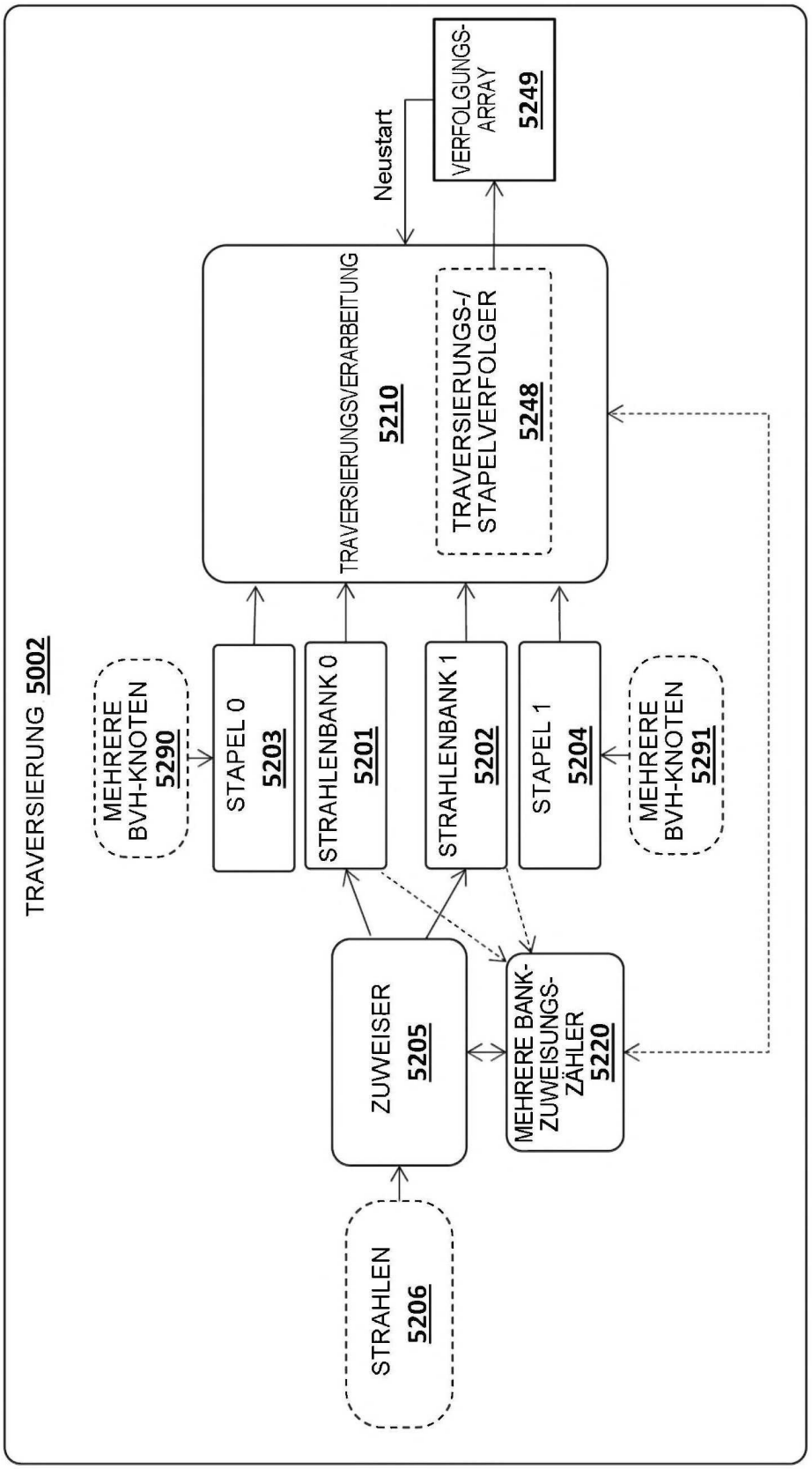

TRAVERSIERUNG 5002
MEHRERE BVH-KNOTEN 5290
STAPEL 0 5203
STRAHLENBANK 0 5201
STRAHLENBANK 1 5202
STAPEL 1 5204
MEHRERE BVH-KNOTEN 5291
TRAVERSIERUNGSVERARBEITUNG 5210
TRAVERSIERUNGS-/ STAPELVERFOLGER 5248
Neustart
VERFOLGUNGS-ARRAY 5249
ZUWEISER 5205
MEHRERE BANK-ZUWEISUNGS-ZÄHLER 5220
STRAHLEN 5206

FIG. 52A

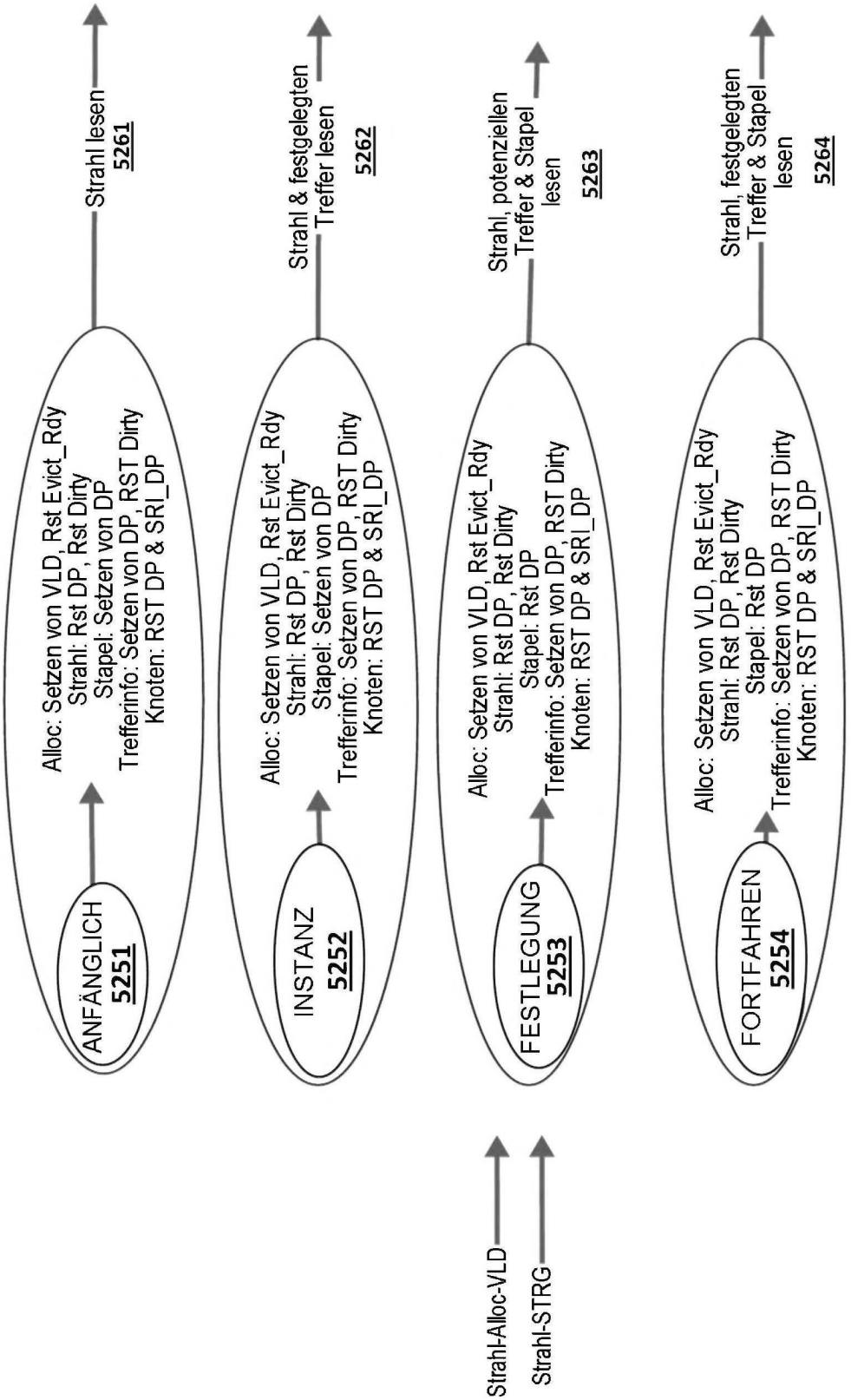

Strahl-Alloc-VLD
Strahl-STRG
ANFÄNGLICH
5251
Alloc: Setzen von VLD, Rst Evict_Rdy
Strahl: Rst DP, Rst Dirty
Stapel: Setzen von DP
Trefferinfo: Setzen von DP, RST Dirty
Knoten: RST DP & SRI_DP
Strahl lesen
5261
INSTANZ
5252
Alloc: Setzen von VLD, Rst Evict_Rdy
Strahl: Rst DP, Rst Dirty
Stapel: Setzen von DP
Trefferinfo: Setzen von DP, RST Dirty
Knoten: RST DP & SRI_DP
Strahl & festgelegten Treffer lesen
5262
FESTLEGUNG
5253
Alloc: Setzen von VLD, Rst Evict_Rdy
Strahl: Rst DP, Rst Dirty
Stapel: Rst DP
Trefferinfo: Setzen von DP, RST Dirty
Knoten: RST DP & SRI_DP
Strahl, potenziellen Treffer & Stapel lesen
5263
FORTFAHREN
5254
Alloc: Setzen von VLD, Rst Evict_Rdy
Strahl: Rst DP, Rst Dirty
Stapel: Rst DP
Trefferinfo: Setzen von DP, RST Dirty
Knoten: RST DP & SRI_DP
Strahl, festgelegten Treffer & Stapel lesen
5264

FIG. 52B

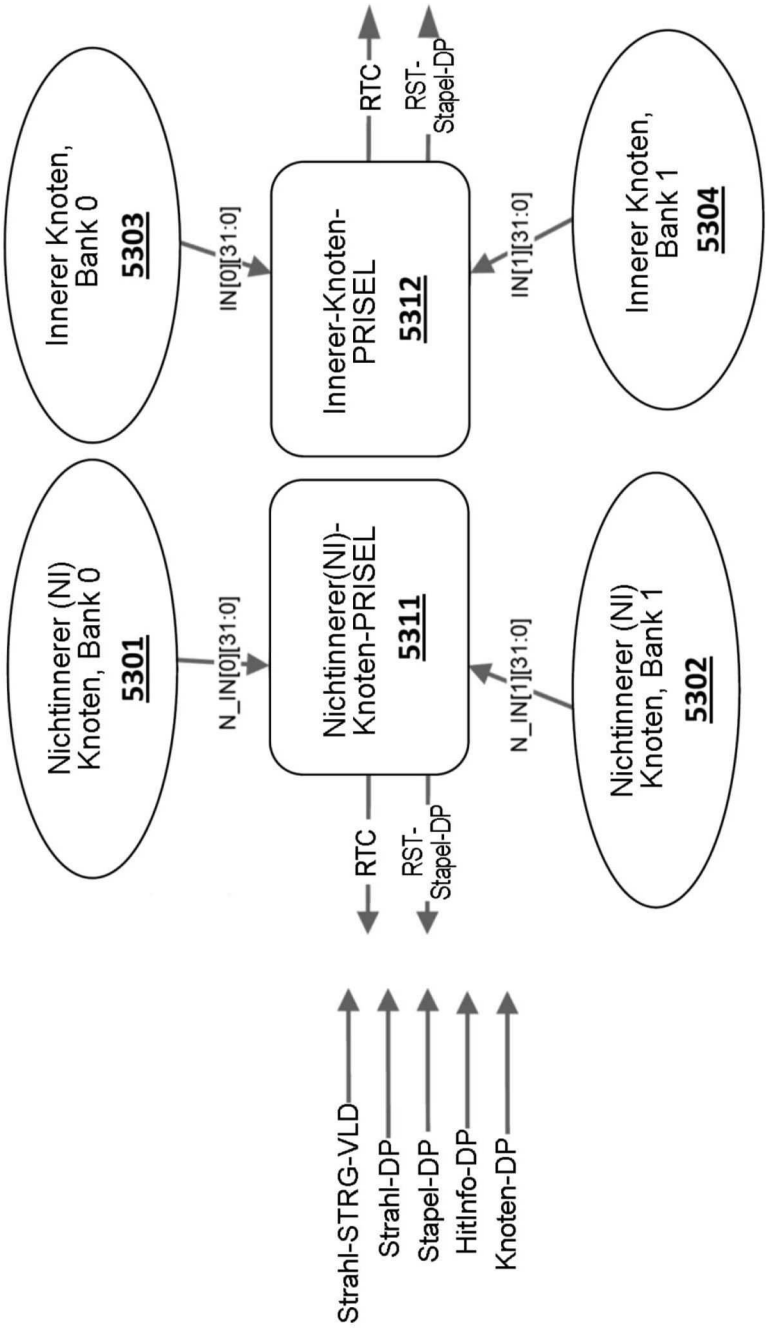

Innerer Knoten, Bank 0
5303
Nichtinnerer (NI) Knoten, Bank 0
5301
IN[0][31:0]
N_IN[0][31:0]
Innerer-Knoten-PRISEL
5312
Nichtinnerer(NI)-Knoten-PRISEL
5311
RTC
RST-Stapel-DP
IN[1][31:0]
N_IN[1][31:0]
Innerer Knoten, Bank 1
5304
Nichtinnerer (NI) Knoten, Bank 1
5302
RTC
RST-Stapel-DP
Strahl-STRG-VLD
Strahl-DP
Stapel-DP
HitInfo-DP
Knoten-DP

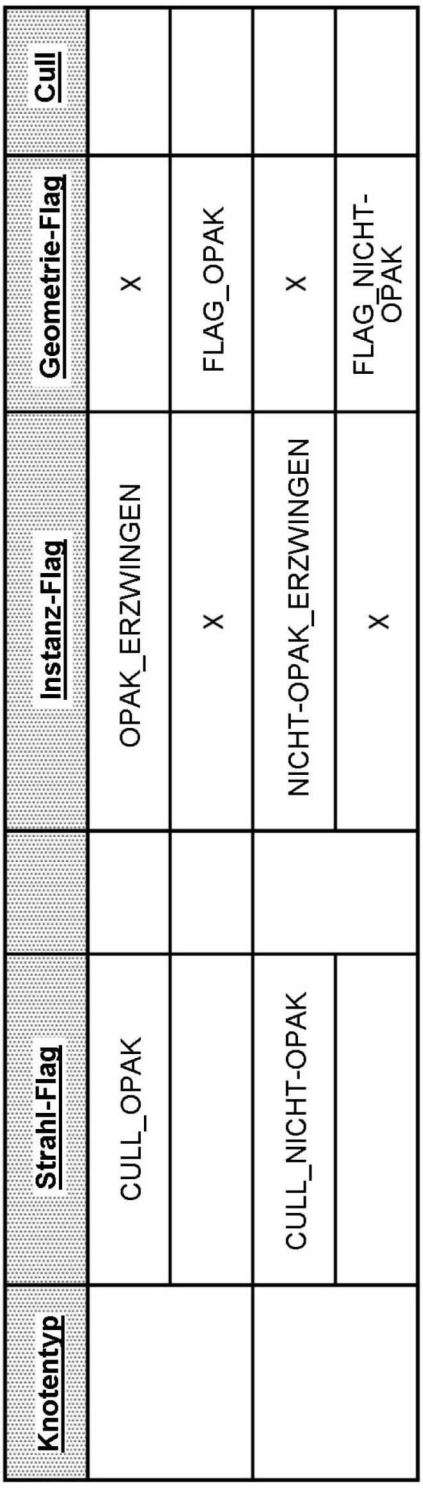
FIG. 53

Strahl-Flag	Instanz-Flag	Geometrie-Flag	Final-Flag	
FLAG_KEINES	FLAG_KEINES	FLAG_OPAK	Opak	
		FLAG_NICHT-OPAK	Nicht-opak	
	OPAK_ERZWINGEN	X	Opak	*Diese schließen sich gegenseitig aus*
	NICHT-OPAK_ERZWINGEN		Nicht-opak	
OPAK_ERZWINGEN	X		Opak	*Diese schließen sich gegenseitig aus*
NICHT-OPAK_ERZWINGEN			Nicht-opak	

FIG. 54

Strahl-Flag	Instanz-Flag	Geometrie-Flag	Cull
CULL-OPAK	OPAK_ERZWINGEN	X	Wahr
	X	FLAG_OPAK	
CULL-NICHT-OPAK	NICHT-OPAK_ERZWINGEN	X	
	X	FLAG_NICHT-OPAK	

FIG. 55A



Knotentyp	Strahl-Flag		Instanz-Flag	Geometrie-Flag	Cull
	CULL_OPAK		OPAK_ERZWINGEN	X	
			X	FLAG_OPAK	
	CULL_NICHT-OPAK		NICHT-OPAK_ERZWINGEN	X	
			X	FLAG_NICHT-OPAK	

FIG. 55B

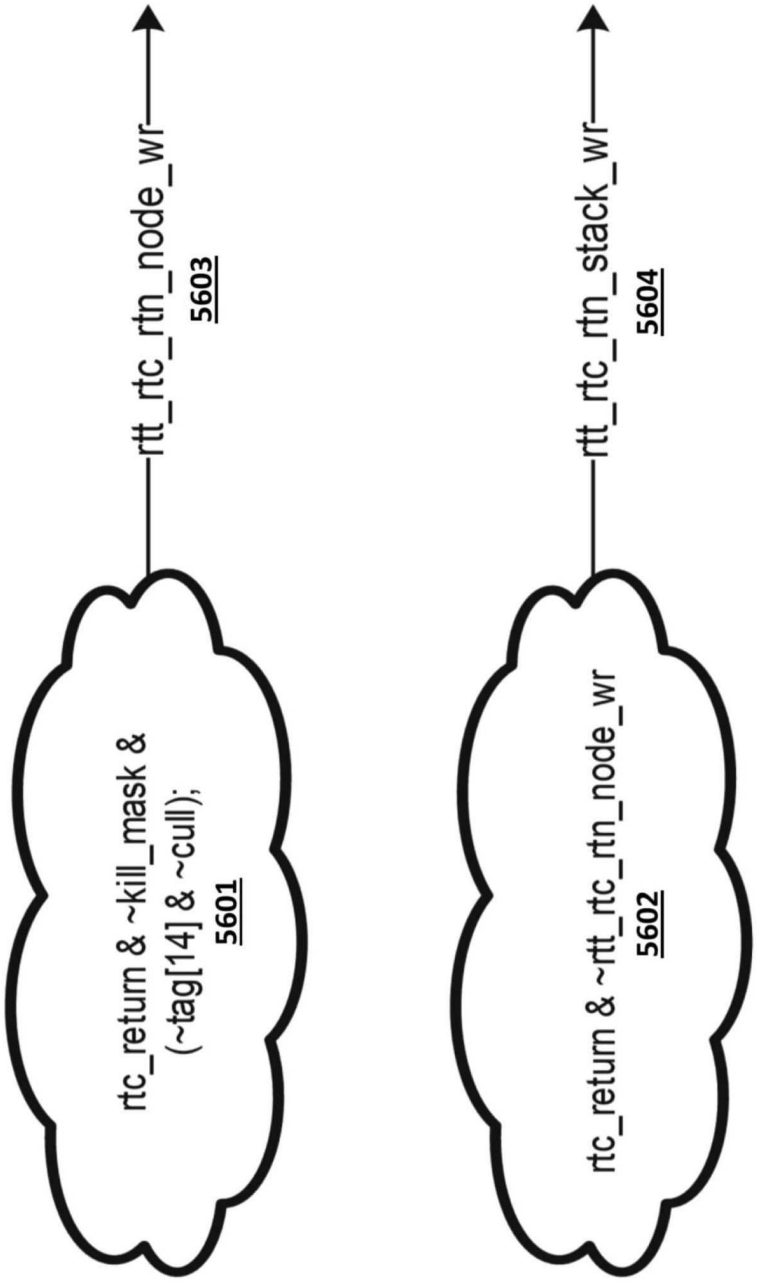

rtc_return & ~kill_mask &
(~tag[14] & ~cull);
5601
rtt_rtc_rtn_node_wr
5603
rtc_return & ~rtt_rtc_rtn_node_wr
5602
rtt_rtc_rtn_stack_wr
5604

FIG. 56

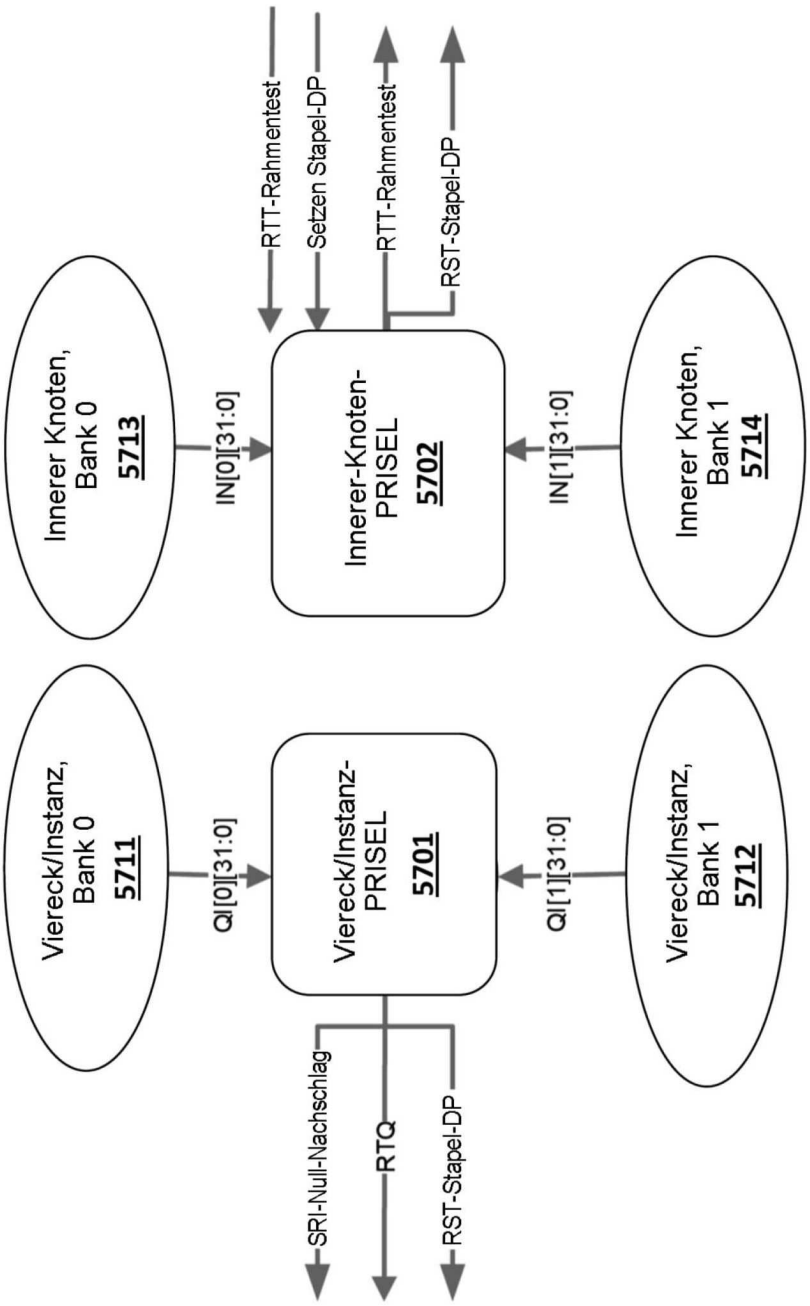

Innerer Knoten, Bank 0 5713
Viereck/Instanz, Bank 0 5711
IN[0][31:0]
QI[0][31:0]
RTT-Rahmentest
Setzen Stapel-DP
RTT-Rahmentest
RST-Stapel-DP
Innerer-Knoten-PRISEL 5702
Viereck/Instanz-PRISEL 5701
IN[1][31:0]
QI[1][31:0]
Innerer Knoten, Bank 1 5714
Viereck/Instanz, Bank 1 5712
SRI-Null-Nachschlag
RTQ
RST-Stapel-DP

FIG. 57

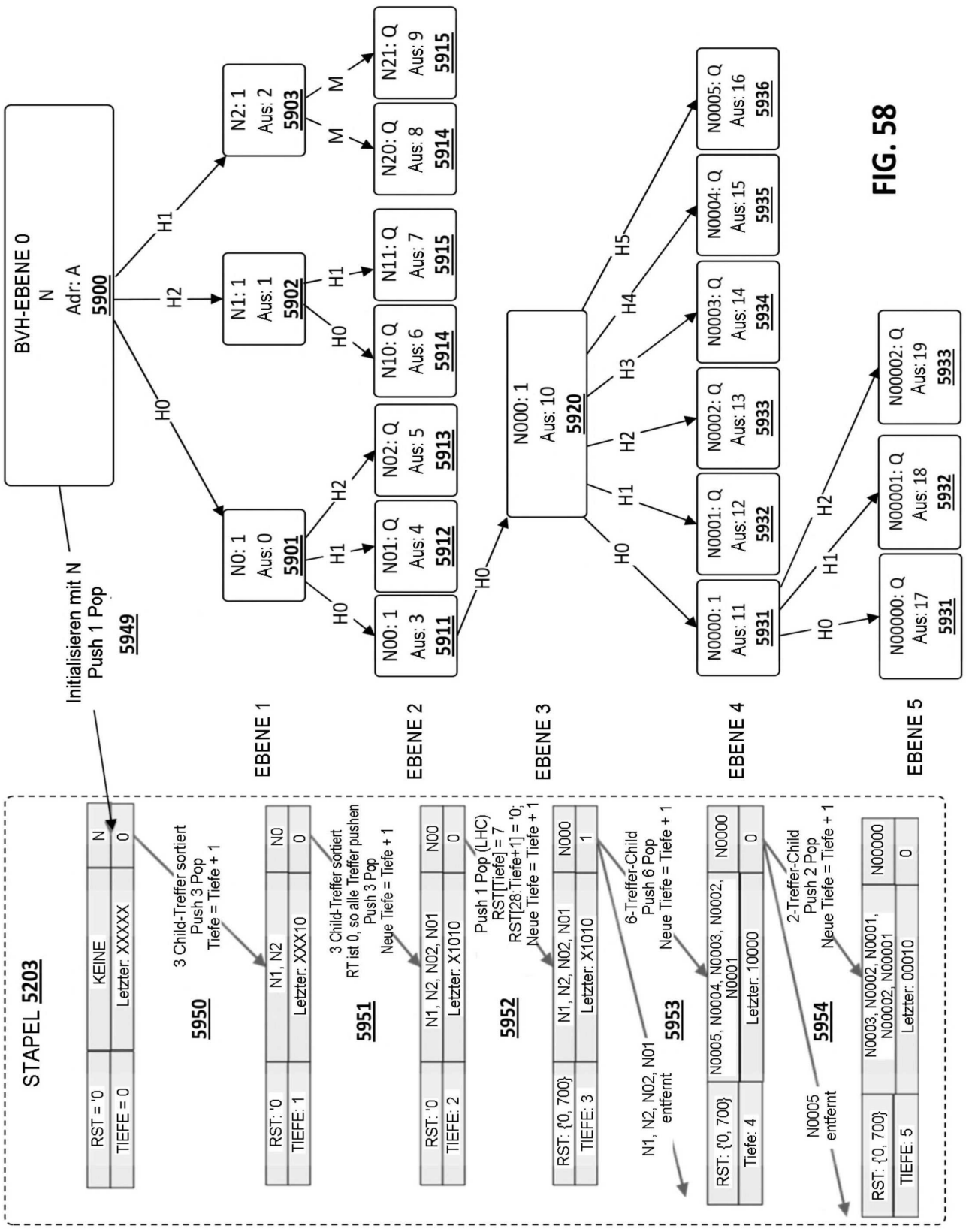

BVH-EBENE 0
N
Adr: A
5900
Initialisieren mit N
Push 1 Pop
5949
H0
H1
H2
N0: 1
Aus: 0
5901
N1: 1
Aus: 1
5902
N2: 1
Aus: 2
5903
N00: 1
Aus: 3
5911
N01: Q
Aus: 4
5912
N02: Q
Aus: 5
5913
N10: Q
Aus: 6
5914
N11: Q
Aus: 7
5915
N20: Q
Aus: 8
5914
N21: Q
Aus: 9
5915
N000: 1
Aus: 10
5920
N0000: 1
Aus: 11
5931
N0001: Q
Aus: 12
5932
N0002: Q
Aus: 13
5933
N0003: Q
Aus: 14
5934
N0004: Q
Aus: 15
5935
N0005: Q
Aus: 16
5936
N00000: Q
Aus: 17
5931
N00001: Q
Aus: 18
5932
N00002: Q
Aus: 19
5933
EBENE 1
EBENE 2
EBENE 3
EBENE 4
EBENE 5
STAPEL 5203
RST = '0
TIEFE = 0
KEINE
Letzter: XXXXX
N
0
3 Child-Treffer sortiert
Push 3 Pop
Tiefe = Tiefe + 1
5950
RST: '0
TIEFE: 1
N1, N2
Letzter: XXX10
N0
0
3 Child-Treffer sortiert
RT ist 0, so alle Treffer pushen
Push 3 Pop
Neue Tiefe = Tiefe + 1
5951
RST: '0
TIEFE: 2
N1, N2, N02, N01
Letzter: X1010
N00
0
Push 1 Pop (LHC)
RST[Tiefe] = 7
RST[28:Tiefe+1] = '0;
Neue Tiefe = Tiefe + 1
5952
RST: {0, 700}
TIEFE: 3
N1, N2, N02, N01
Letzter: X1010
N000
1
6-Treffer-Child
Push 6 Pop
Neue Tiefe = Tiefe + 1
N1, N2, N02, N01
entfernt
5953
RST: {0, 700}
Tiefe: 4
N0005, N0004, N0003, N0002, N0001
Letzter: 10000
N0000
0
2-Treffer-Child
Push 2 Pop
Neue Tiefe = Tiefe + 1
N0005
entfernt
5954
RST: {0, 700}
TIEFE: 5
N0003, N0002, N0001, N00002, N00001
Letzter: 00010
N00000
0

