US011769290B2

(12) **United States Patent**
Benthin et al.

(10) **Patent No.: US 11,769,290 B2**
(45) **Date of Patent: *Sep. 26, 2023**

(54) **APPARATUS AND METHOD FOR IMPLEMENTING BOUNDING VOLUME HIERARCHY (BVH) OPERATIONS ON TESSELATION HARDWARE**

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventors: **Carsten Benthin**, Voelklingen (DE); **Gabor Liktor**, San Francisco, CA (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **17/557,968**

(22) Filed: **Dec. 21, 2021**

(65) **Prior Publication Data**

US 2022/0222886 A1 Jul. 14, 2022

Related U.S. Application Data

(63) Continuation of application No. 16/803,614, filed on Feb. 27, 2020, now Pat. No. 11,210,841, which is a continuation of application No. 15/482,701, filed on Apr. 7, 2017, now Pat. No. 10,614,611.

(51) **Int. Cl.**
G06T 15/06 (2011.01)
G06T 15/00 (2011.01)

(52) **U.S. Cl.**
CPC ***G06T 15/06*** (2013.01); ***G06T 15/005*** (2013.01)

(58) **Field of Classification Search**
None
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

2010/0097372 A1* 4/2010 Gies G06T 15/06 345/419
2011/0018874 A1 1/2011 Hasselgren et al.
2013/0155080 A1* 6/2013 Nordlund G06T 15/005 345/522

OTHER PUBLICATIONS

Final Office Action, U.S. Appl. No. 15/482,701, dated Jan. 10, 2019, 21 pages.
Final Office Action, U.S. Appl. No. 16/803,614, dated Apr. 29, 2021, 17 pages.
Non-Final Office Action, U.S. Appl. No. 15/482,701, dated Apr. 19, 2018, 24 pages.
Non-Final Office Action, U.S. Appl. No. 15/482,701, dated May 22, 2019, 22 pages.

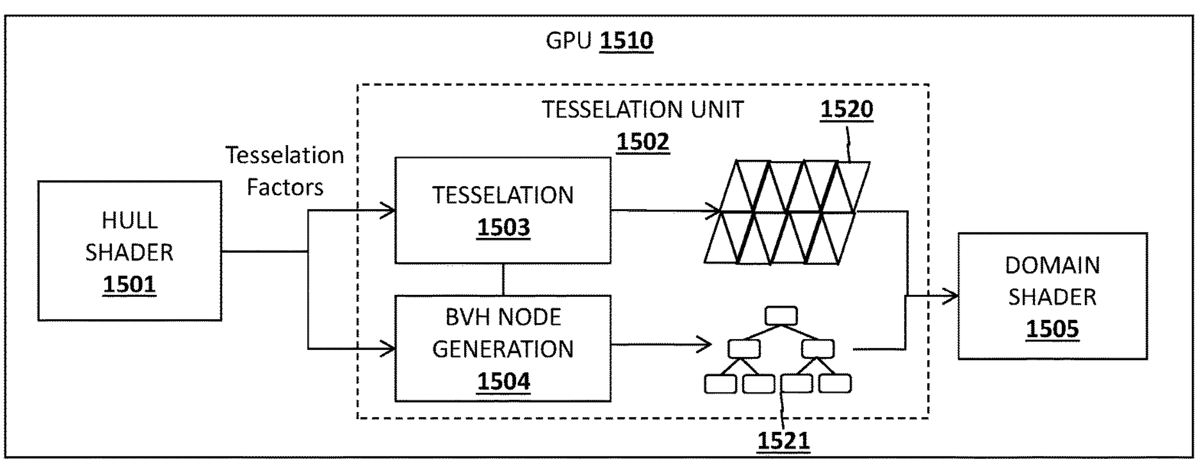
(Continued)

Primary Examiner — Tapas Mazumder
(74) *Attorney, Agent, or Firm* — NICHOLSON DE VOS WEBSTER & ELLIOTT LLP

(57) **ABSTRACT**

An apparatus and method are described for using tessellation hardware to generate bounding volume hierarchies (BVHs) and perform other ray tracing operations. For example, one embodiment of an apparatus comprises: a shader to output a plurality of tessellation factors and one or more input surfaces; and a tessellation circuit comprising first circuitry and/or logic to tessellate each input surface to generate a new set of primitives and second circuitry and/or logic to concurrently generate a bounding volume hierarchy (BVH) **1521** based on the new set of primitives.

18 Claims, 32 Drawing Sheets



(56) **References Cited**

OTHER PUBLICATIONS

Non-Final Rejection dated Jan. 21, 2021 for U.S. Appl. No. 16/803,614.
Notice of Allowance, U.S. Appl. No. 15/482,701, dated Dec. 10, 2019, 10 pages.
Notice of Allowance, U.S. Appl. No. 16/803,614, dated Aug. 18, 2021, 9 pages.

* cited by examiner

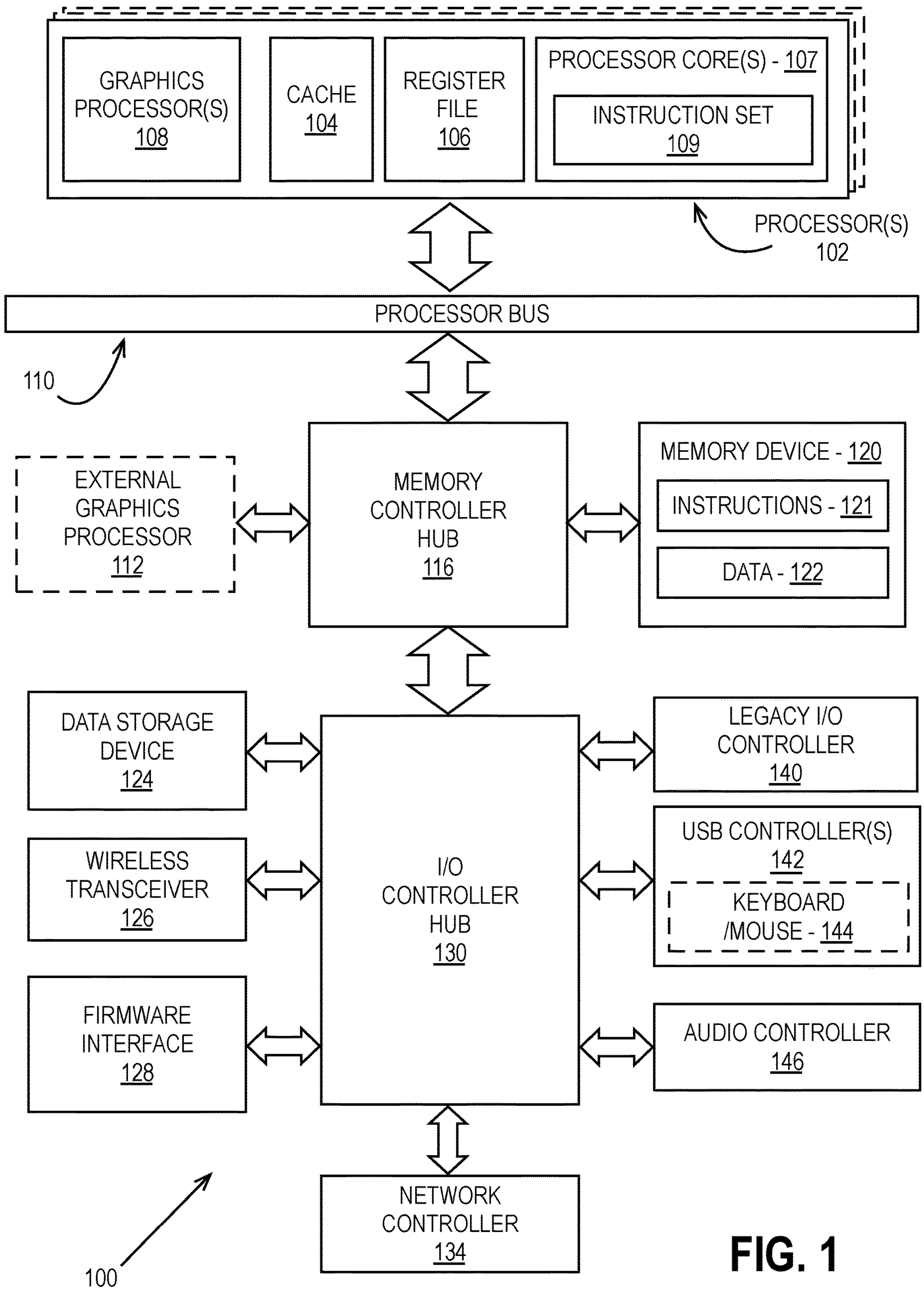

GRAPHICS PROCESSOR(S) 108
CACHE 104
REGISTER FILE 106
PROCESSOR CORE(S) - 107
INSTRUCTION SET 109
PROCESSOR(S) 102
PROCESSOR BUS
110
EXTERNAL GRAPHICS PROCESSOR 112
MEMORY CONTROLLER HUB 116
MEMORY DEVICE - 120
INSTRUCTIONS - 121
DATA - 122
DATA STORAGE DEVICE 124
WIRELESS TRANSCEIVER 126
FIRMWARE INTERFACE 128
I/O CONTROLLER HUB 130
LEGACY I/O CONTROLLER 140
USB CONTROLLER(S) 142
KEYBOARD /MOUSE - 144
AUDIO CONTROLLER 146
NETWORK CONTROLLER 134
100

FIG. 1

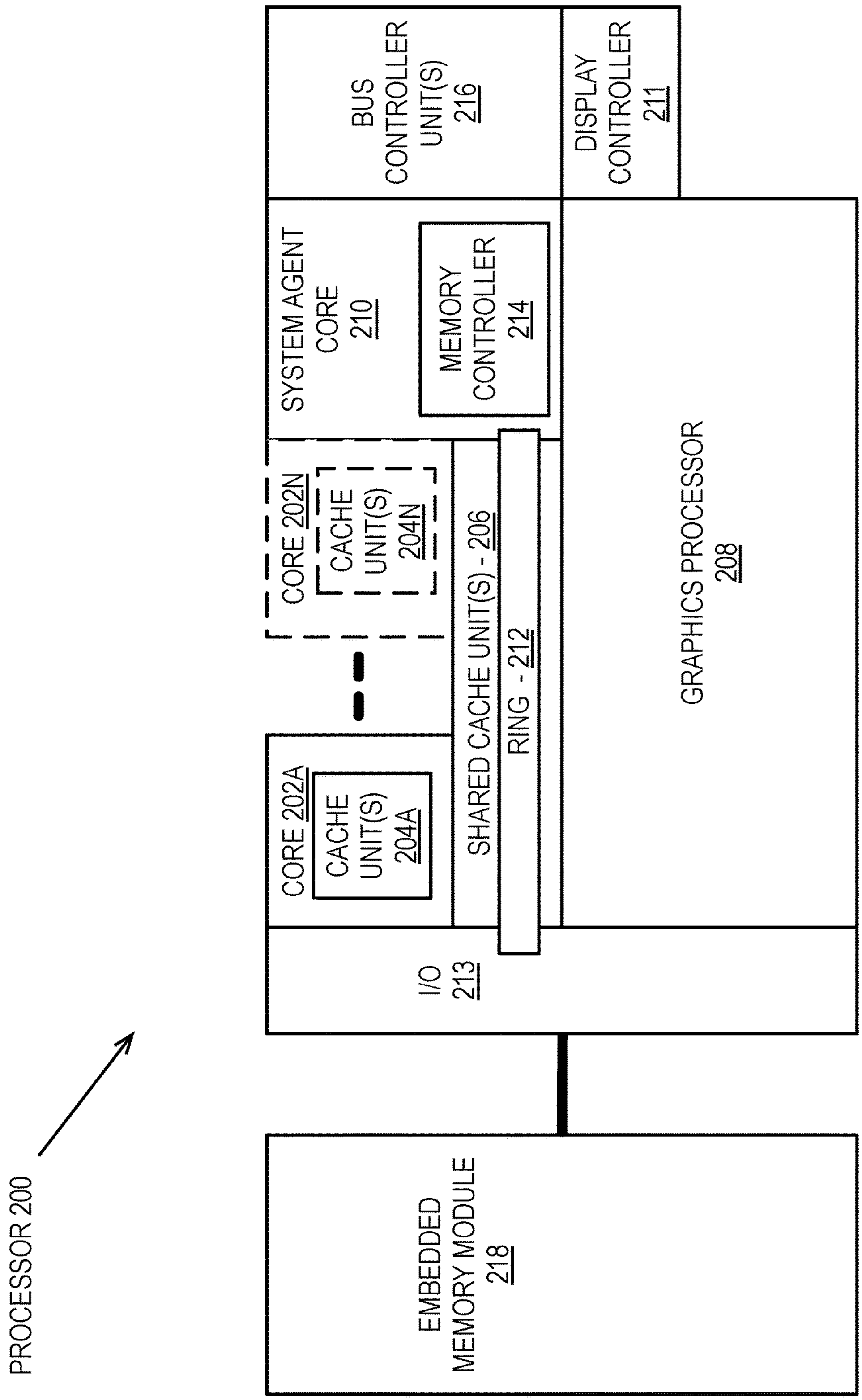

PROCESSOR 200
BUS CONTROLLER UNIT(S) 216
DISPLAY CONTROLLER 211
SYSTEM AGENT CORE 210
MEMORY CONTROLLER 214
CORE 202N
CACHE UNIT(S) 204N
SHARED CACHE UNIT(S) - 206
RING - 212
GRAPHICS PROCESSOR 208
CORE 202A
CACHE UNIT(S) 204A
I/O 213
EMBEDDED MEMORY MODULE 218

FIG. 2

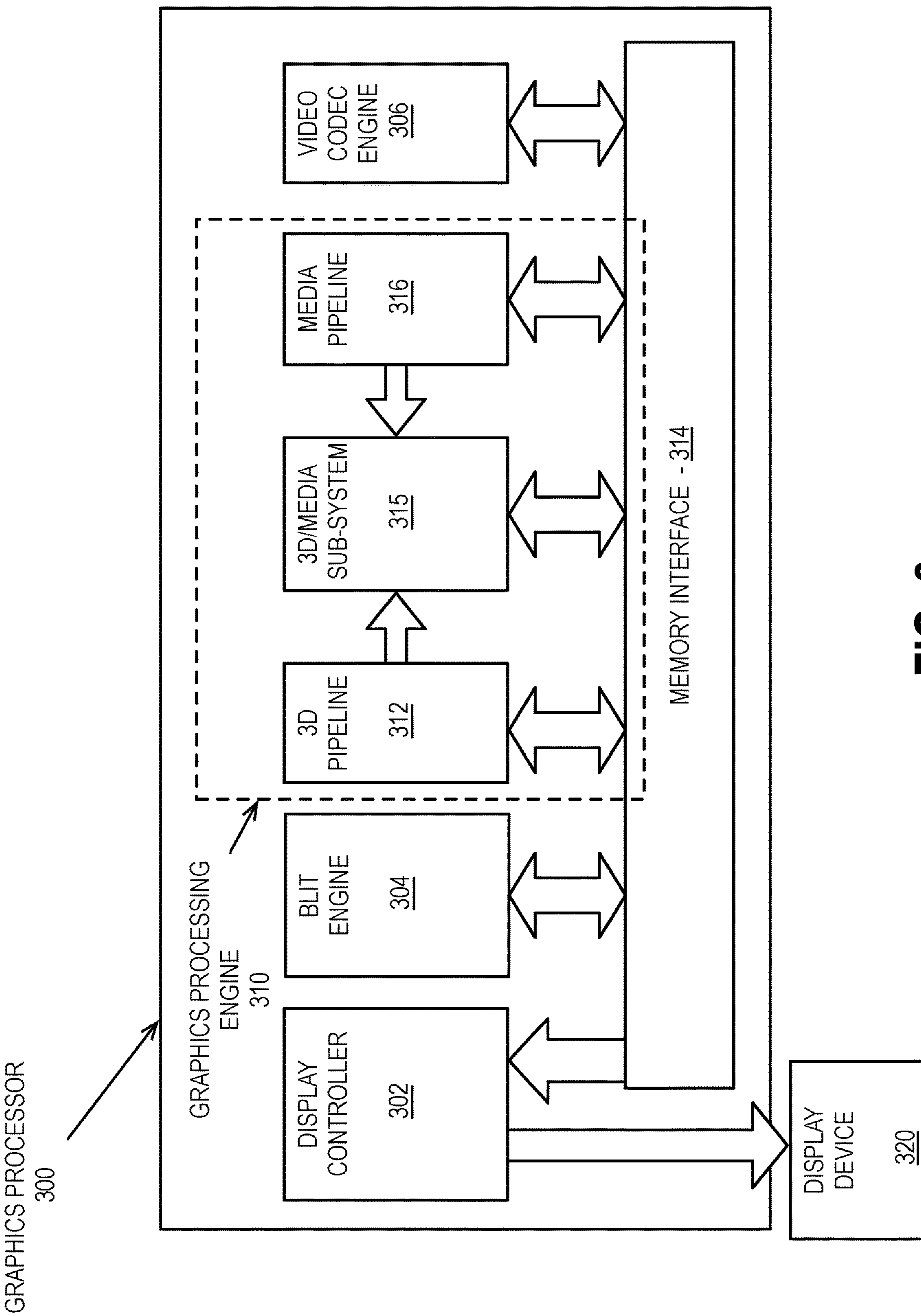

GRAPHICS PROCESSOR
300
GRAPHICS PROCESSING ENGINE
310
DISPLAY CONTROLLER
302
BLIT ENGINE
304
3D PIPELINE
312
3D/MEDIA SUB-SYSTEM
315
MEDIA PIPELINE
316
VIDEO CODEC ENGINE
306
MEMORY INTERFACE - 314
DISPLAY DEVICE
320

FIG. 3

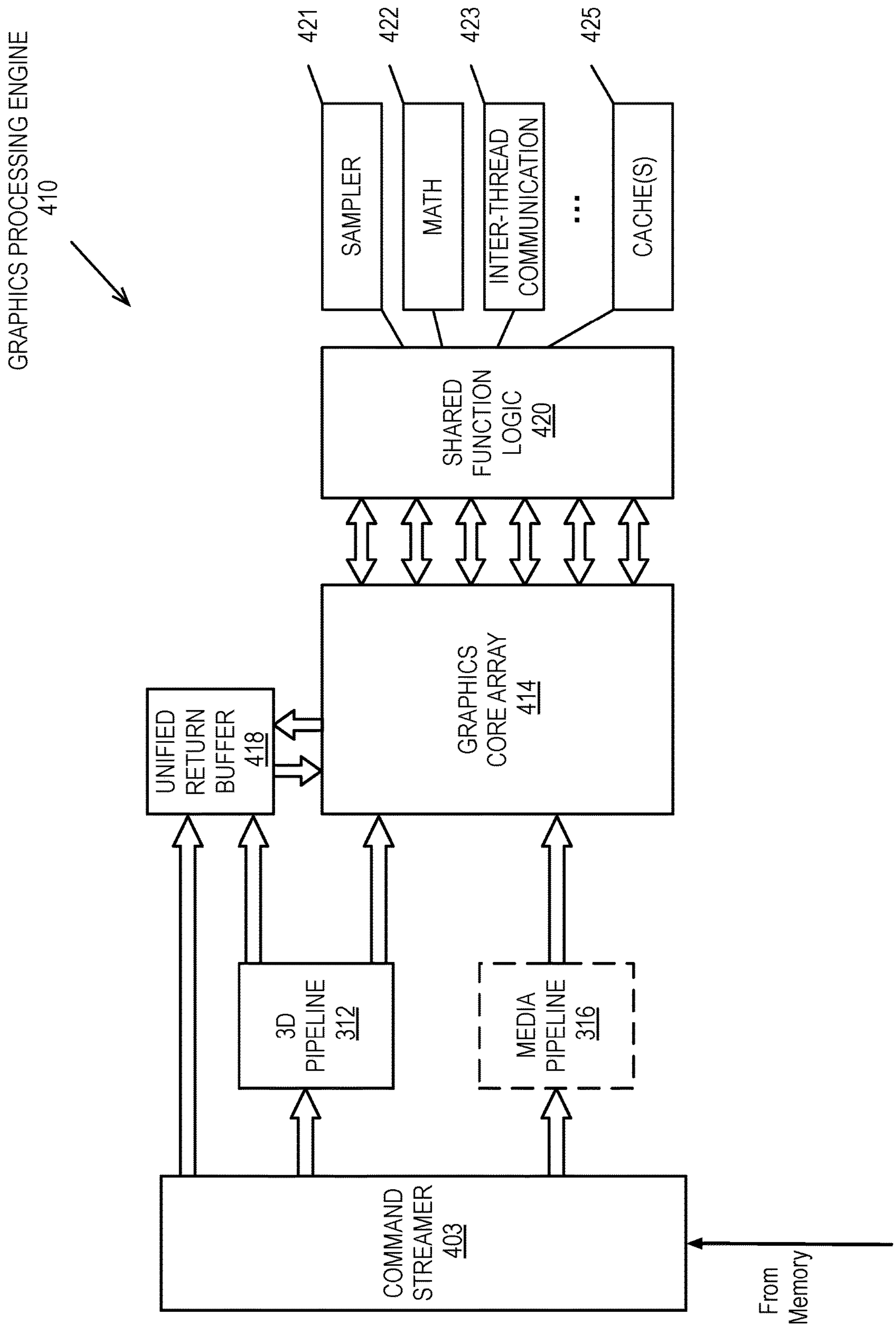

GRAPHICS PROCESSING ENGINE
410
COMMAND
STREAMER
403
From
Memory
3D
PIPELINE
312
MEDIA
PIPELINE
316
UNIFIED
RETURN
BUFFER
418
GRAPHICS
CORE ARRAY
414
SHARED
FUNCTION
LOGIC
420
SAMPLER
MATH
INTER-THREAD
COMMUNICATION
...
CACHE(S)
421
422
423
425