FIG. 58

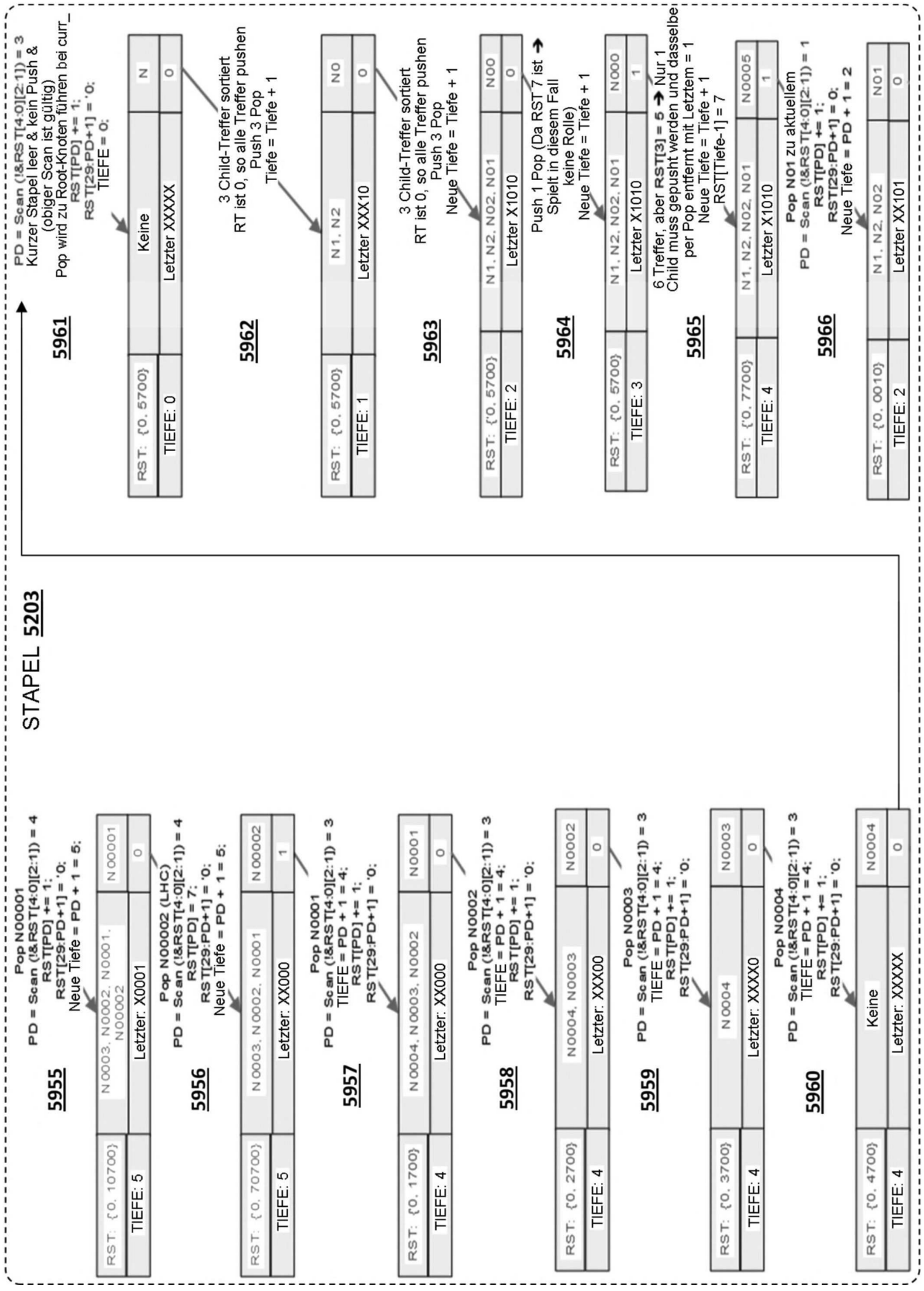

STAPEL 5203
5955
Pop N00001
PD = Scan (!&RST[4:0][2:1]) = 4
RST[PD] += 1;
RST[29:PD+1] = '0;
Neue Tiefe = PD + 1 = 5;
RST: {'0, 10700}
TIEFE: 5
N0003, N0002, N0001, N00002
N00001
Letzter: X0001
0
5956
Pop N00002 (LHC)
PD = Scan (!&RST[4:0][2:1]) = 4
RST[PD] = 7;
RST[29:PD+1] = '0;
Neue Tiefe = PD + 1 = 5;
RST: {'0, 70700}
TIEFE: 5
N0003, N0002, N0001
N00002
Letzter: XX000
1
5957
Pop N0001
PD = Scan (!&RST[4:0][2:1]) = 3
TIEFE = PD + 1 = 4;
RST[PD] += 1;
RST[29:PD+1] = '0;
RST: {'0, 1700}
TIEFE: 4
N0004, N0003, N0002
N0001
Letzter: XX000
0
5958
Pop N0002
PD = Scan (!&RST[4:0][2:1]) = 3
TIEFE = PD + 1 = 4;
RST[PD] += 1;
RST[29:PD+1] = '0;
RST: {'0, 2700}
TIEFE: 4
N0004, N0003
N0002
Letzter: XXX00
0
5959
Pop N0003
PD = Scan (!&RST[4:0][2:1]) = 3
TIEFE = PD + 1 = 4;
RST[PD] += 1;
RST[29:PD+1] = '0;
RST: {'0, 3700}
TIEFE: 4
N0004
N0003
Letzter: XXXX0
0
5960
Pop N0004
PD = Scan (!&RST[4:0][2:1]) = 3
TIEFE = PD + 1 = 4;
RST[PD] += 1;
RST[29:PD+1] = '0;
RST: {'0, 4700}
TIEFE: 4
Keine
N0004
Letzter: XXXXX
0
PD = Scan (!&RST[4:0][2:1]) = 3
Kurzer Stapel leer & kein Push &
(obiger Scan ist gültig)
Pop wird zu Root-Knoten führen bei curr_
RST[PD] += 1;
RST[29:PD+1] = '0;
TIEFE = 0;
5961
RST: {'0, 5700}
TIEFE: 0
Keine
N
Letzter XXXXX
0
3 Child-Treffer sortiert
RT ist 0, so alle Treffer pushen
Push 3 Pop
Tiefe = Tiefe + 1
5962
RST: {'0, 5700}
TIEFE: 1
N1, N2
N0
Letzter XXX10
0
3 Child-Treffer sortiert
RT ist 0, so alle Treffer pushen
Push 3 Pop
Neue Tiefe = Tiefe + 1
5963
RST: {'0, 5700}
TIEFE: 2
N1, N2, N02, N01
N00
Letzter X1010
0
Push 1 Pop (Da RST 7 ist →
Spielt in diesem Fall
keine Rolle)
Neue Tiefe = Tiefe + 1
5964
RST: {'0, 5700}
TIEFE: 3
N1, N2, N02, N01
N000
Letzter X1010
1
6 Treffer, aber RST[3] = 5 → Nur 1
Child muss gepusht werden und dasselbe
per Pop entfernt mit Letztem = 1
Neue Tiefe = Tiefe + 1
RST[Tiefe-1] = 7
5965
RST: {'0, 7700}
TIEFE: 4
N1, N2, N02, N01
N0005
Letzter X1010
1
Pop N01 zu aktuellem
PD = Scan (!&RST[4:0][2:1]) = 1
RST[PD] += 1;
RST[29:PD+1] = 0;
Neue Tiefe = PD + 1 = 2
5966
RST: {'0, 0010}
TIEFE: 2
N1, N2, N02
N01
Letzter XX101
0

FIG. 59A

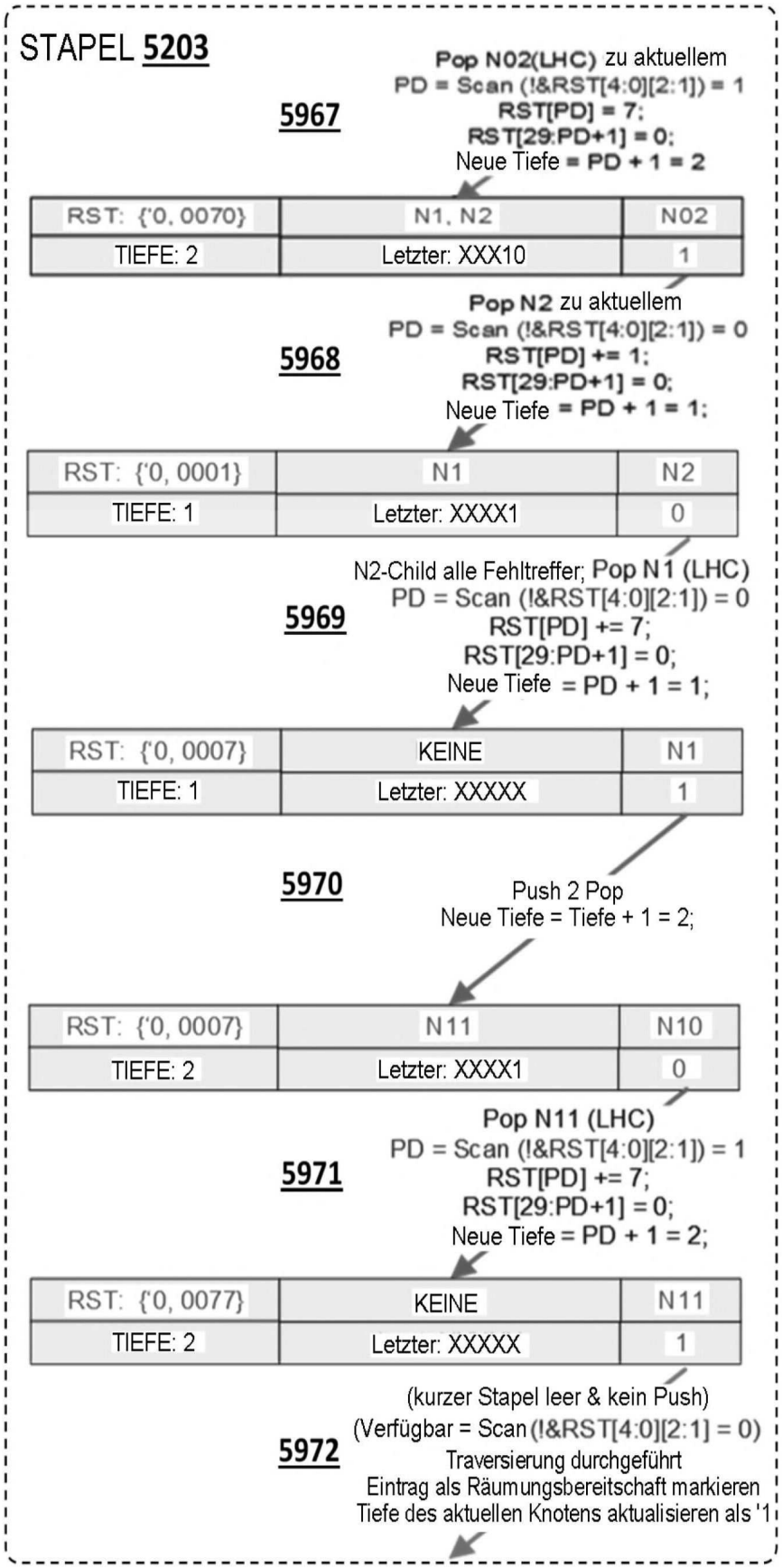

STAPEL 5203
5967
Pop N02(LHC) zu aktuellem
PD = Scan (!&RST[4:0][2:1]) = 1
RST[PD] = 7;
RST[29:PD+1] = 0;
Neue Tiefe = PD + 1 = 2
RST: {'0, 0070}
N1, N2
N02
TIEFE: 2
Letzter: XXX10
1
5968
Pop N2 zu aktuellem
PD = Scan (!&RST[4:0][2:1]) = 0
RST[PD] += 1;
RST[29:PD+1] = 0;
Neue Tiefe = PD + 1 = 1;
RST: {'0, 0001}
N1
N2
TIEFE: 1
Letzter: XXXX1
0
5969
N2-Child alle Fehltreffer; Pop N1 (LHC)
PD = Scan (!&RST[4:0][2:1]) = 0
RST[PD] += 7;
RST[29:PD+1] = 0;
Neue Tiefe = PD + 1 = 1;
RST: {'0, 0007}
KEINE
N1
TIEFE: 1
Letzter: XXXXX
1
5970
Push 2 Pop
Neue Tiefe = Tiefe + 1 = 2;
RST: {'0, 0007}
N11
N10
TIEFE: 2
Letzter: XXXX1
0
5971
Pop N11 (LHC)
PD = Scan (!&RST[4:0][2:1]) = 1
RST[PD] += 7;
RST[29:PD+1] = 0;
Neue Tiefe = PD + 1 = 2;
RST: {'0, 0077}
KEINE
N11
TIEFE: 2
Letzter: XXXXX
1
5972
(kurzer Stapel leer & kein Push)
(Verfügbar = Scan (!&RST[4:0][2:1] = 0)
Traversierung durchgeführt
Eintrag als Räumungsbereitschaft markieren
Tiefe des aktuellen Knotens aktualisieren als '1

FIG. 59B

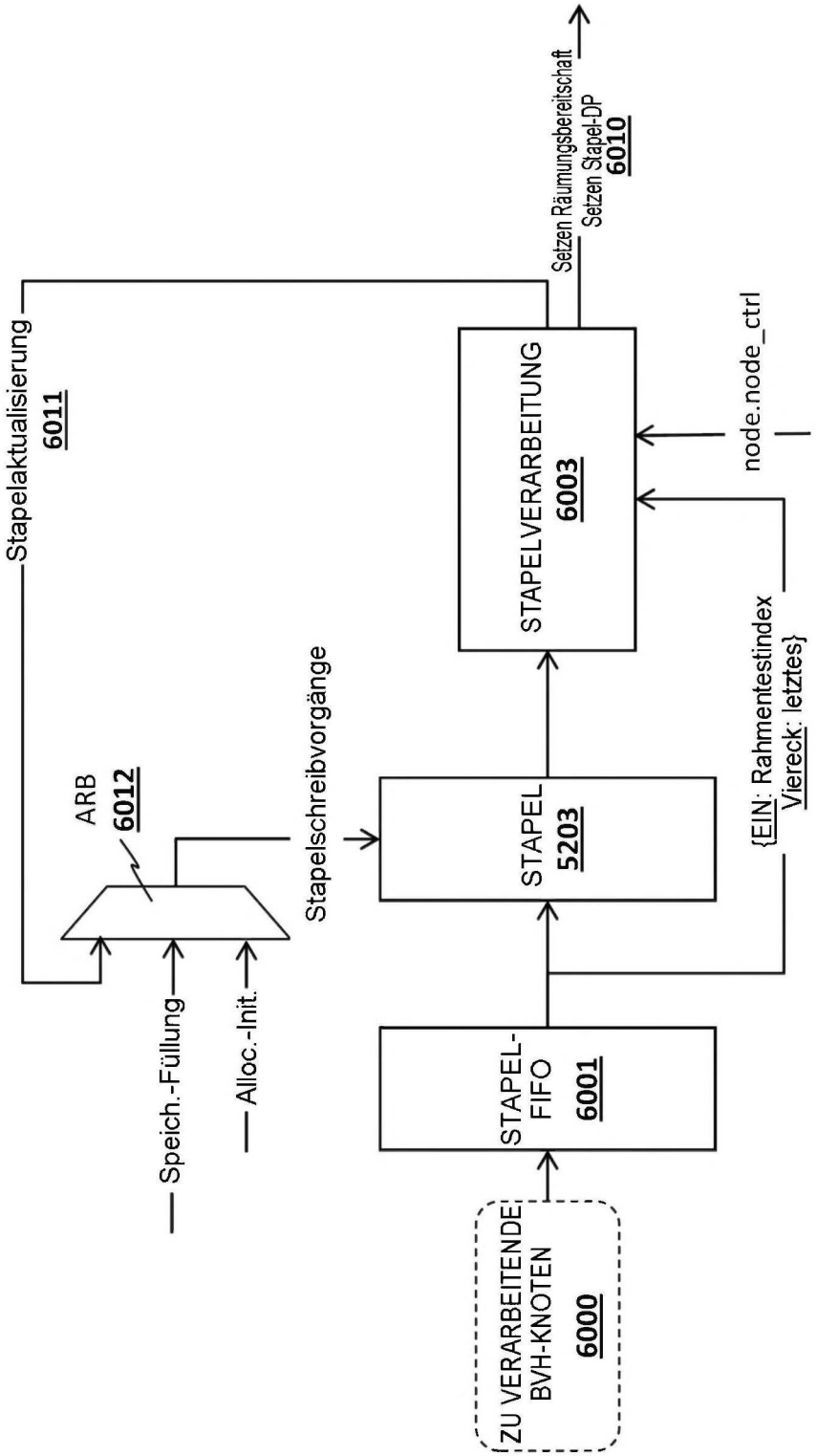
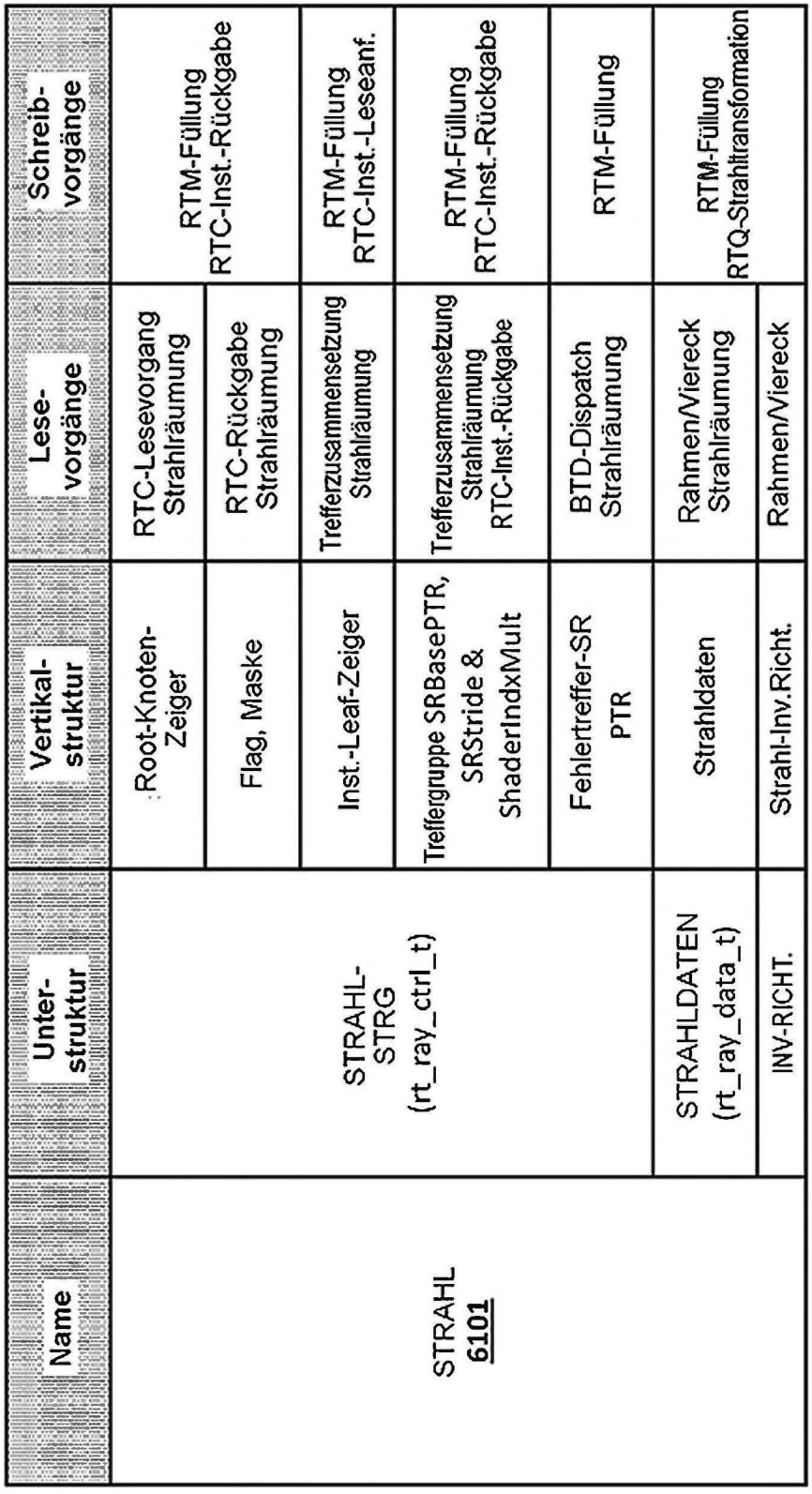

ZU VERARBEITENDE BVH-KNOTEN
6000
STAPEL-FIFO
6001
STAPEL
5203
STAPELVERARBEITUNG
6003
Stapelschreibvorgänge
ARB
6012
Speich.-Füllung
Alloc.-Init.
Stapelaktualisierung
6011
Setzen Räumungsbereitschaft
Setzen Stapel-DP
6010
node.node_ctrl
{EIN: Rahmentestindex
Viereck: letztes}

FIG. 60



Name	Unter-struktur	Vertikal-struktur	Lese-vorgänge	Schreib-vorgänge
STRAHL **6101**	STRAHL-STRG (rt_ray_ctrl_t)	Root-Knoten-Zeiger	RTC-Lesevorgang Strahlräumung	RTM-Füllung RTC-Inst.-Rückgabe
		Flag, Maske	RTC-Rückgabe Strahlräumung	
		Inst.-Leaf-Zeiger	Trefferzusammensetzung Strahlräumung	RTM-Füllung RTC-Inst.-Leseanf.
		Treffergruppe SRBasePTR, SRStride & ShaderIndxMult	Trefferzusammensetzung Strahlräumung RTC-Inst.-Rückgabe	RTM-Füllung RTC-Inst.-Rückgabe
		Fehlertreffer-SR PTR	BTD-Dispatch Strahlräumung	RTM-Füllung
	STRAHLDATEN (rt_ray_data_t)	Strahldaten	Rahmen/Viereck Strahlräumung	RTM-Füllung RTQ-Strahltransformation
	INV-RICHT.	Strahl-Inv.Richt.	Rahmen/Viereck	

FIG. 61A

Name	Unter-struktur	Vertikal-struktur	Lese-vorgänge	Schreib-vorgänge
TREFFER **6102**	Trefferinfo (rt_hitinfo_t)	Treffer T	Viereck & Räumung	RTM-Füllung Festgelegter Treffer
		Treffer	Räumung	
STAPEL **6103**	Stapel			
		Stapel	Stapel-Verarb.-Start Räumung	RTM-Füllung Stapel-Verarb.-Ende
		Aktueller Stapel	RTC-Leseanf. Stapel-Verarb.-Start Räumung	RTM-Füllung Stapel-Verarb.-Ende
		Knoten	RTC-Antwort	
		Knoten-Strg	Festgelegter Treffer	
		Knoteninfo	Prisel	
		Knotendaten (Vereinigung)	Rahmen/Viereck	

FIG. 61B

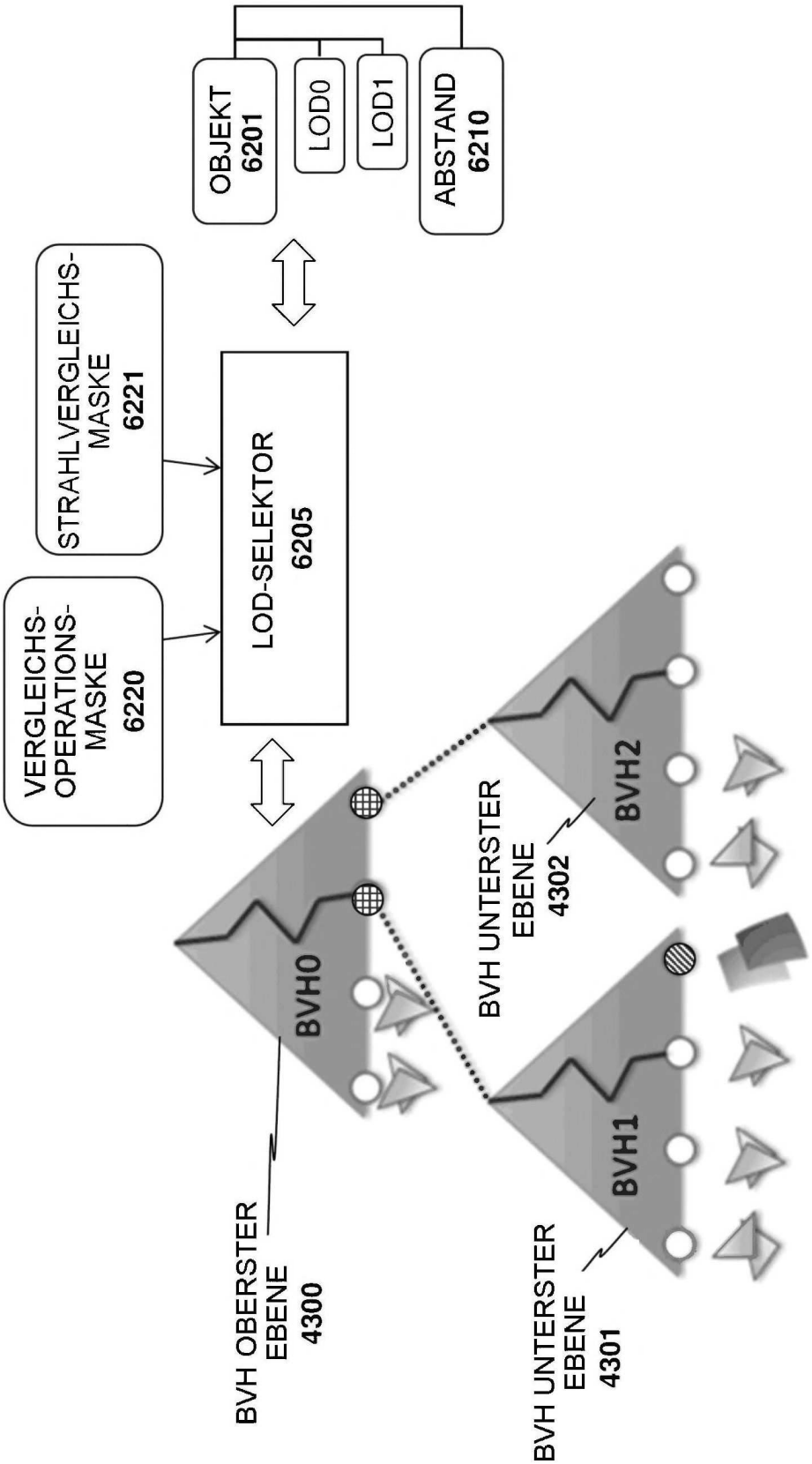

OBJEKT
6201
LOD0
LOD1
ABSTAND
6210
STRAHLVERGLEICHS-
MASKE
6221
VERGLEICHS-
OPERATIONS-
MASKE
6220
LOD-SELEKTOR
6205
BVH0
BVH1
BVH2
BVH OBERSTER
EBENE
4300
BVH UNTERSTER
EBENE
4302
BVH UNTERSTER
EBENE
4301

FIG. 62

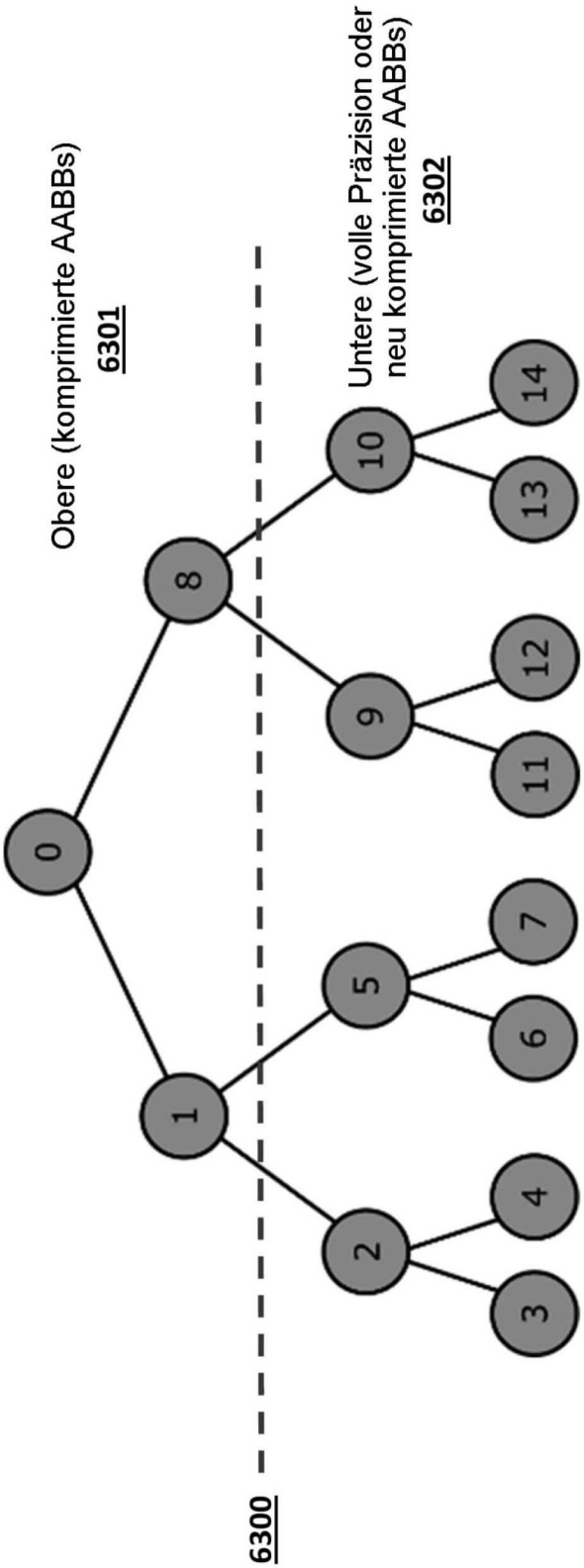

Obere (komprimierte AABBs)
6301
Untere (volle Präzision oder neu komprimierte AABBs)
6302
6300
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

FIG. 63

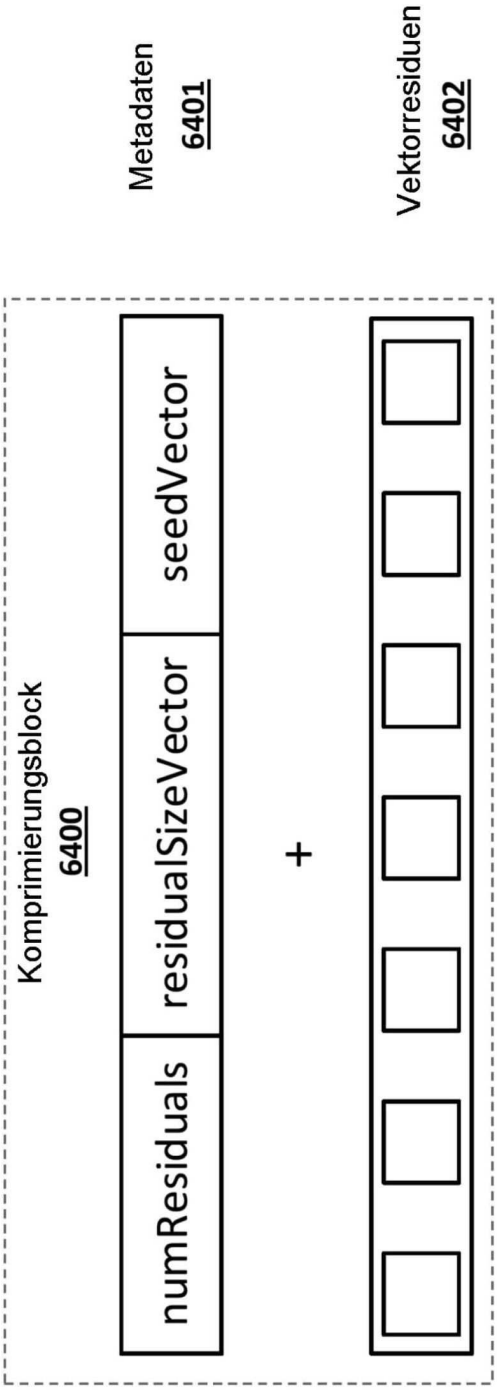

Komprimierungsblock
6400
numResiduals
residualSizeVector
seedVector
+
Metadaten
6401
Vektorresiduen
6402

FIG. 64

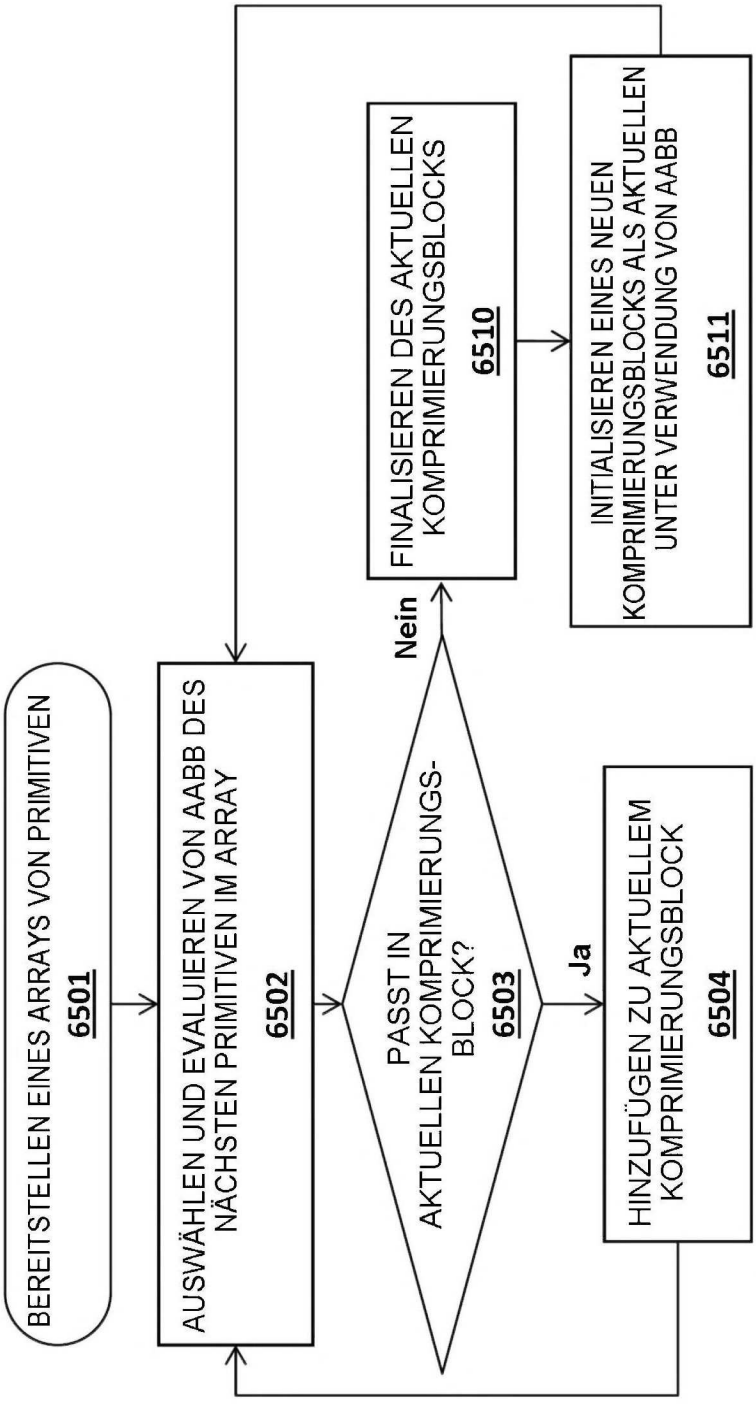

BEREITSTELLEN EINES ARRAYS VON PRIMITIVEN
6501
AUSWÄHLEN UND EVALUIEREN VON AABB DES NÄCHSTEN PRIMITIVEN IM ARRAY
6502
PASST IN AKTUELLEN KOMPRIMIERUNGS-BLOCK?
6503
Nein
Ja
FINALISIEREN DES AKTUELLEN KOMPRIMIERUNGSBLOCKS
6510
INITIALISIEREN EINES NEUEN KOMPRIMIERUNGSBLOCKS ALS AKTUELLEN UNTER VERWENDUNG VON AABB
6511
HINZUFÜGEN ZU AKTUELLEM KOMPRIMIERUNGSBLOCK
6504

FIG. 65


Block-Offset-Indexkomprimierungsblock
6610
numIndices
residualSizeVector
seedVector
+
Metadaten
6603
Indexresiduen
6602

FIG. 66

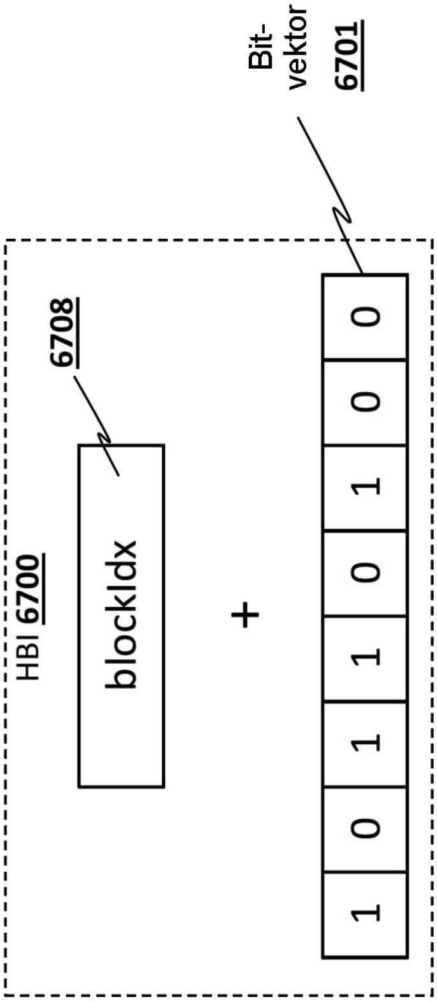


FIG. 67A

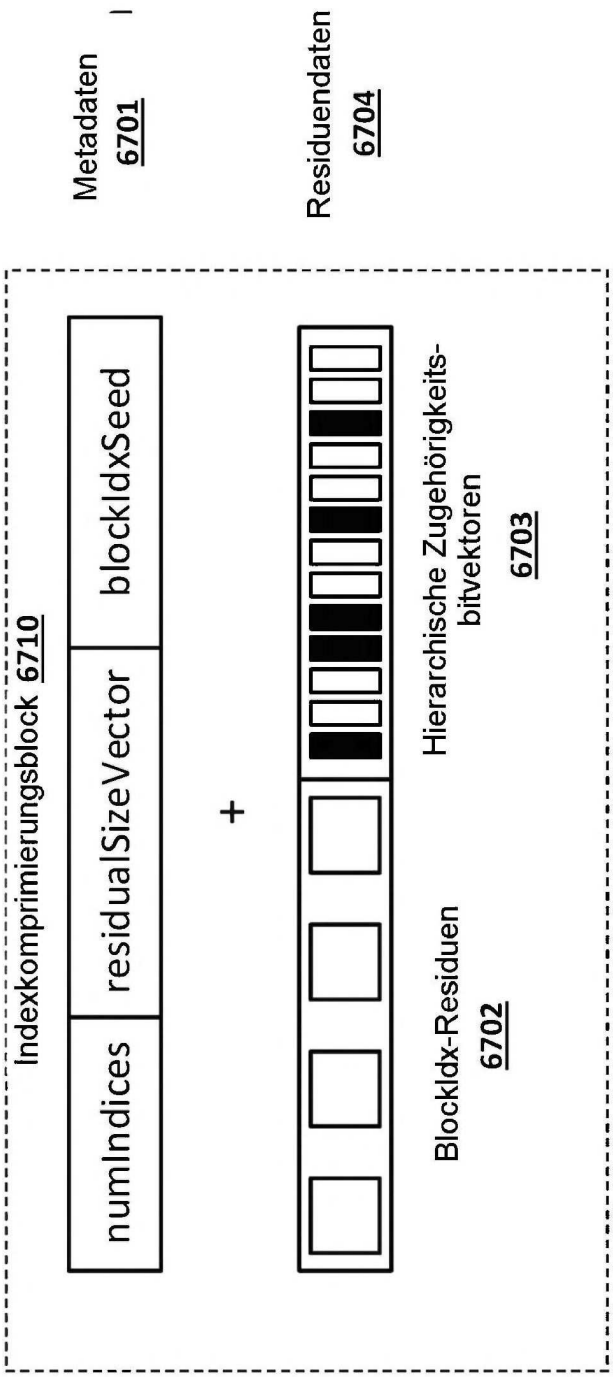

Indexkomprimierungsblock 6710
numIndices
residualSizeVector
blockIdxSeed
+
Metadaten 6701
Residuendaten 6704
Blockldx-Residuen 6702
Hierarchische Zugehörigkeits-bitvektoren 6703

FIG. 67B

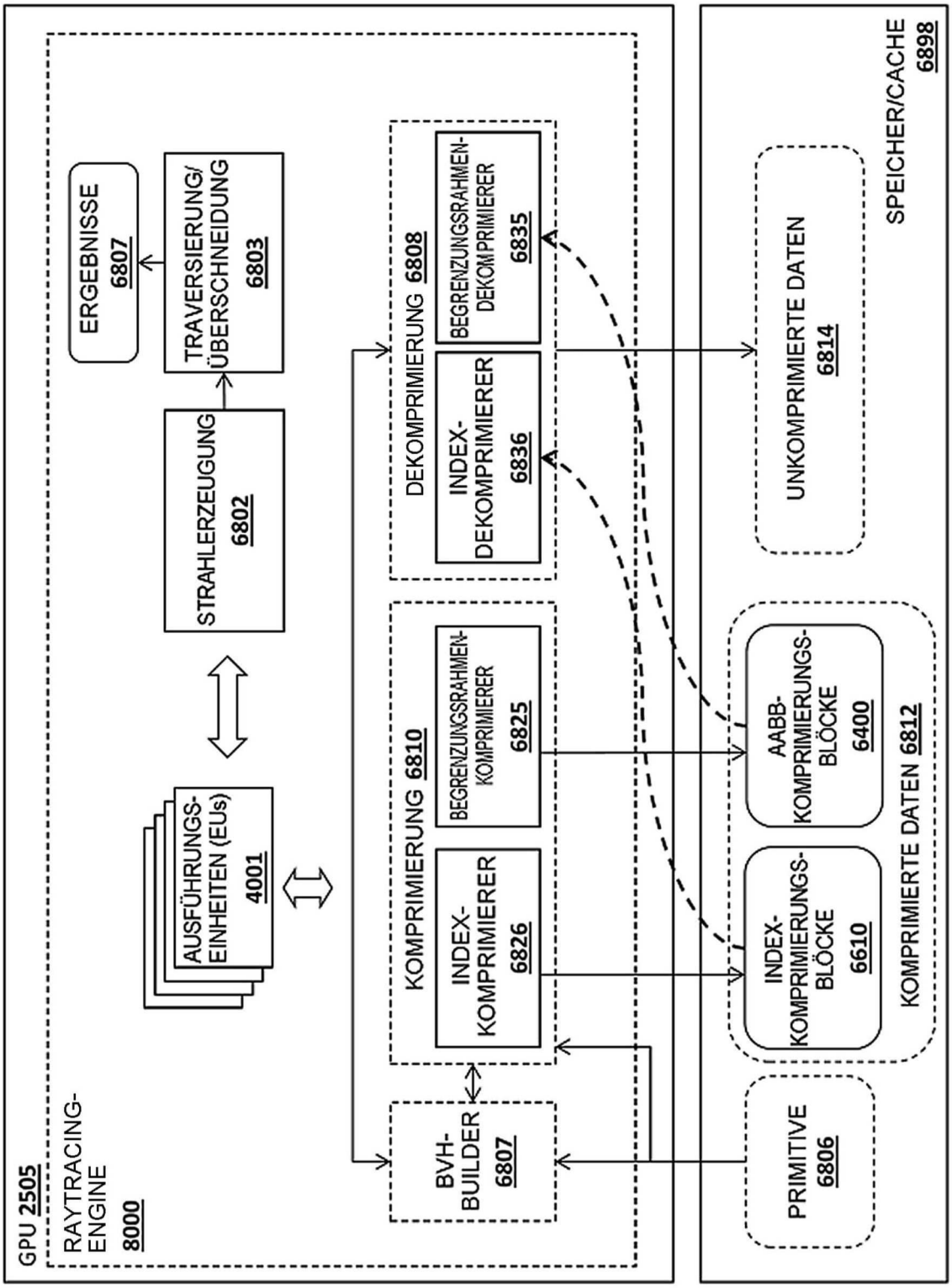

GPU 2505
RAYTRACING-ENGINE 8000
AUSFÜHRUNGS-EINHEITEN (EUs) 4001
STRAHLERZEUGUNG 6802
TRAVERSIERUNG/ ÜBERSCHNEIDUNG 6803
ERGEBNISSE 6807
BVH-BUILDER 6807
KOMPRIMIERUNG 6810
INDEX-KOMPRIMIERER 6826
BEGRENZUNGSRAHMEN-KOMPRIMIERER 6825
DEKOMPRIMIERUNG 6808
INDEX-DEKOMPRIMIERER 6836
BEGRENZUNGSRAHMEN-DEKOMPRIMIERER 6835
SPEICHER/CACHE 6898
PRIMITIVE 6806
KOMPRIMIERTE DATEN 6812
INDEX-KOMPRIMIERUNGS-BLÖCKE 6610
AABB-KOMPRIMIERUNGS-BLÖCKE 6400
UNKOMPRIMIERTE DATEN 6814

FIG. 68

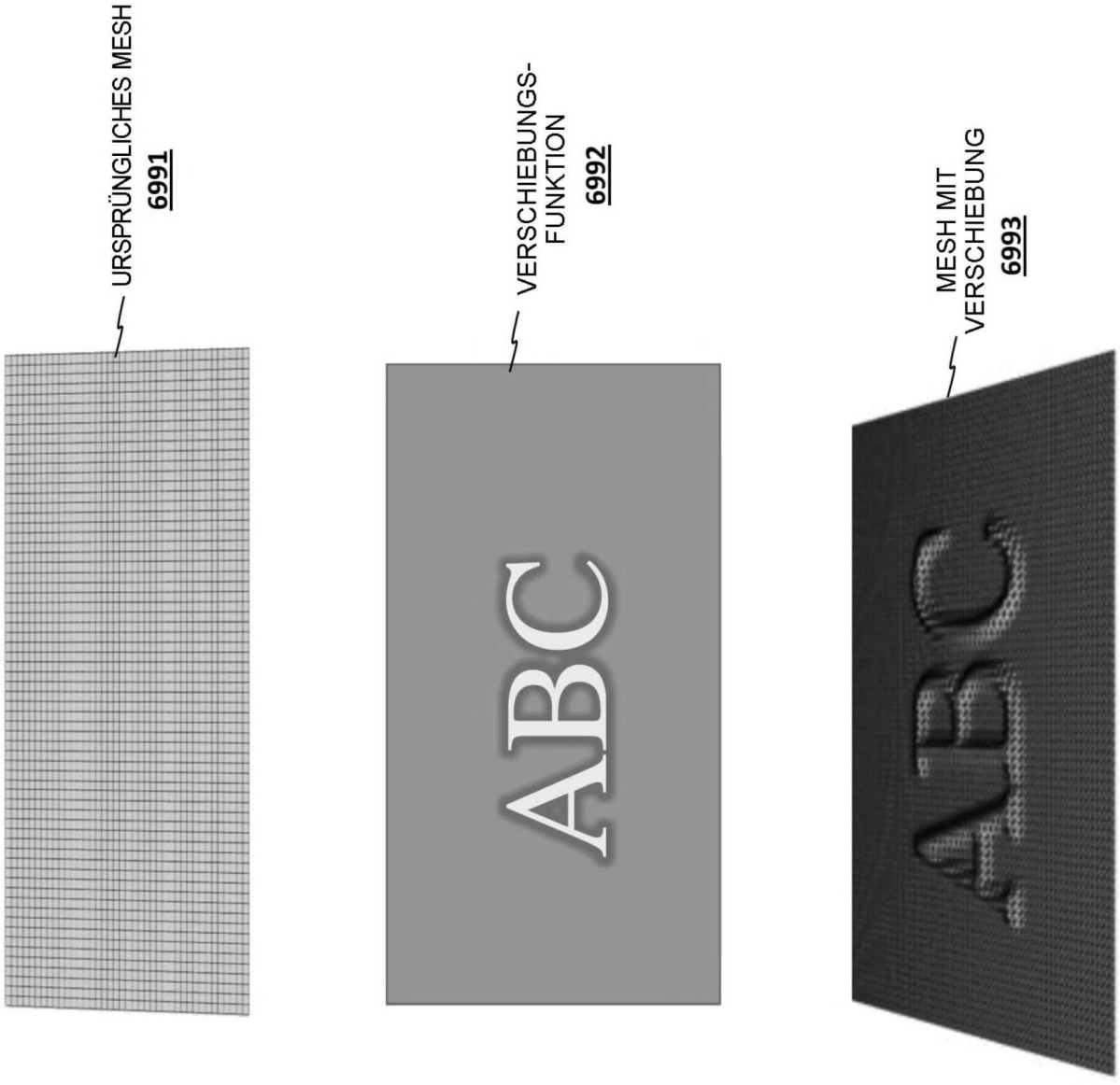

URSPRÜNGLICHES MESH
6991
VERSCHIEBUNGS-
FUNKTION
6992
MESH MIT
VERSCHIEBUNG
6993
ABC

FIG. 69A

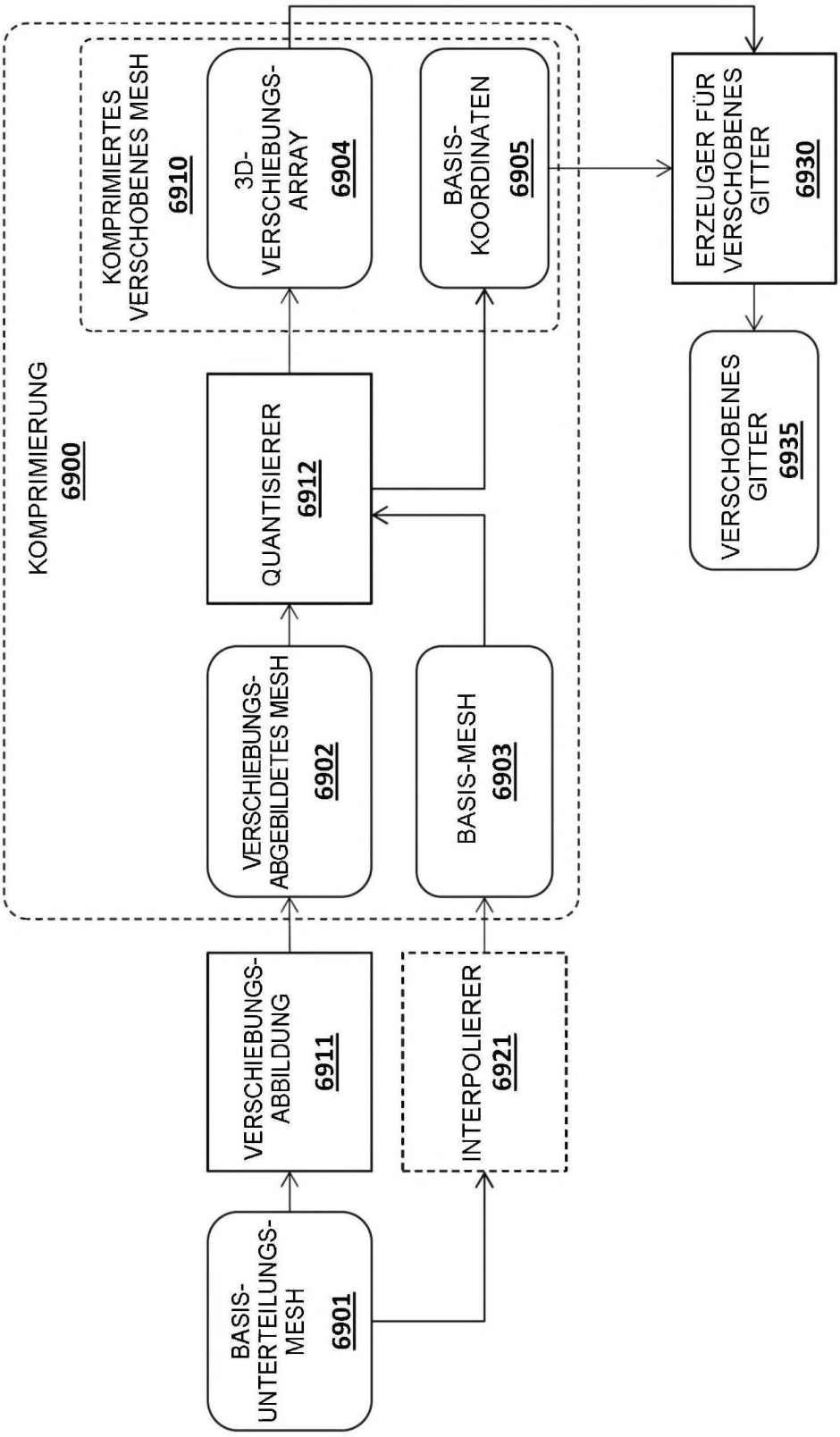

KOMPRIMIERUNG
6900
KOMPRIMIERTES VERSCHOBENES MESH
6910
3D-VERSCHIEBUNGS-ARRAY
6904
BASIS-KOORDINATEN
6905
QUANTISIERER
6912
VERSCHIEBUNGS-ABGEBILDETES MESH
6902
BASIS-MESH
6903
VERSCHIEBUNGS-ABBILDUNG
6911
INTERPOLIERER
6921
BASIS-UNTERTEILUNGS-MESH
6901
ERZEUGER FÜR VERSCHOBENES GITTER
6930
VERSCHOBENES GITTER
6935