FIG. 4

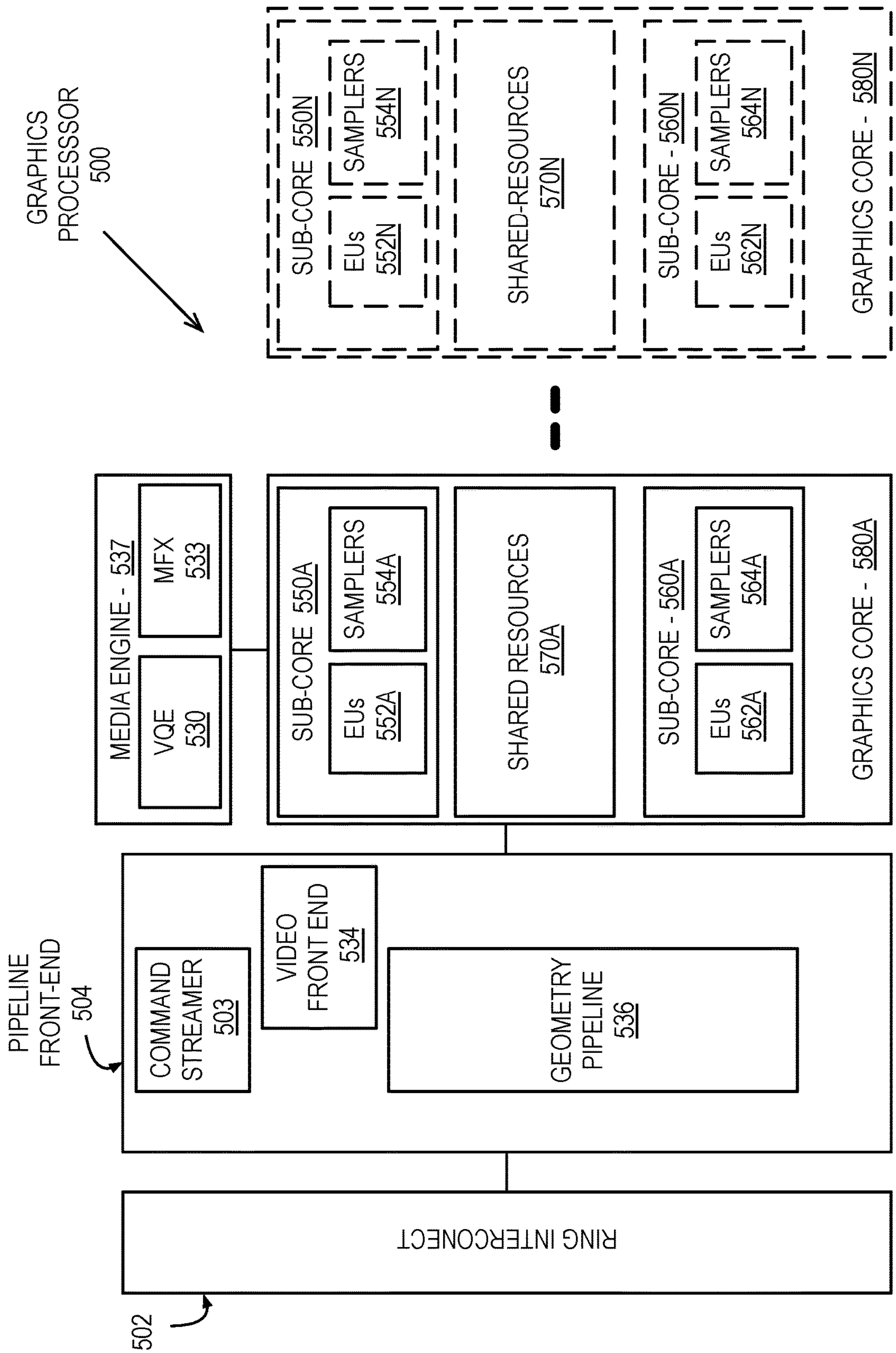
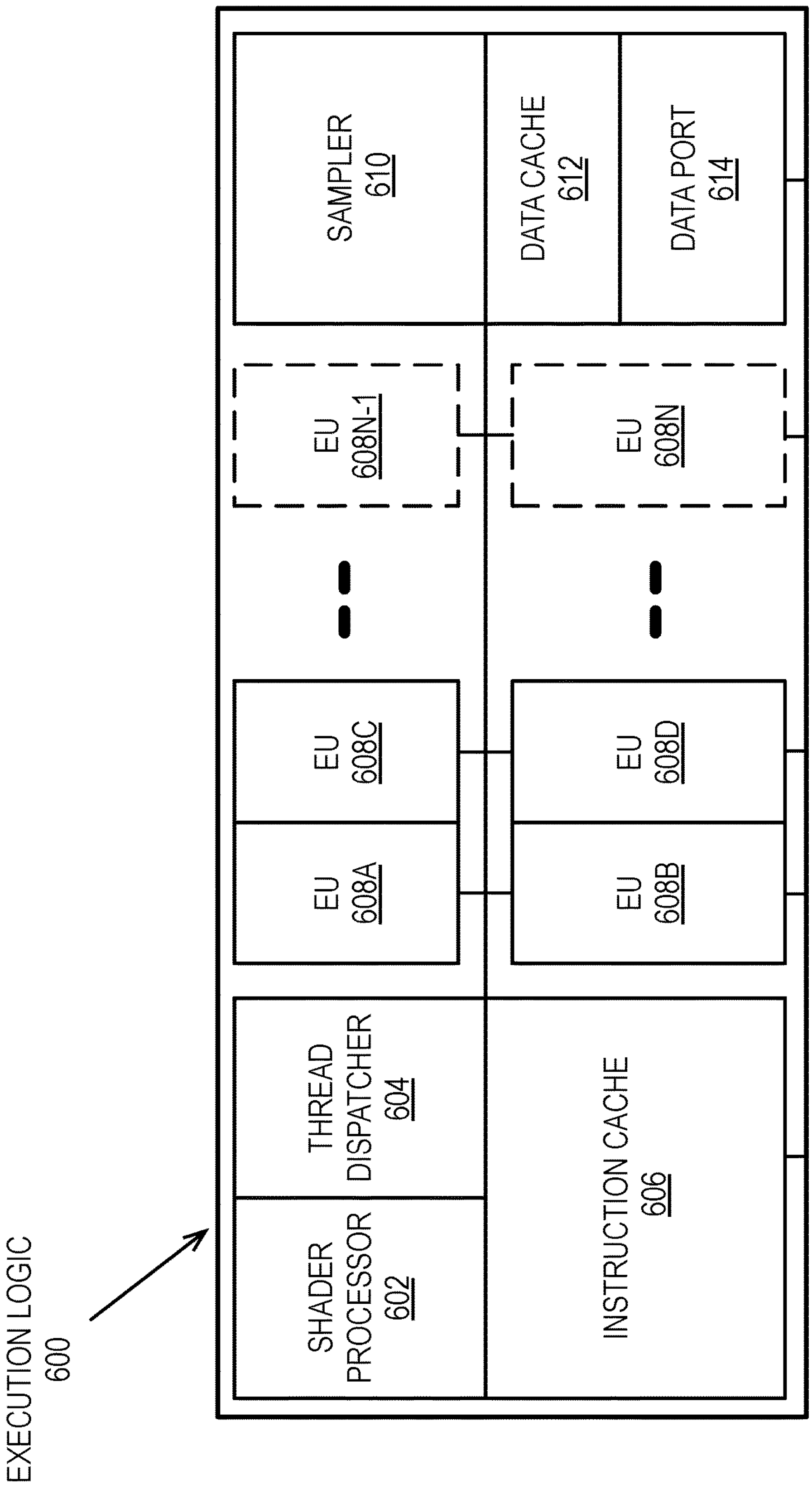

GRAPHICS PROCESSSOR 500
SUB-CORE 550N
EUs 552N
SAMPLERS 554N
SHARED-RESOURCES 570N
SUB-CORE - 560N
EUs 562N
SAMPLERS 564N
GRAPHICS CORE - 580N
MEDIA ENGINE - 537
VQE 530
MFX 533
SUB-CORE 550A
EUs 552A
SAMPLERS 554A
SHARED RESOURCES 570A
SUB-CORE - 560A
EUs 562A
SAMPLERS 564A
GRAPHICS CORE - 580A
PIPELINE FRONT-END 504
COMMAND STREAMER 503
VIDEO FRONT END 534
GEOMETRY PIPELINE 536
RING INTERCONECT
502

FIG. 5



EXECUTION LOGIC 600
SHADER PROCESSOR 602
THREAD DISPATCHER 604
INSTRUCTION CACHE 606
EU 608A
EU 608B
EU 608C
EU 608D
EU 608N-1
EU 608N
SAMPLER 610
DATA CACHE 612
DATA PORT 614

FIG. 6

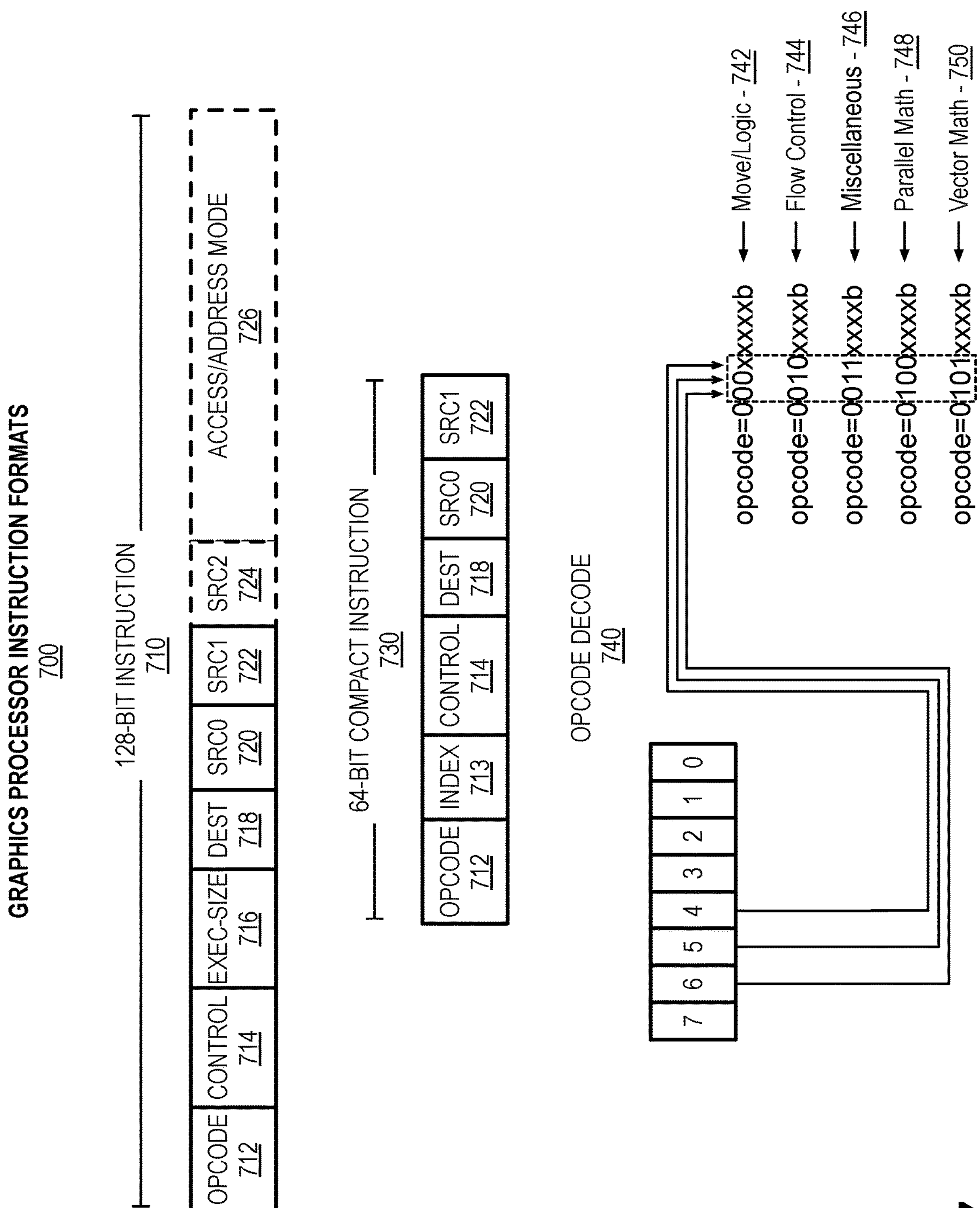

GRAPHICS PROCESSOR INSTRUCTION FORMATS
700
128-BIT INSTRUCTION
710
OPCODE
712
CONTROL
714
EXEC-SIZE
716
DEST
718
SRC0
720
SRC1
722
SRC2
724
ACCESS/ADDRESS MODE
726
64-BIT COMPACT INSTRUCTION
730
OPCODE
712
INDEX
713
CONTROL
714
DEST
718
SRC0
720
SRC1
722
OPCODE DECODE
740
7
6
5
4
3
2
1
0
opcode=0000xxxxb
opcode=0010xxxxb
opcode=0011xxxxb
opcode=0100xxxxb
opcode=0101xxxxb
Move/Logic - 742
Flow Control - 744
Miscellaneous - 746
Parallel Math - 748
Vector Math - 750

FIG. 7

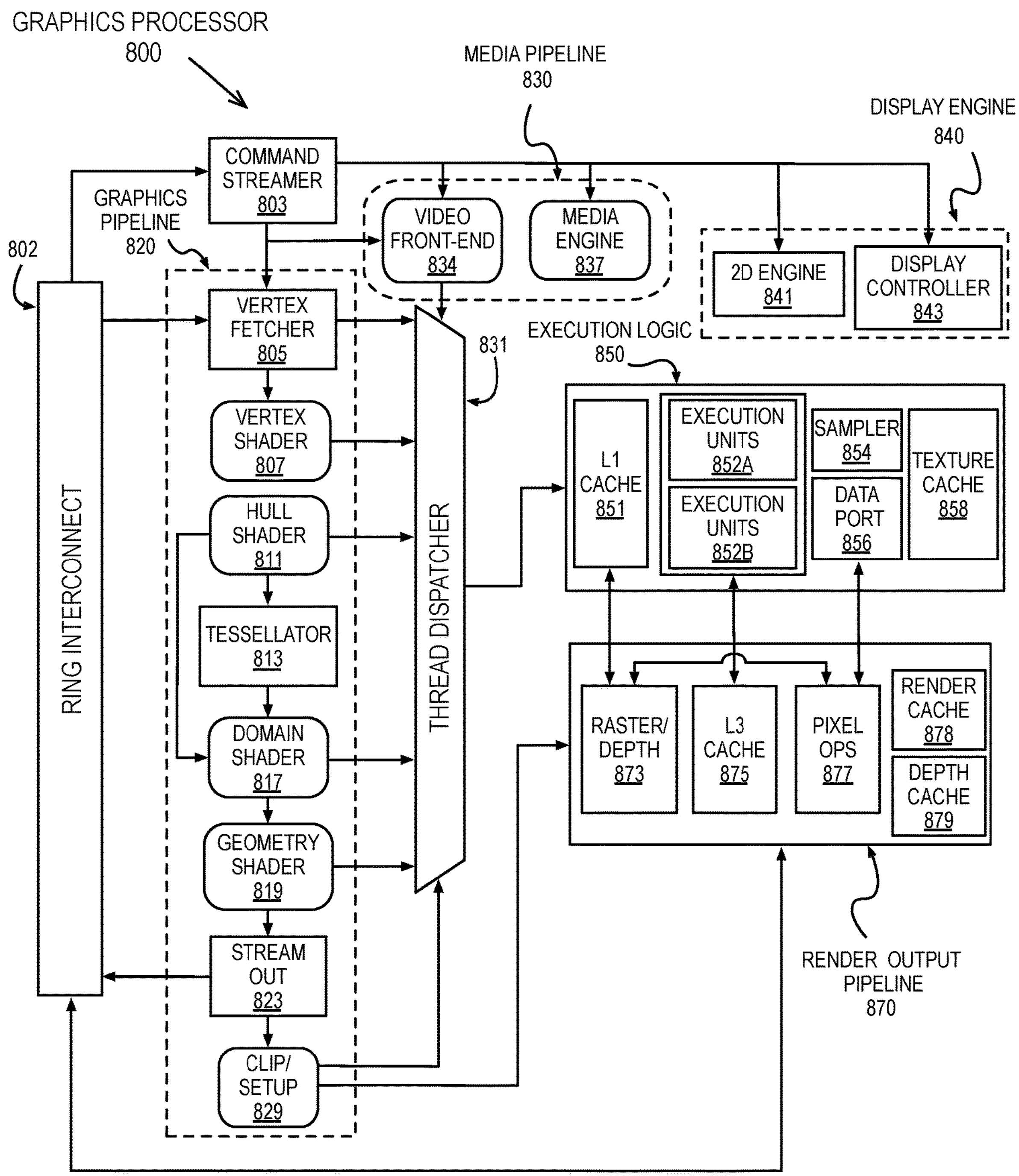

GRAPHICS PROCESSOR
800
MEDIA PIPELINE
830
DISPLAY ENGINE
840
COMMAND
STREAMER
803
GRAPHICS
PIPELINE
820
802
VIDEO
FRONT-END
834
MEDIA
ENGINE
837
2D ENGINE
841
DISPLAY
CONTROLLER
843
VERTEX
FETCHER
805
831
EXECUTION LOGIC
850
VERTEX
SHADER
807
L1
CACHE
851
EXECUTION
UNITS
852A
SAMPLER
854
TEXTURE
CACHE
858
HULL
SHADER
811
EXECUTION
UNITS
852B
DATA
PORT
856
RING INTERCONNECT
THREAD DISPATCHER
TESSELLATOR
813
DOMAIN
SHADER
817
RASTER/
DEPTH
873
L3
CACHE
875
PIXEL
OPS
877
RENDER
CACHE
878
DEPTH
CACHE
879
GEOMETRY
SHADER
819
STREAM
OUT
823
RENDER OUTPUT
PIPELINE
870
CLIP/
SETUP
829

FIG. 8

FIG. 9A

GRAPHICS PROCESSOR COMMAND FORMAT
900

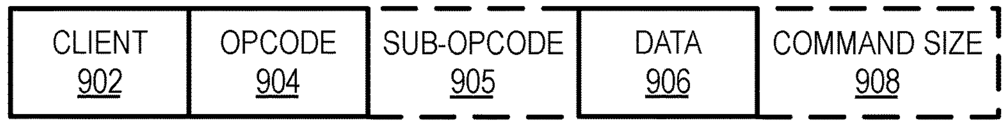
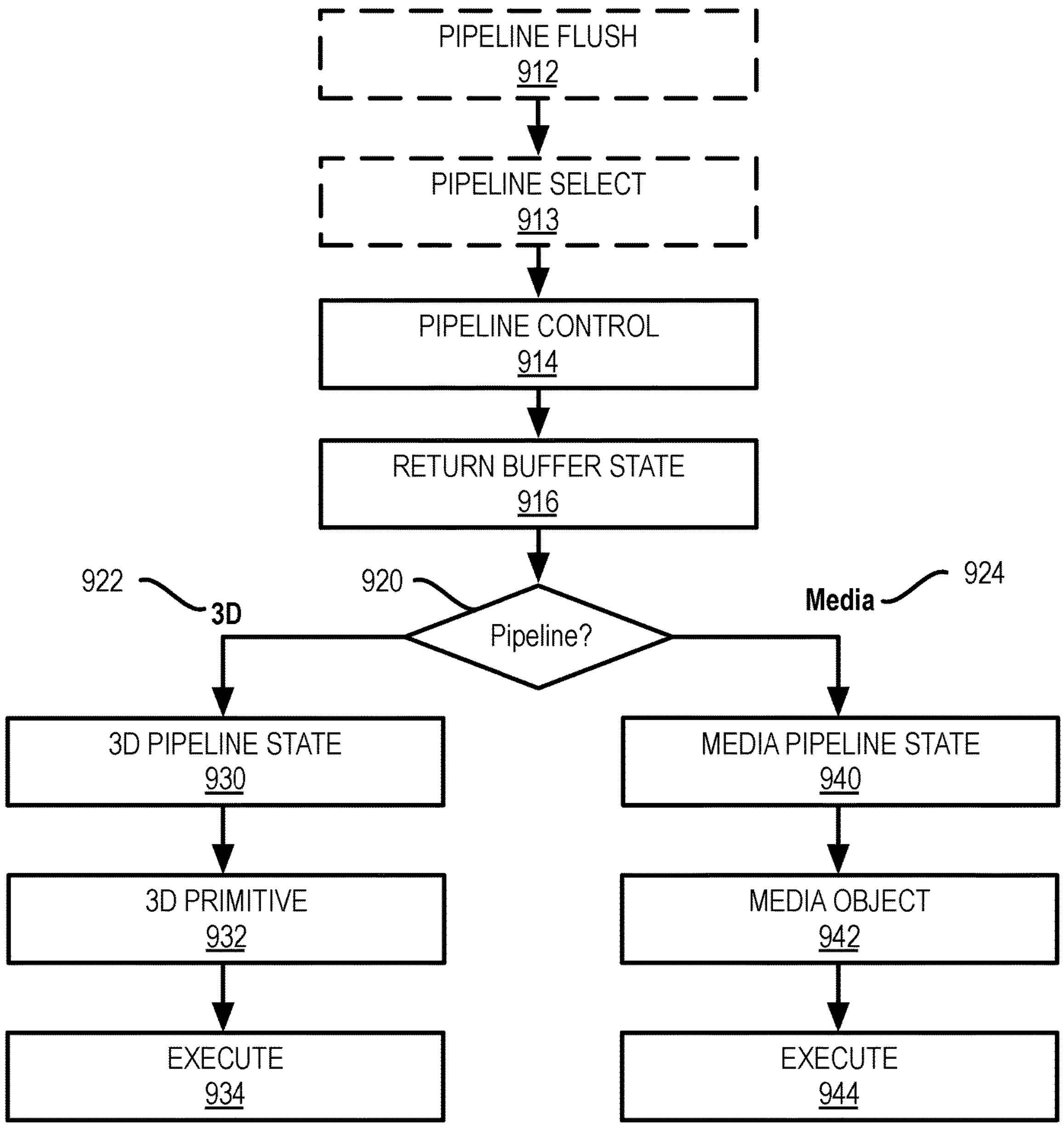