FIG. 69B

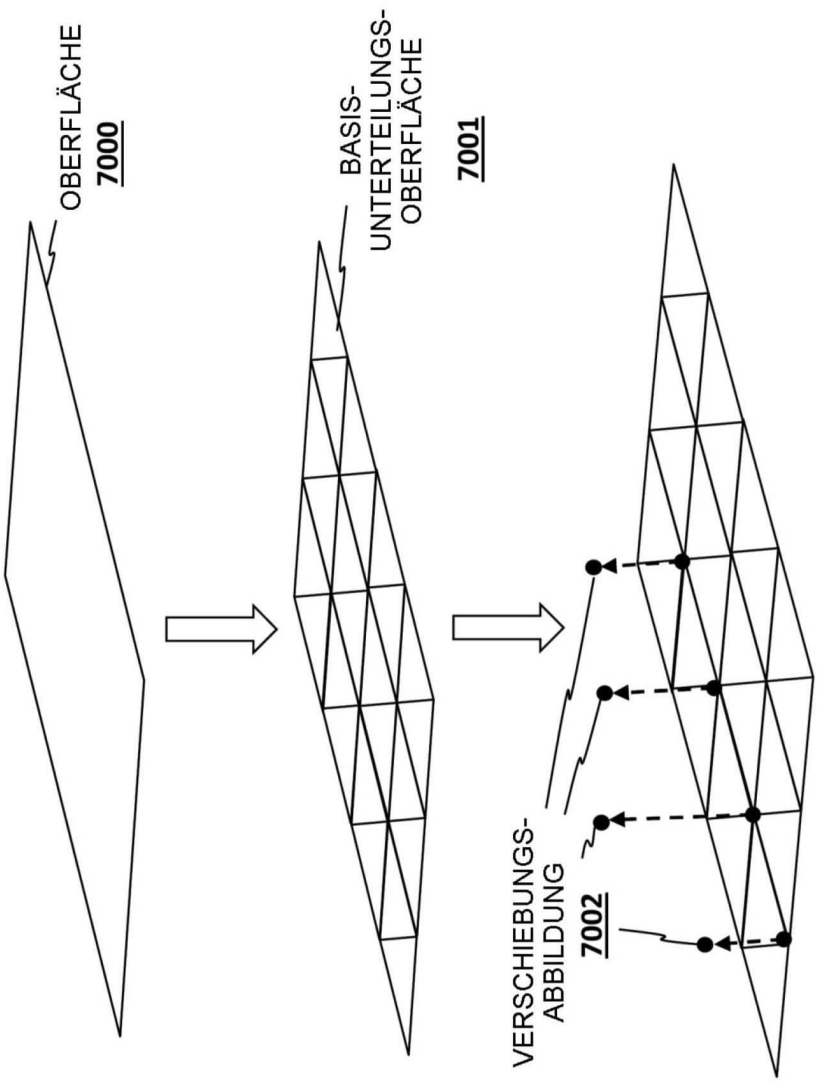

OBERFLÄCHE
7000
BASIS-
UNTERTEILUNGS-
OBERFLÄCHE
7001
VERSCHIEBUNGS-
ABBILDUNG
7002

FIG. 70A

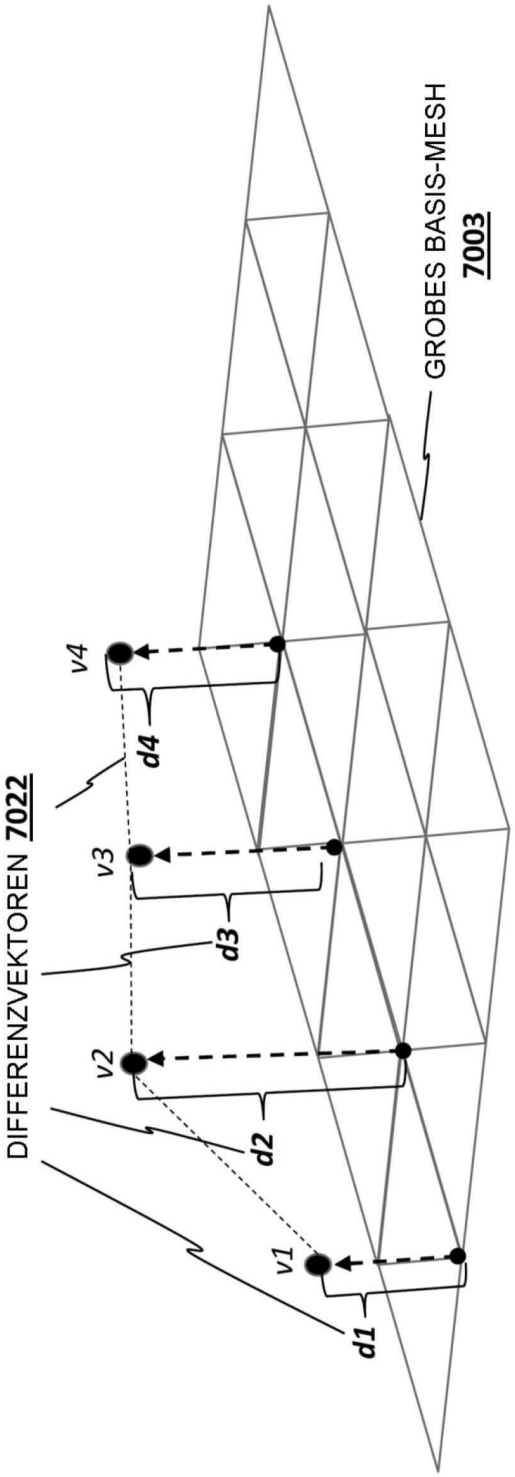

GROBES BASIS-MESH
7003
DIFFERENZVEKTOREN 7022
v4
v3
v2
v1
d4
d3
d2
d1

FIG. 70B

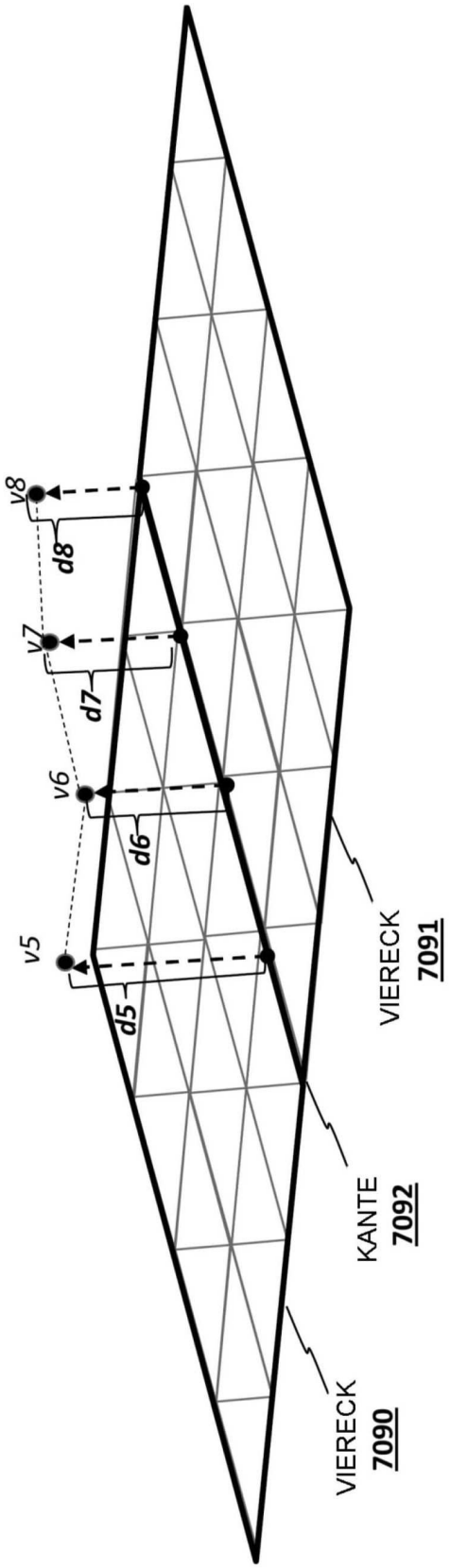


FIG. 70C

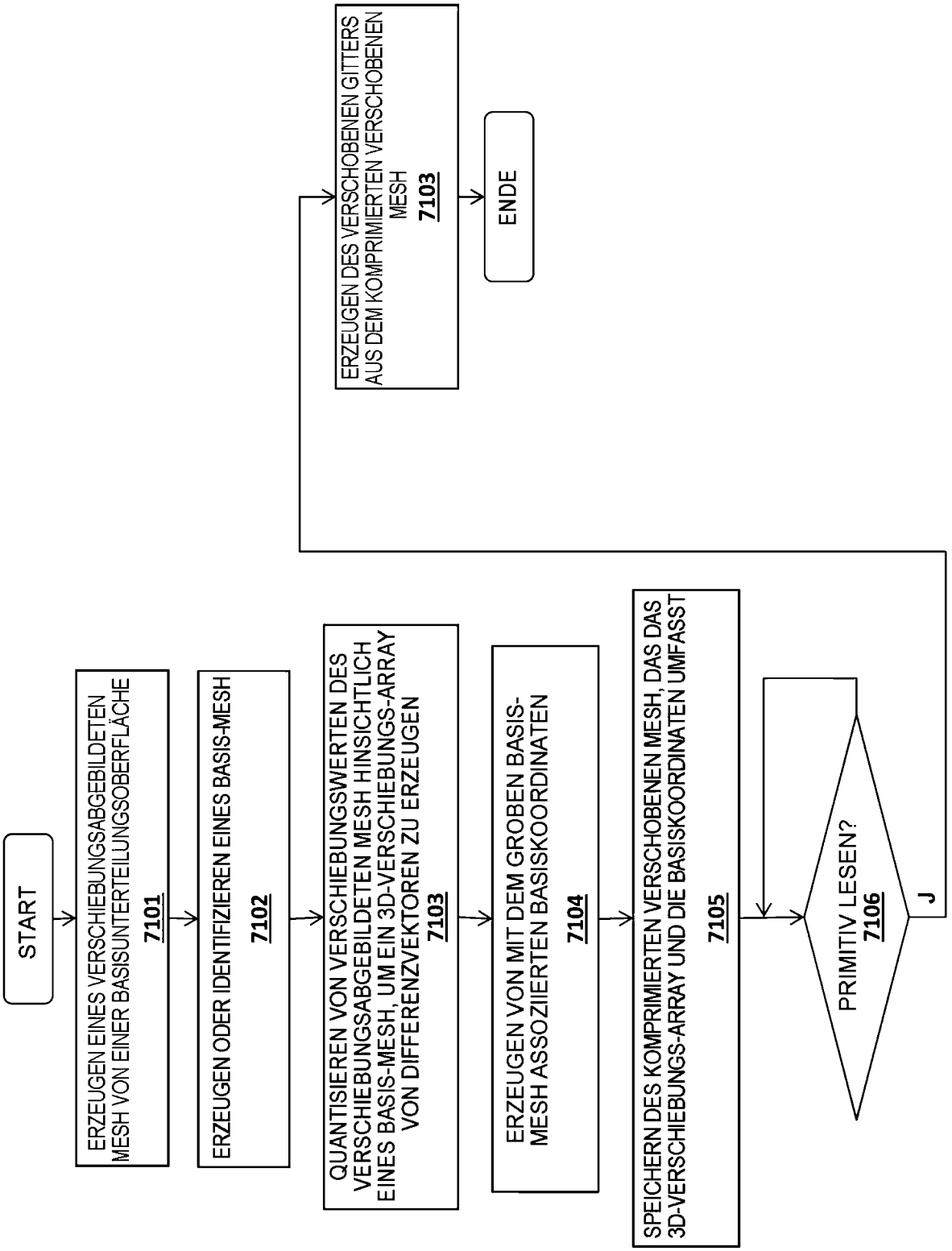

START
ERZEUGEN EINES VERSCHIEBUNGSABGEBILDETEN MESH VON EINER BASISUNTERTEILUNGSOBERFLÄCHE
7101
ERZEUGEN ODER IDENTIFIZIEREN EINES BASIS-MESH
7102
QUANTISIEREN VON VERSCHIEBUNGSWERTEN DES VERSCHIEBUNGSABGEBILDETEN MESH HINSICHTLICH EINES BASIS-MESH, UM EIN 3D-VERSCHIEBUNGS-ARRAY VON DIFFERENZVEKTOREN ZU ERZEUGEN
7103
ERZEUGEN VON MIT DEM GROBEN BASIS-MESH ASSOZIIERTEN BASISKOORDINATEN
7104
SPEICHERN DES KOMPRIMIERTEN VERSCHOBENEN MESH, DAS DAS 3D-VERSCHIEBUNGS-ARRAY UND DIE BASISKOORDINATEN UMFASST
7105
PRIMITIV LESEN?
7106
J
ERZEUGEN DES VERSCHOBENEN GITTERS AUS DEM KOMPRIMIERTEN VERSCHOBENEN MESH
7103
ENDE

FIG. 71

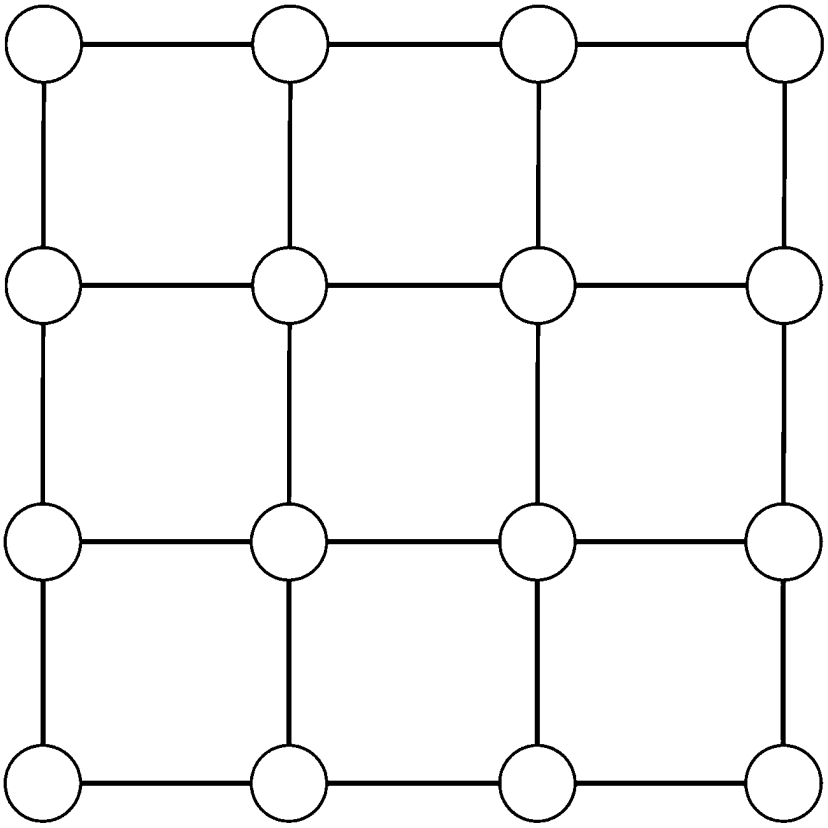


FIG. 72

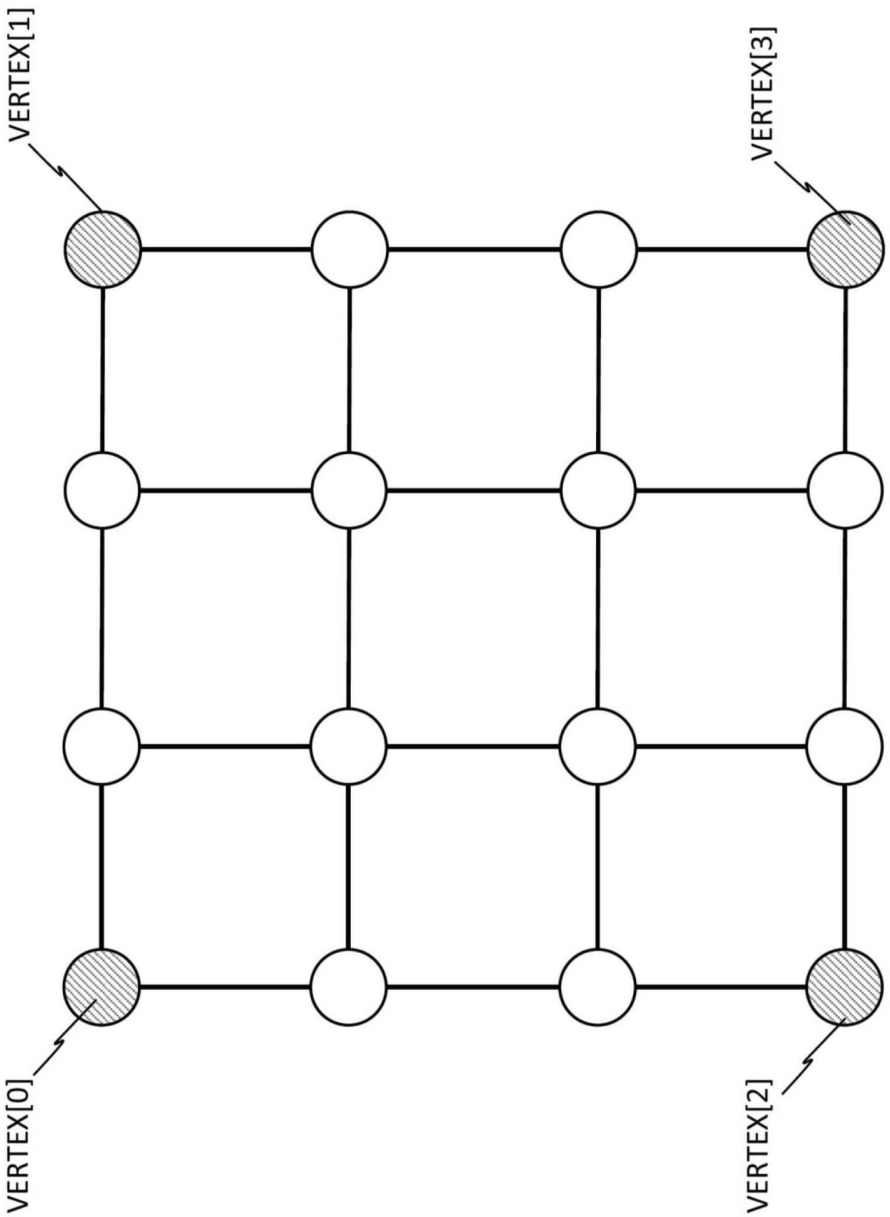


FIG. 73

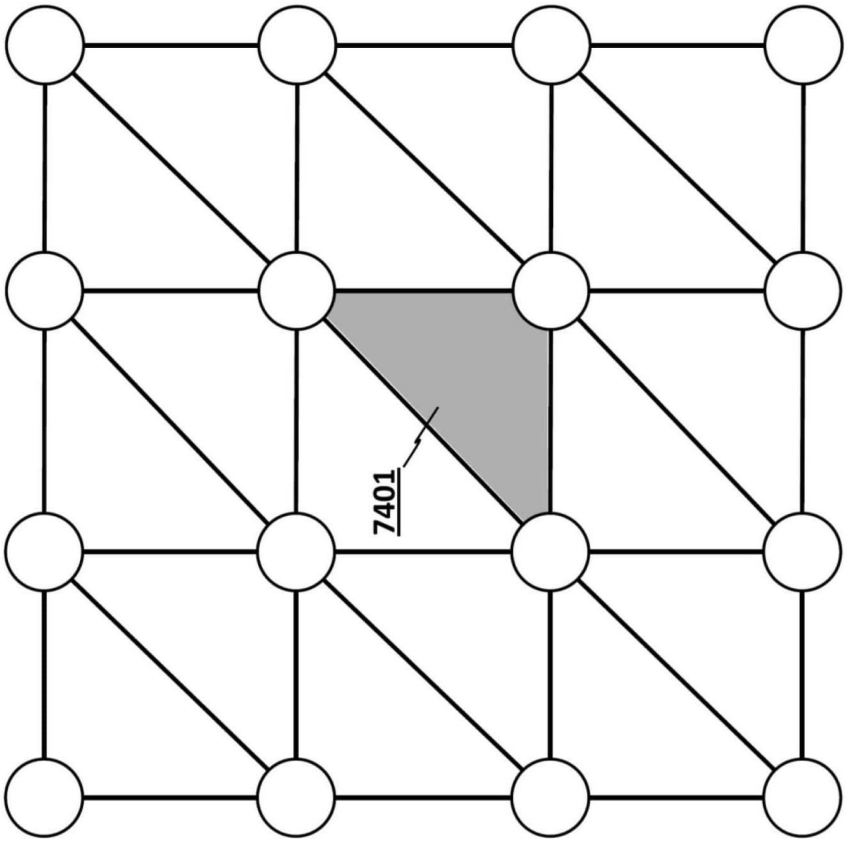


FIG. 74

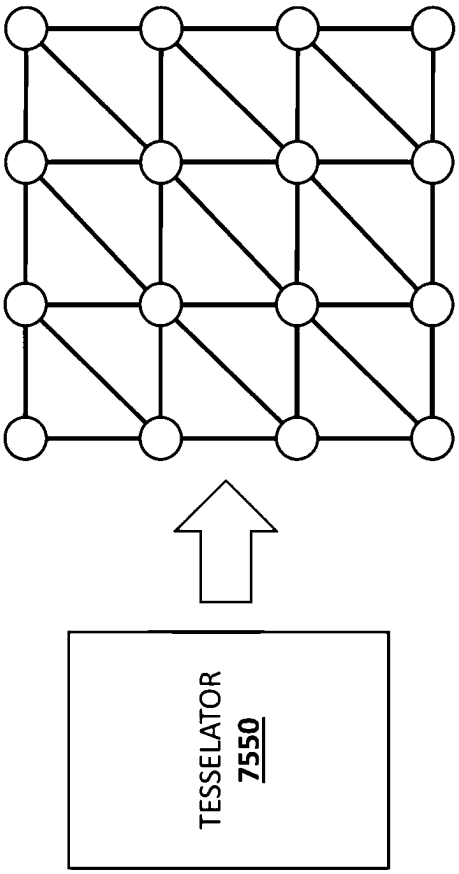


FIG. 75

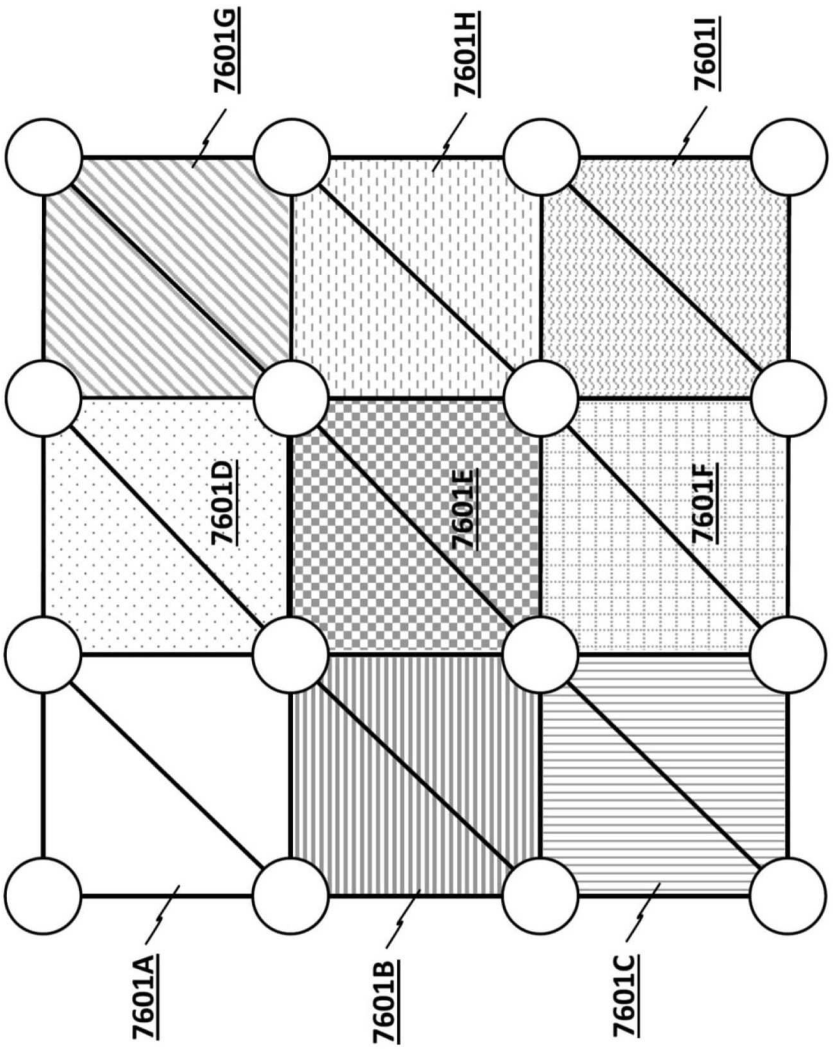


FIG. 76

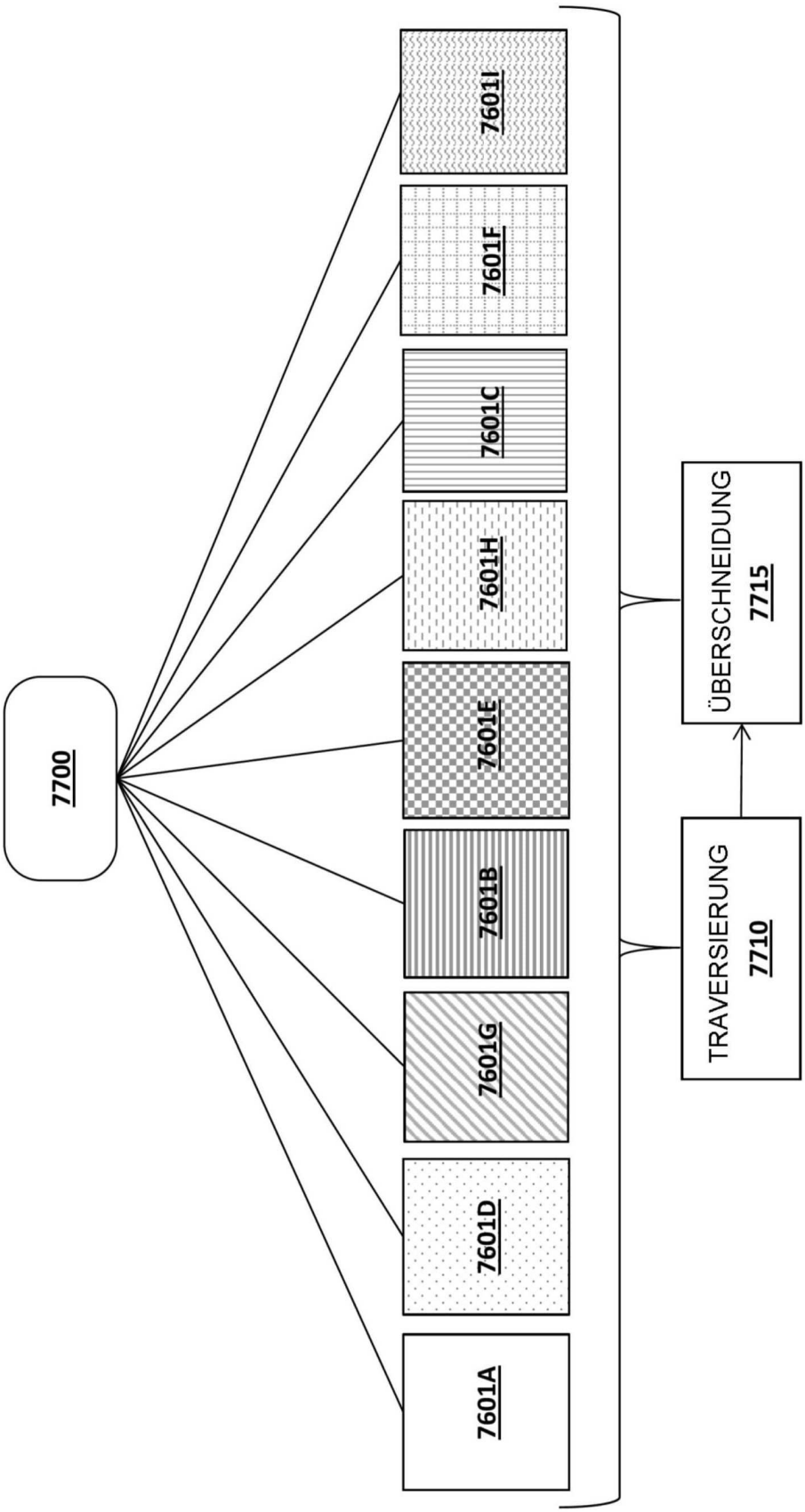

7700
7601A
7601D
7601G
7601B
7601E
7601H
7601C
7601F
7601I
TRAVERSIERUNG
7710
ÜBERSCHNEIDUNG
7715

FIG. 77

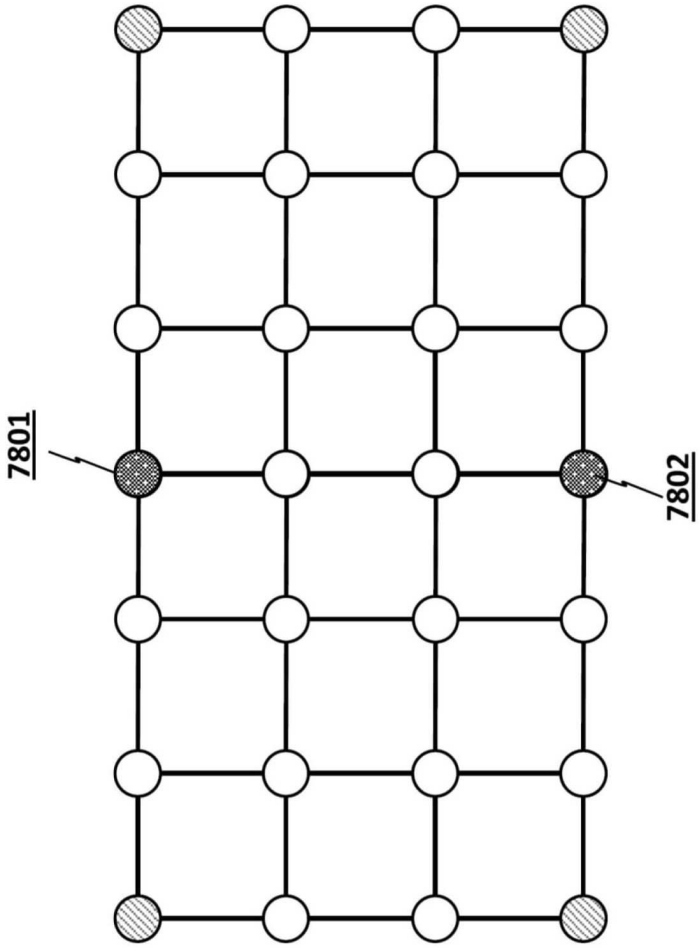


FIG. 78

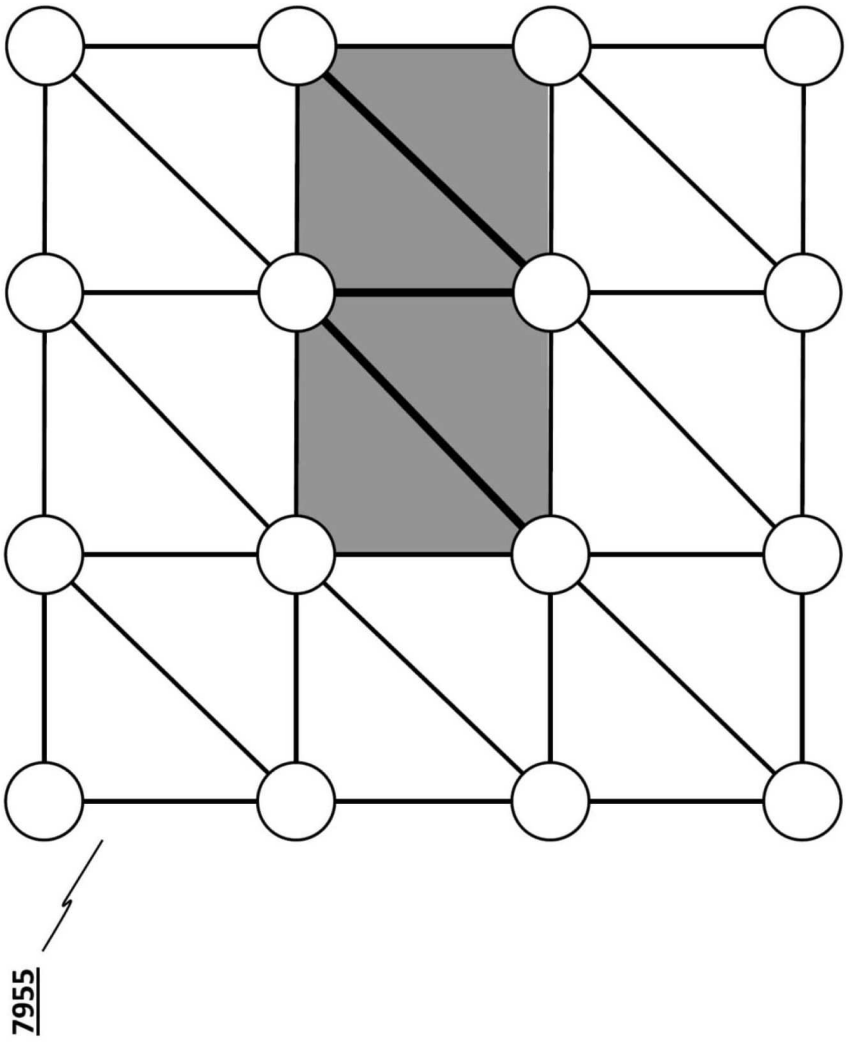


FIG. 79

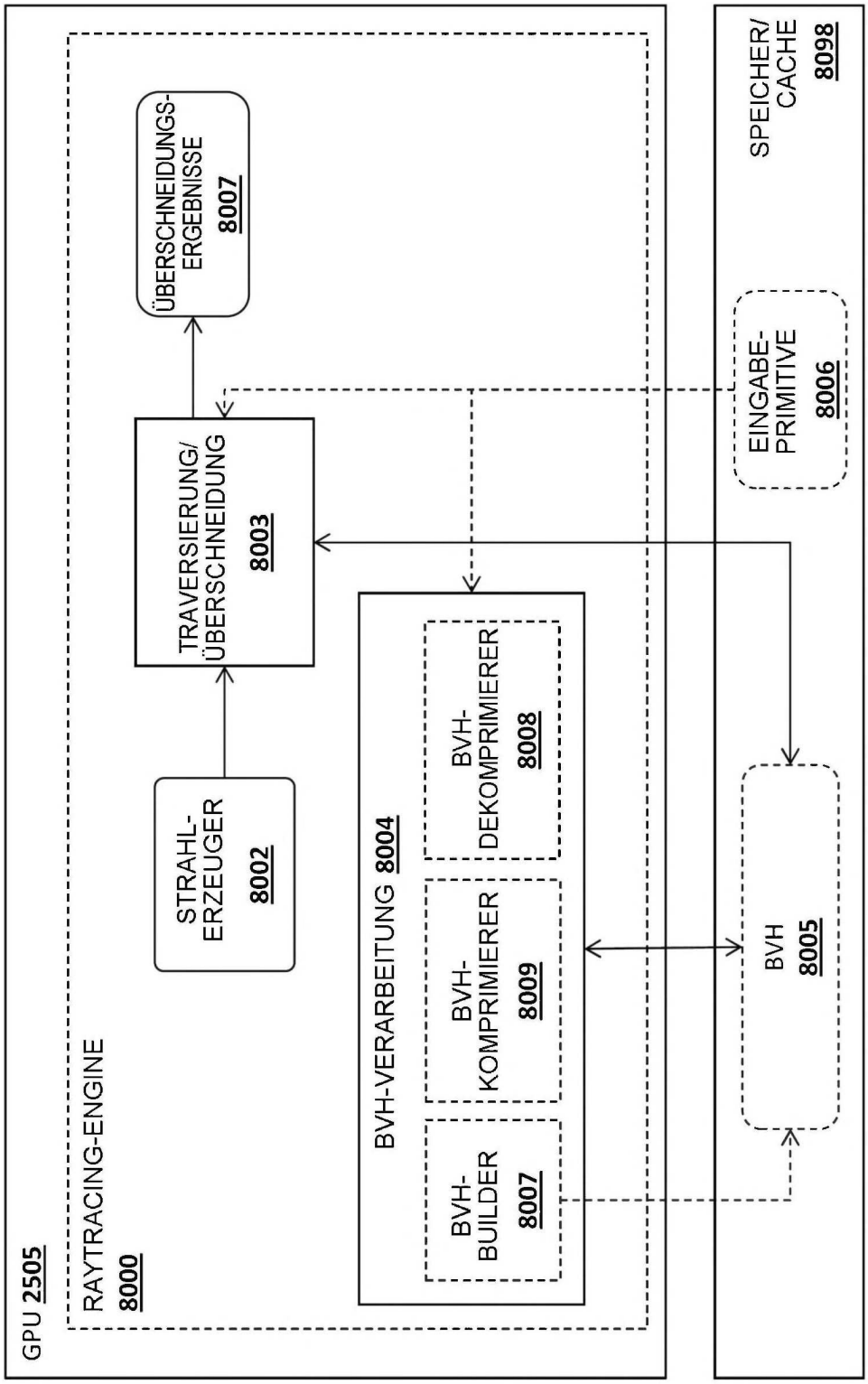

GPU 2505
RAYTRACING-ENGINE 8000
ÜBERSCHNEIDUNGS-ERGEBNISSE 8007
TRAVERSIERUNG/ ÜBERSCHNEIDUNG 8003
STRAHL-ERZEUGER 8002
BVH-VERARBEITUNG 8004
BVH-DEKOMPRIMIERER 8008
BVH-KOMPRIMIERER 8009
BVH-BUILDER 8007
SPEICHER/ CACHE 8098
EINGABE-PRIMITIVE 8006
BVH 8005