FIG. 9B

GRAPHICS PROCESSOR COMMAND SEQUENCE
910



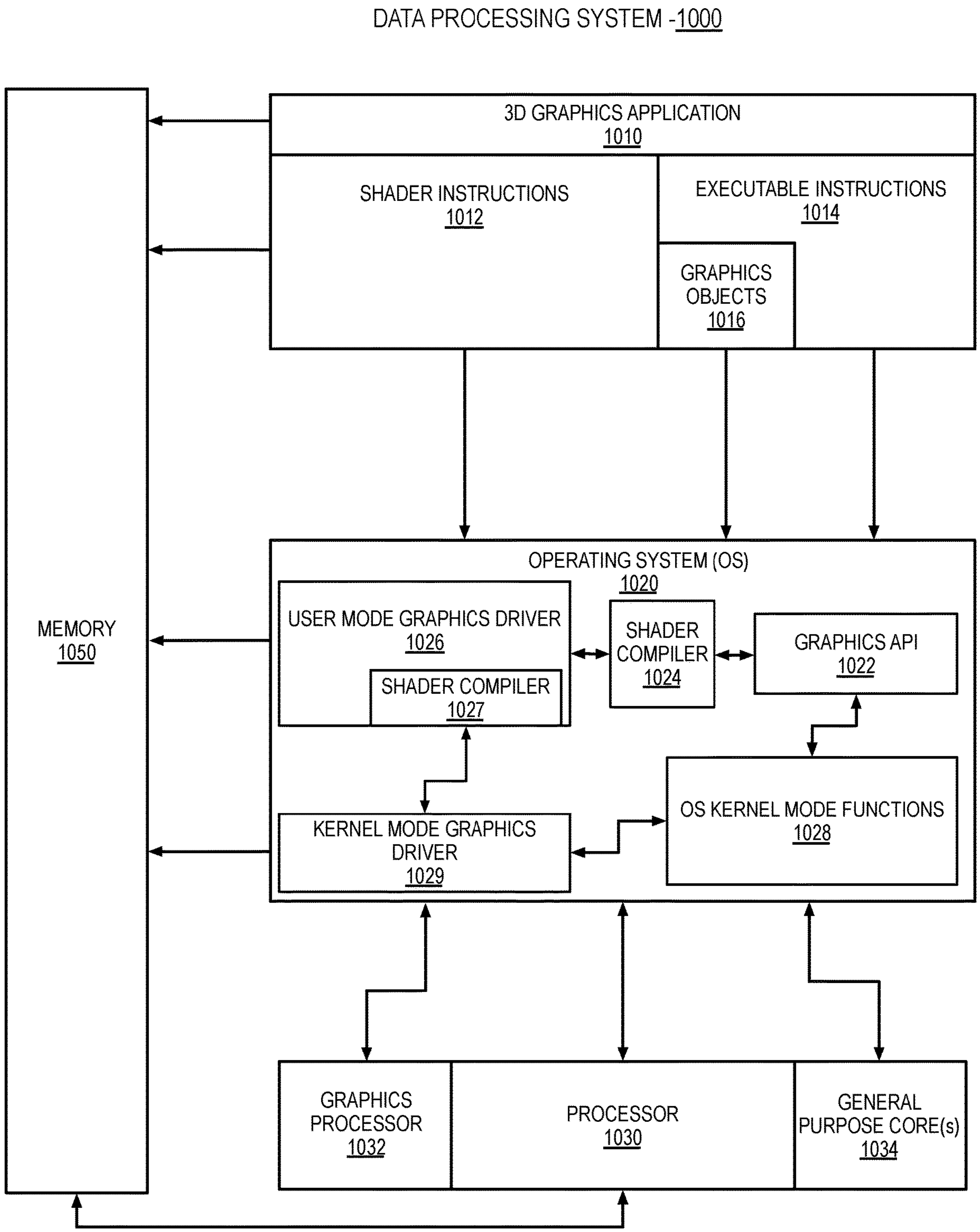

DATA PROCESSING SYSTEM -1000
3D GRAPHICS APPLICATION
1010
SHADER INSTRUCTIONS
1012
EXECUTABLE INSTRUCTIONS
1014
GRAPHICS
OBJECTS
1016
MEMORY
1050
OPERATING SYSTEM (OS)
1020
USER MODE GRAPHICS DRIVER
1026
SHADER COMPILER
1027
SHADER
COMPILER
1024
GRAPHICS API
1022
KERNEL MODE GRAPHICS
DRIVER
1029
OS KERNEL MODE FUNCTIONS
1028
GRAPHICS
PROCESSOR
1032
PROCESSOR
1030
GENERAL
PURPOSE CORE(s)
1034

FIG. 10

IP CORE DEVELOPMENT - 1100

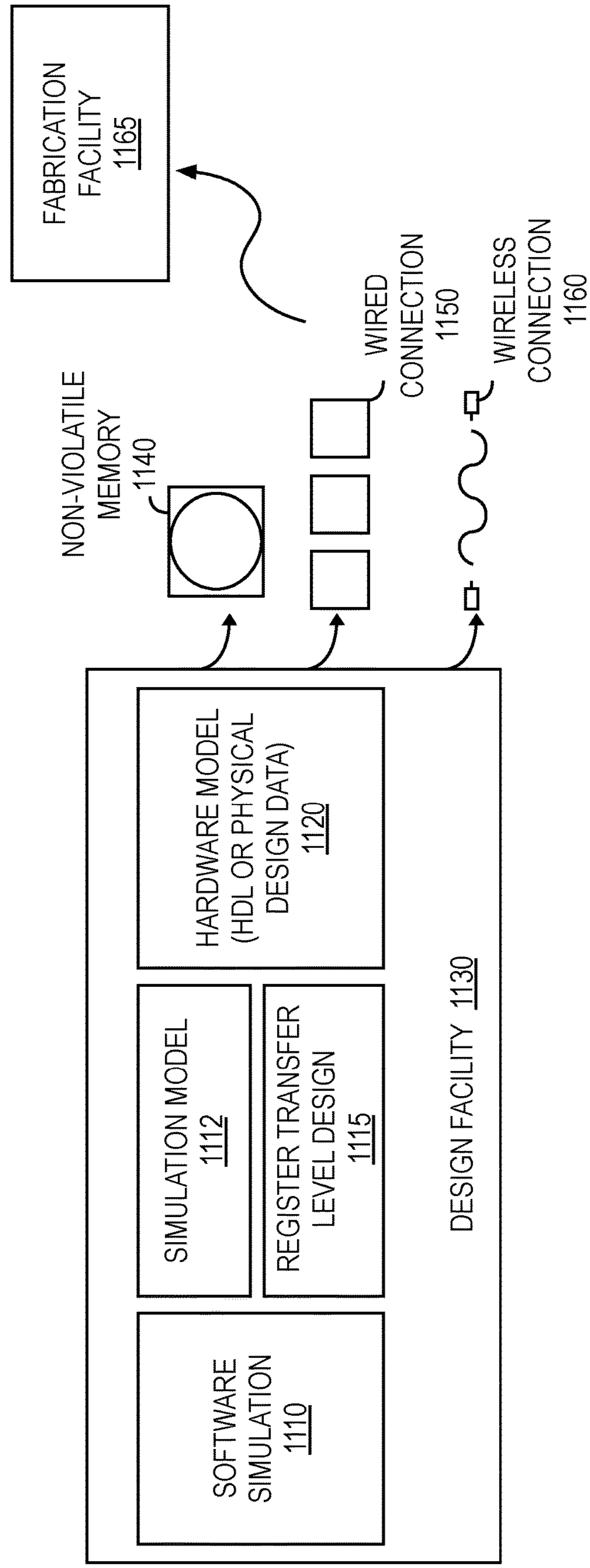


FIG. 11

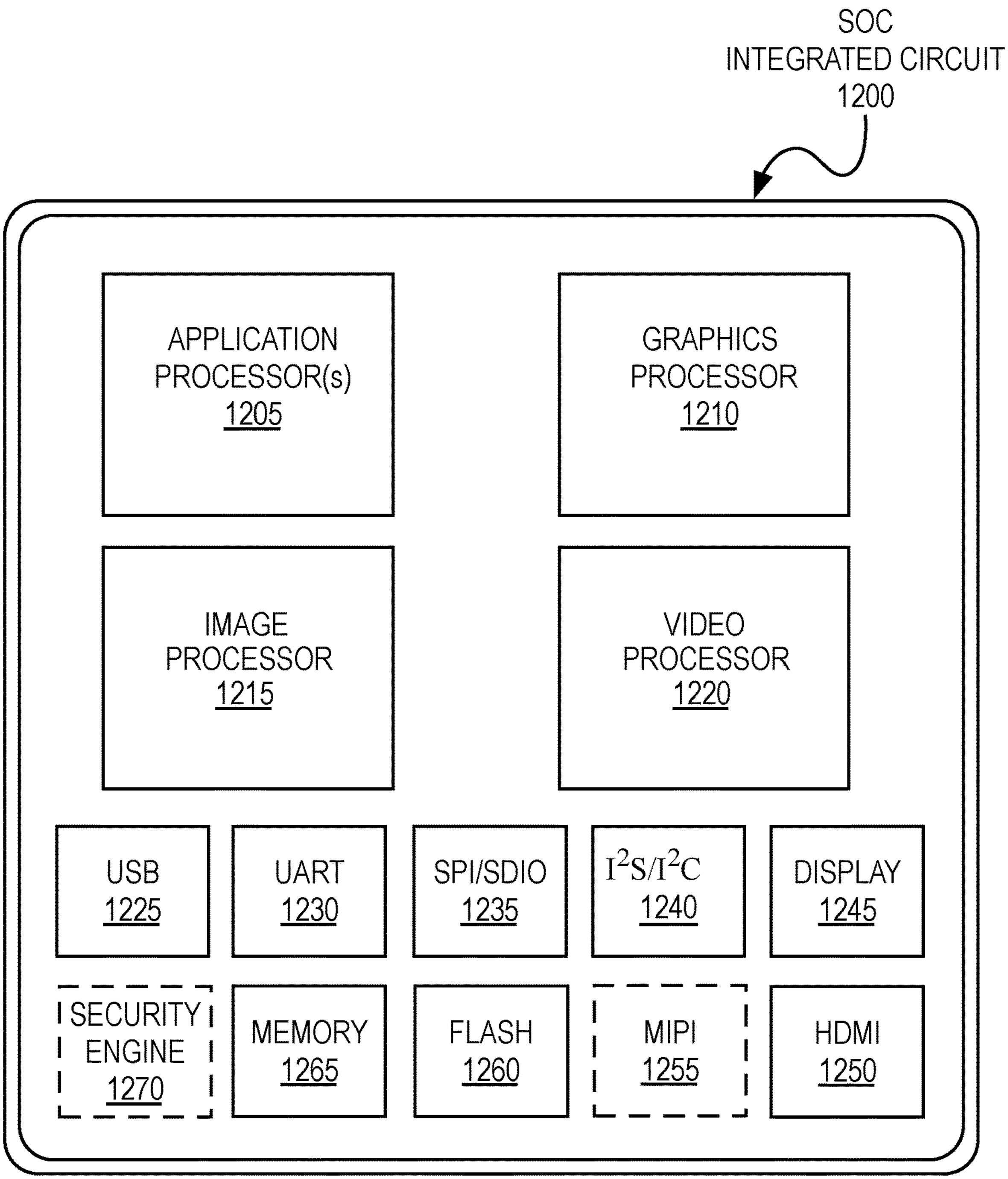

SOC
INTEGRATED CIRCUIT
1200
APPLICATION
PROCESSOR(s)
1205
GRAPHICS
PROCESSOR
1210
IMAGE
PROCESSOR
1215
VIDEO
PROCESSOR
1220
USB
1225
UART
1230
SPI/SDIO
1235
I2S/I2C
1240
DISPLAY
1245
SECURITY
ENGINE
1270
MEMORY
1265
FLASH
1260
MIPI
1255
HDMI
1250

FIG. 12

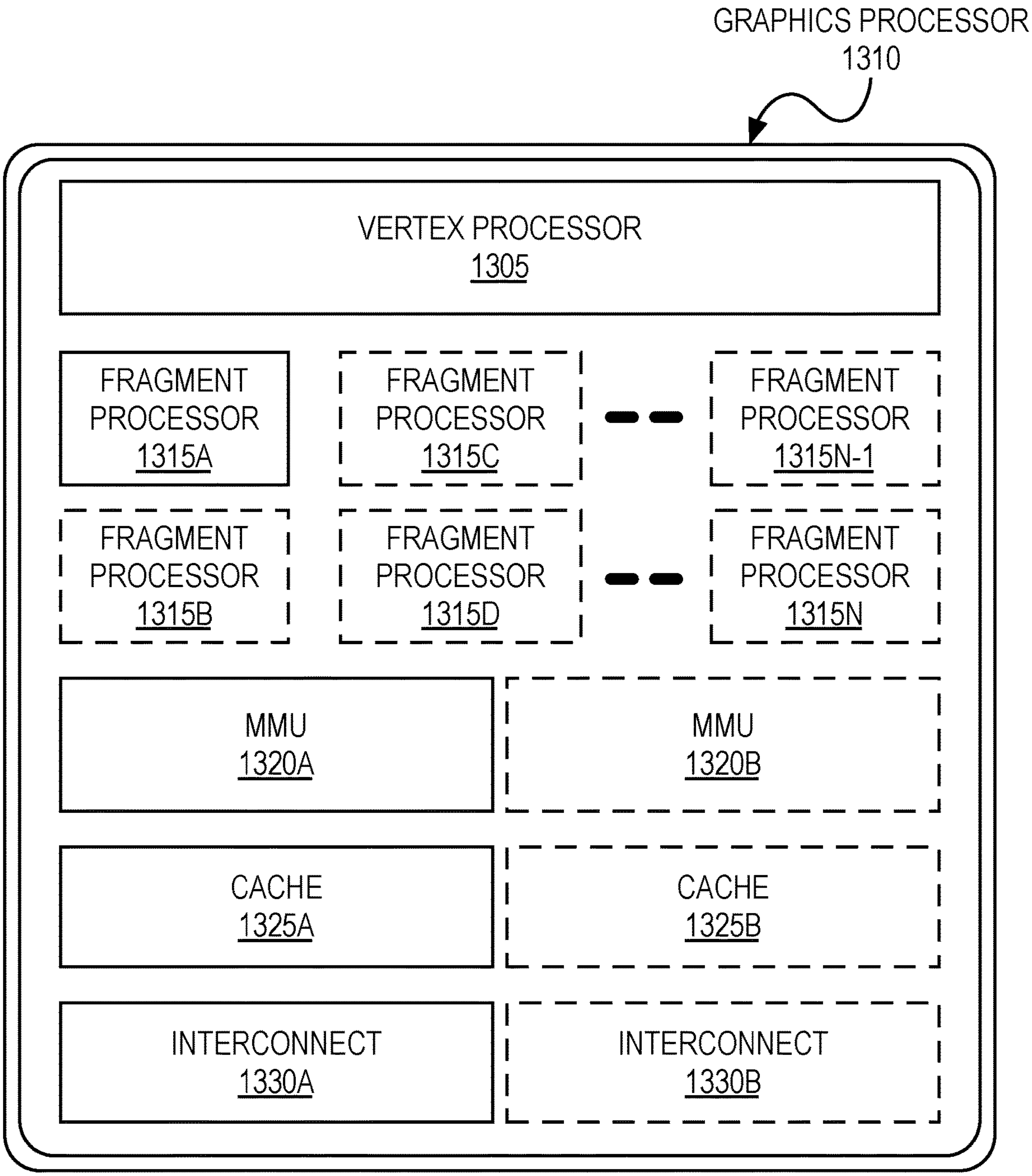

GRAPHICS PROCESSOR
1310
VERTEX PROCESSOR
1305
FRAGMENT PROCESSOR
1315A
FRAGMENT PROCESSOR
1315C
FRAGMENT PROCESSOR
1315N-1
FRAGMENT PROCESSOR
1315B
FRAGMENT PROCESSOR
1315D
FRAGMENT PROCESSOR
1315N
MMU
1320A
MMU
1320B
CACHE
1325A
CACHE
1325B
INTERCONNECT
1330A
INTERCONNECT
1330B

FIG. 13

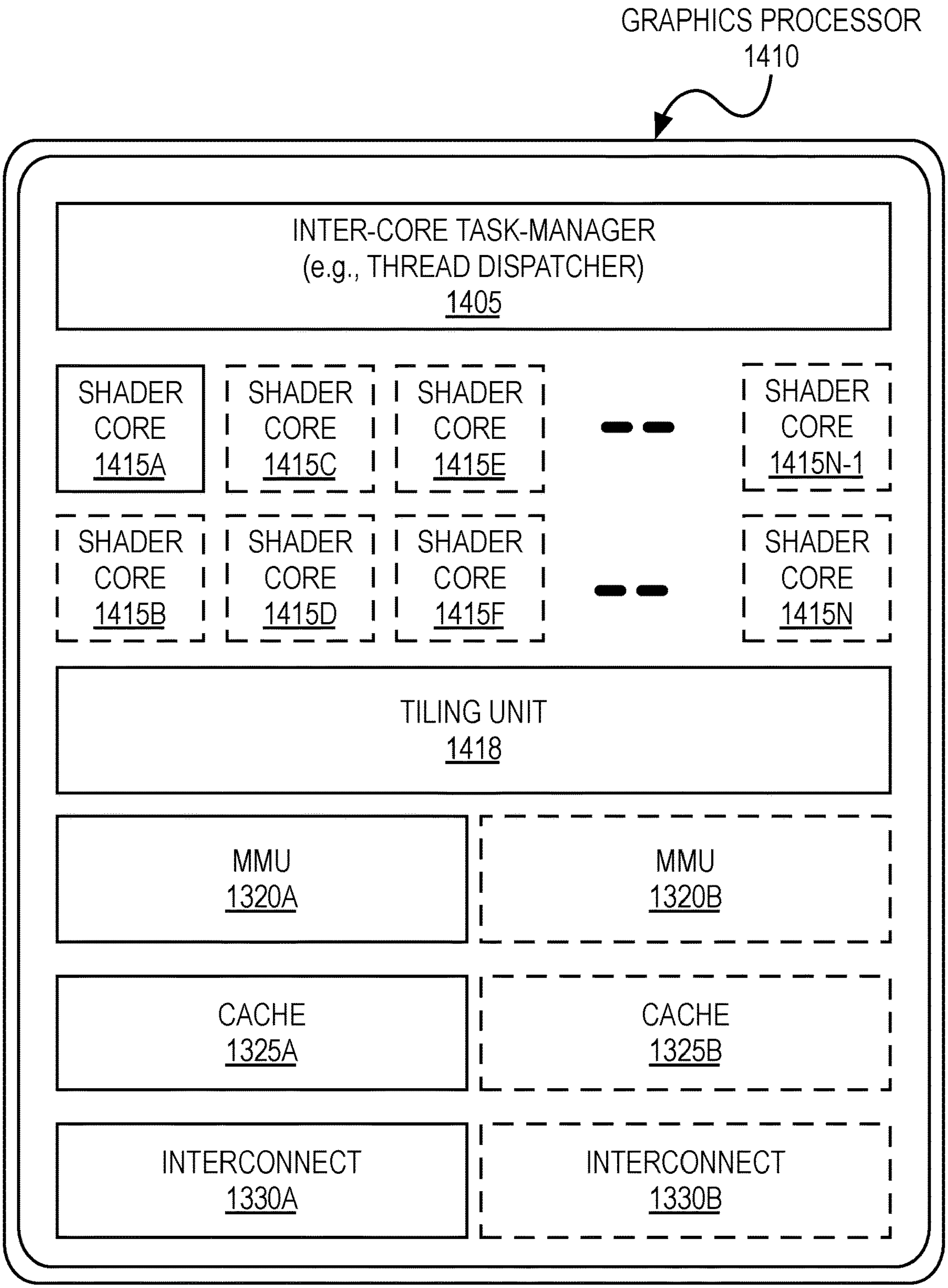

GRAPHICS PROCESSOR
1410
INTER-CORE TASK-MANAGER
(e.g., THREAD DISPATCHER)
1405
SHADER CORE 1415A
SHADER CORE 1415C
SHADER CORE 1415E
SHADER CORE 1415N-1
SHADER CORE 1415B
SHADER CORE 1415D
SHADER CORE 1415F
SHADER CORE 1415N
TILING UNIT
1418
MMU 1320A
MMU 1320B
CACHE 1325A
CACHE 1325B
INTERCONNECT 1330A
INTERCONNECT 1330B

FIG. 14

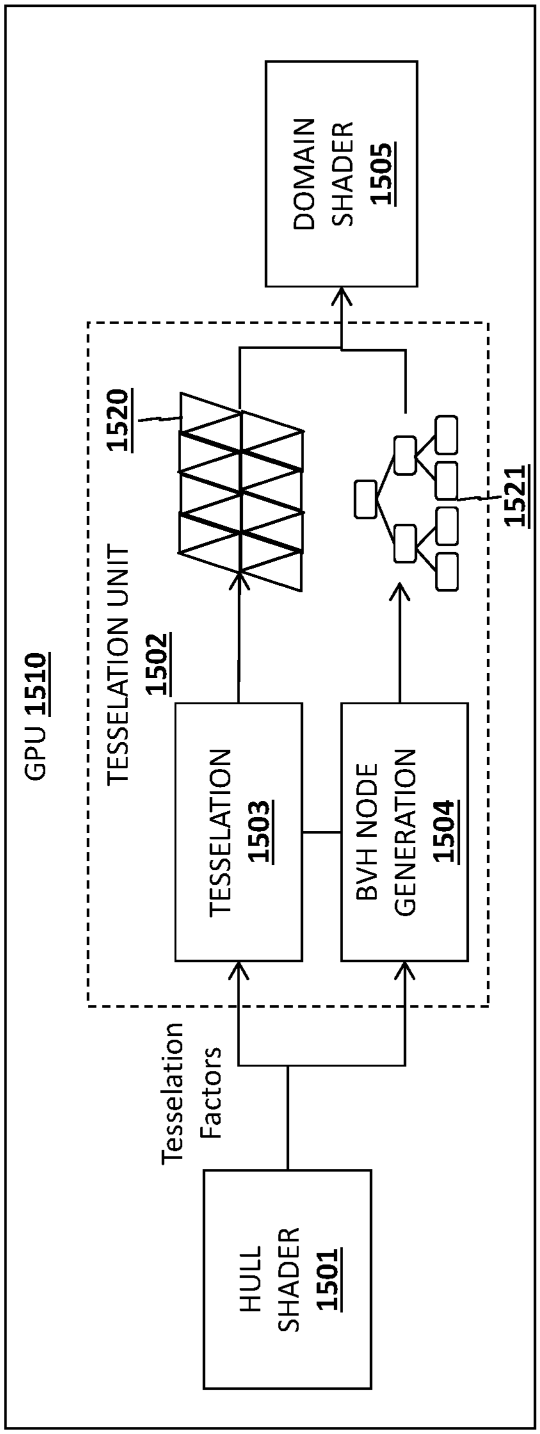

GPU 1510
HULL SHADER 1501
Tesselation Factors
TESSELATION UNIT 1502
TESSELATION 1503
BVH NODE GENERATION 1504
1520
1521
DOMAIN SHADER 1505

FIG. 15

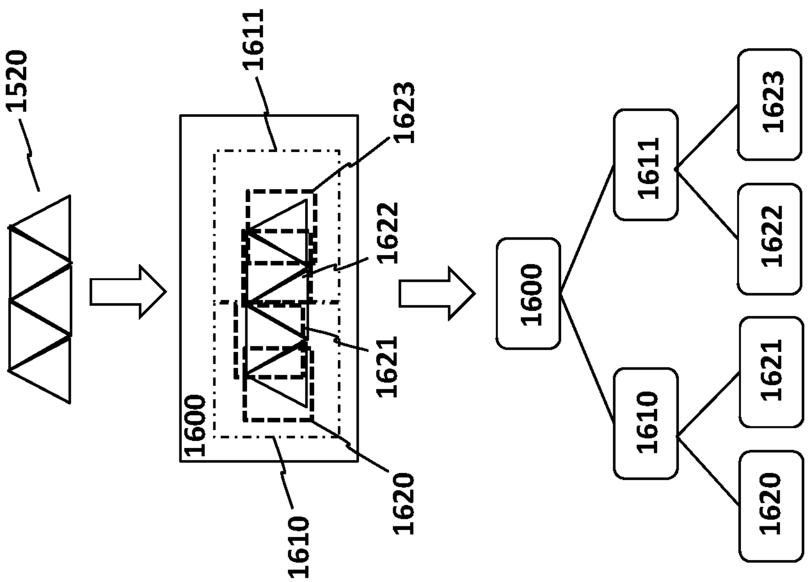

1520
1611
1623
1622
1621
1600
1610
1620
1600
1611
1610
1623
1622
1621
1620

FIG. 16

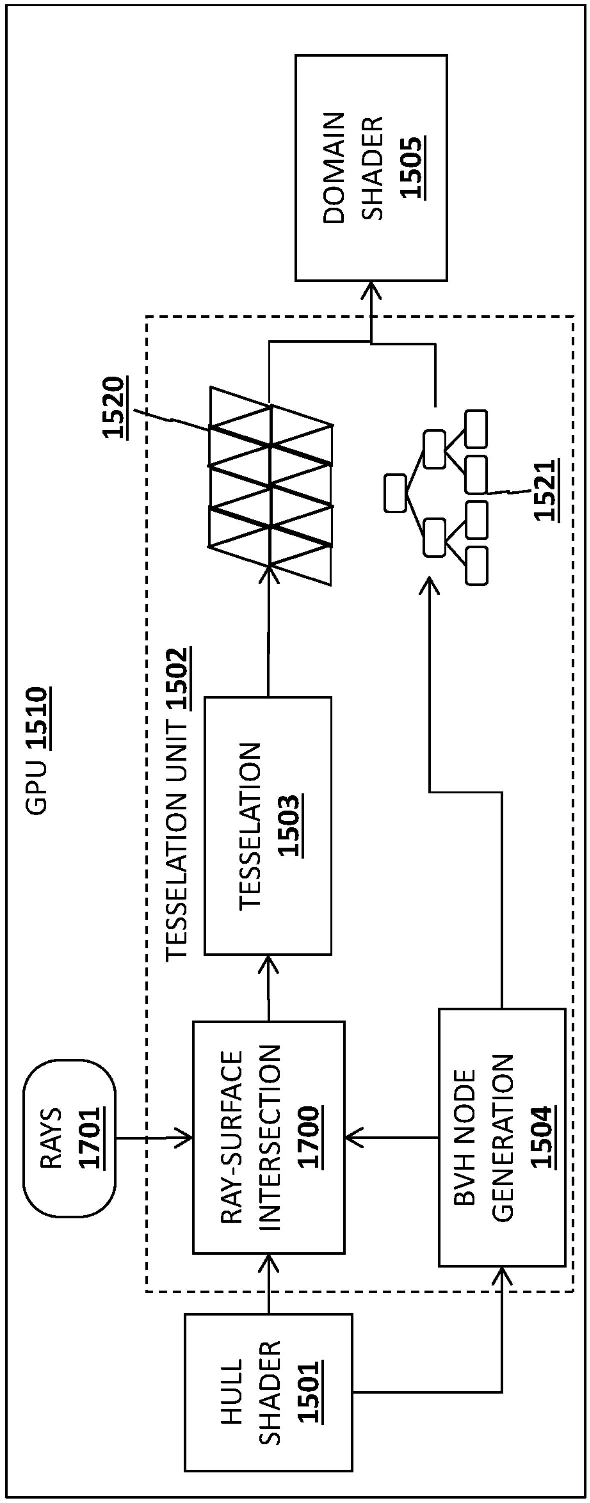

GPU 1510
TESSELATION UNIT 1502
RAYS 1701
HULL SHADER 1501
RAY-SURFACE INTERSECTION 1700
TESSELATION 1503
BVH NODE GENERATION 1504
1520
1521
DOMAIN SHADER 1505

FIG. 17

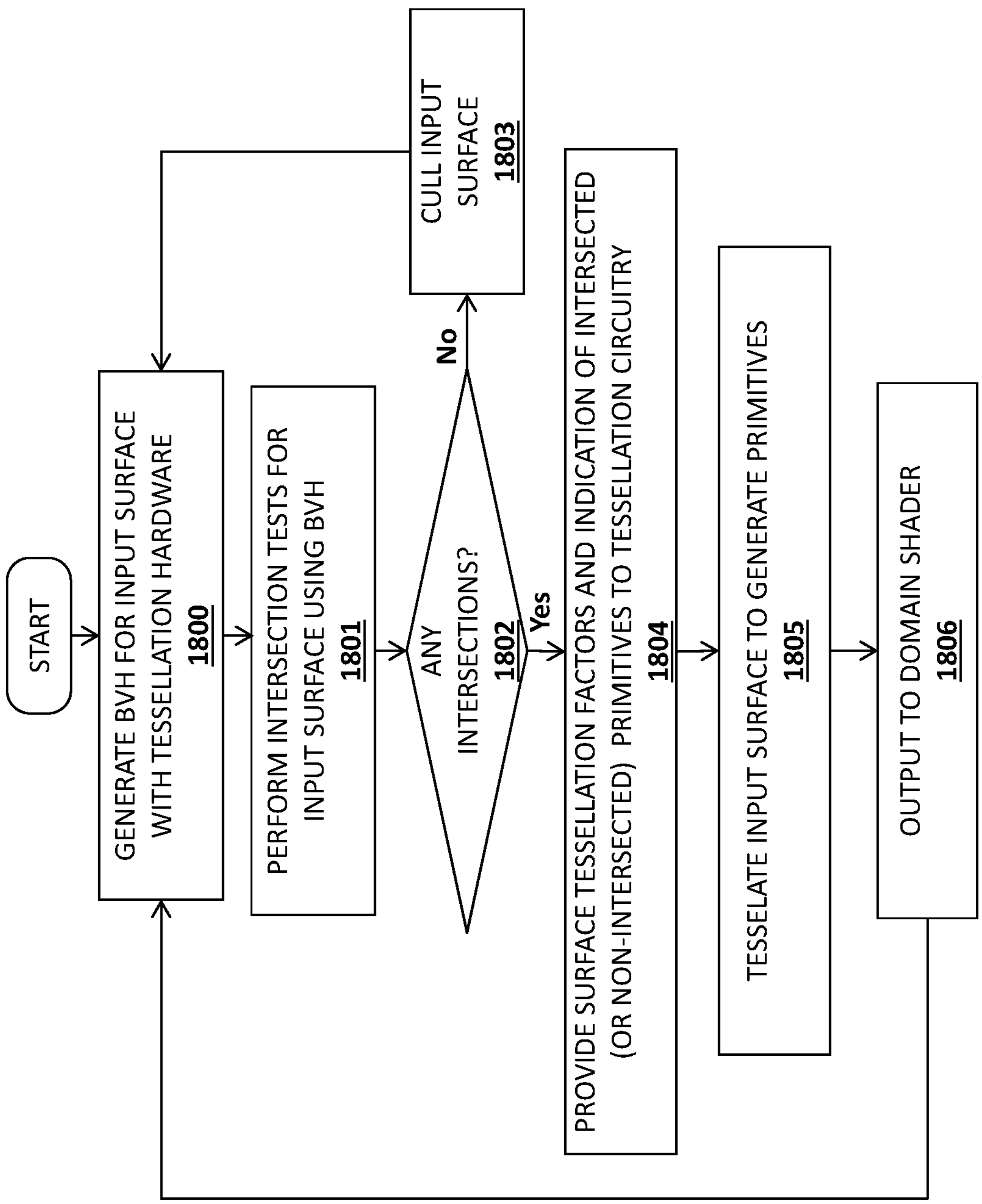

START
GENERATE BVH FOR INPUT SURFACE WITH TESSELLATION HARDWARE
1800
PERFORM INTERSECTION TESTS FOR INPUT SURFACE USING BVH
1801
ANY INTERSECTIONS?
1802
No
Yes
CULL INPUT SURFACE
1803
PROVIDE SURFACE TESSELLATION FACTORS AND INDICATION OF INTERSECTED (OR NON-INTERSECTED) PRIMITIVES TO TESSELLATION CIRCUITRY
1804
TESSELATE INPUT SURFACE TO GENERATE PRIMITIVES
1805
OUTPUT TO DOMAIN SHADER
1806

FIG. 18

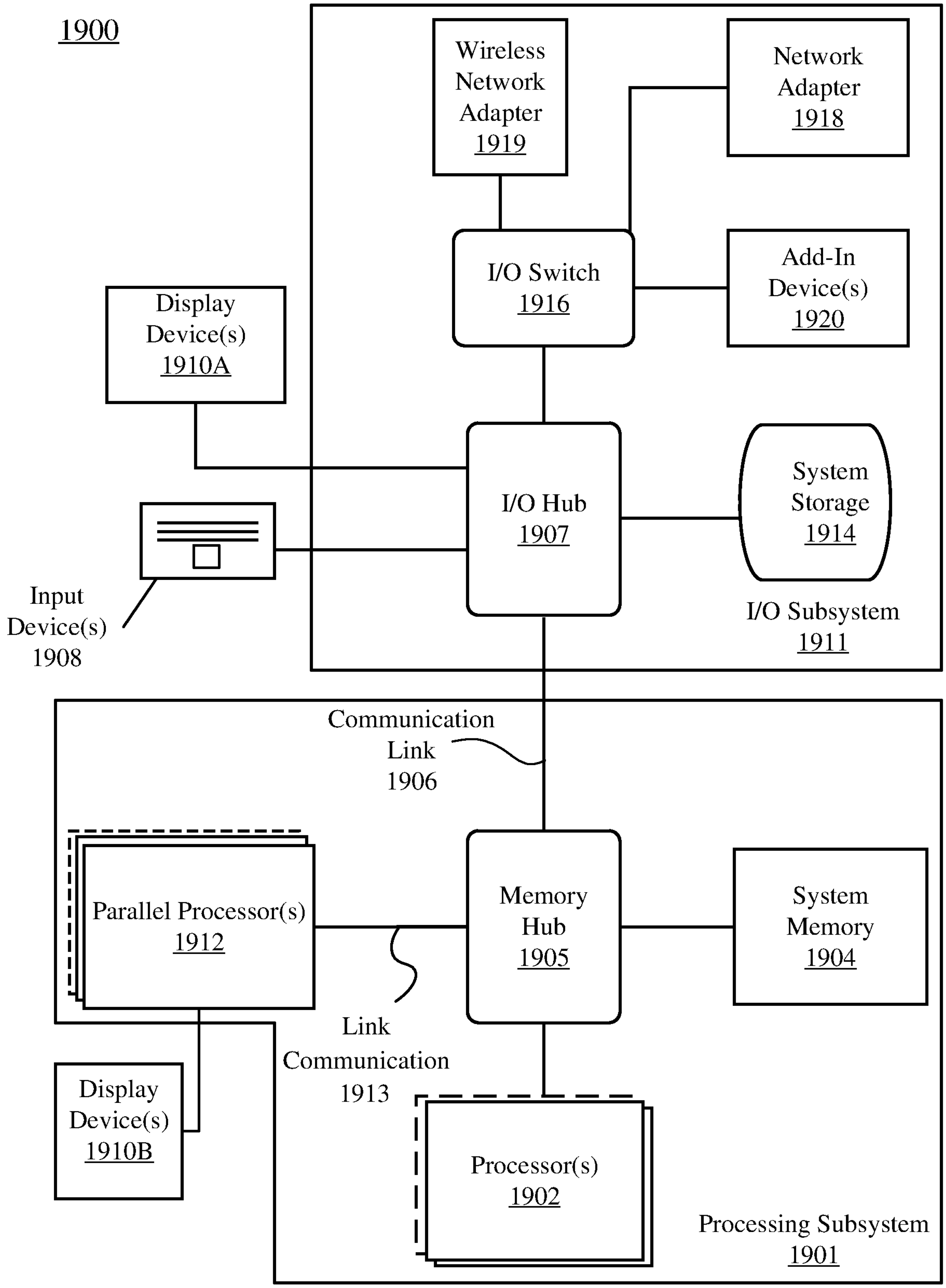

1900
Wireless Network Adapter 1919
Network Adapter 1918
I/O Switch 1916
Add-In Device(s) 1920
Display Device(s) 1910A
I/O Hub 1907
System Storage 1914
Input Device(s) 1908
I/O Subsystem 1911
Communication Link 1906
Parallel Processor(s) 1912
Memory Hub 1905
System Memory 1904
Link Communication 1913
Display Device(s) 1910B
Processor(s) 1902
Processing Subsystem 1901

FIG. 19

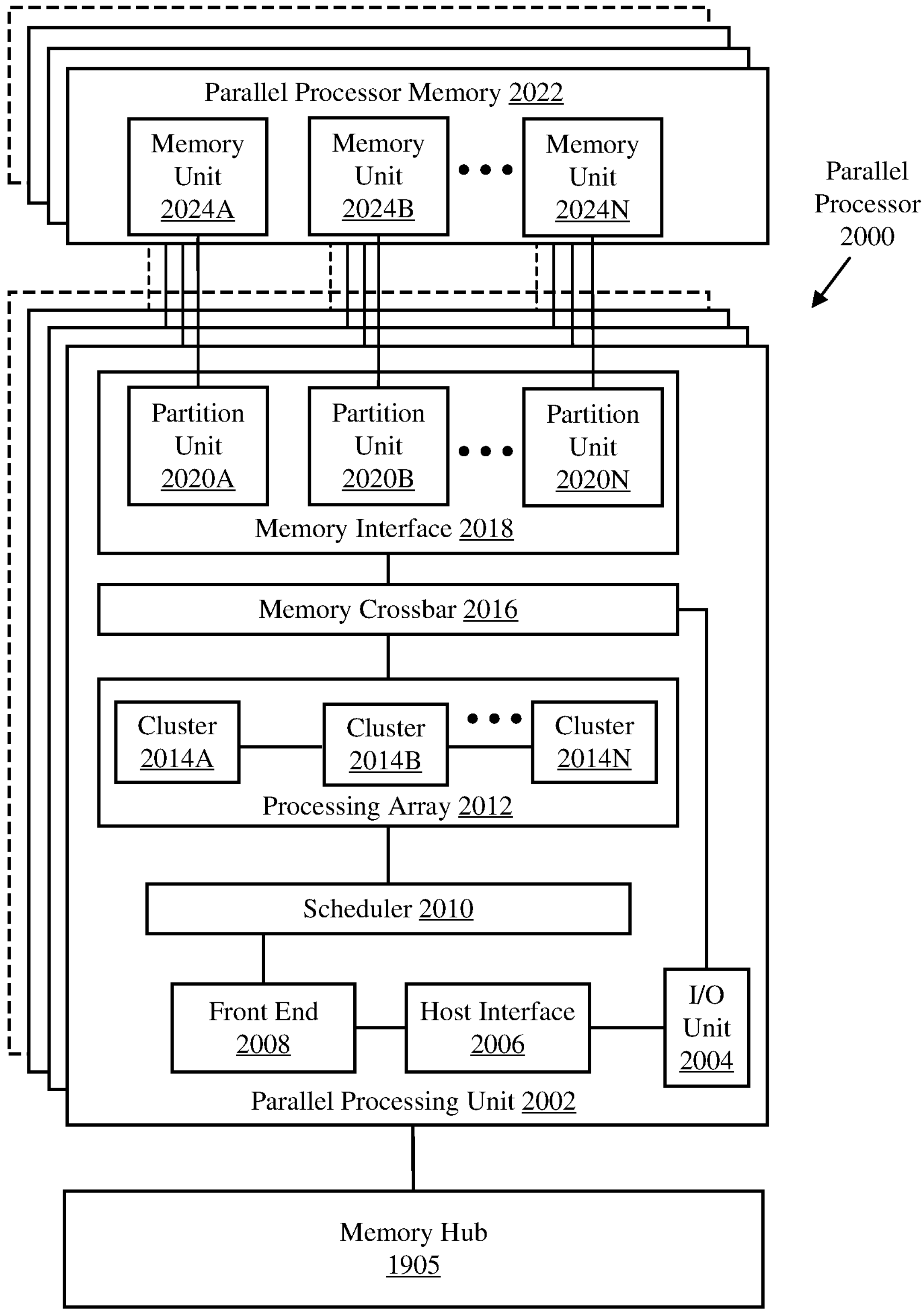

Parallel Processor Memory 2022
Memory Unit 2024A
Memory Unit 2024B
Memory Unit 2024N
Parallel Processor 2000
Partition Unit 2020A
Partition Unit 2020B
Partition Unit 2020N
Memory Interface 2018
Memory Crossbar 2016
Cluster 2014A
Cluster 2014B
Cluster 2014N
Processing Array 2012
Scheduler 2010
Front End 2008
Host Interface 2006
I/O Unit 2004
Parallel Processing Unit 2002
Memory Hub 1905

FIG. 20A

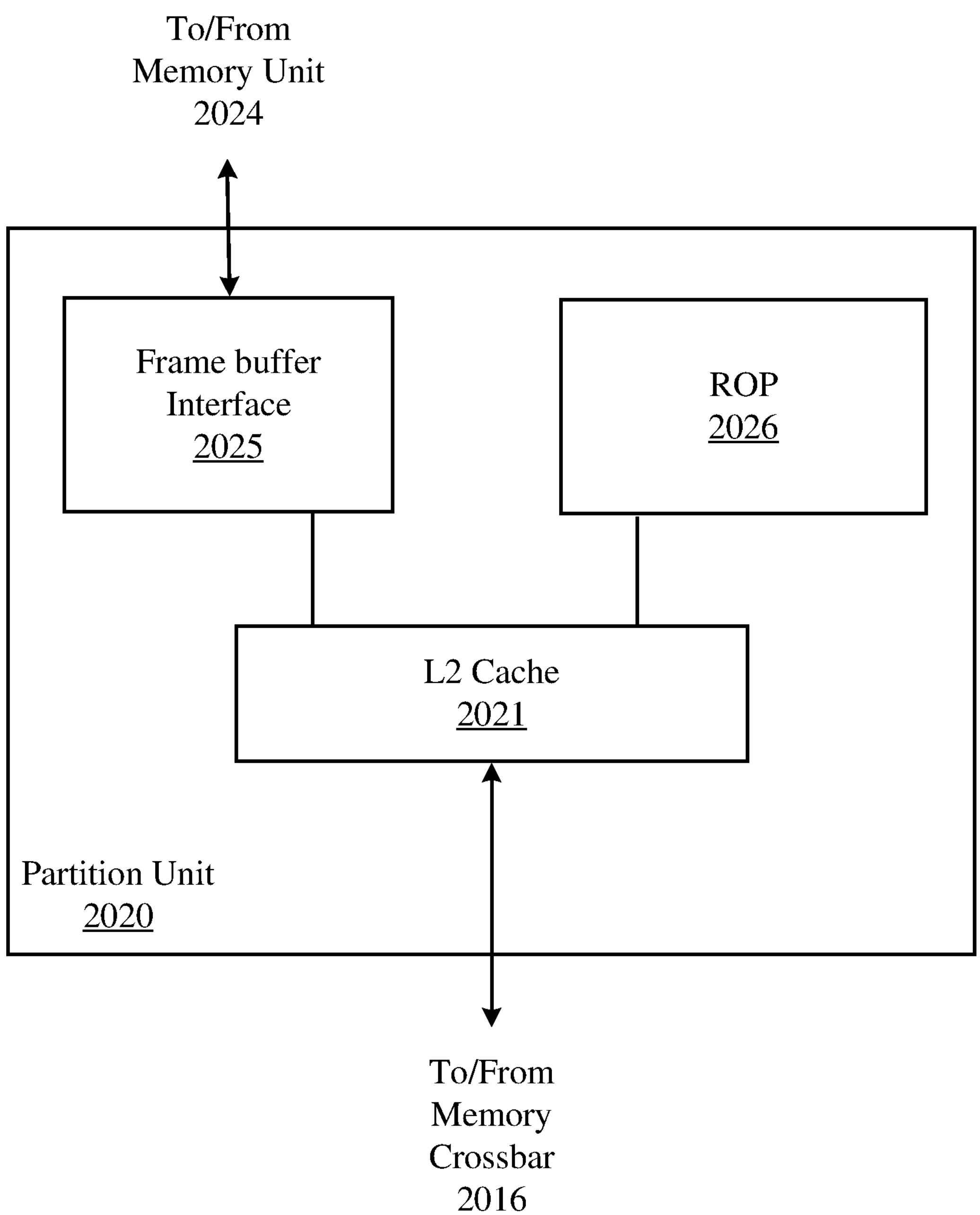

To/From
Memory Unit
2024
Frame buffer
Interface
2025
ROP
2026
L2 Cache
2021
Partition Unit
2020
To/From
Memory
Crossbar
2016