FIG. 80

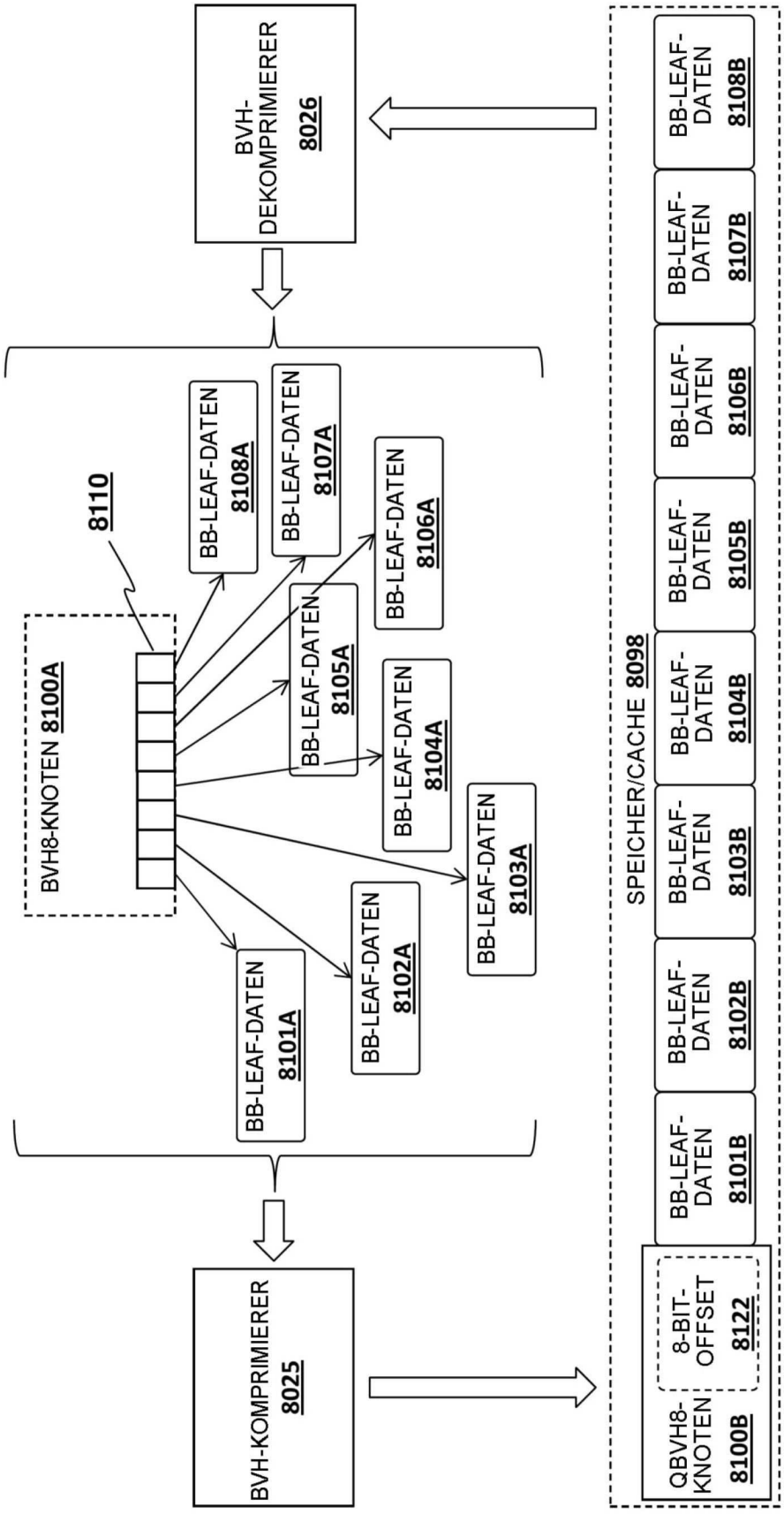

BVH-DEKOMPRIMIERER 8026
BVH8-KNOTEN 8100A
8110
BB-LEAF-DATEN 8108A
BB-LEAF-DATEN 8107A
BB-LEAF-DATEN 8106A
BB-LEAF-DATEN 8105A
BB-LEAF-DATEN 8104A
BB-LEAF-DATEN 8103A
BB-LEAF-DATEN 8102A
BB-LEAF-DATEN 8101A
BVH-KOMPRIMIERER 8025
SPEICHER/CACHE 8098
BB-LEAF-DATEN 8108B
BB-LEAF-DATEN 8107B
BB-LEAF-DATEN 8106B
BB-LEAF-DATEN 8105B
BB-LEAF-DATEN 8104B
BB-LEAF-DATEN 8103B
BB-LEAF-DATEN 8102B
BB-LEAF-DATEN 8101B
8-BIT-OFFSET 8122
QBVH8-KNOTEN 8100B

FIG. 81

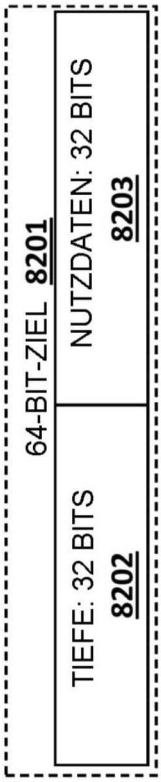


FIG. 82A

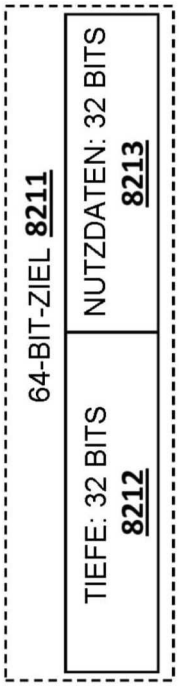


FIG. 82B

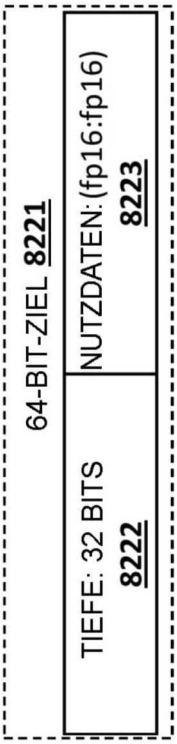


FIG. 82C

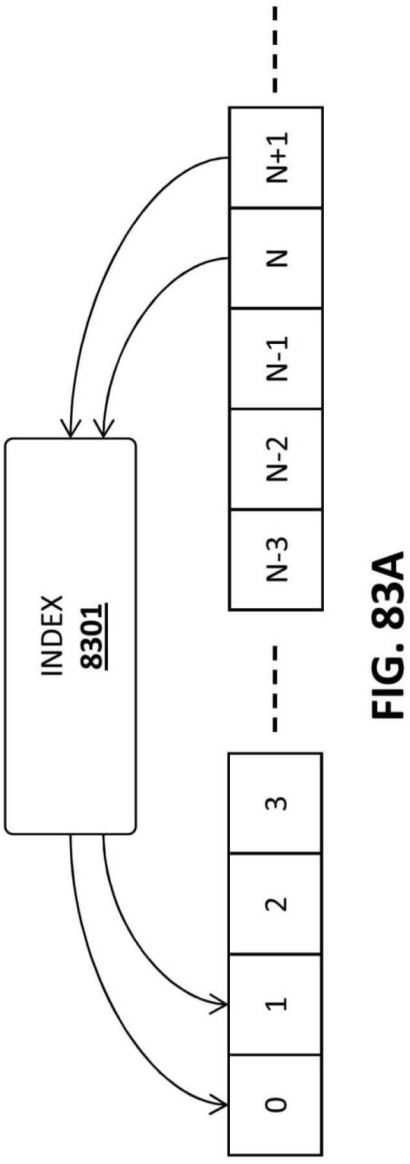


FIG. 83A

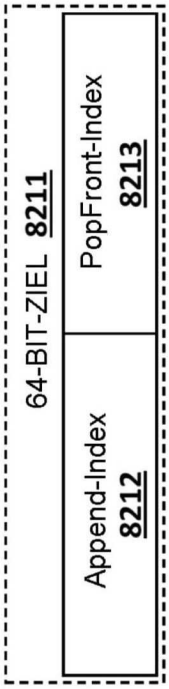


FIG. 83B

Standard-Atomics

```
Producers()
{
 For_all_jobs_for_this_prodcuer()
  {
    for(;;) // Erhalten eines gültigen Write-/AppendIndex
    {
      uint PopIndex = HeadBuffer[0];
      uint WriteIndex = TailBuffer[0], old = 0xffffffff;

      if(((WriteIndex+1)%SIZE_POW2) == (PopIndex% SIZE_POW2))
        continue; // Warteschlange voll

      InterlockedCompareExchange(TailBuffer[0],WriteIndex,
                                 WriteIndex+1,old);

      if(old == AppendIndex) break;
    }

    // Warten, dass die alten Inhalte verbraucht werden
    while( 0 == job_entry_consumed_buffer[WriteIndex] );

    WriteJobEntry(WriteIndex);

    // Signal, dass der neue Eintrag für Verbraucher bereit ist
    job_entry_ready_buffer[WriteIndex] = 1;
  }
}
```

64-Bit-Ringpuffer-Atomics

```
Producers()
{
  For_all_jobs_for_this_prodcuer()
  {
    uint writeIndex=0xffffffff;

    //Erhalten eines gültigen writeIndex - bleibt in der Schleife, bis die Warteschlange nicht mehr voll ist
    while( writeIndex == 0xffffffff )
      InterlockedAppend(QueueBuffer[0],SIZE_POW2, writeIndex );

    // Warten, dass die alten Inhalte verbraucht werden
    while( 0 == job_entry_consumed_buffer[writeIndex] );

    WriteJobEntry(writeIndex);

    // Signal, dass der neue Eintrag für Verbraucher bereit ist
    job_entry_ready_buffer[writeIndex] = 1;
  }
}
```

FIG. 84A

Standard-Atomics

```
Consumers()
{
  for(;;) // endlose Schleife (in Realität nicht nützlich)
  {
    for(;;)// Versuchen, den nächsten Pop-Index in Besitz zu nehmen.
    {
      uint PopIndex = HeadBuffer[0];
      uint AppendIndex = TailBuffer[0], old = 0xffffffff;

      if(((AppendIndex)%SIZE_POW2) == (PopIndex% SIZE_POW2))
        continue; // Warteschlange leer

      InterlockedCompareExchange(HeadBuffer[0], PopIndex,
                                 PopIndex+1,old);

      if( old == PopIndex )
         break;
    }

    // Warten, dass der Index für Verbrauch bereit ist
    while(job_entry_ready_buffer[PopIndex] == 0 );

    JobEntry e = ReadJobEntry(PopIndex);

    ConsumeJobEntry(e);

    job_entry_consumed_buffer[PopIndex] = 1;
  }
}
```

64-Bit-Ringpuffer-Atomics

```
Consumers()
{
  for(;;) // endlose Schleife (in Realität nicht nützlich)
  {
    uint readIndex=0xffffffff;

    // Erhalten eines gültigen readIndex
    while(readIndex==0xffffffff)
      InterlockedPopFront(QueueBuffer[0], SIZE_POW2,readIndex );

    // Warten, dass der Index für Verbrauch bereit ist
    while(job_entry_ready_buffer[readIndex] == 0 );

    JobEntry e = ReadJobEntry(readIndex);

    ConsumeJobEntry(e);

    job_entry_consumed_buffer[readIndex] = 1;
  }
}
```

FIG. 84B

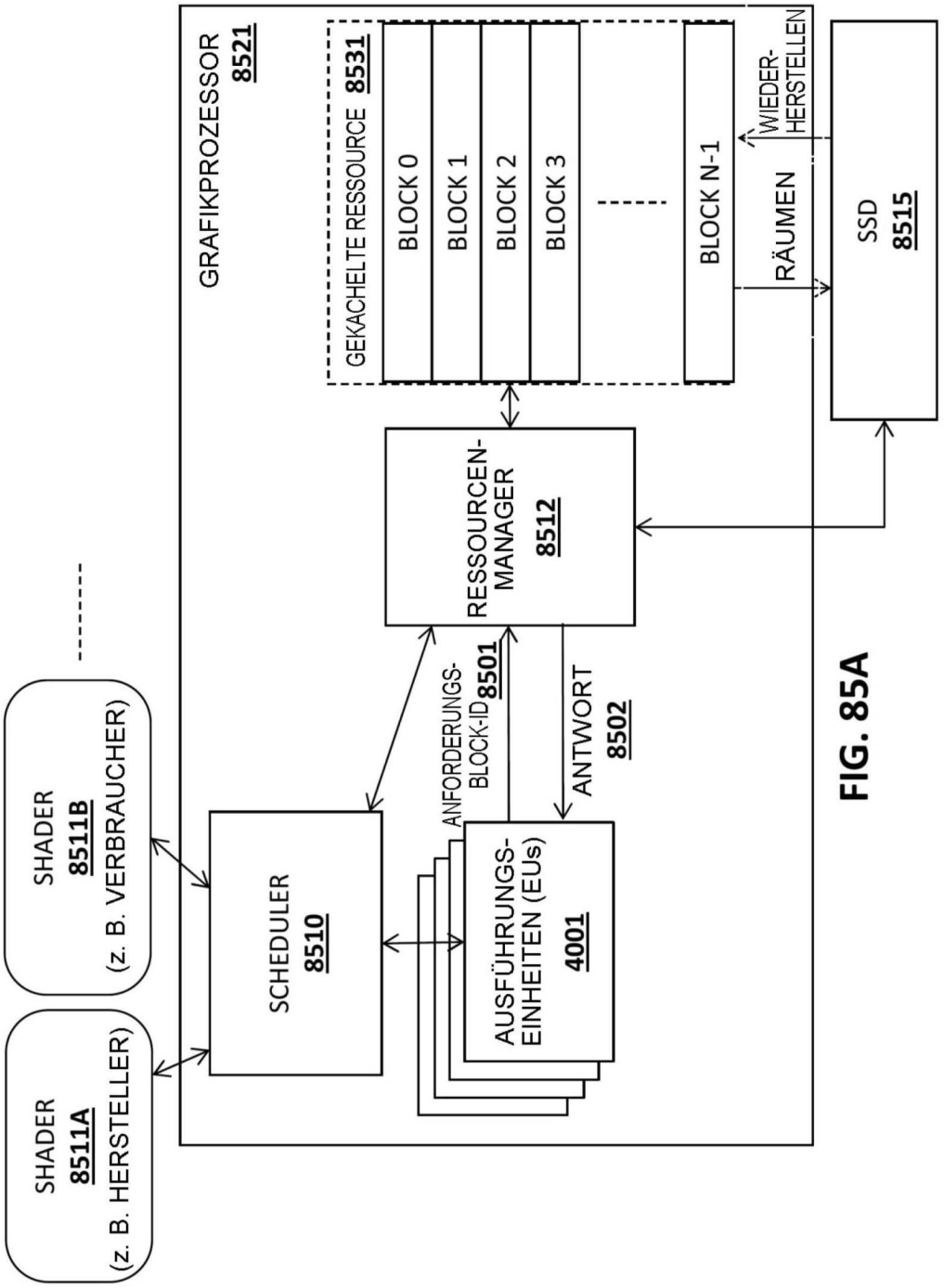

GRAFIKPROZESSOR 8521
GEKACHELTE RESSOURCE 8531
BLOCK 0
BLOCK 1
BLOCK 2
BLOCK 3
BLOCK N-1
WIEDER-HERSTELLEN
RÄUMEN
SSD 8515
RESSOURCEN-MANAGER 8512
ANFORDERUNGS-BLOCK-ID 8501
ANTWORT 8502
SCHEDULER 8510
AUSFÜHRUNGS-EINHEITEN (EUs) 4001
SHADER 8511B (z. B. VERBRAUCHER)
SHADER 8511A (z. B. HERSTELLER)

FIG. 85A

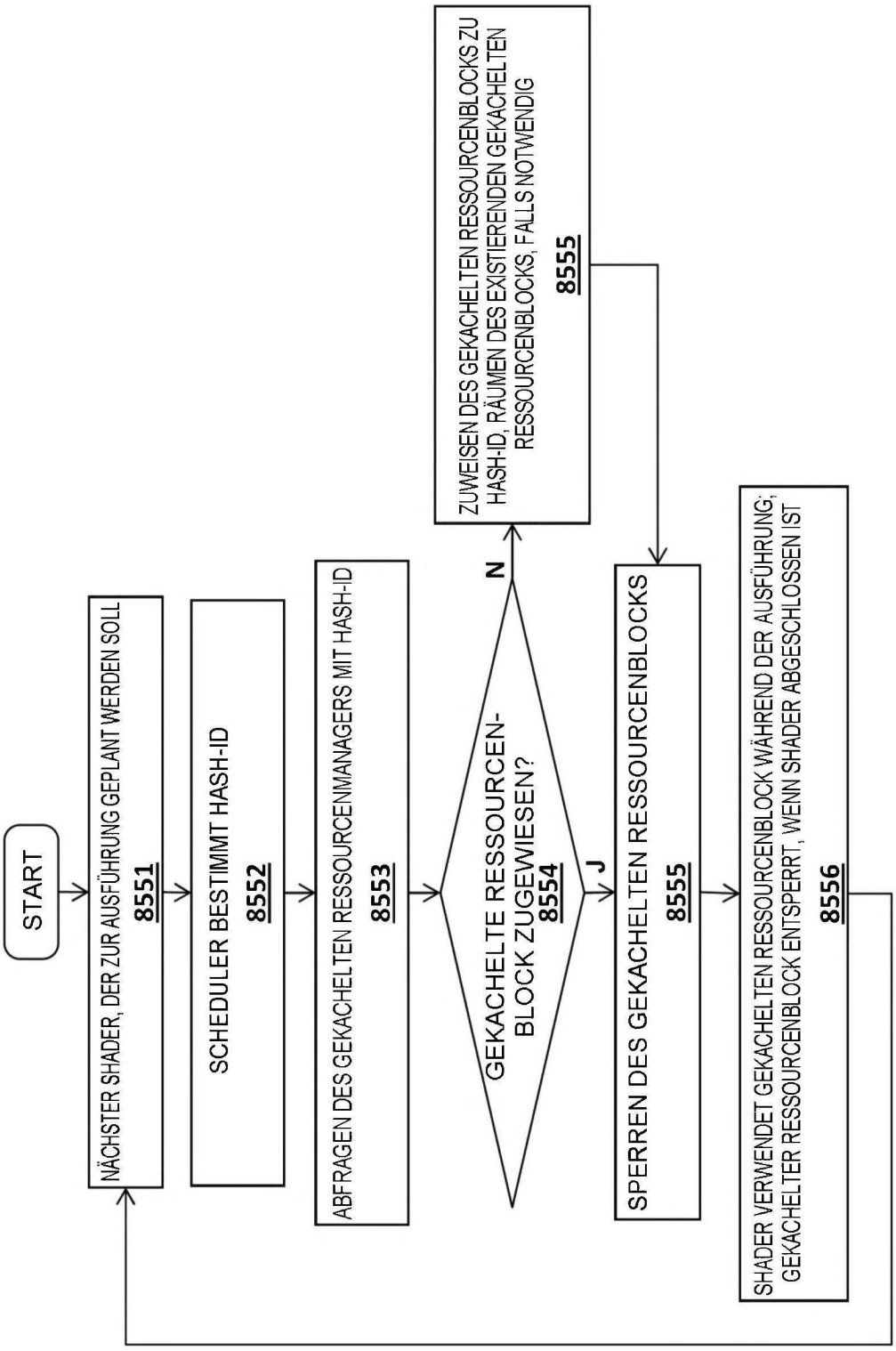

START
NÄCHSTER SHADER, DER ZUR AUSFÜHRUNG GEPLANT WERDEN SOLL
8551
SCHEDULER BESTIMMT HASH-ID
8552
ABFRAGEN DES GEKACHELTEN RESSOURCENMANAGERS MIT HASH-ID
8553
GEKACHELTE RESSOURCEN-BLOCK ZUGEWIESEN?
8554
N
J
ZUWEISEN DES GEKACHELTEN RESSOURCENBLOCKS ZU HASH-ID, RÄUMEN DES EXISTIERENDEN GEKACHELTEN RESSOURCENBLOCKS, FALLS NOTWENDIG
8555
SPERREN DES GEKACHELTEN RESSOURCENBLOCKS
8555
SHADER VERWENDET GEKACHELTEN RESSOURCENBLOCK WÄHREND DER AUSFÜHRUNG; GEKACHELTER RESSOURCENBLOCK ENTSPERRT, WENN SHADER ABGESCHLOSSEN IST
8556

FIG. 85B

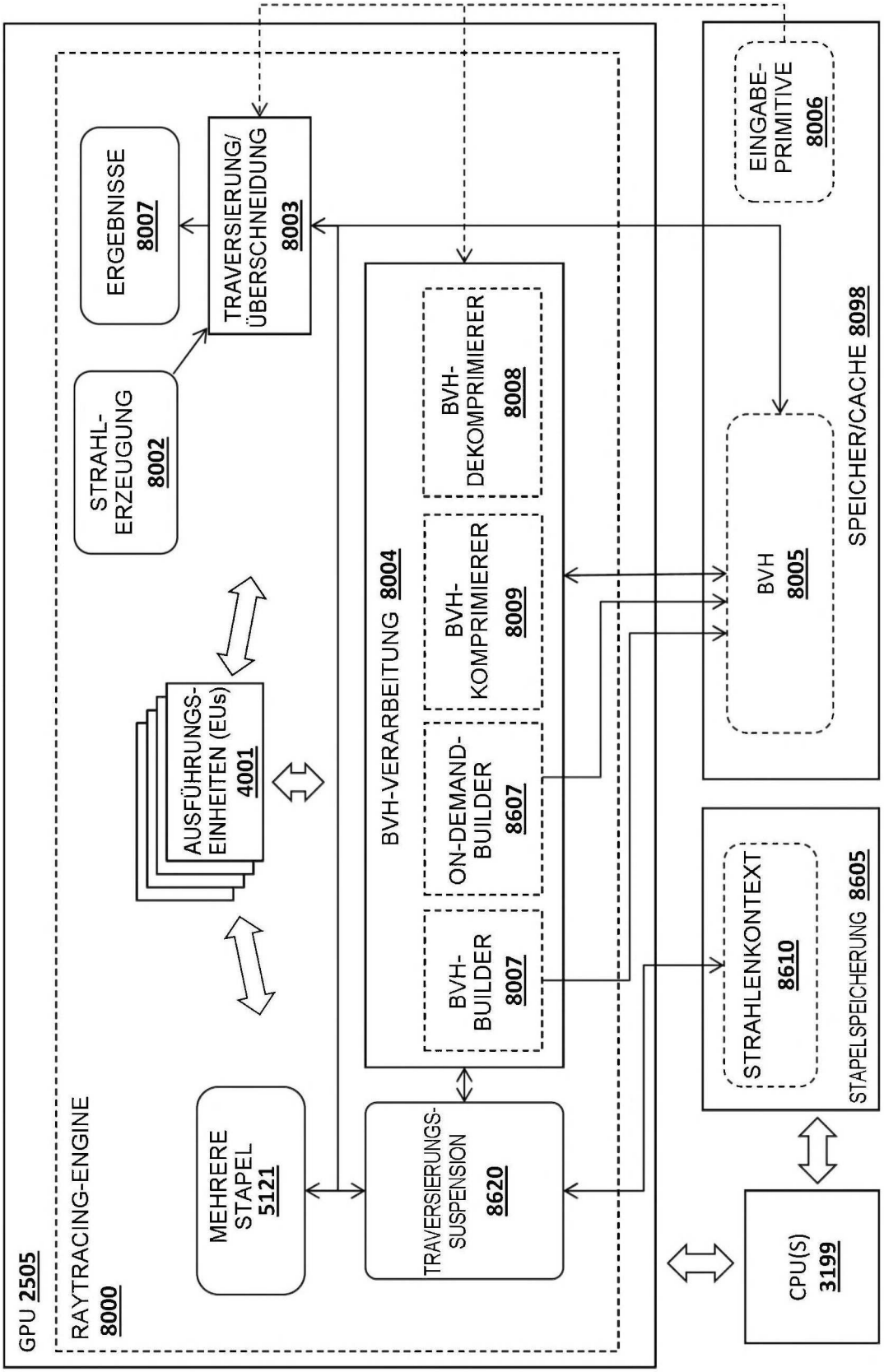

GPU 2505
RAYTRACING-ENGINE 8000
ERGEBNISSE 8007
TRAVERSIERUNG/ ÜBERSCHNEIDUNG 8003
STRAHL-ERZEUGUNG 8002
AUSFÜHRUNGS-EINHEITEN (EUs) 4001
MEHRERE STAPEL 5121
BVH-VERARBEITUNG 8004
BVH-DEKOMPRIMIERER 8008
BVH-KOMPRIMIERER 8009
ON-DEMAND-BUILDER 8607
BVH-BUILDER 8007
TRAVERSIERUNGS-SUSPENSION 8620
EINGABE-PRIMITIVE 8006
BVH 8005
SPEICHER/CACHE 8098
STRAHLENKONTEXT 8610
STAPELSPEICHERUNG 8605
CPU(S) 3199