FIG. 20B

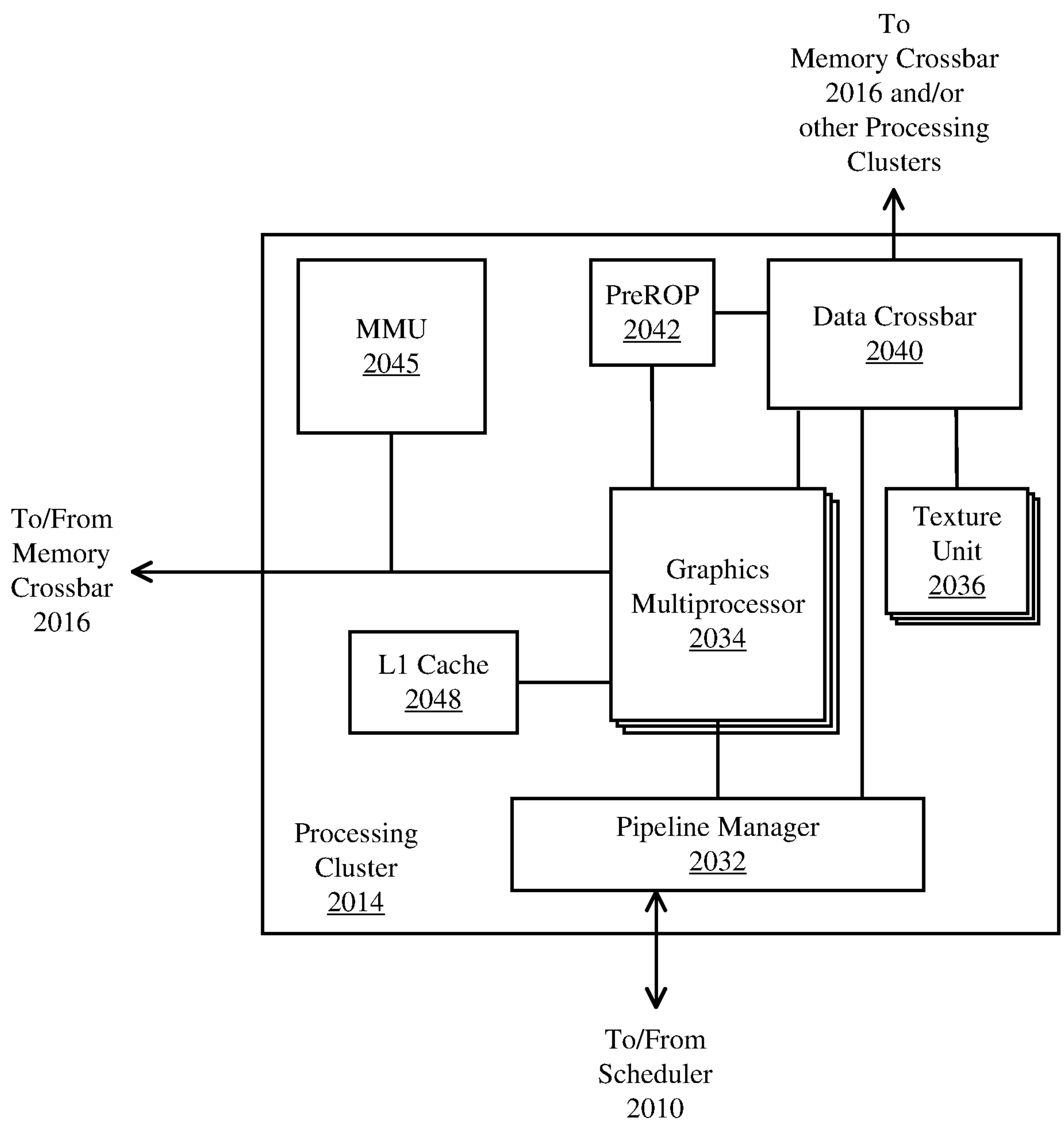
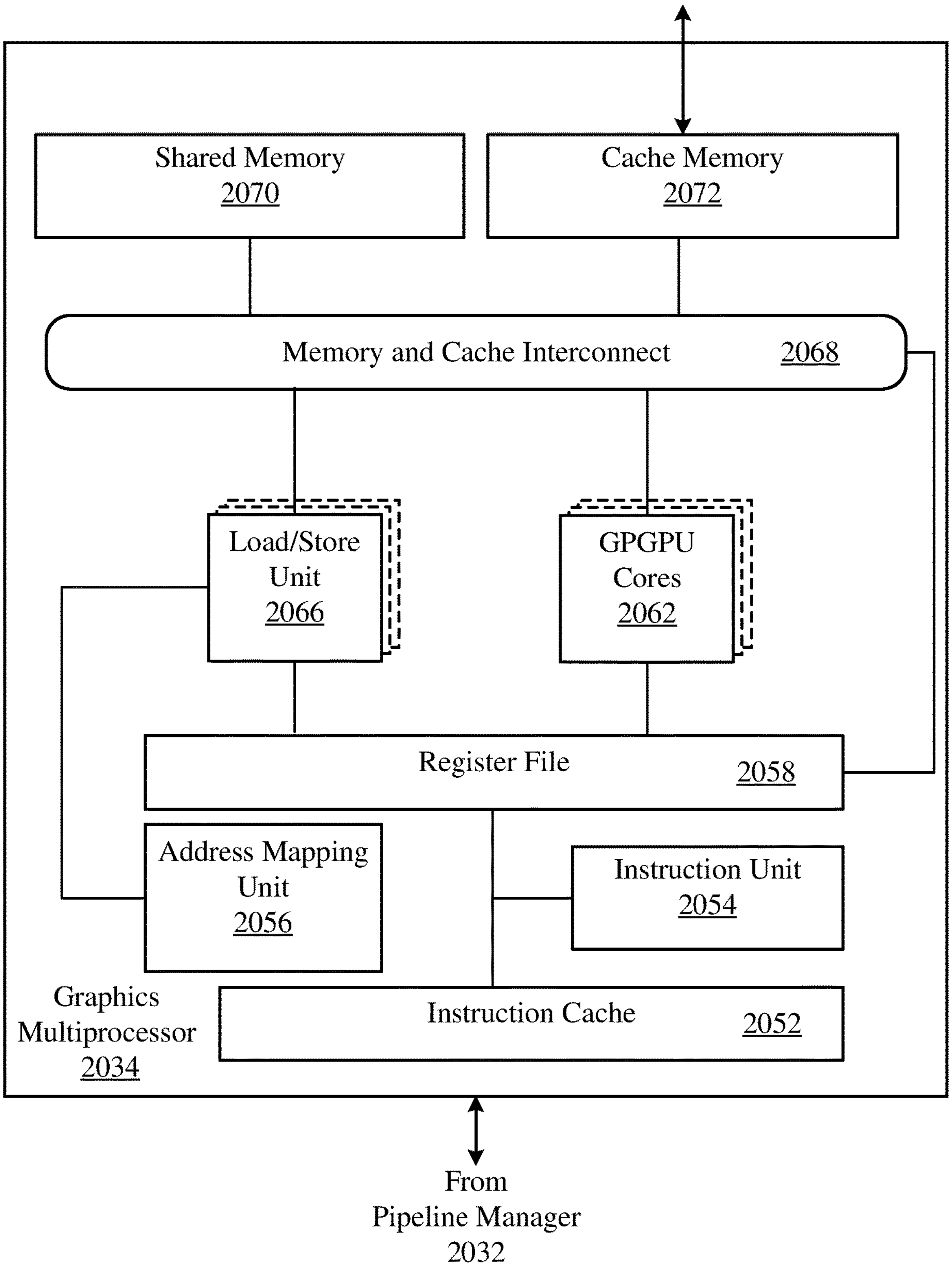

To
Memory Crossbar
2016 and/or
other Processing
Clusters
MMU
2045
PreROP
2042
Data Crossbar
2040
To/From
Memory
Crossbar
2016
Graphics
Multiprocessor
2034
Texture
Unit
2036
L1 Cache
2048
Processing
Cluster
2014
Pipeline Manager
2032
To/From
Scheduler
2010

FIG. 20C



Shared Memory
2070
Cache Memory
2072
Memory and Cache Interconnect
2068
Load/Store
Unit
2066
GPGPU
Cores
2062
Register File
2058
Address Mapping
Unit
2056
Instruction Unit
2054
Graphics
Multiprocessor
2034
Instruction Cache
2052
From
Pipeline Manager
2032

FIG. 20D

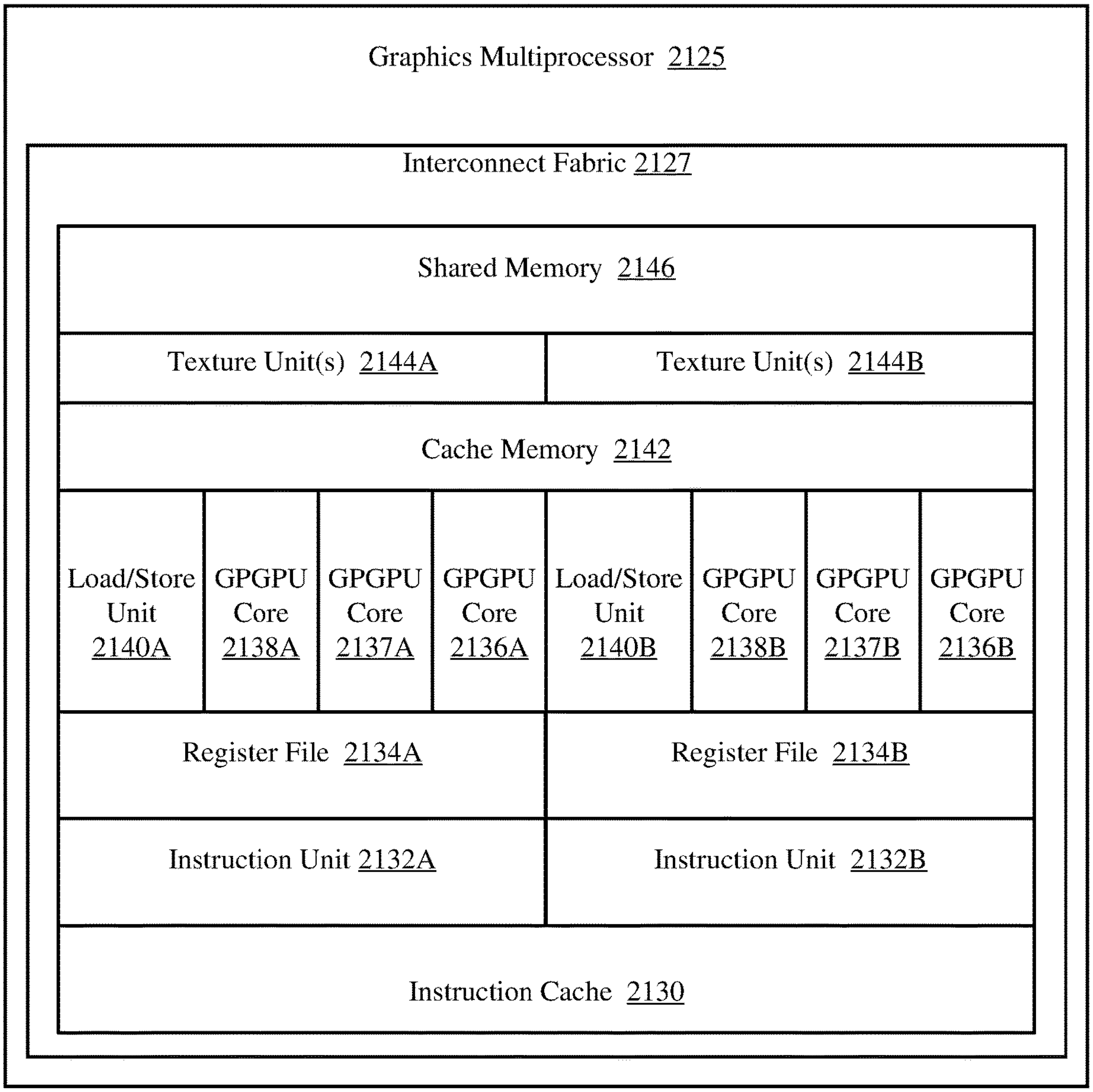

Graphics Multiprocessor 2125
Interconnect Fabric 2127
Shared Memory 2146
Texture Unit(s) 2144A
Texture Unit(s) 2144B
Cache Memory 2142
Load/Store Unit 2140A
GPGPU Core 2138A
GPGPU Core 2137A
GPGPU Core 2136A
Load/Store Unit 2140B
GPGPU Core 2138B
GPGPU Core 2137B
GPGPU Core 2136B
Register File 2134A
Register File 2134B
Instruction Unit 2132A
Instruction Unit 2132B
Instruction Cache 2130

FIG. 21A

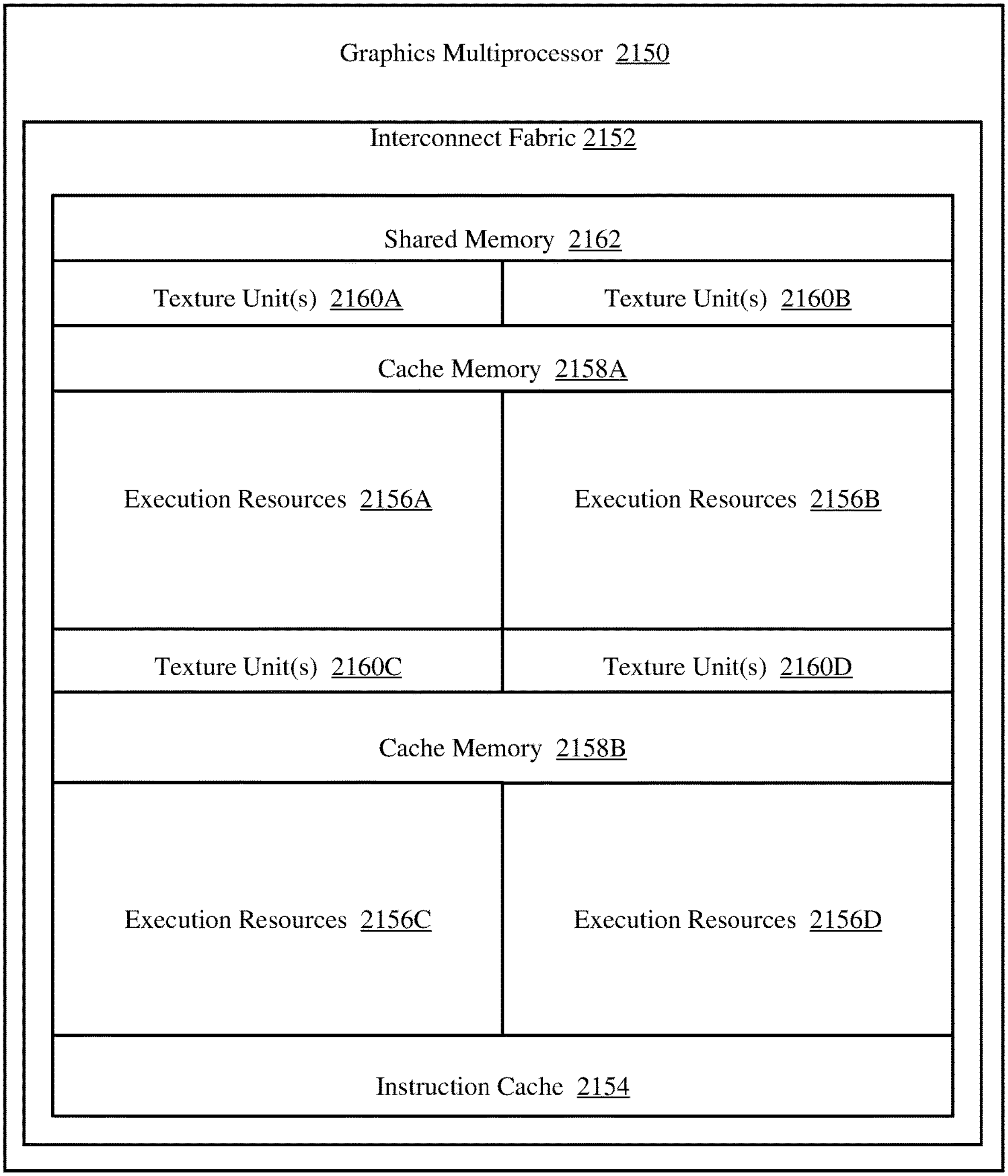

Graphics Multiprocessor 2150
Interconnect Fabric 2152
Shared Memory 2162
Texture Unit(s) 2160A
Texture Unit(s) 2160B
Cache Memory 2158A
Execution Resources 2156A
Execution Resources 2156B
Texture Unit(s) 2160C
Texture Unit(s) 2160D
Cache Memory 2158B
Execution Resources 2156C
Execution Resources 2156D
Instruction Cache 2154

FIG. 21B

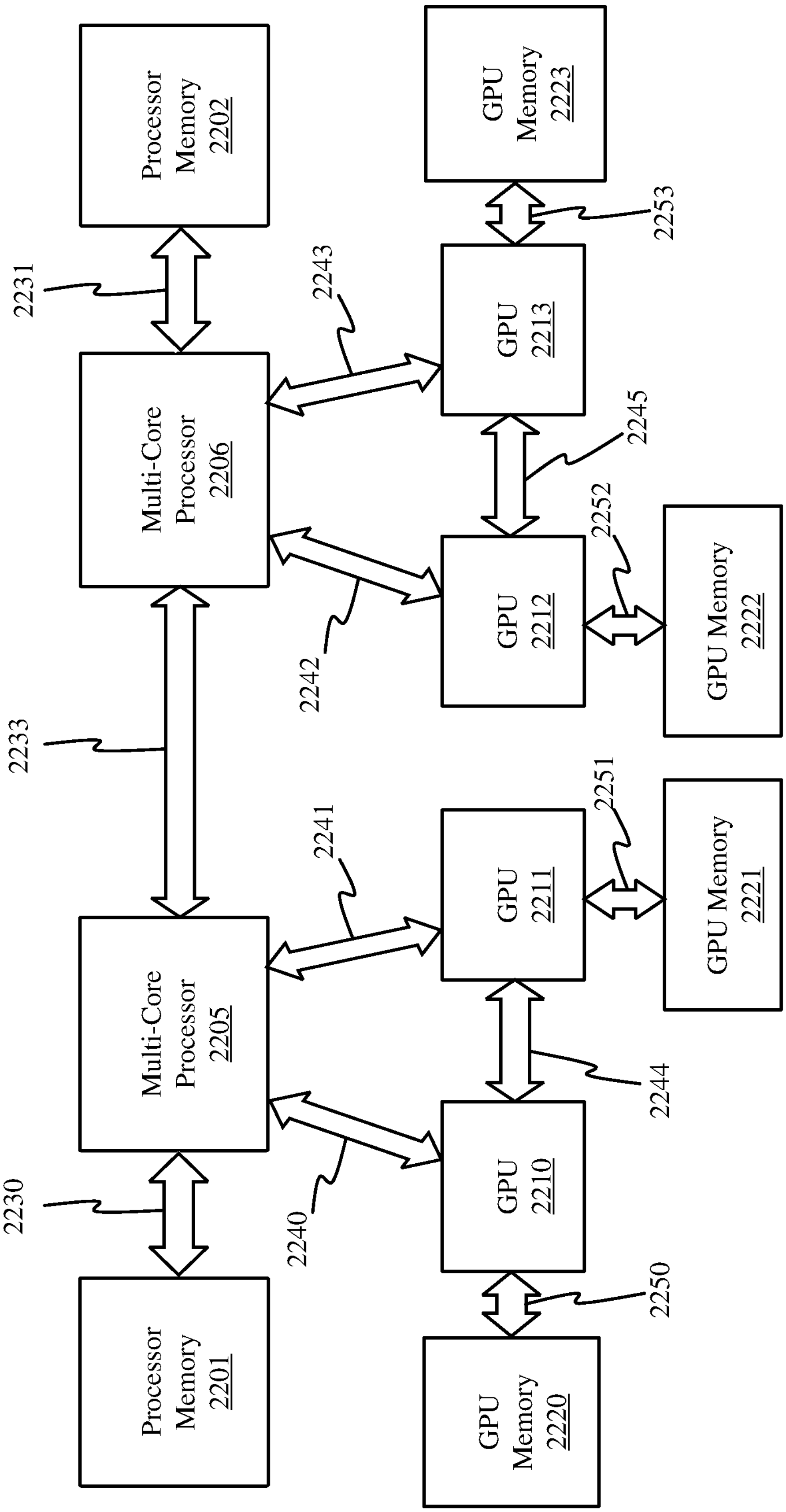

Processor Memory 2202
GPU Memory 2223
2231
2243
2253
GPU 2213
Multi-Core Processor 2206
2245
2252
GPU Memory 2222
GPU 2212
2242
2233
2241
2251
GPU 2211
GPU Memory 2221
Multi-Core Processor 2205
2244
GPU 2210
2230
2240
2250
Processor Memory 2201
GPU Memory 2220

FIG. 22A

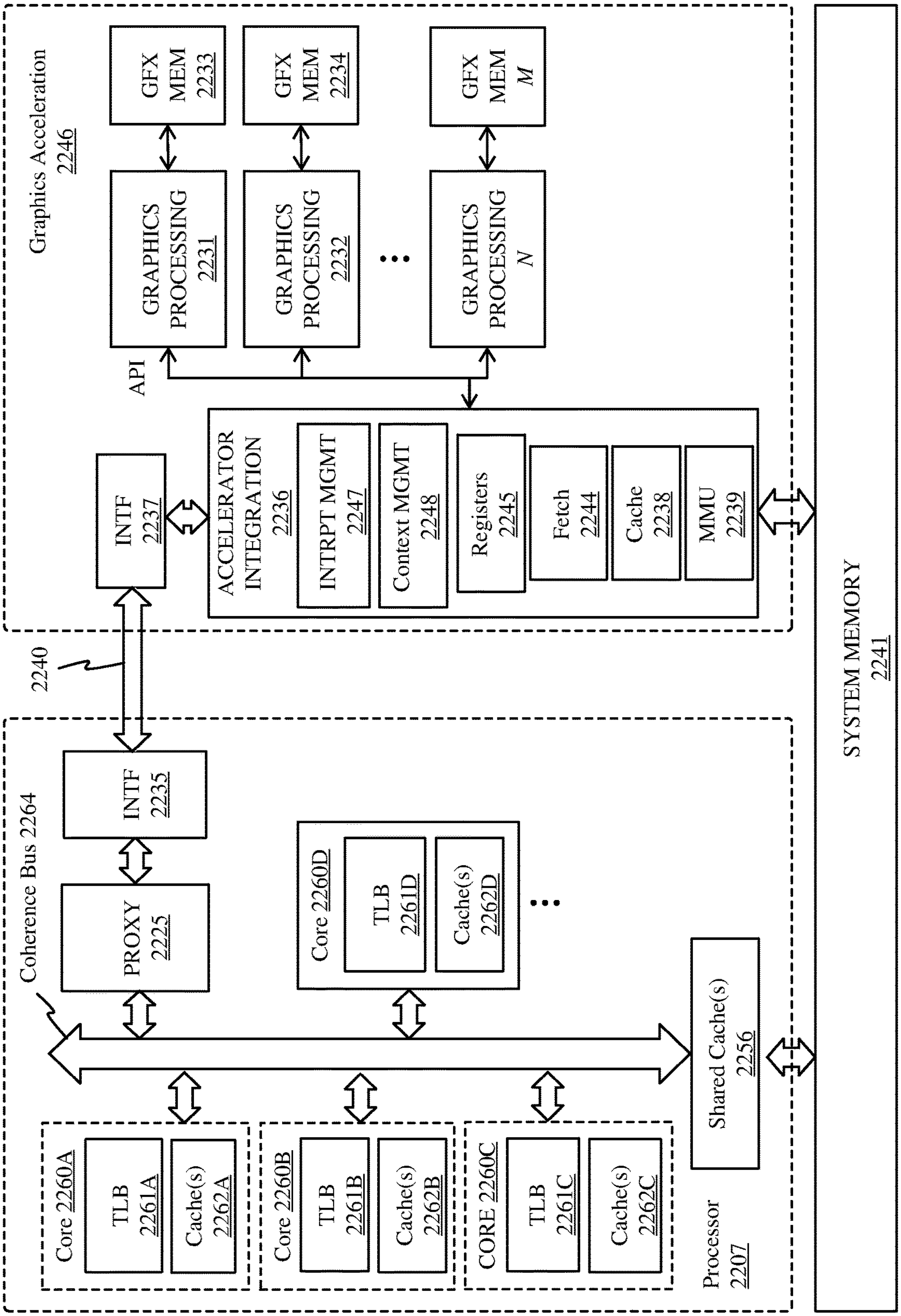

Graphics Acceleration 2246
GFX MEM 2233
GFX MEM 2234
GFX MEM M
GRAPHICS PROCESSING 2231
GRAPHICS PROCESSING 2232
GRAPHICS PROCESSING N
API
INTF 2237
ACCELERATOR INTEGRATION 2236
INTRPT MGMT 2247
Context MGMT 2248
Registers 2245
Fetch 2244
Cache 2238
MMU 2239
2240
Coherence Bus 2264
INTF 2235
PROXY 2225
Core 2260D
TLB 2261D
Cache(s) 2262D
Shared Cache(s) 2256
Core 2260A
TLB 2261A
Cache(s) 2262A
Core 2260B
TLB 2261B
Cache(s) 2262B
CORE 2260C
TLB 2261C
Cache(s) 2262C
Processor 2207
SYSTEM MEMORY 2241