FIG. 86A

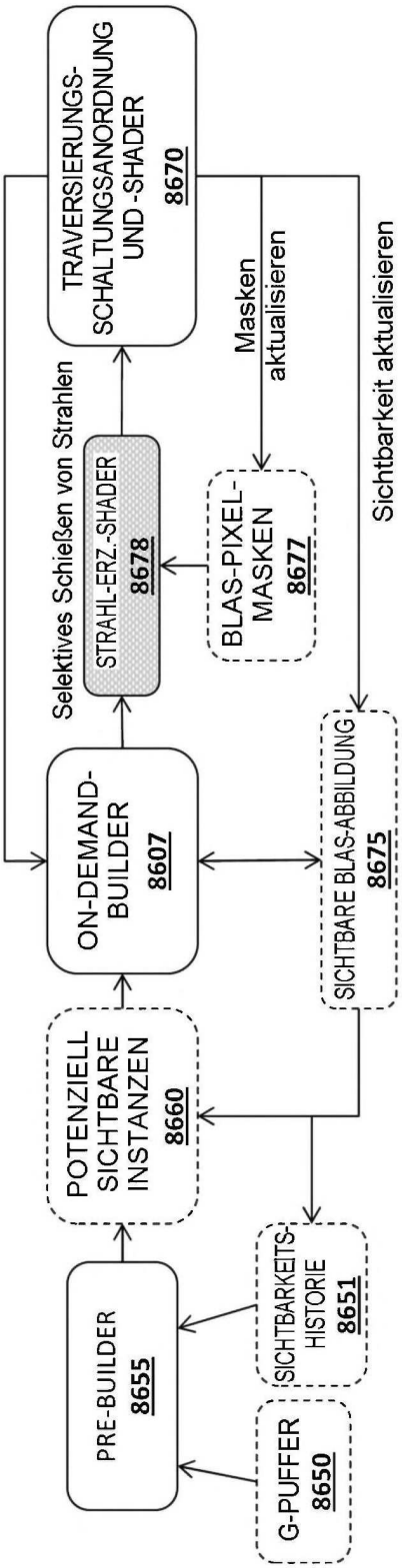

TRAVERSIERUNGS-SCHALTUNGSANORDNUNG UND -SHADER 8670
Masken aktualisieren
Sichtbarkeit aktualisieren
Selektives Schießen von Strahlen
STRAHL-ERZ.-SHADER 8678
BLAS-PIXEL-MASKEN 8677
ON-DEMAND-BUILDER 8607
SICHTBARE BLAS-ABBILDUNG 8675
POTENZIELL SICHTBARE INSTANZEN 8660
PRE-BUILDER 8655
SICHTBARKEITS-HISTORIE 8651
G-PUFFER 8650

FIG. 86B

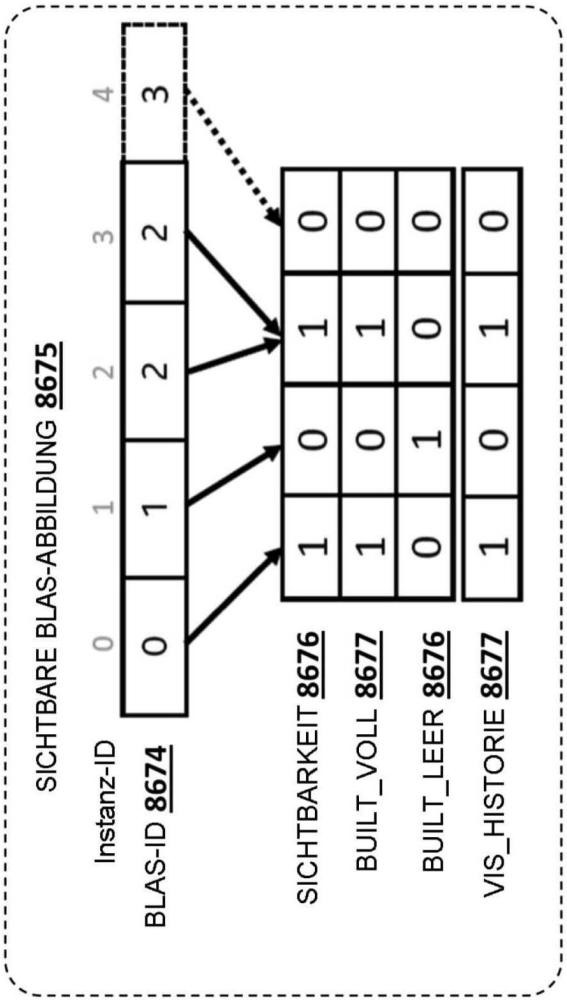

SICHTBARE BLAS-ABBILDUNG 8675
Instanz-ID
BLAS-ID 8674
0
1
2
3
4
0
1
2
3
SICHTBARKEIT 8676
BUILT_VOLL 8677
BUILT_LEER 8676
VIS_HISTORIE 8677
1 0 1 0
1 0 1 0
0 1 0 0
1 0 1 0

FIG. 86C

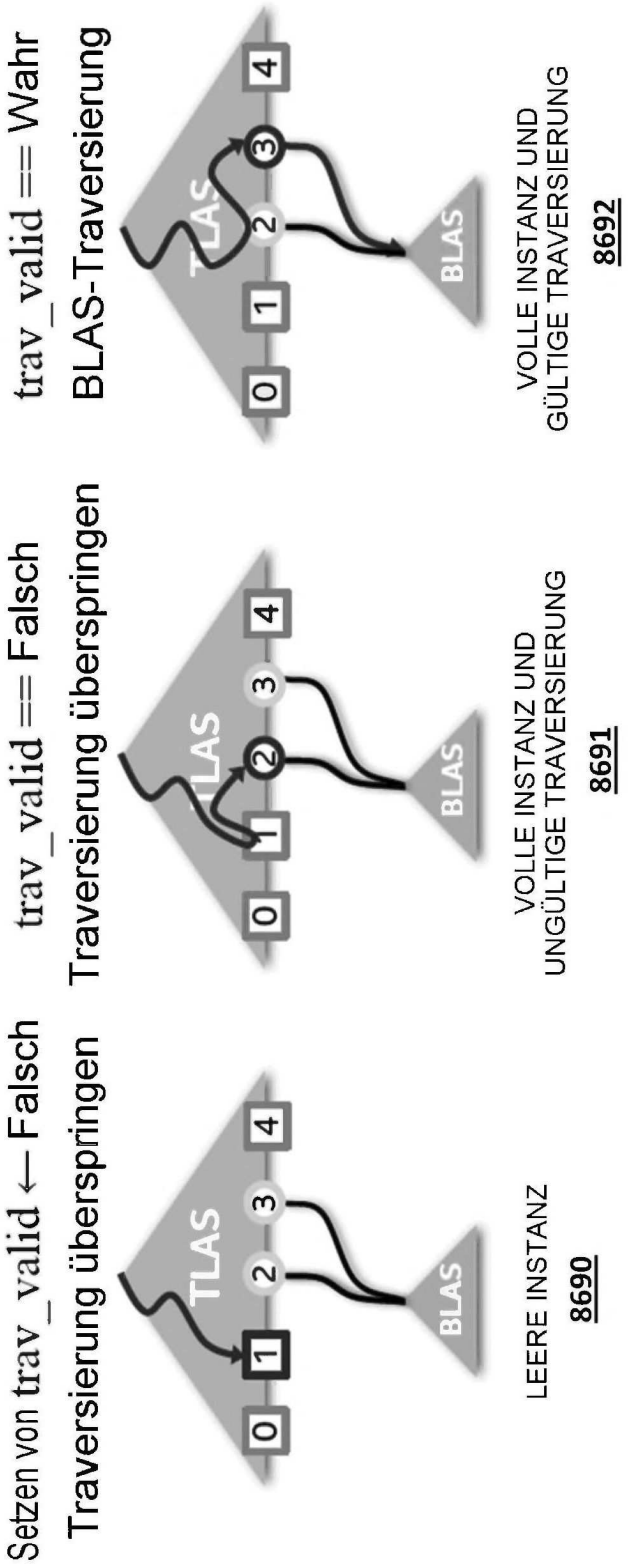

Setzen von trav_valid ← Falsch
Traversierung überspringen
TLAS
BLAS
0
1
2
3
4
LEERE INSTANZ
8690
trav_valid == Falsch
Traversierung überspringen
VOLLE INSTANZ UND
UNGÜLTIGE TRAVERSIERUNG
8691
trav_valid == Wahr
BLAS-Traversierung
VOLLE INSTANZ UND
GÜLTIGE TRAVERSIERUNG
8692

FIG. 86D

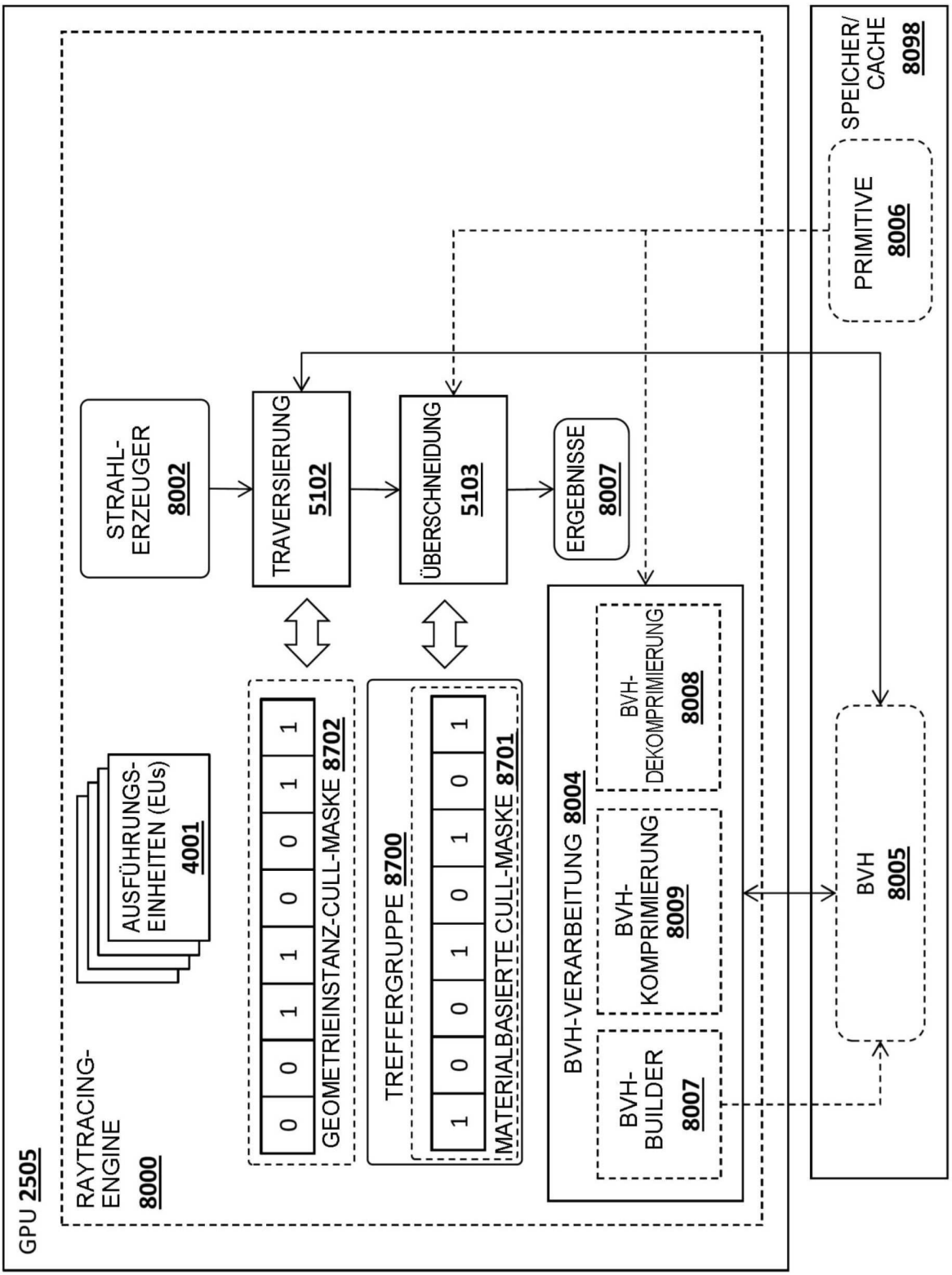

GPU 2505
RAYTRACING-ENGINE 8000
AUSFÜHRUNGS-EINHEITEN (EUs) 4001
STRAHL-ERZEUGER 8002
TRAVERSIERUNG 5102
ÜBERSCHNEIDUNG 5103
ERGEBNISSE 8007
GEOMETRIEINSTANZ-CULL-MASKE 8702
0 0 1 1 0 0 1 1
TREFFERGRUPPE 8700
MATERIALBASIERTE CULL-MASKE 8701
1 0 0 1 0 1 0 1
BVH-VERARBEITUNG 8004
BVH-BUILDER 8007
BVH-KOMPRIMIERUNG 8009
BVH-DEKOMPRIMIERUNG 8008
SPEICHER/CACHE 8098
PRIMITIVE 8006
BVH 8005

FIG. 87

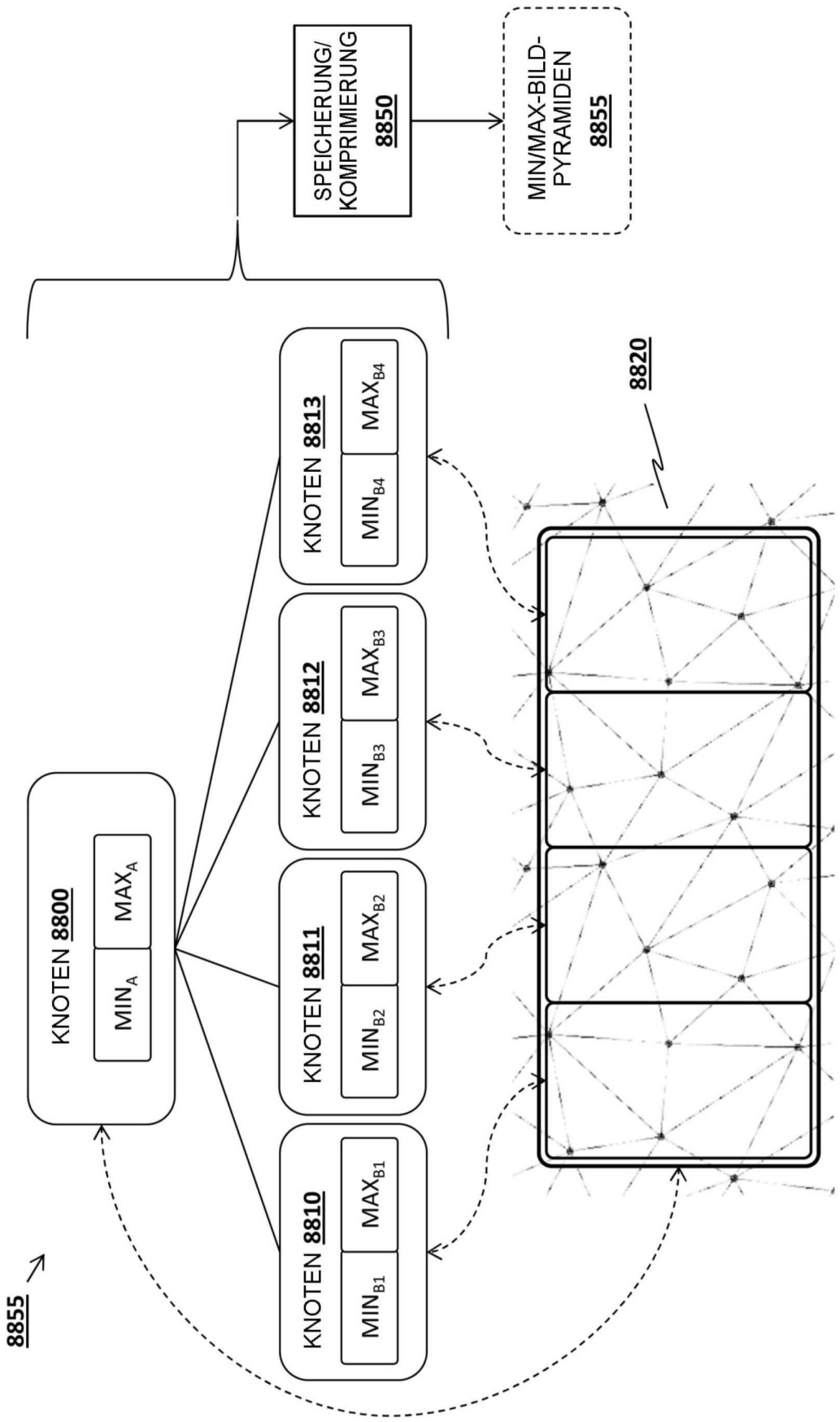

8855
KNOTEN 8800
MIN$_A$
MAX$_A$
KNOTEN 8810
MIN$_{B1}$
MAX$_{B1}$
KNOTEN 8811
MIN$_{B2}$
MAX$_{B2}$
KNOTEN 8812
MIN$_{B3}$
MAX$_{B3}$
KNOTEN 8813
MIN$_{B4}$
MAX$_{B4}$
SPEICHERUNG/ KOMPRIMIERUNG 8850
MIN/MAX-BILD-PYRAMIDEN 8855
8820

FIG. 88

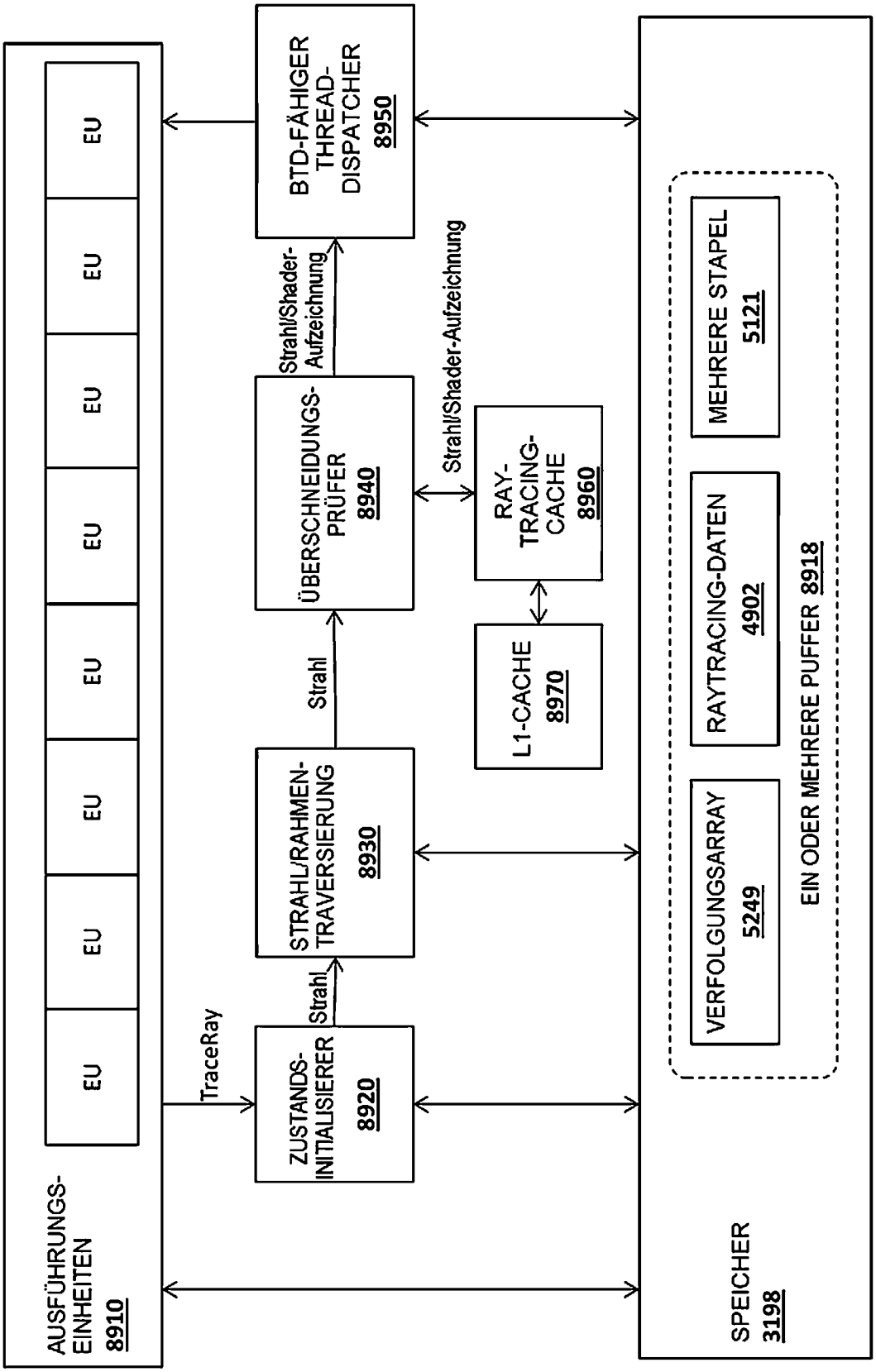

AUSFÜHRUNGS-EINHEITEN 8910
EU
EU
EU
EU
EU
EU
EU
EU
TraceRay
ZUSTANDS-INITIALISIERER 8920
Strahl
STRAHL/RAHMEN-TRAVERSIERUNG 8930
Strahl
ÜBERSCHNEIDUNGS-PRÜFER 8940
Strahl/Shader-Aufzeichnung
BTD-FÄHIGER THREAD-DISPATCHER 8950
Strahl/Shader-Aufzeichnung
RAY-TRACING-CACHE 8960
L1-CACHE 8970
SPEICHER 3198
EIN ODER MEHRERE PUFFER 8918
VERFOLGUNGSARRAY 5249
RAYTRACING-DATEN 4902
MEHRERE STAPEL 5121

FIG. 89A

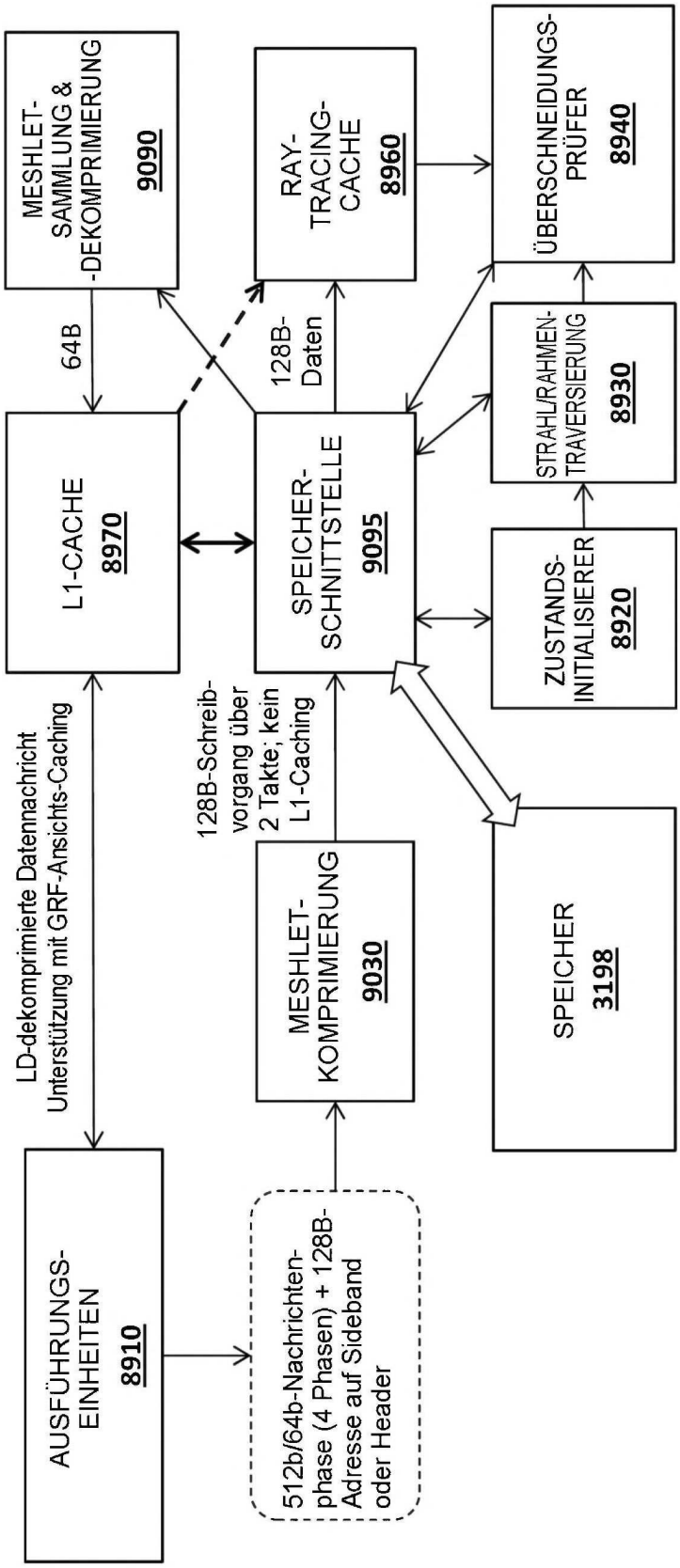

MESHLET-SAMMLUNG & -DEKOMPRIMIERUNG 9090
RAY-TRACING-CACHE 8960
ÜBERSCHNEIDUNGS-PRÜFER 8940
64B
128B-Daten
L1-CACHE 8970
SPEICHER-SCHNITTSTELLE 9095
STRAHL/RAHMEN-TRAVERSIERUNG 8930
ZUSTANDS-INITIALISIERER 8920
LD-dekomprimierte Datennachricht Unterstützung mit GRF-Ansichts-Caching
128B-Schreib-vorgang über 2 Takte; kein L1-Caching
MESHLET-KOMPRIMIERUNG 9030
SPEICHER 3198
AUSFÜHRUNGS-EINHEITEN 8910
512b/64b-Nachrichten-phase (4 Phasen) + 128B-Adresse auf Sideband oder Header

FIG. 89B

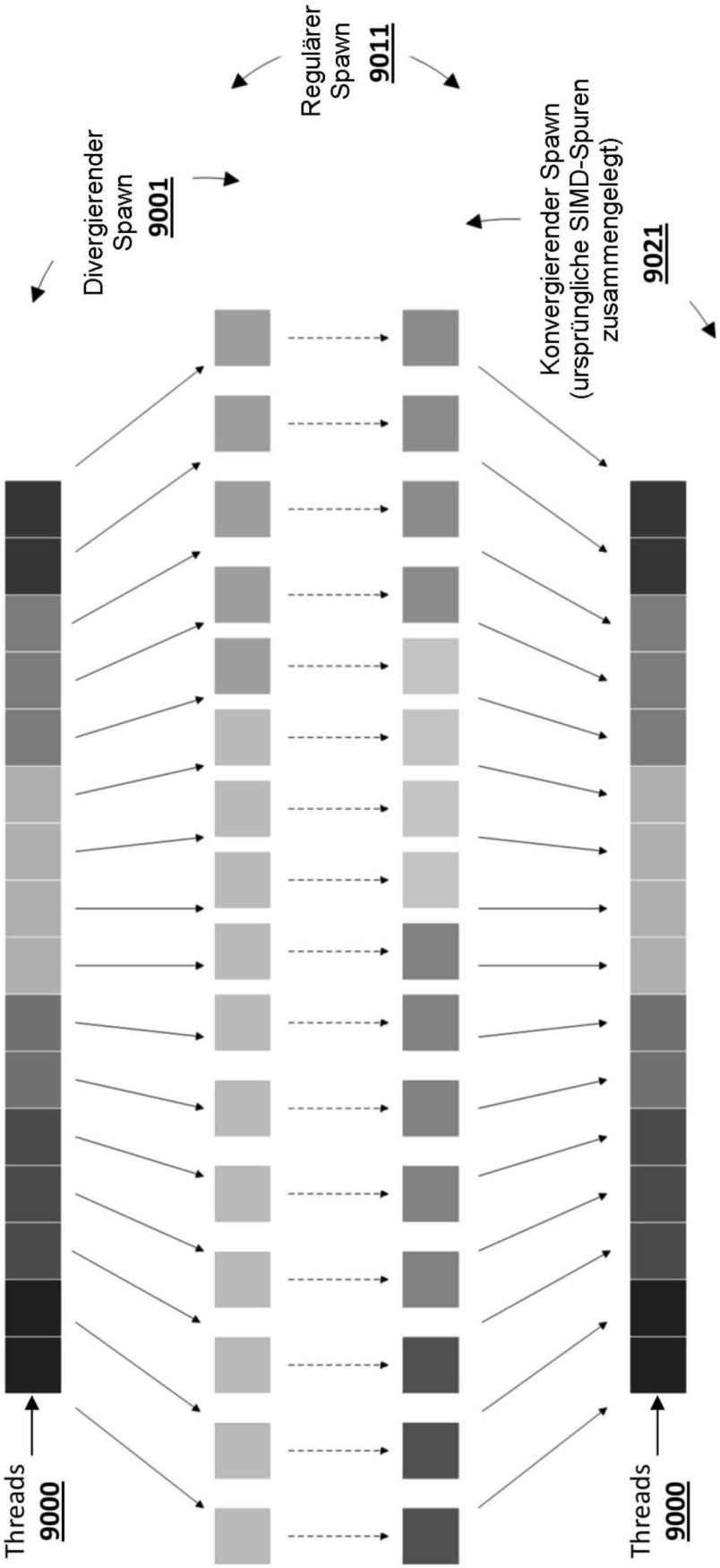

Divergierender Spawn 9001
Regulärer Spawn 9011
Konvergierender Spawn (ursprüngliche SIMD-Spuren zusammengelegt) 9021
Threads 9000
Threads 9000

FIG. 90

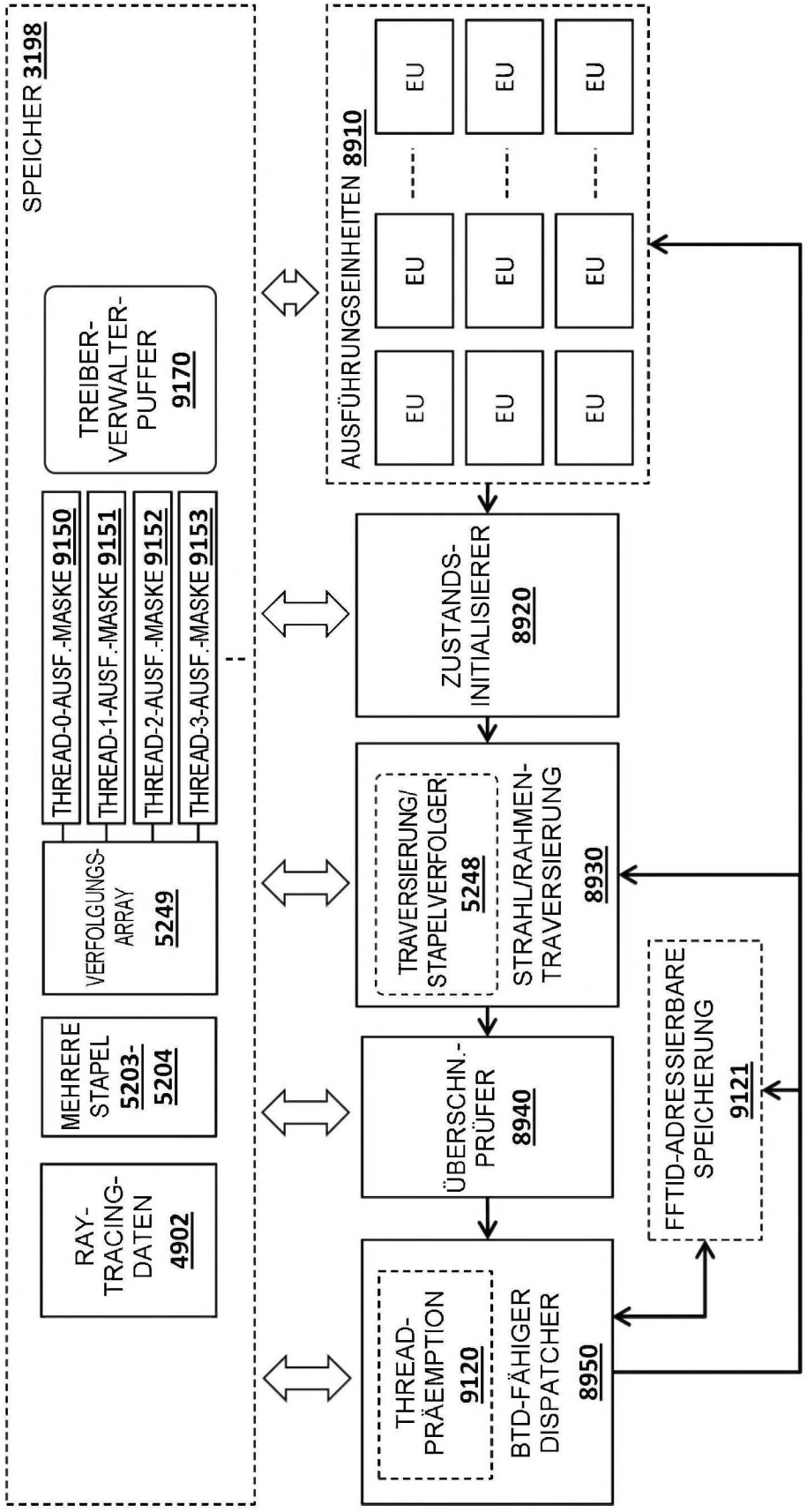

SPEICHER 3198
TREIBER-VERWALTER-PUFFER 9170
THREAD-0-AUSF.-MASKE 9150
THREAD-1-AUSF.-MASKE 9151
THREAD-2-AUSF.-MASKE 9152
THREAD-3-AUSF.-MASKE 9153
VERFOLGUNGS-ARRAY 5249
MEHRERE STAPEL 5203-5204
RAY-TRACING-DATEN 4902
AUSFÜHRUNGSEINHEITEN 8910
EU
ZUSTANDS-INITIALISIERER 8920
TRAVERSIERUNG/STAPELVERFOLGER 5248
STRAHL/RAHMEN-TRAVERSIERUNG 8930
ÜBERSCHN.-PRÜFER 8940
FFTID-ADRESSIERBARE SPEICHERUNG 9121
THREAD-PRÄEMPTION 9120
BTD-FÄHIGER DISPATCHER 8950

FIG. 91

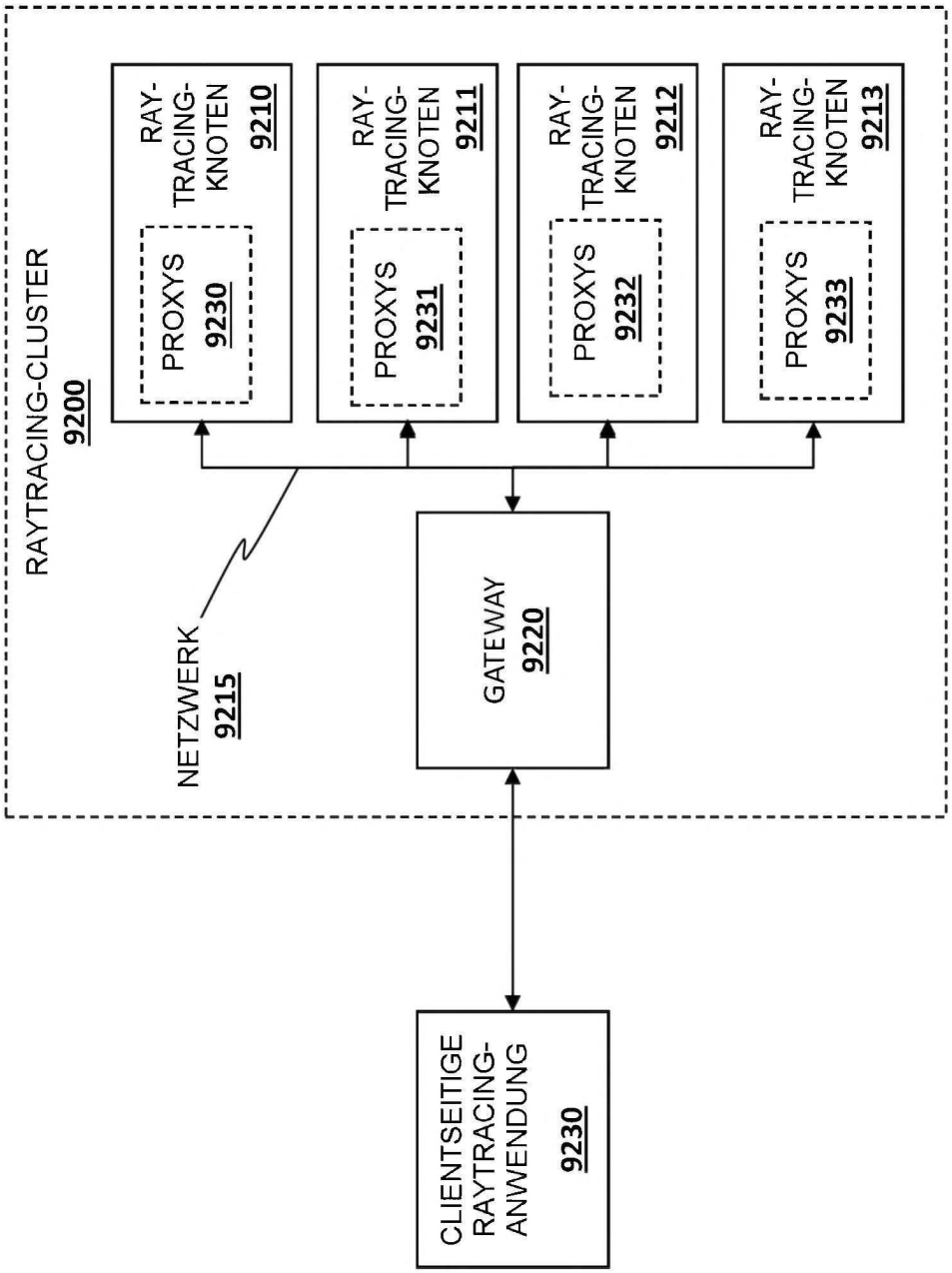

RAYTRACING-CLUSTER
9200
RAY-TRACING-KNOTEN 9210
PROXYS 9230
RAY-TRACING-KNOTEN 9211
PROXYS 9231
RAY-TRACING-KNOTEN 9212
PROXYS 9232
RAY-TRACING-KNOTEN 9213
PROXYS 9233
NETZWERK 9215
GATEWAY 9220
CLIENTSEITIGE RAYTRACING-ANWENDUNG 9230

FIG. 92

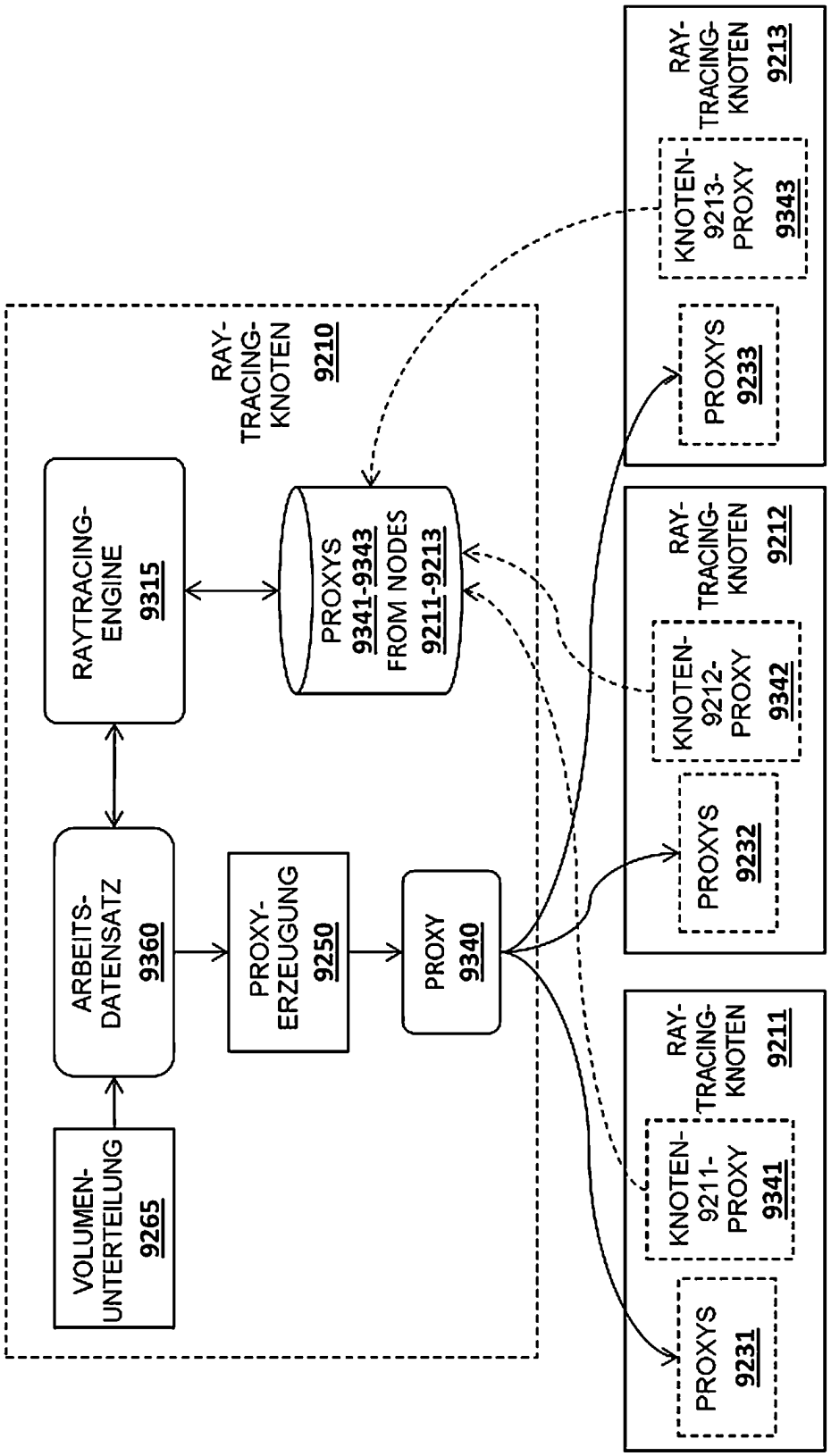

RAY-TRACING-KNOTEN 9210
VOLUMEN-UNTERTEILUNG 9265
ARBEITS-DATENSATZ 9360
RAYTRACING-ENGINE 9315
PROXY-ERZEUGUNG 9250
PROXY 9340
PROXYS 9341-9343 FROM NODES 9211-9213
RAY-TRACING-KNOTEN 9211
KNOTEN-9211-PROXY 9341
PROXYS 9231
RAY-TRACING-KNOTEN 9212
KNOTEN-9212-PROXY 9342
PROXYS 9232
RAY-TRACING-KNOTEN 9213
KNOTEN-9213-PROXY 9343
PROXYS 9233

FIG. 93

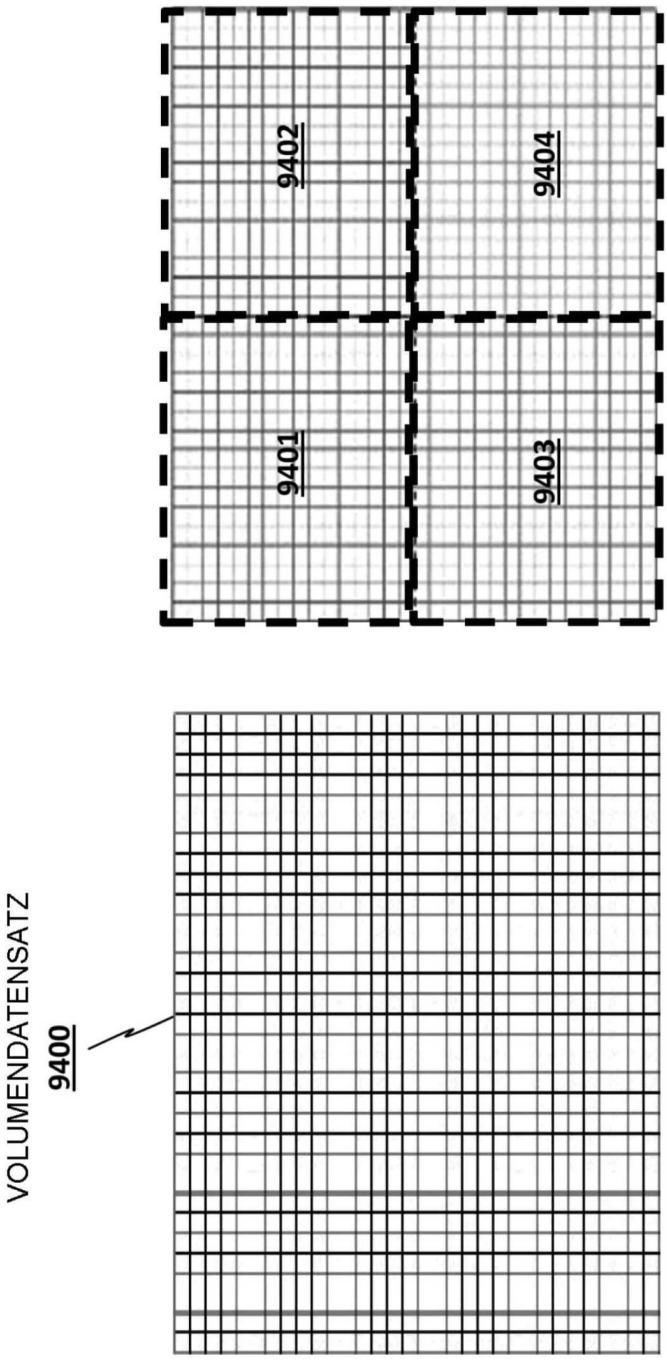


FIG. 94

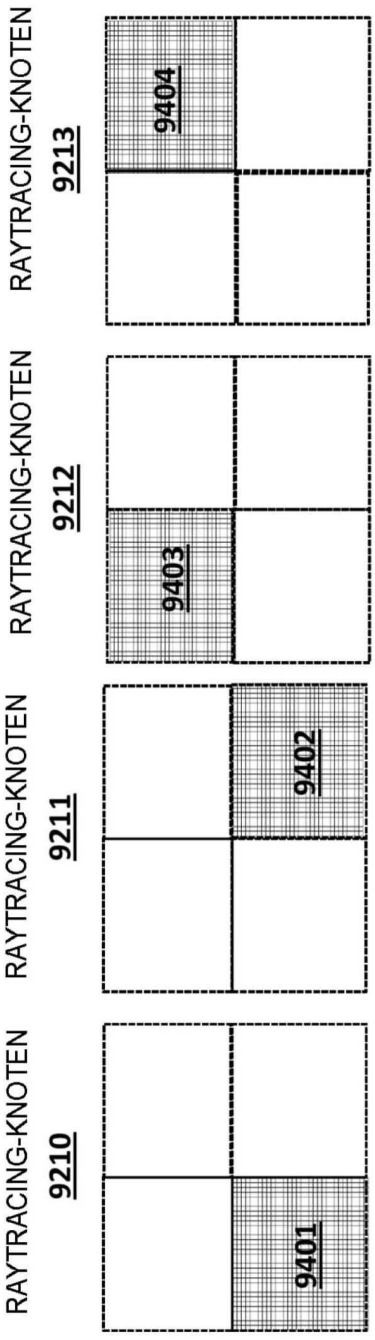

RAYTRACING-KNOTEN
9210
9401
RAYTRACING-KNOTEN
9211
9402
RAYTRACING-KNOTEN
9212
9403
RAYTRACING-KNOTEN
9213
9404

FIG. 95

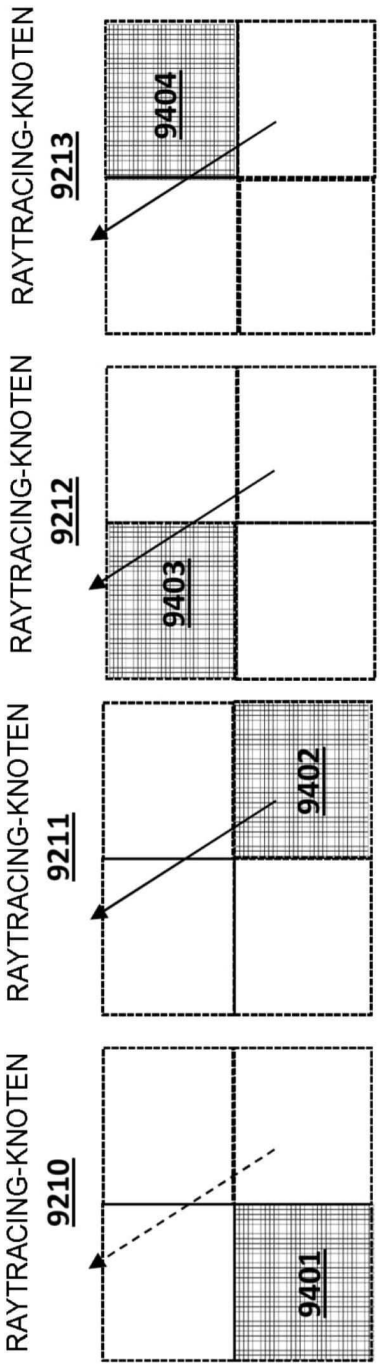

RAYTRACING-KNOTEN
9213
9404
RAYTRACING-KNOTEN
9212
9403
RAYTRACING-KNOTEN
9211
9402
RAYTRACING-KNOTEN
9210
9401

FIG. 96

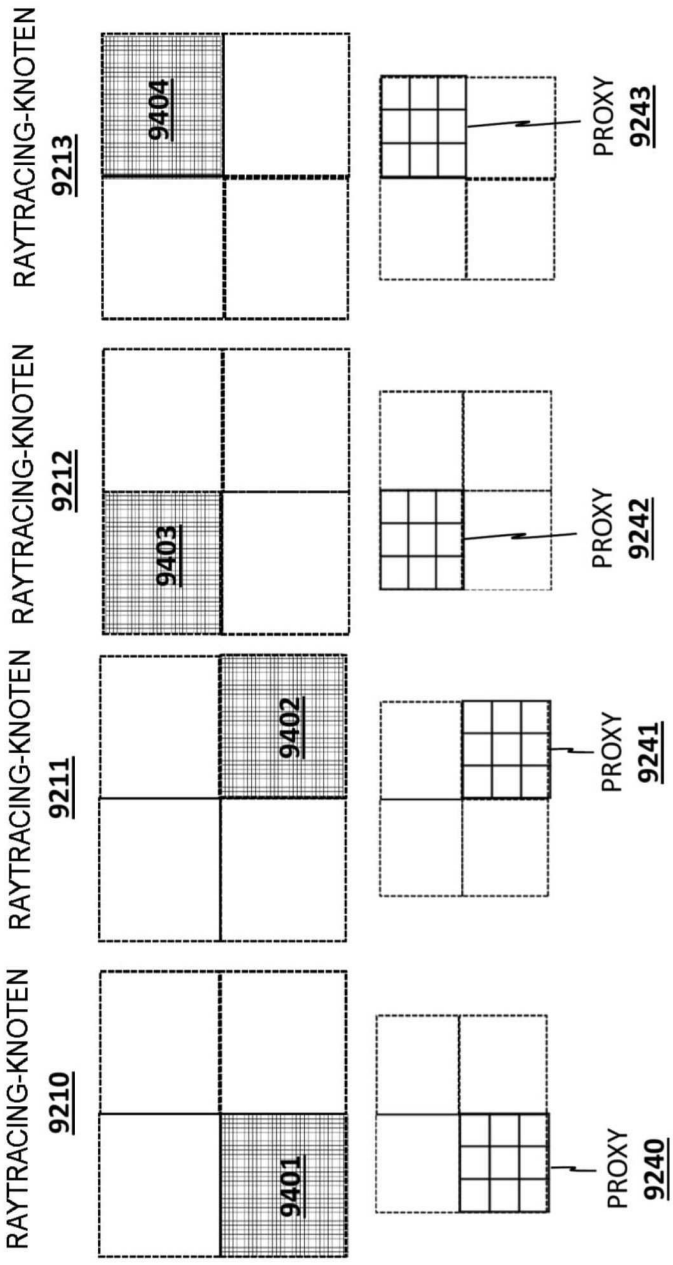

RAYTRACING-KNOTEN
9210
RAYTRACING-KNOTEN
9211
RAYTRACING-KNOTEN
9212
RAYTRACING-KNOTEN
9213
9401
9402
9403
9404
PROXY
9240
PROXY
9241
PROXY
9242
PROXY
9243

FIG. 97

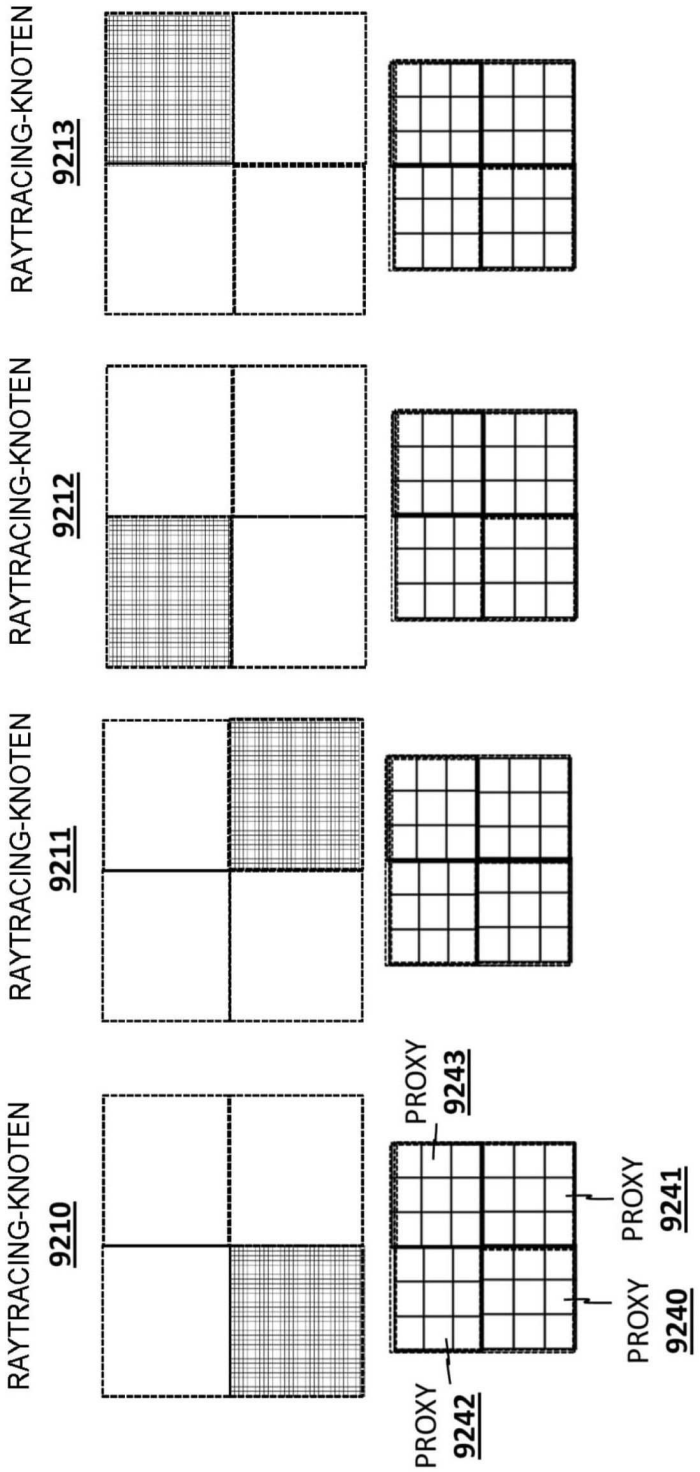

RAYTRACING-KNOTEN
9213
RAYTRACING-KNOTEN
9212
RAYTRACING-KNOTEN
9211
RAYTRACING-KNOTEN
9210
PROXY
9243
PROXY
9241
PROXY
9240
PROXY
9242

FIG. 98

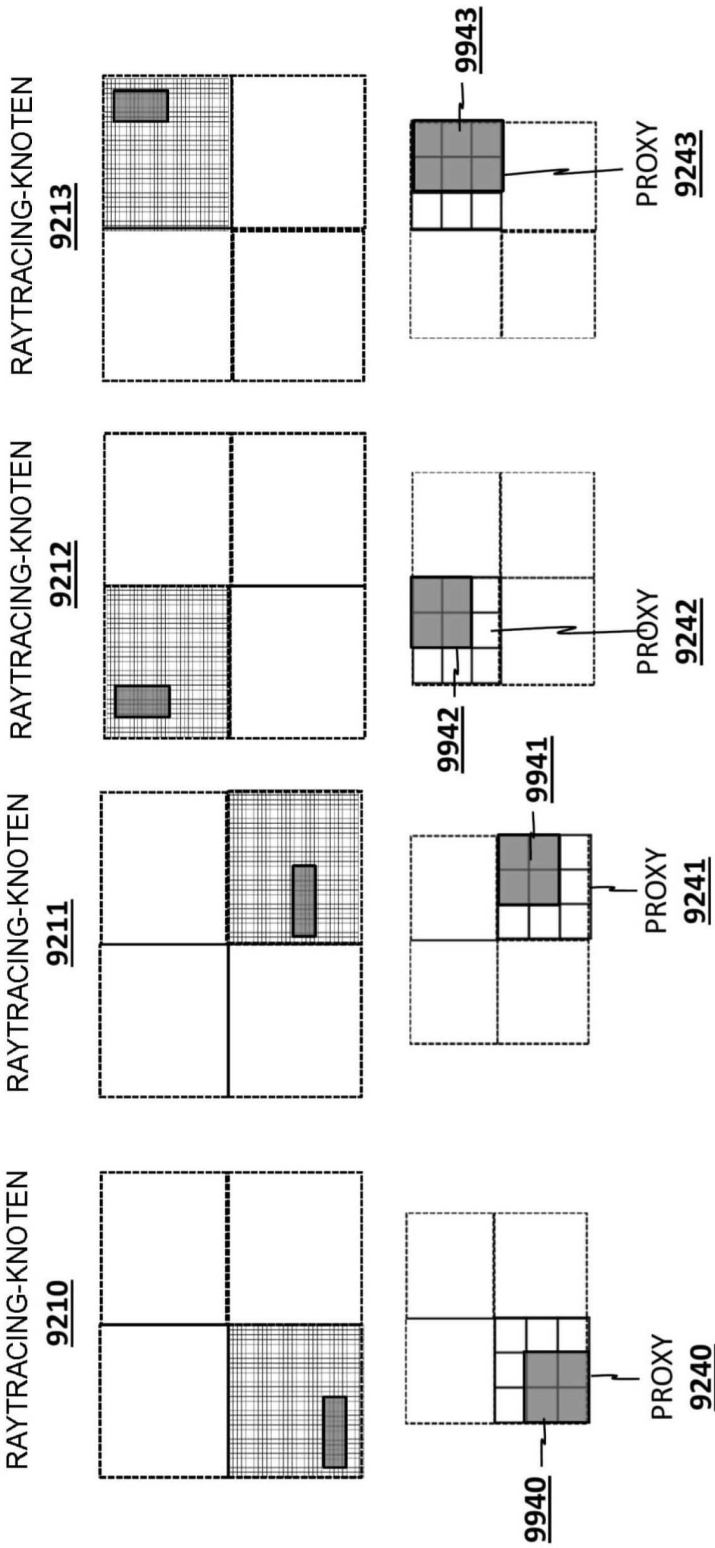

RAYTRACING-KNOTEN
9213
RAYTRACING-KNOTEN
9212
RAYTRACING-KNOTEN
9211
RAYTRACING-KNOTEN
9210
9943
PROXY
9243
9942
PROXY
9242
9941
PROXY
9241
9940
PROXY
9240