FIG. 22B

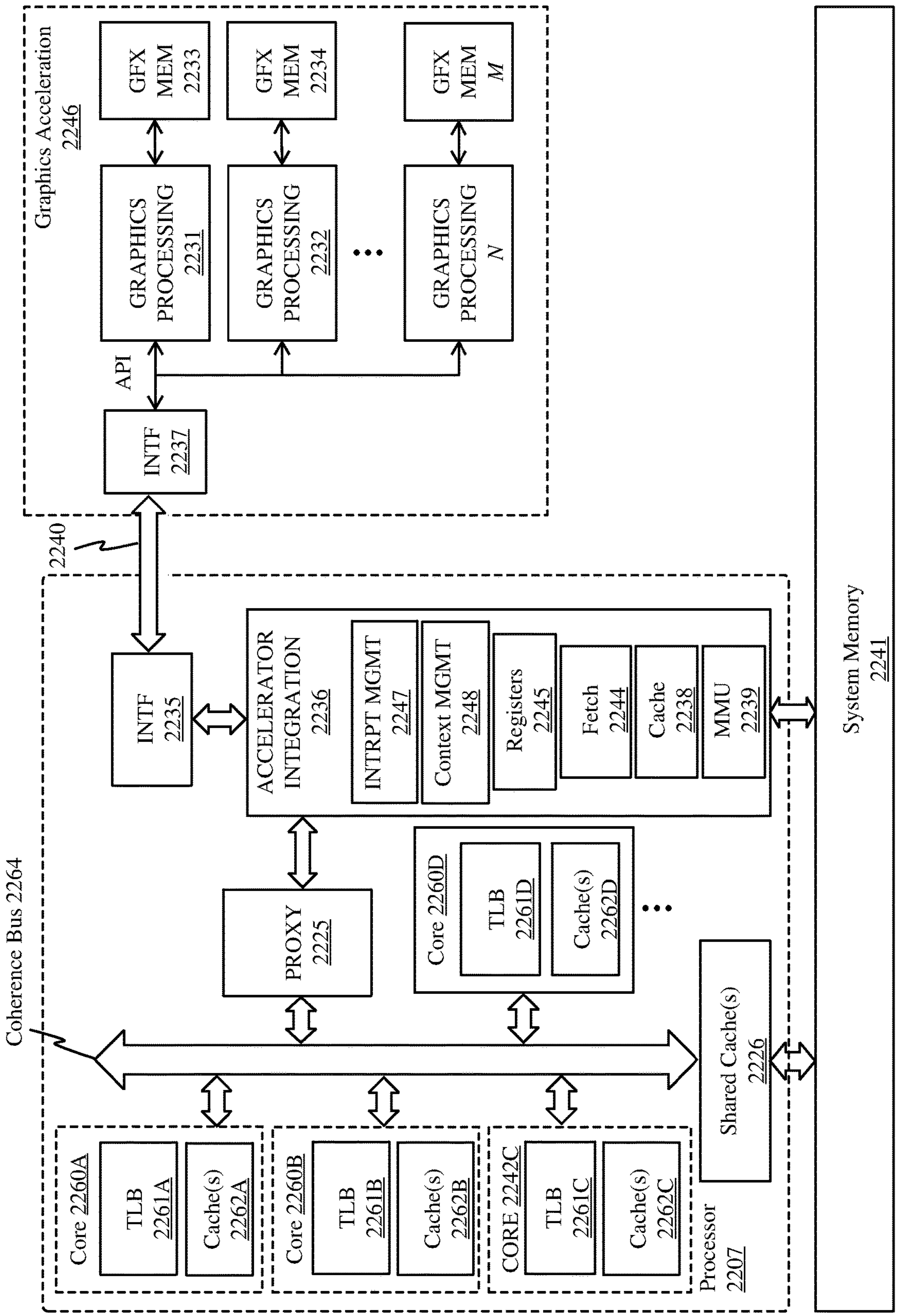

Graphics Acceleration
2246
GFX MEM 2233
GFX MEM 2234
GFX MEM M
GRAPHICS PROCESSING 2231
GRAPHICS PROCESSING 2232
GRAPHICS PROCESSING N
API
INTF 2237
2240
INTF 2235
ACCELERATOR INTEGRATION 2236
INTRPT MGMT 2247
Context MGMT 2248
Registers 2245
Fetch 2244
Cache 2238
MMU 2239
System Memory
2241
Coherence Bus 2264
PROXY 2225
Core 2260D
TLB 2261D
Cache(s) 2262D
Shared Cache(s) 2226
Core 2260A
TLB 2261A
Cache(s) 2262A
Core 2260B
TLB 2261B
Cache(s) 2262B
CORE 2242C
TLB 2261C
Cache(s) 2262C
Processor
2207

FIG. 22C

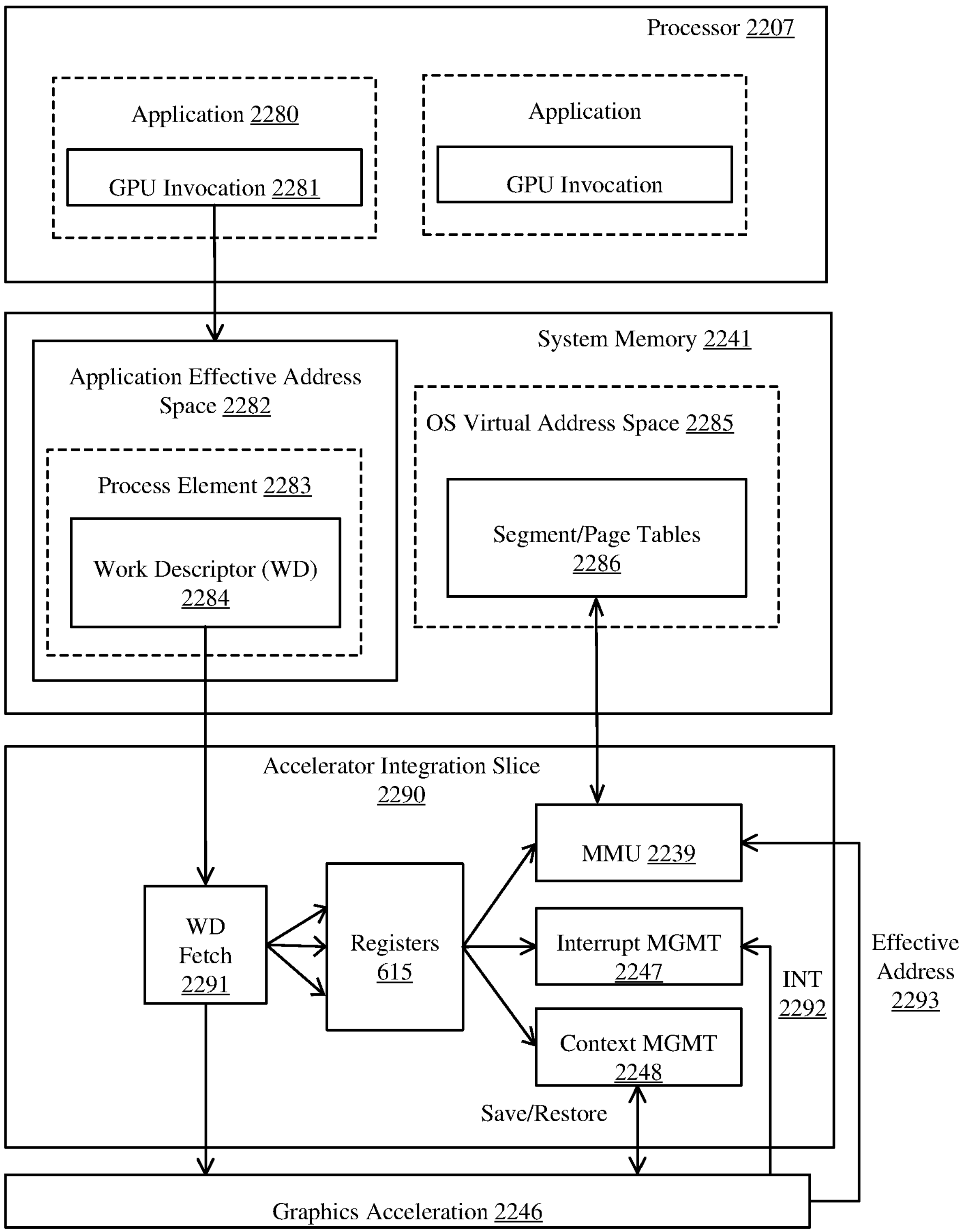

Processor 2207
Application 2280
GPU Invocation 2281
Application
GPU Invocation
System Memory 2241
Application Effective Address Space 2282
Process Element 2283
Work Descriptor (WD) 2284
OS Virtual Address Space 2285
Segment/Page Tables 2286
Accelerator Integration Slice 2290
WD Fetch 2291
Registers 615
MMU 2239
Interrupt MGMT 2247
Context MGMT 2248
Save/Restore
INT 2292
Effective Address 2293
Graphics Acceleration 2246

FIG. 22D

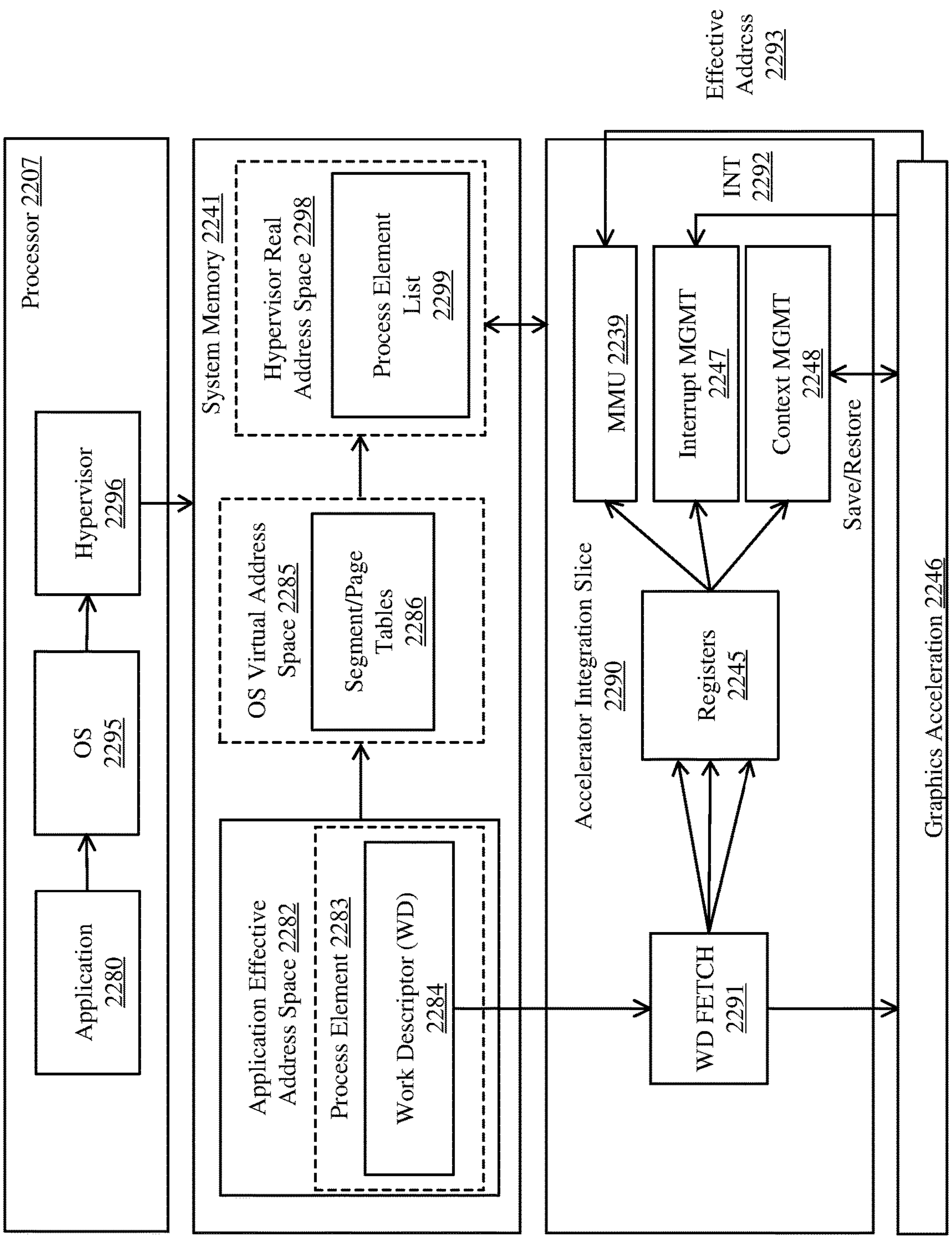

Processor 2207
Application 2280
OS 2295
Hypervisor 2296
System Memory 2241
Application Effective Address Space 2282
Process Element 2283
Work Descriptor (WD) 2284
OS Virtual Address Space 2285
Segment/Page Tables 2286
Hypervisor Real Address Space 2298
Process Element List 2299
Accelerator Integration Slice 2290
WD FETCH 2291
Registers 2245
MMU 2239
Interrupt MGMT 2247
Context MGMT 2248
Save/Restore
INT 2292
Effective Address 2293
Graphics Acceleration 2246

FIG. 22E

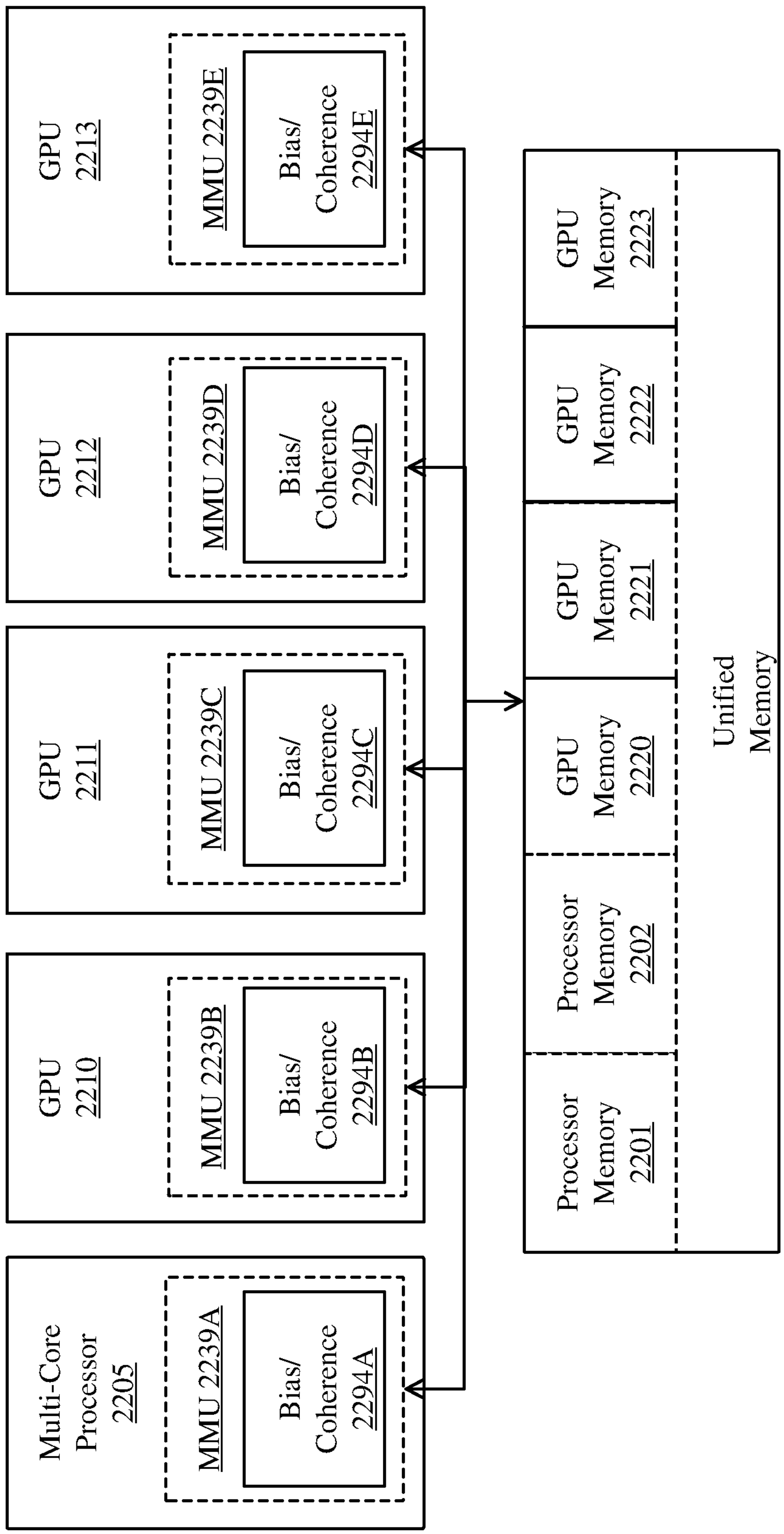

GPU 2213
MMU 2239E
Bias/ Coherence 2294E
GPU 2212
MMU 2239D
Bias/ Coherence 2294D
GPU 2211
MMU 2239C
Bias/ Coherence 2294C
GPU 2210
MMU 2239B
Bias/ Coherence 2294B
Multi-Core Processor 2205
MMU 2239A
Bias/ Coherence 2294A
GPU Memory 2223
GPU Memory 2222
GPU Memory 2221
GPU Memory 2220
Processor Memory 2202
Processor Memory 2201
Unified Memory

FIG. 22F

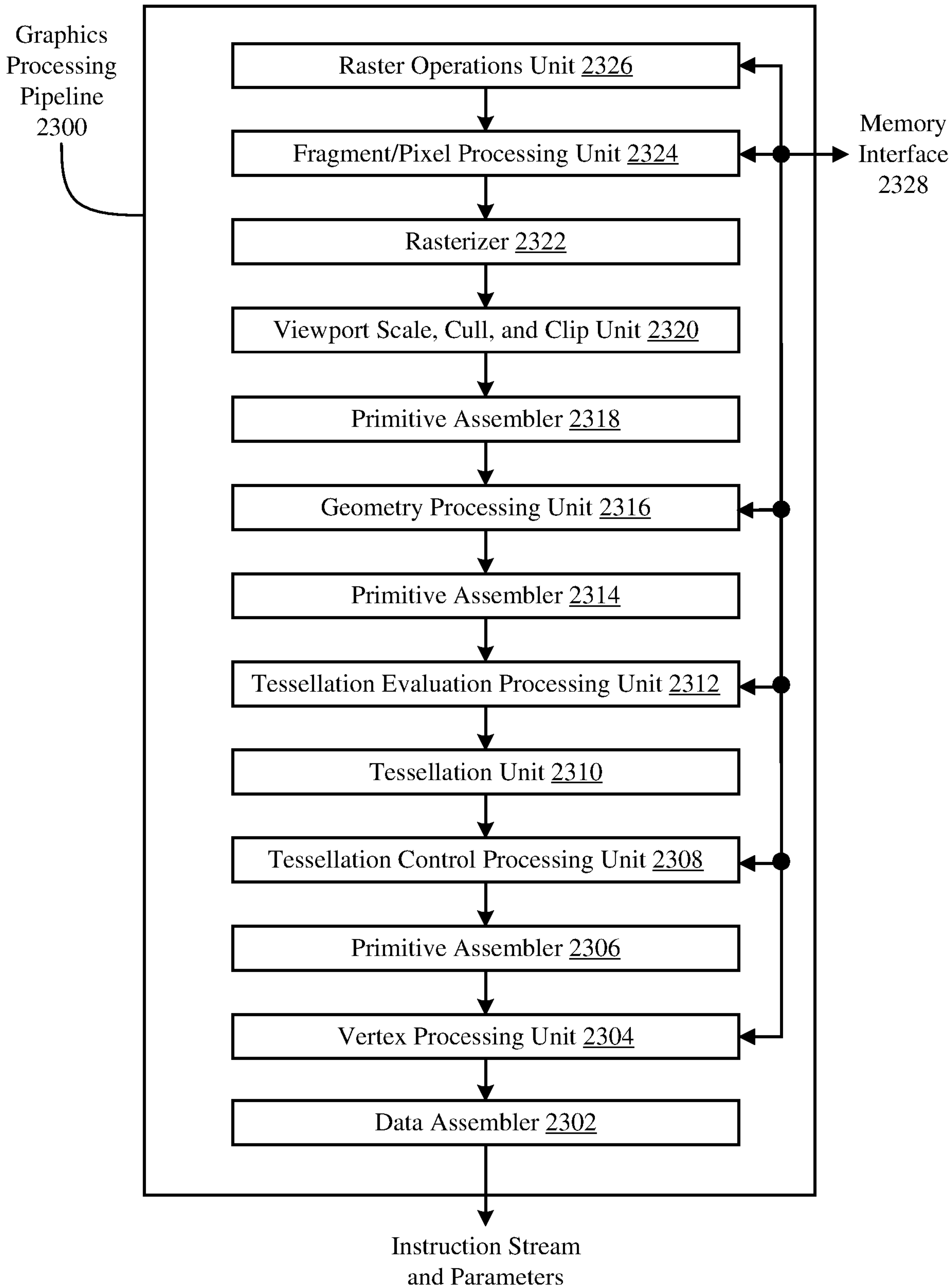

Graphics
Processing
Pipeline
2300
Raster Operations Unit 2326
Fragment/Pixel Processing Unit 2324
Memory
Interface
2328
Rasterizer 2322
Viewport Scale, Cull, and Clip Unit 2320
Primitive Assembler 2318
Geometry Processing Unit 2316
Primitive Assembler 2314
Tessellation Evaluation Processing Unit 2312
Tessellation Unit 2310
Tessellation Control Processing Unit 2308
Primitive Assembler 2306
Vertex Processing Unit 2304
Data Assembler 2302
Instruction Stream
and Parameters

FIG. 23

APPARATUS AND METHOD FOR IMPLEMENTING BOUNDING VOLUME HIERARCHY (BVH) OPERATIONS ON TESSELATION HARDWARE

BACKGROUND

Field of the Invention

This invention relates generally to the field of graphics processors. More particularly, the invention relates to an apparatus and method for implementing bounding volume hierarchy operations on tessellation hardware.

Description of the Related Art

Ray tracing is a graphics rendering technique that is becoming increasingly popular in a number of applications and contexts. For example, ray tracing may be used to compute a global illumination solution for a graphics scene. In general, ray tracing may be used in computer graphics to determine visibility by directing one or more rays from a vantage point described by the ray's position vector along a line of sight described by the ray's direction vector. To determine the nearest visible surface along that line of sight requires that the ray to be effectively tested for intersection against all of the geometry within the virtual scene and retain the nearest intersection.

However, determining the intersections of each ray with the geometries of a scene may be computationally complex and resource intensive, thereby limiting a real-time ray tracing solution.