FIG. 99

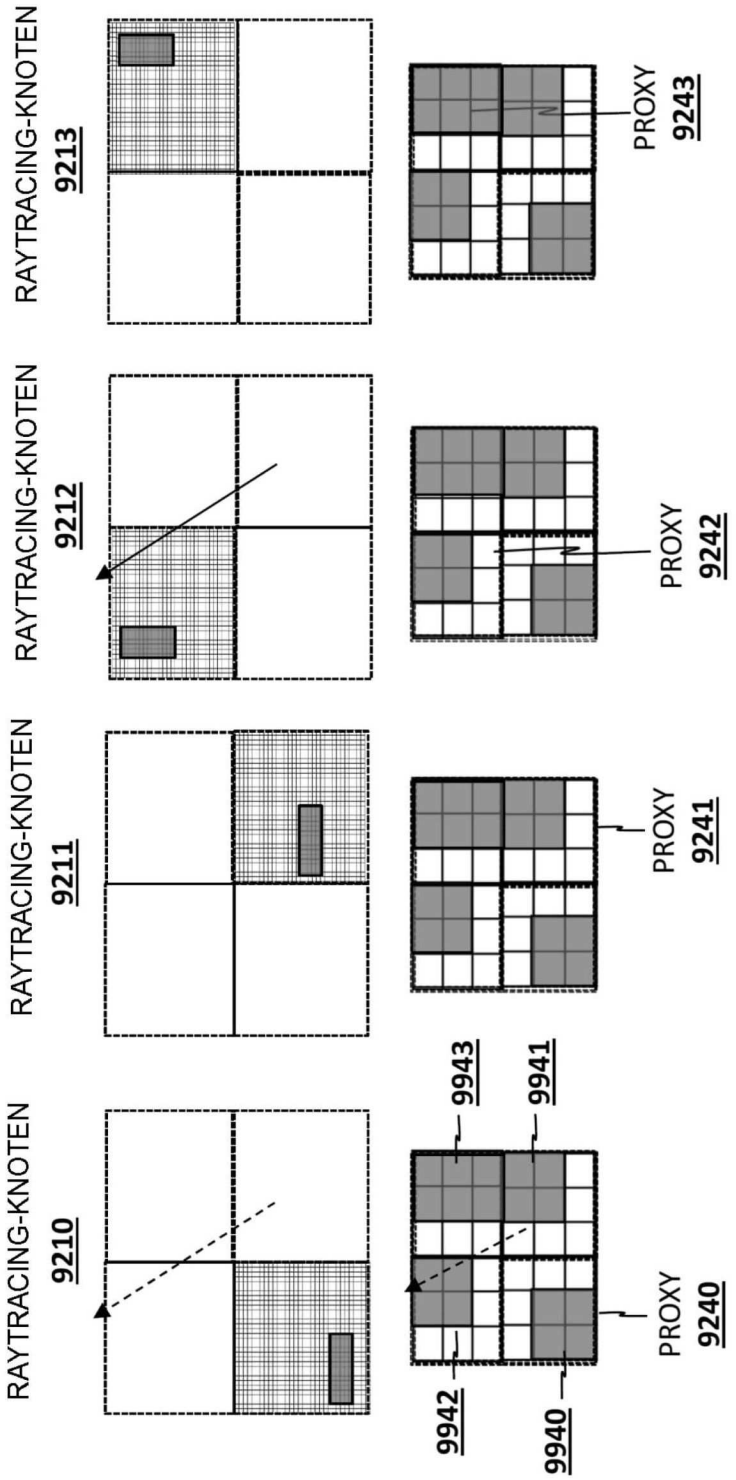

RAYTRACING-KNOTEN
9213
RAYTRACING-KNOTEN
9212
RAYTRACING-KNOTEN
9211
RAYTRACING-KNOTEN
9210
PROXY
9243
PROXY
9242
PROXY
9241
PROXY
9240
9943
9941
9942
9940

FIG. 100

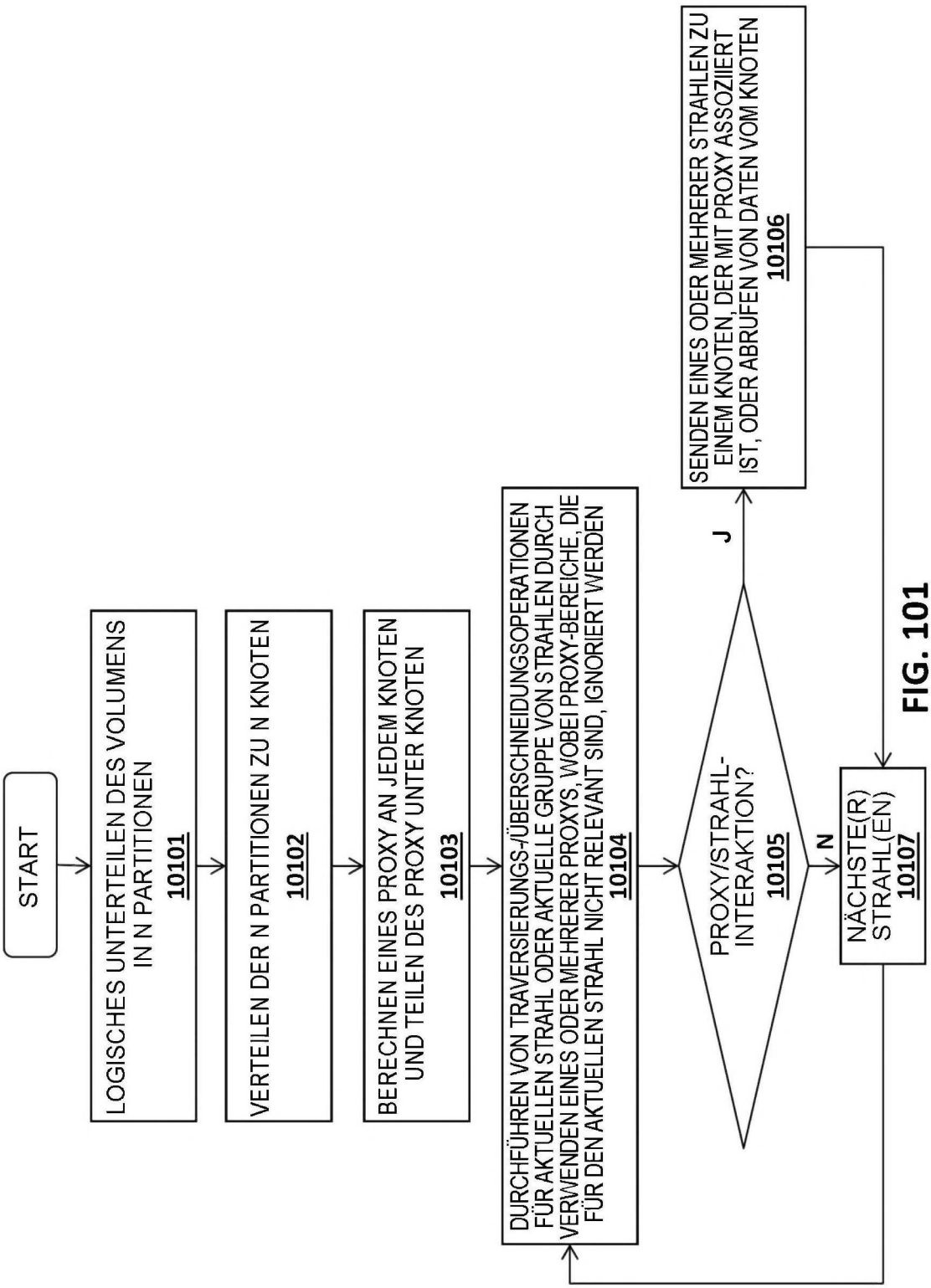

START
LOGISCHES UNTERTEILEN DES VOLUMENS IN N PARTITIONEN
10101
VERTEILEN DER N PARTITIONEN ZU N KNOTEN
10102
BERECHNEN EINES PROXY AN JEDEM KNOTEN UND TEILEN DES PROXY UNTER KNOTEN
10103
DURCHFÜHREN VON TRAVERSIERUNGS-/ÜBERSCHNEIDUNGSOPERATIONEN FÜR AKTUELLEN STRAHL ODER AKTUELLE GRUPPE VON STRAHLEN DURCH VERWENDEN EINES ODER MEHRERER PROXYS, WOBEI PROXY-BEREICHE, DIE FÜR DEN AKTUELLEN STRAHL NICHT RELEVANT SIND, IGNORIERT WERDEN
10104
PROXY/STRAHL-INTERAKTION?
10105
J
SENDEN EINES ODER MEHRERER STRAHLEN ZU EINEM KNOTEN, DER MIT PROXY ASSOZIIERT IST, ODER ABRUFEN VON DATEN VOM KNOTEN
10106
N
NÄCHSTE(R) STRAHL(EN)
10107

FIG. 101

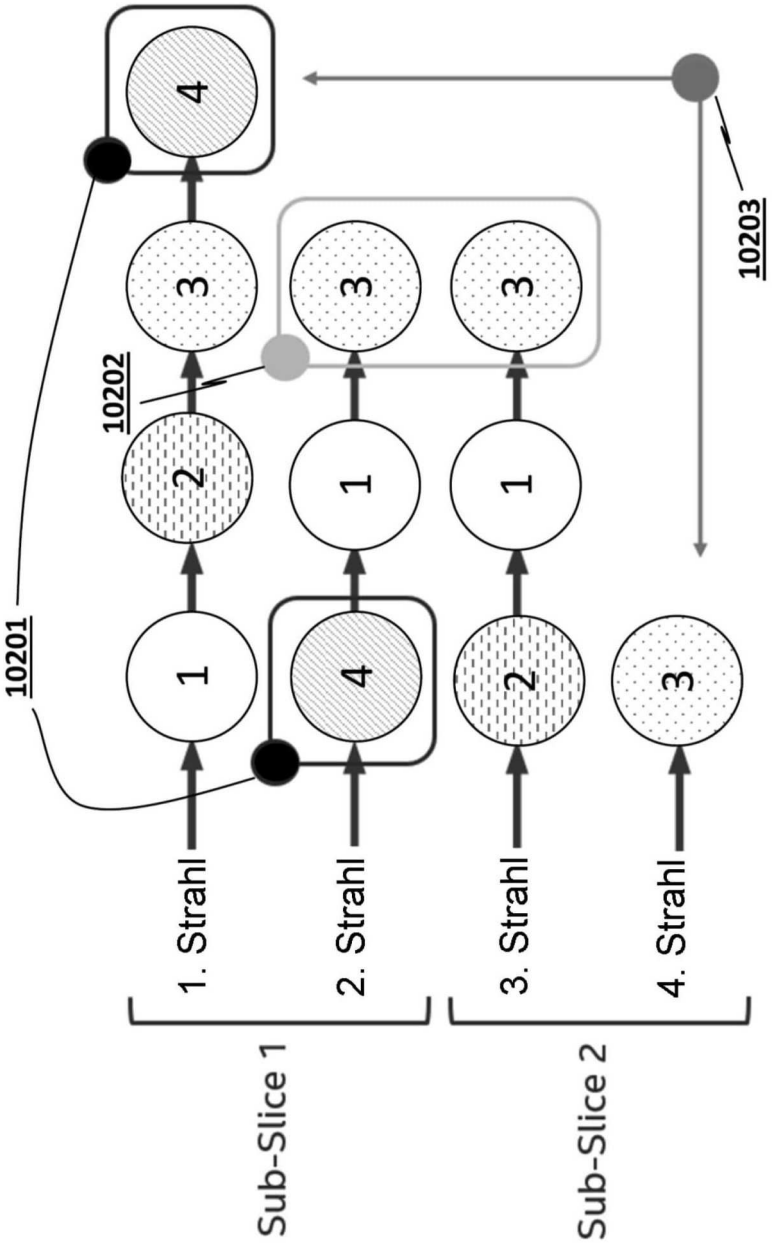

10201
10202
10203
1. Strahl
2. Strahl
3. Strahl
4. Strahl
Sub-Slice 1
Sub-Slice 2

FIG. 102

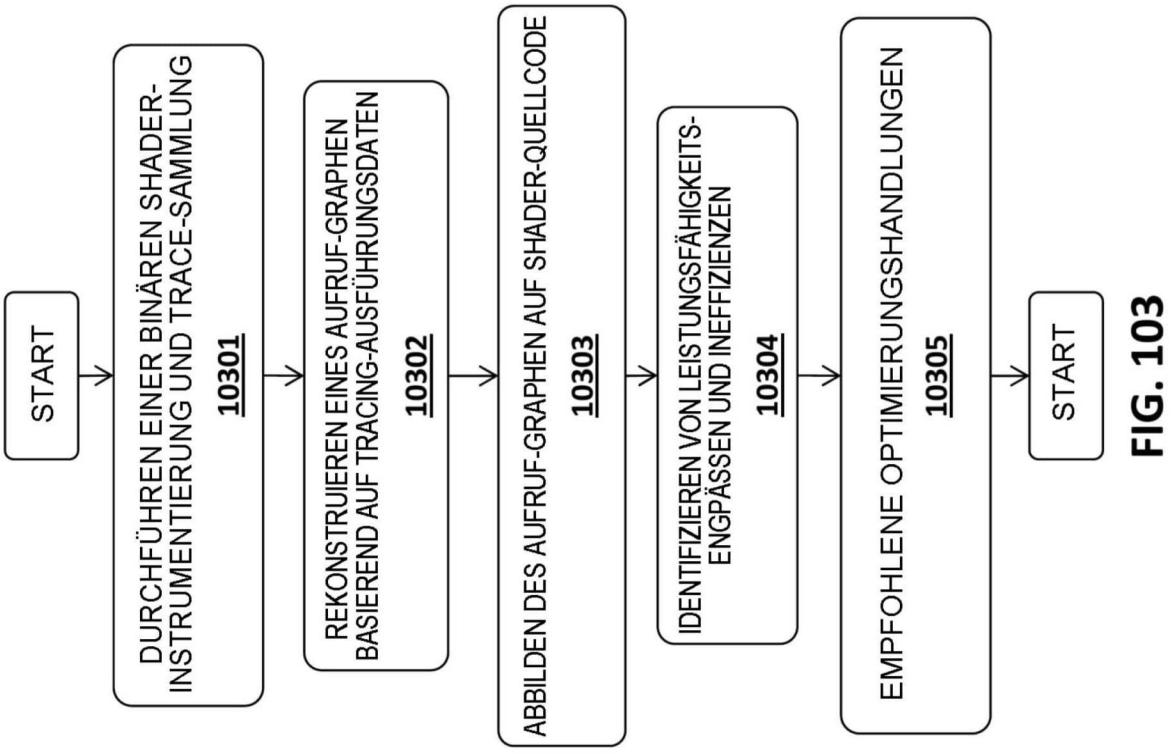

START
DURCHFÜHREN EINER BINÄREN SHADER-INSTRUMENTIERUNG UND TRACE-SAMMLUNG
10301
REKONSTRUIEREN EINES AUFRUF-GRAPHEN BASIEREND AUF TRACING-AUSFÜHRUNGSDATEN
10302
ABBILDEN DES AUFRUF-GRAPHEN AUF SHADER-QUELLCODE
10303
IDENTIFIZIEREN VON LEISTUNGSFÄHIGKEITS-ENGPÄSSEN UND INEFFIZIENZEN
10304
EMPFOHLENE OPTIMIERUNGSHANDLUNGEN
10305
START

FIG. 103

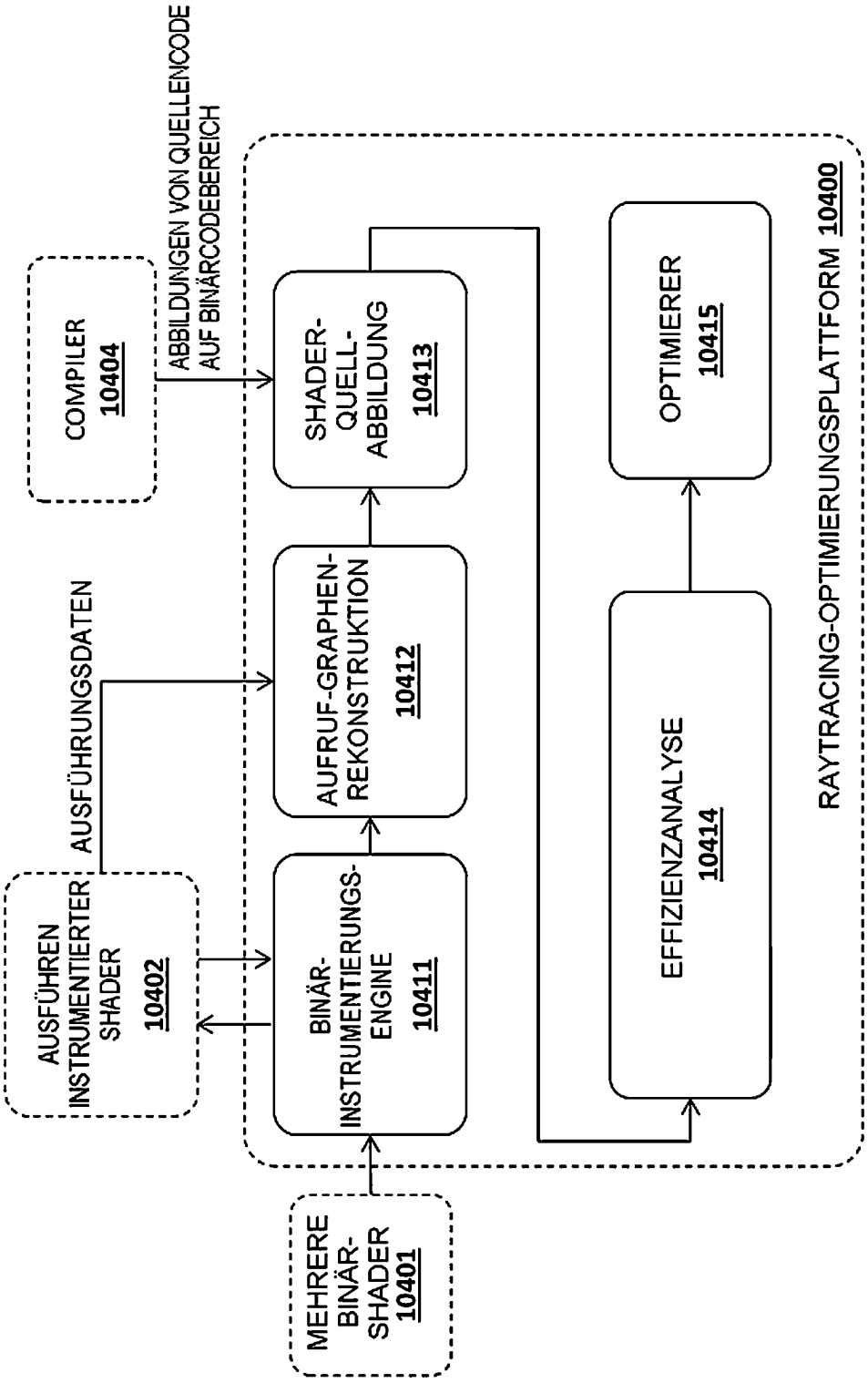

MEHRERE BINÄR-SHADER 10401
AUSFÜHREN INSTRUMENTIERTER SHADER 10402
AUSFÜHRUNGSDATEN
COMPILER 10404
ABBILDUNGEN VON QUELLENCODE AUF BINÄRCODEBEREICH
RAYTRACING-OPTIMIERUNGSPLATTFORM 10400
BINÄR-INSTRUMENTIERUNGS-ENGINE 10411
AUFRUF-GRAPHEN-REKONSTRUKTION 10412
SHADER-QUELL-ABBILDUNG 10413
EFFIZIENZANALYSE 10414
OPTIMIERER 10415

FIG. 104

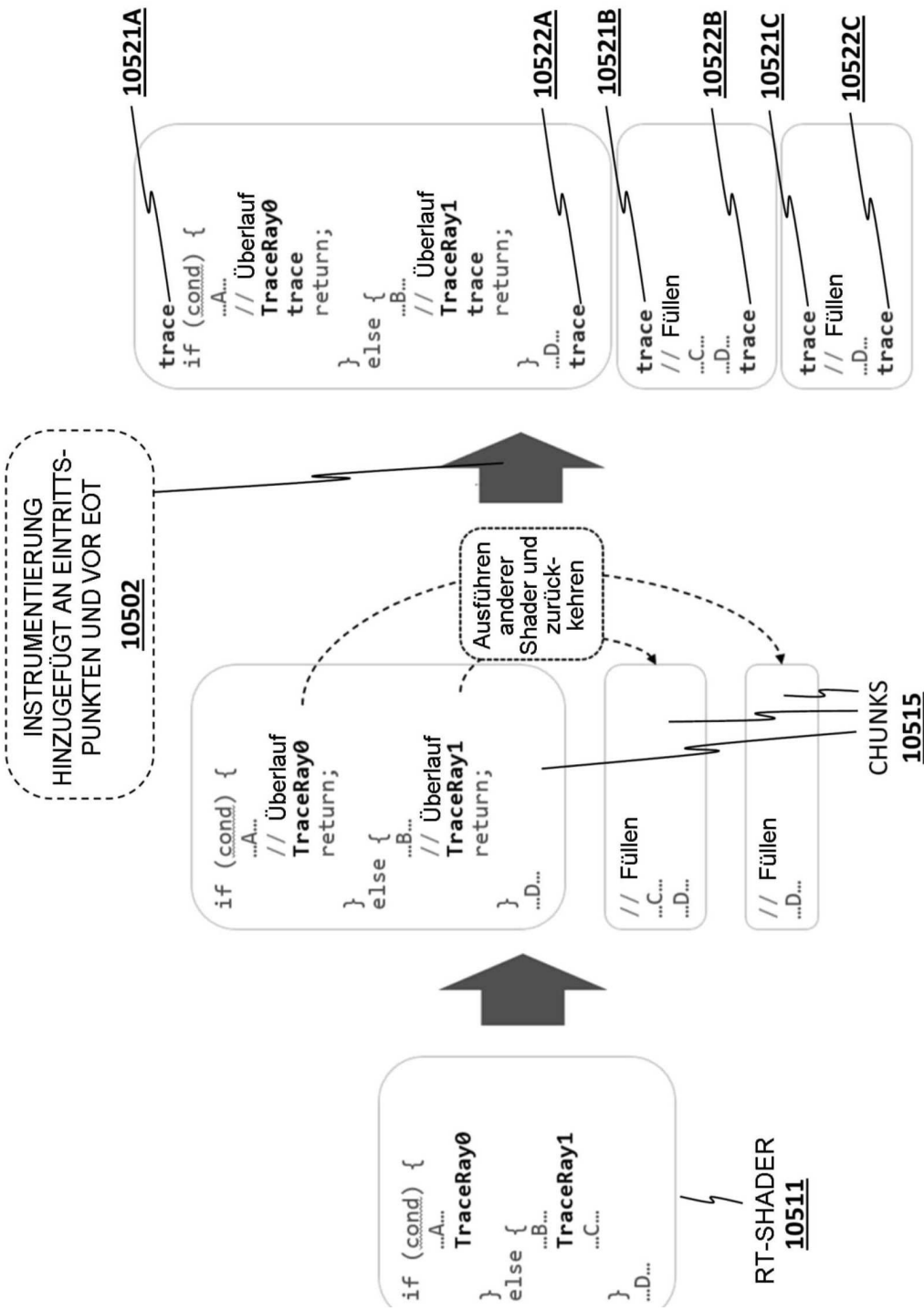

RT-SHADER
10511
if (cond) {
…A…
TraceRay0
}
else {
…B…
TraceRay1
…C…
}
…D…
INSTRUMENTIERUNG HINZUGEFÜGT AN EINTRITTS-PUNKTEN UND VOR EOT
10502
CHUNKS
10515
Ausführen anderer Shader und zurück-kehren
// Füllen
…C…
…D…
// Füllen
…D…
10521A
10522A
10521B
10522B
10521C
10522C
trace
if (cond) {
…A…
// Überlauf
TraceRay0
trace
return;
}
else {
…B…
// Überlauf
TraceRay1
trace
return;
}
…D…
trace
// Füllen
…C…
…D…
trace
trace
// Füllen
…D…
trace

FIG. 105

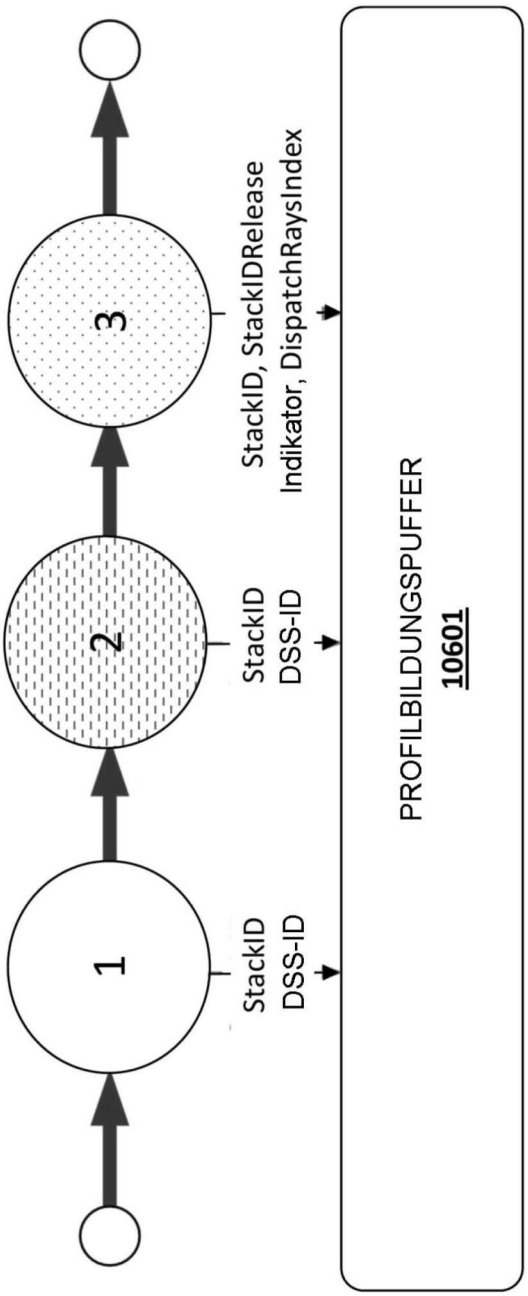

1
2
3
StackID
DSS-ID
StackID
DSS-ID
StackID, StackIDRelease
Indikator, DispatchRaysIndex
PROFILBILDUNGSPUFFER
10601

FIG. 106

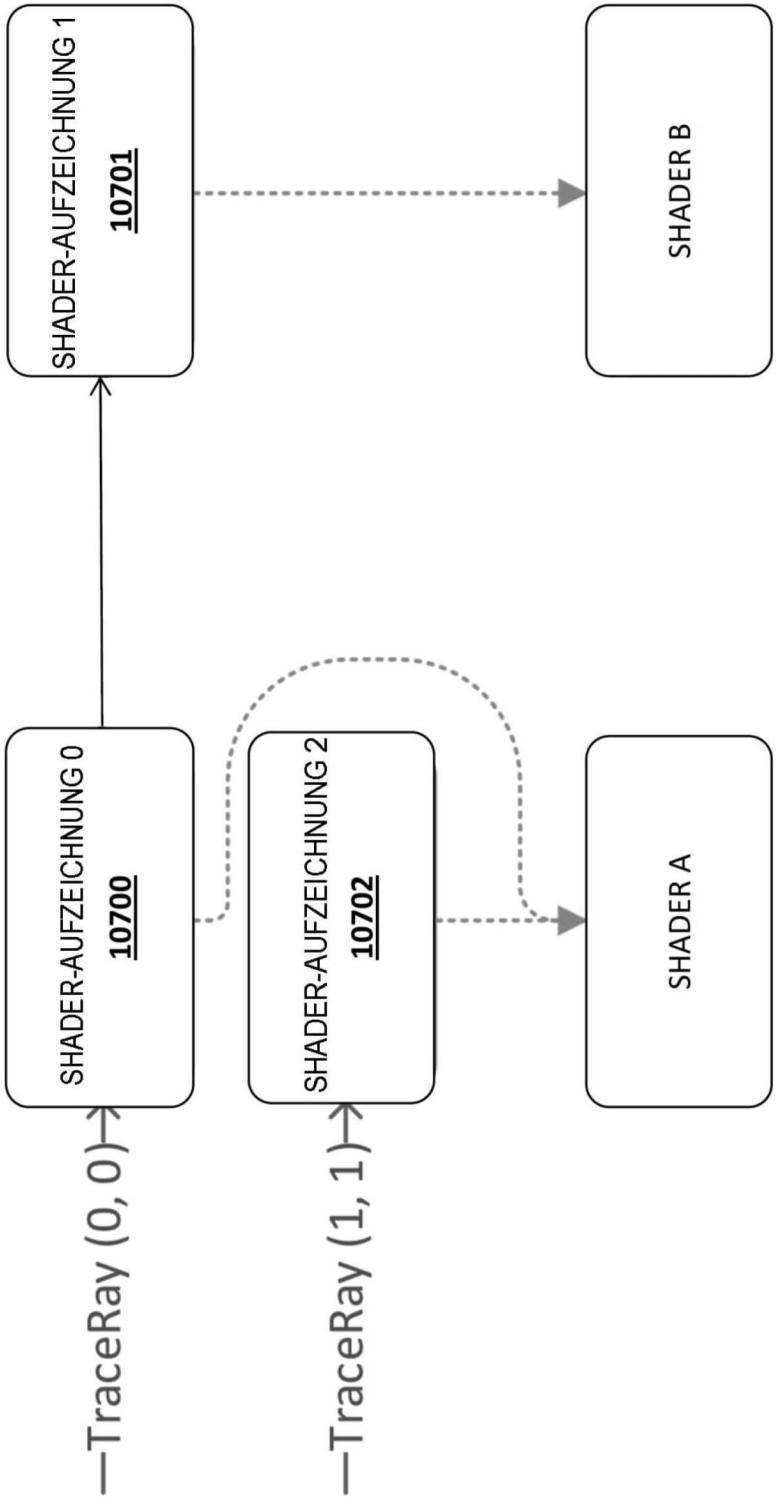


FIG. 107