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

FIG. **1** is a block diagram of an embodiment of a computer system with a processor having one or more processor cores and graphics processors;

FIG. **2** is a block diagram of one embodiment of a processor having one or more processor cores, an integrated memory controller, and an integrated graphics processor;

FIG. **3** is a block diagram of one embodiment of a graphics processor which may be a discreet graphics processing unit, or may be graphics processor integrated with a plurality of processing cores;

FIG. **4** is a block diagram of an embodiment of a graphics-processing engine for a graphics processor;

FIG. **5** is a block diagram of another embodiment of a graphics processor;

FIG. **6** is a block diagram of thread execution logic including an array of processing elements;

FIG. **7** illustrates a graphics processor execution unit instruction format according to an embodiment;

FIG. **8** is a block diagram of another embodiment of a graphics processor which includes a graphics pipeline, a media pipeline, a display engine, thread execution logic, and a render output pipeline;

FIG. **9**A is a block diagram illustrating a graphics processor command format according to an embodiment;

FIG. **9**B is a block diagram illustrating a graphics processor command sequence according to an embodiment;

FIG. **10** illustrates exemplary graphics software architecture for a data processing system according to an embodiment;

FIG. **11** illustrates an exemplary IP core development system that may be used to manufacture an integrated circuit to perform operations according to an embodiment;

FIG. **12** illustrates an exemplary system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment;

FIG. **13** illustrates an exemplary graphics processor of a system on a chip integrated circuit that may be fabricated using one or more IP cores;

FIG. **14** illustrates an additional exemplary graphics processor of a system on a chip integrated circuit that may be fabricated using one or more IP cores

FIG. **15** illustrates one embodiment in which a tessellation unit generates a bounding volume hierarchy (BVH);

FIG. **16** illustrates a mapping of an exemplary set of primitives and a concurrently-generated BVH;

FIG. **17** illustrates one embodiment which uses the BVH to test the input surface and its primitives for intersection with a set of N rays before tessellation;

FIG. **18** illustrates a method in accordance with one embodiment of the invention;

FIG. **19** is a block diagram illustrating a computer system configured to implement one or more aspects of the embodiments described herein;

FIG. **20**A-**20**D illustrate a parallel processor components, according to an embodiment;

FIGS. **21**A-**21**B are block diagrams of graphics multiprocessors, according to embodiments;

FIG. **22**A-**22**F illustrate an exemplary architecture in which a plurality of GPUs are communicatively coupled to a plurality of multi-core processors; and

FIG. **23** illustrates a graphics processing pipeline, according to an embodiment.

DETAILED DESCRIPTION

In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the embodiments of the invention described below. It will be apparent, however, to one skilled in the art that the embodiments of the invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form to avoid obscuring the underlying principles of the embodiments of the invention.

Exemplary Graphics Processor Architectures and Data Types

System Overview

FIG. **1** is a block diagram of a processing system **100**, according to an embodiment. In various embodiments the system **100** includes one or more processors **102** and one or more graphics processors **108**, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors **102** or processor cores **107**. In one embodiment, the system **100** is a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices.

An embodiment of system **100** can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In some embodiments system **100** is a mobile phone, smart phone, tablet computing device or mobile Internet device. Data processing system **100** can also include, couple with, or be integrated within a wearable device, such as a smart watch wearable device, smart eyewear device, augmented reality device, or virtual reality device. In some embodiments, data processing system **100** is a television or set top box device having one or more processors **102** and a graphical interface generated by one or more graphics processors **108**.

In some embodiments, the one or more processors **102** each include one or more processor cores **107** to process instructions which, when executed, perform operations for system and user software. In some embodiments, each of the one or more processor cores **107** is configured to process a specific instruction set **109**. In some embodiments, instruction set **109** may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a Very Long Instruction Word (VLIW). Multiple processor cores **107** may each process a different instruction set **109**, which may include instructions to facilitate the emulation of other instruction sets. Processor core **107** may also include other processing devices, such a Digital Signal Processor (DSP).

In some embodiments, the processor **102** includes cache memory **104**. Depending on the architecture, the processor **102** can have a single internal cache or multiple levels of internal cache. In some embodiments, the cache memory is shared among various components of the processor **102**. In some embodiments, the processor **102** also uses an external cache (e.g., a Level-3 (L3) cache or Last Level Cache (LLC)) (not shown), which may be shared among processor cores **107** using known cache coherency techniques. A register file **106** is additionally included in processor **102** which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor **102**.

In some embodiments, processor **102** is coupled with a processor bus **110** to transmit communication signals such as address, data, or control signals between processor **102** and other components in system **100**. In one embodiment the system **100** uses an exemplary 'hub' system architecture, including a memory controller hub **116** and an Input Output (I/O) controller hub **130**. A memory controller hub **116** facilitates communication between a memory device and other components of system **100**, while an I/O Controller Hub (ICH) **130** provides connections to I/O devices via a local I/O bus. In one embodiment, the logic of the memory controller hub **116** is integrated within the processor.

Memory device **120** can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. In one embodiment the memory device **120** can operate as system memory for the system **100**, to store data **122** and instructions **121** for use when the one or more processors **102** executes an application or process. Memory controller hub **116** also couples with an optional external graphics processor **112**, which may communicate with the one or more graphics processors **108** in processors **102** to perform graphics and media operations.

In some embodiments, ICH **130** enables peripherals to connect to memory device **120** and processor **102** via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller **146**, a firmware interface **128**, a wireless transceiver **126** (e.g., Wi-Fi, Bluetooth), a data storage device **124** (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller **140** for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers **142** connect input devices, such as keyboard and mouse **144** combinations. A network controller **134** may also couple with ICH **130**. In some embodiments, a high-performance network controller (not shown) couples with processor bus **110**. It will be appreciated that the system **100** shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, the I/O controller hub **130** may be integrated within the one or more processor **102**, or the memory controller hub **116** and I/O controller hub **130** may be integrated into a discreet external graphics processor, such as the external graphics processor **112**.

FIG. **2** is a block diagram of an embodiment of a processor **200** having one or more processor cores **202**A-**202**N, an integrated memory controller **214**, and an integrated graphics processor **208**. Those elements of FIG. **2** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. Processor **200** can include additional cores up to and including additional core **202**N represented by the dashed lined boxes. Each of processor cores **202**A-**202**N includes one or more internal cache units **204**A-**204**N. In some embodiments each processor core also has access to one or more shared cached units **206**.

The internal cache units **204**A-**204**N and shared cache units **206** represent a cache memory hierarchy within the processor **200**. The cache memory hierarchy may include at least one level of instruction and data cache within each processor core and one or more levels of shared mid-level cache, such as a Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the LLC. In some embodiments, cache coherency logic maintains coherency between the various cache units **206** and **204**A-**204**N.

In some embodiments, processor **200** may also include a set of one or more bus controller units **216** and a system agent core **210**. The one or more bus controller units **216** manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). System agent core **210** provides management functionality for the various processor components. In some embodiments, system agent core **210** includes one or more integrated memory controllers **214** to manage access to various external memory devices (not shown).

In some embodiments, one or more of the processor cores **202**A-**202**N include support for simultaneous multi-threading. In such embodiment, the system agent core **210** includes components for coordinating and operating cores **202**A-**202**N during multi-threaded processing. System agent core **210** may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of processor cores **202**A-**202**N and graphics processor **208**.

In some embodiments, processor **200** additionally includes graphics processor **208** to execute graphics processing operations. In some embodiments, the graphics processor **208** couples with the set of shared cache units **206**, and the system agent core **210**, including the one or more integrated memory controllers **214**. In some embodiments, a display controller **211** is coupled with the graphics processor **208** to drive graphics processor output to one or more coupled displays. In some embodiments, display controller

211 may be a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor **208** or system agent core **210**.

In some embodiments, a ring based interconnect unit **212** is used to couple the internal components of the processor **200**. However, an alternative interconnect unit may be used, such as a point-to-point interconnect, a switched interconnect, or other techniques, including techniques well known in the art. In some embodiments, graphics processor **208** couples with the ring interconnect **212** via an I/O link **213**.

The exemplary I/O link **213** represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module **218**, such as an eDRAM module. In some embodiments, each of the processor cores **202**A-**202**N and graphics processor **208** use embedded memory modules **218** as a shared Last Level Cache.

In some embodiments, processor cores **202**A-**202**N are homogenous cores executing the same instruction set architecture. In another embodiment, processor cores **202**A-**202**N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores **202**A-**202**N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set. In one embodiment processor cores **202**A-**202**N are heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. Additionally, processor **200** can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.

FIG. **3** is a block diagram of a graphics processor **300**, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores. In some embodiments, the graphics processor communicates via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor memory. In some embodiments, graphics processor **300** includes a memory interface **314** to access memory. Memory interface **314** can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

In some embodiments, graphics processor **300** also includes a display controller **302** to drive display output data to a display device **320**. Display controller **302** includes hardware for one or more overlay planes for the display and composition of multiple layers of video or user interface elements. In some embodiments, graphics processor **300** includes a video codec engine **306** to encode, decode, or transcode media to, from, or between one or more media encoding formats, including, but not limited to Moving Picture Experts Group (MPEG) formats such as MPEG-2, Advanced Video Coding (AVC) formats such as H.264/MPEG-4 AVC, as well as the Society of Motion Picture & Television Engineers (SMPTE) 421M/VC-1, and Joint Photographic Experts Group (JPEG) formats such as JPEG, and Motion JPEG (MJPEG) formats.

In some embodiments, graphics processor **300** includes a block image transfer (BLIT) engine **304** to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations are performed using one or more components of graphics processing engine (GPE) **310**. In some embodiments, GPE **310** is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

In some embodiments, GPE **310** includes a 3D pipeline **312** for performing 3D operations, such as rendering three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline **312** includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media sub-system **315**. While 3D pipeline **312** can be used to perform media operations, an embodiment of GPE **310** also includes a media pipeline **316** that is specifically used to perform media operations, such as video post-processing and image enhancement.

In some embodiments, media pipeline **316** includes fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of video codec engine **306**. In some embodiments, media pipeline **316** additionally includes a thread spawning unit to spawn threads for execution on 3D/Media sub-system **315**. The spawned threads perform computations for the media operations on one or more graphics execution units included in 3D/Media sub-system **315**.

In some embodiments, 3D/Media subsystem **315** includes logic for executing threads spawned by 3D pipeline **312** and media pipeline **316**. In one embodiment, the pipelines send thread execution requests to 3D/Media subsystem **315**, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. In some embodiments, 3D/Media subsystem **315** includes one or more internal caches for thread instructions and data. In some embodiments, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

Graphics Processing Engine

FIG. **4** is a block diagram of a graphics processing engine **410** of a graphics processor in accordance with some embodiments. In one embodiment, the graphics processing engine (GPE) **410** is a version of the GPE **310** shown in FIG. **3**. Elements of FIG. **4** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. For example, the 3D pipeline **312** and media pipeline **316** of FIG. **3** are illustrated. The media pipeline **316** is optional in some embodiments of the GPE **410** and may not be explicitly included within the GPE **410**. For example and in at least one embodiment, a separate media and/or image processor is coupled to the GPE **410**.

In some embodiments, GPE **410** couples with or includes a command streamer **403**, which provides a command stream to the 3D pipeline **312** and/or media pipelines **316**. In some embodiments, command streamer **403** is coupled with memory, which can be system memory, or one or more of internal cache memory and shared cache memory. In some embodiments, command streamer **403** receives commands from the memory and sends the commands to 3D pipeline **312** and/or media pipeline **316**. The commands are directives fetched from a ring buffer, which stores commands for the 3D pipeline **312** and media pipeline **316**. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The commands for the 3D pipeline **312** can also include references to data stored in memory, such as but not limited to vertex and geometry data for the 3D pipeline **312** and/or image data and memory objects for the media pipeline **316**. The 3D pipeline **312** and media pipeline **316** process the commands and data by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to a graphics core array **414**.

In various embodiments the 3D pipeline **312** can execute one or more shader programs, such as vertex shaders, geometry shaders, pixel shaders, fragment shaders, compute shaders, or other shader programs, by processing the instructions and dispatching execution threads to the graphics core array **414**. The graphics core array **414** provides a unified block of execution resources. Multi-purpose execution logic (e.g., execution units) within the graphic core array **414** includes support for various 3D API shader languages and can execute multiple simultaneous execution threads associated with multiple shaders.

In some embodiments the graphics core array **414** also includes execution logic to perform media functions, such as video and/or image processing. In one embodiment, the execution units additionally include general-purpose logic that is programmable to perform parallel general purpose computational operations, in addition to graphics processing operations. The general purpose logic can perform processing operations in parallel or in conjunction with general purpose logic within the processor core(s) **107** of FIG. **1** or core **202**A-**202**N as in FIG. **2**.

Output data generated by threads executing on the graphics core array **414** can output data to memory in a unified return buffer (URB) **418**. The URB **418** can store data for multiple threads. In some embodiments the URB **418** may be used to send data between different threads executing on the graphics core array **414**. In some embodiments the URB **418** may additionally be used for synchronization between threads on the graphics core array and fixed function logic within the shared function logic **420**.

In some embodiments, graphics core array **414** is scalable, such that the array includes a variable number of graphics cores, each having a variable number of execution units based on the target power and performance level of GPE **410**. In one embodiment the execution resources are dynamically scalable, such that execution resources may be enabled or disabled as needed.

The graphics core array **414** couples with shared function logic **420** that includes multiple resources that are shared between the graphics cores in the graphics core array. The shared functions within the shared function logic **420** are hardware logic units that provide specialized supplemental functionality to the graphics core array **414**. In various embodiments, shared function logic **420** includes but is not limited to sampler **421**, math **422**, and inter-thread communication (ITC) **423** logic. Additionally, some embodiments implement one or more cache(s) **425** within the shared function logic **420**. A shared function is implemented where the demand for a given specialized function is insufficient for inclusion within the graphics core array **414**. Instead a single instantiation of that specialized function is implemented as a stand-alone entity in the shared function logic **420** and shared among the execution resources within the graphics core array **414**. The precise set of functions that are shared between the graphics core array **414** and included within the graphics core array **414** varies between embodiments.

FIG. **5** is a block diagram of another embodiment of a graphics processor **500**. Elements of FIG. **5** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

In some embodiments, graphics processor **500** includes a ring interconnect **502**, a pipeline front-end **504**, a media engine **537**, and graphics cores **580**A-**580**N. In some embodiments, ring interconnect **502** couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In some embodiments, the graphics processor is one of many processors integrated within a multi-core processing system.

In some embodiments, graphics processor **500** receives batches of commands via ring interconnect **502**. The incoming commands are interpreted by a command streamer **503** in the pipeline front-end **504**. In some embodiments, graphics processor **500** includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) **580**A-**580**N. For 3D geometry processing commands, command streamer **503** supplies commands to geometry pipeline **536**. For at least some media processing commands, command streamer **503** supplies the commands to a video front end **534**, which couples with a media engine **537**. In some embodiments, media engine **537** includes a Video Quality Engine (VQE) **530** for video and image post-processing and a multi-format encode/decode (MFX) **533** engine to provide hardware-accelerated media data encode and decode. In some embodiments, geometry pipeline **536** and media engine **537** each generate execution threads for the thread execution resources provided by at least one graphics core **580**A.

In some embodiments, graphics processor **500** includes scalable thread execution resources featuring modular cores **580**A-**580**N (sometimes referred to as core slices), each having multiple sub-cores **550**A-**550**N, **560**A-**560**N (sometimes referred to as core sub-slices). In some embodiments, graphics processor **500** can have any number of graphics cores **580**A through **580**N. In some embodiments, graphics processor **500** includes a graphics core **580**A having at least a first sub-core **550**A and a second sub-core **560**A. In other embodiments, the graphics processor is a low power processor with a single sub-core (e.g., **550**A). In some embodiments, graphics processor **500** includes multiple graphics cores **580**A-**580**N, each including a set of first sub-cores **550**A-**550**N and a set of second sub-cores **560**A-**560**N. Each sub-core in the set of first sub-cores **550**A-**550**N includes at least a first set of execution units **552**A-**552**N and media/texture samplers **554**A-**554**N. Each sub-core in the set of second sub-cores **560**A-**560**N includes at least a second set of execution units **562**A-**562**N and samplers **564**A-**564**N. In some embodiments, each sub-core **550**A-**550**N, **560**A-**560**N shares a set of shared resources **570**A-**570**N. In some embodiments, the shared resources include shared cache memory and pixel operation logic. Other shared resources may also be included in the various embodiments of the graphics processor.

Execution Units

FIG. **6** illustrates thread execution logic **600** including an array of processing elements employed in some embodiments of a GPE. Elements of FIG. **6** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

In some embodiments, thread execution logic **600** includes a shader processor **602**, a thread dispatcher **604**, instruction cache **606**, a scalable execution unit array including a plurality of execution units **608**A-**608**N, a sampler **610**, a data cache **612**, and a data port **614**. In one embodiment the scalable execution unit array can dynamically scale by enabling or disabling one or more execution units (e.g., any of execution unit **608**A, **608**B, **608**C, **608**D, through **608**N−1 and **608**N) based on the computational requirements of a workload. In one embodiment the included components are interconnected via an interconnect fabric that links to each of the components. In some embodiments, thread execution logic **600** includes one or more connections to memory, such as system memory or cache memory, through one or more of instruction cache **606**, data port **614**, sampler **610**, and execution units **608**A-**608**N. In some embodiments, each execution unit (e.g. **608**A) is a stand-alone programmable general purpose computational unit that is capable of executing multiple simultaneous hardware threads while processing multiple data elements in parallel for each thread. In various embodiments, the array of execution units **608**A-**608**N is scalable to include any number individual execution units.

In some embodiments, the execution units **608**A-**608**N are primarily used to execute shader programs. A shader processor **602** can process the various shader programs and dispatch execution threads associated with the shader programs via a thread dispatcher **604**. In one embodiment the thread dispatcher includes logic to arbitrate thread initiation requests from the graphics and media pipelines and instantiate the requested threads on one or more execution unit in the execution units **608**A-**608**N. For example, the geometry pipeline (e.g., **536** of FIG. **5**) can dispatch vertex, tessellation, or geometry shaders to the thread execution logic **600** (FIG. **6**) for processing. In some embodiments, thread dispatcher **604** can also process runtime thread spawning requests from the executing shader programs.

In some embodiments, the execution units **608**A-**608**N support an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders). Each of the execution units **608**A-**608**N is capable of multi-issue single instruction multiple data (SIMD) execution and multi-threaded operation enables an efficient execution environment in the face of higher latency memory accesses. Each hardware thread within each execution unit has a dedicated high-bandwidth register file and associated independent thread-state. Execution is multi-issue per clock to pipelines capable of integer, single and double precision floating point operations, SIMD branch capability, logical operations, transcendental operations, and other miscellaneous operations. While waiting for data from memory or one of the shared functions, dependency logic within the execution units **608**A-**608**N causes a waiting thread to sleep until the requested data has been returned. While the waiting thread is sleeping, hardware resources may be devoted to processing other threads. For example, during a delay associated with a vertex shader operation, an execution unit can perform operations for a pixel shader, fragment shader, or another type of shader program, including a different vertex shader.

Each execution unit in execution units **608**A-**608**N operates on arrays of data elements. The number of data elements is the "execution size," or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical Arithmetic Logic Units (ALUs) or Floating Point Units (FPUs) for a particular graphics processor. In some embodiments, execution units **608**A-**608**N support integer and floating-point data types.

The execution unit instruction set includes SIMD instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (Quad-Word (QW) size data elements), eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

One or more internal instruction caches (e.g., **606**) are included in the thread execution logic **600** to cache thread instructions for the execution units. In some embodiments, one or more data caches (e.g., **612**) are included to cache thread data during thread execution. In some embodiments, a sampler **610** is included to provide texture sampling for 3D operations and media sampling for media operations. In some embodiments, sampler **610** includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

During execution, the graphics and media pipelines send thread initiation requests to thread execution logic **600** via thread spawning and dispatch logic. Once a group of geometric objects has been processed and rasterized into pixel data, pixel processor logic (e.g., pixel shader logic, fragment shader logic, etc.) within the shader processor **602** is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In some embodiments, a pixel shader or fragment shader calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. In some embodiments, pixel processor logic within the shader processor **602** then executes an application programming interface (API)-supplied pixel or fragment shader program. To execute the shader program, the shader processor **602** dispatches threads to an execution unit (e.g., **608**A) via thread dispatcher **604**. In some embodiments, pixel shader **602** uses texture sampling logic in the sampler **610** to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

In some embodiments, the data port **614** provides a memory access mechanism for the thread execution logic **600** output processed data to memory for processing on a graphics processor output pipeline. In some embodiments, the data port **614** includes or couples to one or more cache memories (e.g., data cache **612**) to cache data for memory access via the data port.

FIG. **7** is a block diagram illustrating a graphics processor instruction formats **700** according to some embodiments. In one or more embodiment, the graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. In some embodiments, instruction format **700** described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

In some embodiments, the graphics processor execution units natively support instructions in a 128-bit instruction format **710**. A 64-bit compacted instruction format **730** is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit instruction format **710** provides access to all instruction options, while some options and operations are restricted in the 64-bit instruction format **730**. The native instructions available in the 64-bit instruction format **730** vary by embodiment. In some embodiments, the instruction is compacted in part using a set of index values in an index field **713**. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit instruction format **710**.

For each format, instruction opcode **712** defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. In some embodiments, instruction control field **714** enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For instructions in the 128-bit instruction format **710** an exec-size field **716** limits the number of data channels that will be executed in parallel. In some embodiments, exec-size field **716** is not available for use in the 64-bit compact instruction format **730**.

Some execution unit instructions have up to three operands including two source operands, src0 **720**, src1 **722**, and one destination **718**. In some embodiments, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 **724**), where the instruction opcode **712** determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

In some embodiments, the 128-bit instruction format **710** includes an access/address mode field **726** specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction.

In some embodiments, the 128-bit instruction format **710** includes an access/address mode field **726**, which specifies an address mode and/or an access mode for the instruction. In one embodiment the access mode is used to define a data access alignment for the instruction. Some embodiments support access modes including a 16-byte aligned access mode and a 1-byte aligned access mode, where the byte alignment of the access mode determines the access alignment of the instruction operands. For example, when in a first mode, the instruction may use byte-aligned addressing for source and destination operands and when in a second mode, the instruction may use 16-byte-aligned addressing for all source and destination operands.

In one embodiment, the address mode portion of the access/address mode field **726** determines whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed based on an address register value and an address immediate field in the instruction.

In some embodiments instructions are grouped based on opcode **712** bit-fields to simplify Opcode decode **740**. For an 8-bit opcode, bits **4**, **5**, and **6** allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group **742** includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group **742** shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group **744** (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group **746** includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group **748** includes component-wise arithmetic instructions (e.g., add, multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group **748** performs the arithmetic operations in parallel across data channels. The vector math group **750** includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

Graphics Pipeline

FIG. **8** is a block diagram of another embodiment of a graphics processor **800**. Elements of FIG. **8** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

In some embodiments, graphics processor **800** includes a graphics pipeline **820**, a media pipeline **830**, a display engine **840**, thread execution logic **850**, and a render output pipeline **870**. In some embodiments, graphics processor **800** is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor **800** via a ring interconnect **802**. In some embodiments, ring interconnect **802** couples graphics processor **800** to other processing components, such as other graphics processors or general-purpose processors. Commands from ring interconnect **802** are interpreted by a command streamer **803**, which supplies instructions to individual components of graphics pipeline **820** or media pipeline **830**.

In some embodiments, command streamer **803** directs the operation of a vertex fetcher **805** that reads vertex data from memory and executes vertex-processing commands provided by command streamer **803**. In some embodiments, vertex fetcher **805** provides vertex data to a vertex shader **807**, which performs coordinate space transformation and lighting operations to each vertex. In some embodiments, vertex fetcher **805** and vertex shader **807** execute vertex-processing instructions by dispatching execution threads to execution units **852**A-**852**B via a thread dispatcher **831**.

In some embodiments, execution units **852**A-**852**B are an array of vector processors having an instruction set for performing graphics and media operations. In some embodiments, execution units **852**A-**852**B have an attached L1 cache **851** that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

In some embodiments, graphics pipeline **820** includes tessellation components to perform hardware-accelerated tessellation of 3D objects. In some embodiments, a programmable hull shader **811** configures the tessellation operations. A programmable domain shader **817** provides back-end evaluation of tessellation output. A tessellator **813** operates at the direction of hull shader **811** and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to graphics pipeline **820**. In some embodiments, if tessellation is not used, tessellation components (e.g., hull shader **811**, tessellator **813**, and domain shader **817**) can be bypassed.

In some embodiments, complete geometric objects can be processed by a geometry shader **819** via one or more threads dispatched to execution units **852**A-**852**B, or can proceed directly to the clipper **829**. In some embodiments, the geometry shader operates on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader **819** receives input from the vertex shader **807**. In some embodiments, geometry shader **819** is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

Before rasterization, a clipper **829** processes vertex data. The clipper **829** may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In some embodiments, a rasterizer and depth test component **873** in the render output pipeline **870** dispatches pixel shaders to convert the geometric objects into their per pixel representations. In some embodiments, pixel shader logic is included in thread execution logic **850**. In some embodiments, an application can bypass the rasterizer and depth test component **873** and access un-rasterized vertex data via a stream out unit **823**.

The graphics processor **800** has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units **852**A-**852**B and associated cache(s) **851**, texture and media sampler **854**, and texture/sampler cache **858** interconnect via a data port **856** to perform memory access and communicate with render output pipeline components of the processor. In some embodiments, sampler **854**, caches **851**, **858** and execution units **852**A-**852**B each have separate memory access paths.

In some embodiments, render output pipeline **870** contains a rasterizer and depth test component **873** that converts vertex-based objects into an associated pixel-based representation. In some embodiments, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache **878** and depth cache **879** are also available in some embodiments. A pixel operations component **877** performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine **841**, or substituted at display time by the display controller **843** using overlay display planes. In some embodiments, a shared L3 cache **875** is available to all graphics components, allowing the sharing of data without the use of main system memory.

In some embodiments, graphics processor media pipeline **830** includes a media engine **837** and a video front end **834**. In some embodiments, video front end **834** receives pipeline commands from the command streamer **803**. In some embodiments, media pipeline **830** includes a separate command streamer. In some embodiments, video front-end **834** processes media commands before sending the command to the media engine **837**. In some embodiments, media engine **837** includes thread spawning functionality to spawn threads for dispatch to thread execution logic **850** via thread dispatcher **831**.

In some embodiments, graphics processor **800** includes a display engine **840**. In some embodiments, display engine **840** is external to processor **800** and couples with the graphics processor via the ring interconnect **802**, or some other interconnect bus or fabric. In some embodiments, display engine **840** includes a 2D engine **841** and a display controller **843**. In some embodiments, display engine **840** contains special purpose logic capable of operating independently of the 3D pipeline. In some embodiments, display controller **843** couples with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

In some embodiments, graphics pipeline **820** and media pipeline **830** are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL), Open Computing Language (OpenCL), and/or Vulkan graphics and compute API, all from the Khronos Group. In some embodiments, support may also be provided for the Direct3D library from the Microsoft Corporation. In some embodiments, a combination of these libraries may be supported. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

Graphics Pipeline Programming

FIG. **9**A is a block diagram illustrating a graphics processor command format **900** according to some embodiments. FIG. **9**B is a block diagram illustrating a graphics processor command sequence **910** according to an embodiment. The solid lined boxes in FIG. **9**A illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format **900** of FIG. **9**A includes data fields to identify a target client **902** of the command, a command operation code (opcode) **904**, and the relevant data **906** for the command. A sub-opcode **905** and a command size **908** are also included in some commands.

In some embodiments, client **902** specifies the client unit of the graphics device that processes the command data. In some embodiments, a graphics processor command parser examines the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. In some embodiments, the graphics processor client units include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit has a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads the opcode **904** and, if present, sub-opcode **905** to determine the operation to perform. The client unit performs the command using information in data field **906**. For some commands an explicit command size **908** is expected to specify the size of the command. In some embodiments, the command parser automatically determines the size of at least some of the commands based on the command opcode. In some embodiments commands are aligned via multiples of a double word.

The flow diagram in FIG. **9**B shows an exemplary graphics processor command sequence **910**. In some embodiments, software or firmware of a data processing system that features an embodiment of a graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only as embodiments are not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.

In some embodiments, the graphics processor command sequence **910** may begin with a pipeline flush command **912** to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In some embodiments, the 3D pipeline **922** and the media pipeline **924** do not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. In some embodiments, pipeline flush command **912** can be used for pipeline synchronization or before placing the graphics processor into a low power state.

In some embodiments, a pipeline select command **913** is used when a command sequence requires the graphics processor to explicitly switch between pipelines. In some embodiments, a pipeline select command **913** is required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. In some embodiments, a pipeline flush command **912** is required immediately before a pipeline switch via the pipeline select command **913**.

In some embodiments, a pipeline control command **914** configures a graphics pipeline for operation and is used to program the 3D pipeline **922** and the media pipeline **924**. In some embodiments, pipeline control command **914** configures the pipeline state for the active pipeline. In one embodiment, the pipeline control command **914** is used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.

In some embodiments, commands for the return buffer state **916** are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write intermediate data during processing. In some embodiments, the graphics processor also uses one or more return buffers to store output data and to perform cross thread communication. In some embodiments, configuring the return buffer state **916** includes selecting the size and number of return buffers to use for a set of pipeline operations.

The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination **920**, the command sequence is tailored to the 3D pipeline **922** beginning with the 3D pipeline state **930** or the media pipeline **924** beginning at the media pipeline state **940**.

The commands to configure the 3D pipeline state **930** include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based on the particular 3D API in use. In some embodiments, 3D pipeline state **930** commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

In some embodiments, 3D primitive **932** command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive **932** command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive **932** command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. In some embodiments, 3D primitive **932** command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, 3D pipeline **922** dispatches shader execution threads to graphics processor execution units.

In some embodiments, 3D pipeline **922** is triggered via an execute **934** command or event. In some embodiments, a register write triggers command execution. In some embodiments execution is triggered via a 'go' or 'kick' command in the command sequence. In one embodiment, command execution is triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

In some embodiments, the graphics processor command sequence **910** follows the media pipeline **924** path when performing media operations. In general, the specific use and manner of programming for the media pipeline **924** depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. In some embodiments, the media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general purpose processing cores. In one embodiment, the media pipeline also includes elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

In some embodiments, media pipeline **924** is configured in a similar manner as the 3D pipeline **922**. A set of commands to configure the media pipeline state **940** are dispatched or placed into a command queue before the media object commands **942**. In some embodiments, commands for the media pipeline state **940** include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. In some embodiments, commands for the media pipeline state **940** also support the use of one or more pointers to "indirect" state elements that contain a batch of state settings.

In some embodiments, media object commands **942** supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. In some embodiments, all media pipeline states must be valid before issuing a media object command **942**. Once the pipeline state is configured and media object commands **942** are queued, the media pipeline **924** is triggered via an execute command **944** or an equivalent execute event (e.g., register write). Output from media pipeline **924** may then be post processed by operations provided by the 3D pipeline **922** or the media pipeline **924**. In some embodiments, GPGPU operations are configured and executed in a similar manner as media operations.

Graphics Software Architecture

FIG. **10** illustrates exemplary graphics software architecture for a data processing system **1000** according to some embodiments. In some embodiments, software architecture includes a 3D graphics application **1010**, an operating system **1020**, and at least one processor **1030**. In some embodiments, processor **1030** includes a graphics processor **1032** and one or more general-purpose processor core(s) **1034**. The graphics application **1010** and operating system **1020** each execute in the system memory **1050** of the data processing system.

In some embodiments, 3D graphics application **1010** contains one or more shader programs including shader instructions **1012**. The shader language instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL Shader Language (GLSL). The application also includes executable instructions **1014** in a machine language suitable for execution by the general-purpose processor core **1034**. The application also includes graphics objects **1016** defined by vertex data.

In some embodiments, operating system **1020** is a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. The operating system **1020** can support a graphics API **1022** such as the Direct3D API, the OpenGL API, or the Vulkan API. When the Direct3D API is in use, the operating system **1020** uses a front-end shader compiler **1024** to compile any shader instructions **1012** in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. In some embodiments, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application **1010**. In some embodiments, the shader instructions **1012** are provided in an intermediate form, such as a version of the Standard Portable Intermediate Representation (SPIR) used by the Vulkan API.

In some embodiments, user mode graphics driver **1026** contains a back-end shader compiler **1027** to convert the shader instructions **1012** into a hardware specific representation. When the OpenGL API is in use, shader instructions **1012** in the GLSL high-level language are passed to a user mode graphics driver **1026** for compilation. In some embodiments, user mode graphics driver **1026** uses operating system kernel mode functions **1028** to communicate with a kernel mode graphics driver **1029**. In some embodiments, kernel mode graphics driver **1029** communicates with graphics processor **1032** to dispatch commands and instructions.

IP Core Implementations

One or more aspects of at least one embodiment may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as "IP cores," are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model may be supplied to various customers or manufacturing facilities, which load the hardware model on fabrication machines that manufacture the integrated circuit. The integrated circuit may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

FIG. **11** is a block diagram illustrating an IP core development system **1100** that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system **1100** may be used to generate modular, re-usable designs that can be incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility **1130** can generate a software simulation **1110** of an IP core design in a high level programming language (e.g., C/C++). The software simulation **1110** can be used to design, test, and verify the behavior of the IP core using a simulation model **1112**. The simulation model **1112** may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design **1115** can then be created or synthesized from the simulation model **1112**. The RTL design **1115** is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design **1115**, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

The RTL design **1115** or equivalent may be further synthesized by the design facility into a hardware model **1120**, which may be in a hardware description language (HDL), or some other representation of physical design data. The HDL may be further simulated or tested to verify the IP core design. The IP core design can be stored for delivery to a 3rd party fabrication facility **1165** using non-volatile memory **1140** (e.g., hard disk, flash memory, or any non-volatile storage medium). Alternatively, the IP core design may be transmitted (e.g., via the Internet) over a wired connection **1150** or wireless connection **1160**. The fabrication facility **1165** may then fabricate an integrated circuit that is based at least in part on the IP core design. The fabricated integrated circuit can be configured to perform operations in accordance with at least one embodiment described herein.

Exemplary System on a Chip Integrated Circuit

FIGS. **12-14** illustrate exemplary integrated circuits and associated graphics processors that may be fabricated using one or more IP cores, according to various embodiments described herein. In addition to what is illustrated, other logic and circuits may be included, including additional graphics processors/cores, peripheral interface controllers, or general purpose processor cores.

FIG. **12** is a block diagram illustrating an exemplary system on a chip integrated circuit **1200** that may be fabricated using one or more IP cores, according to an embodiment. Exemplary integrated circuit **1200** includes one or more application processor(s) **1205** (e.g., CPUs), at least one graphics processor **1210**, and may additionally include an image processor **1215** and/or a video processor **1220**, any of which may be a modular IP core from the same or multiple different design facilities. Integrated circuit **1200** includes peripheral or bus logic including a USB controller **1225**, UART controller **1230**, an SPI/SDIO controller **1235**, and an I2S/I2C controller **1240**. Additionally, the integrated circuit can include a display device **1245** coupled to one or more of a high-definition multimedia interface (HDMI) controller **1250** and a mobile industry processor interface (MIPI) display interface **1255**. Storage may be provided by a flash memory subsystem **1260** including flash memory and a flash memory controller. Memory interface may be provided via a memory controller **1265** for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine **1270**.

FIG. **13** is a block diagram illustrating an exemplary graphics processor **1310** of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor **1310** can be a variant of the graphics processor **1210** of FIG. **12**. Graphics processor **1310** includes a vertex processor **1305** and one or more fragment processor(s) **1315**A **1315**N (e.g., **1315**A, **1315**B, **1315**C, **1315**D, through **1315**N−1, and **1315**N). Graphics processor **1310** can execute different shader programs via separate logic, such that the vertex processor **1305** is optimized to execute operations for vertex shader programs, while the one or more fragment processor(s) **1315**A-**1315**N execute fragment (e.g., pixel) shading operations for fragment or pixel shader programs. The vertex processor **1305** performs the vertex processing stage of the 3D graphics pipeline and generates primitives and vertex data. The fragment processor(s) **1315**A-**1315**N use the primitive and vertex data generated by the vertex processor **1305** to produce a framebuffer that is displayed on a display device. In one embodiment, the fragment processor(s) **1315**A-**1315**N are optimized to execute fragment shader programs as provided for in the OpenGL API, which may be used to perform similar operations as a pixel shader program as provided for in the Direct 3D API.

Graphics processor **1310** additionally includes one or more memory management units (MMUs) **1320**A-**1320**B, cache(s) **1325**A-**1325**B, and circuit interconnect(s) **1330**A-**1330**B. The one or more MMU(s) **1320**A-**1320**B provide for virtual to physical address mapping for graphics processor **1310**, including for the vertex processor **1305** and/or fragment processor(s) **1315**A-**1315**N, which may reference vertex or image/texture data stored in memory, in addition to vertex or image/texture data stored in the one or more cache(s) **1325**A-**1325**B. In one embodiment the one or more MMU(s) **1320**A-**1320**B may be synchronized with other MMUs within the system, including one or more MMUs associated with the one or more application processor(s) **1205**, image processor **1215**, and/or video processor **1220** of FIG. **12**, such that each processor **1205**-**1220** can participate in a shared or unified virtual memory system. The one or more circuit interconnect(s) **1330**A-**1330**B enable graphics processor **1310** to interface with other IP cores within the SoC, either via an internal bus of the SoC or via a direct connection, according to embodiments.

FIG. **14** is a block diagram illustrating an additional exemplary graphics processor **1410** of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor **1410** can be a variant of the graphics processor **1210** of FIG. **12**. Graphics processor **1410** includes the one or more MMU(s) **1320**A-**1320**B, cache(s) **1325**A-**1325**B, and circuit interconnect(s) **1330**A-**1330**B of the integrated circuit **1300** of FIG. **13**.

Graphics processor **1410** includes one or more shader core(s) **1415**A-**1415**N (e.g., **1415**A, **1415**B, **1415**C, **1415**D, **1415**E, **1415**F, through **1315**N−1, and **1315**N), which provides for a unified shader core architecture in which a single core or type or core can execute all types of programmable shader code, including shader program code to implement vertex shaders, fragment shaders, and/or compute shaders. The exact number of shader cores present can vary among embodiments and implementations. Additionally, graphics processor **1410** includes an inter-core task manager **1405**, which acts as a thread dispatcher to dispatch execution threads to one or more shader core(s) **1415**A-**1415**N and a tiling unit **1418** to accelerate tiling operations for tile-based rendering, in which rendering operations for a scene are subdivided in image space, for example to exploit local spatial coherence within a scene or to optimize use of internal caches.

Apparatus and Method for Implementing Bounding Volume Hierarchy Operations on Tesselation Hardware

Graphics tessellation units include hardware for subdividing a surface into a plurality of smaller surfaces. In particular, during tessellation, each surface is subdivided into additional primitives (triangles, lines, vertices). The number of primitives into which a surface is to be subdivided is specified by a set of tessellation factors (typically provided by a Hull Shader). The end result is that a relatively coarse mesh may be dynamically transformed into a finer mesh with more vertices and triangles. The finer mesh is then shaded and rendered on the display, resulting in improved image quality.

Bounding volume hierarchies are commonly used to improve the efficiency with which operations are performed on graphics primitives and other graphics objects. A BVH is a hierarchical tree structure which is built based on a set of geometric objects. At the top of the tree structure is the root node which encloses all of the geometric objects in a given scene. The individual geometric objects are wrapped in bounding volumes that form the leaf nodes of the tree. These nodes are then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion, eventually resulting in a tree structure with a single bounding volume, represented by the root node, at the top of the tree. Bounding volume hierarchies are used to efficiently support a variety of operations on sets of geometric objects, such as collision detection, primitive culling, and ray traversal/intersection operations used in ray tracing.

In one embodiment, the tessellation hardware generate a bounding volume hierarchy (BVH) for a set of primitives resulting from tessellation of a surface as it generates those primitives. FIG. **15** illustrates an exemplary embodiment in which a tessellation unit **1502** of a GPU **1510** tesselates each input surface to generate a set of triangles or other primitives **1520** and concurrently generates a bounding volume hierarchy (BVH) **1521** associated with the new set of primitives. The tessellation unit **1520** receives surfaces and tessellation factors from a hull shader **1501**. Each surface may be defined, for example, by a plurality of vertices and the tessellation factors may specify how finely the surface is to be tessellated (e.g., how many triangles or other primitives to generate to represent the surface).

In one embodiment, as tessellation circuitry **1503** within the tessellation unit generates the primitives **1520**, BVH node generation circuitry **1504** generates a corresponding BVH **1521** based on the primitives using the same data required for tessellation. For example, the tessellation circuitry **1503** may first split the input surface into two halves, then split those two halves into halves, and so on, until the level of detail has been reached (as specified by the tessellation factors). The BVH node generation circuitry **1504** may define the root BVH node to include the entire input surface. When the input surface is split into two halves, the BVH node generation circuit **1504** may dynamically generate the first two child BVH nodes to include these two halves, then generate the next level of BVH nodes to include the next division and so on, until the entire BVH has been constructed concurrently with the tessellation operations. The results may be provided to a domain shader **1505** which performs shading operations on the new vertices/primitives.

An example is illustrated in FIG. **16** for a set of primitives **1520** generated during tessellation. A first bounding box **1600** (the root node) encompasses the entire set of primitives. The next two boxes **1610-1611** encompass half of the primitives (splitting one primitive down the middle), and the next set of bounding boxes **1620-1623** each enclose an individual primitive (i.e., they are leaf nodes). Thus, given the similarities associated with tessellation and BVH generation, the same tessellation hardware may be used to perform BVH construction concurrently with tessellation.

Apparatus and Method for Fast Topological BVH Generation for Hardware Tessellation

In addition, one embodiment of the invention extends the tessellation hardware to perform ray-triangle intersections on the fly. For example, as illustrated in FIG. **17**, the BVH node generation module **1504** may generate a BVH for a given surface as described above. The ray-surface intersection circuitry **1700** then uses the BVH to test the input surface and its primitives for intersection with a set of N rays before tessellation **1503**. If a particular set of primitives or potentially the entire surface is not intersected by any of the N rays, then this batch of primitives may be culled with respect to the N rays, thereby preserving bandwidth that would otherwise be consumed by the tessellation unit **1502** and other pipeline stages.

In one embodiment, the ray-surface intersection circuitry also includes BVH traversal circuitry to traverse the BVH to determine which primitives (if any) are intersected by one or more of the N rays. If only a subset of the primitives are intersected, then the tessellation circuitry **1503** may only tessellate that portion of the surface which includes primitives which may be intersected by a ray.

A method in accordance with one embodiment is illustrated in FIG. **18**. The method may be implemented within the context of the system architectures described above, but is not limited to any particular processor or system architecture.

At **1800**, a bounding volume hierarchy (BVH) is generated for an input surface with tessellation hardware. As mentioned, this may be accomplished using tessellation factors provided by a hull shader. At **1801**, intersection tests are performed for the input surface for using the BVH to determine whether any primitives of the surface are intersected by any of a set of rays. If not, then the entire input surface may be culled at **1803**. If so, then at **1804** tessellation circuitry is provided with tessellation factors for the surface and/or an indication of those primitives (or sets of primitives) which are intersected (or those which are not intersected). At **1805**, the tessellation circuitry tessellates the input surface to generate the primitives, potentially focusing on only those portions of the surface which are intersected. At **1806**, the results of the tessellation are output to a domain shader which may perform sharing at a primitive-level granularity.

In some embodiments, a graphics processing unit (GPU) is communicatively coupled to host/processor cores to accelerate graphics operations, machine-learning operations, pattern analysis operations, and various general purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or another interconnect (e.g., a high-speed interconnect such as PCIe or NVLink). In other embodiments, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

In the following description, numerous specific details are set forth to provide a more thorough understanding. However, it will be apparent to one of skill in the art that the embodiments described herein may be practiced without one or more of these specific details. In other instances, well-known features have not been described to avoid obscuring the details of the present embodiments.

System Overview

FIG. **19** is a block diagram illustrating a computing system **1900** configured to implement one or more aspects of the embodiments described herein. The computing system **1900** includes a processing subsystem **1901** having one or more processor(s) **1902** and a system memory **1904** communicating via an interconnection path that may include a memory hub **1905**. The memory hub **1905** may be a separate component within a chipset component or may be integrated within the one or more processor(s) **1902**. The memory hub **1905** couples with an I/O subsystem **1911** via a communication link **1906**. The I/O subsystem **1911** includes an I/O hub **1907** that can enable the computing system **1900** to receive input from one or more input device(s) **1908**. Additionally, the I/O hub **1907** can enable a display controller, which may be included in the one or more processor(s) **1902**, to provide outputs to one or more display device(s) **1910**A. In one embodiment the one or more display device(s) **1910**A coupled with the I/O hub **1907** can include a local, internal, or embedded display device.

In one embodiment the processing subsystem **1901** includes one or more parallel processor(s) **1912** coupled to memory hub **1905** via a bus or other communication link **1913**. The communication link **1913** may be one of any number of standards based communication link technologies or protocols, such as, but not limited to PCI Express, or may be a vendor specific communications interface or communications fabric. In one embodiment the one or more parallel processor(s) **1912** form a computationally focused parallel or vector processing system that an include a large number of processing cores and/or processing clusters, such as a many integrated core (MIC) processor. In one embodiment the one or more parallel processor(s) **1912** form a graphics processing subsystem that can output pixels to one of the one or more display device(s) **1910**A coupled via the I/O Hub **1907**. The one or more parallel processor(s) **1912** can also include a display controller and display interface (not shown) to enable a direct connection to one or more display device(s) **1910**B.

Within the I/O subsystem **1911**, a system storage unit **1914** can connect to the I/O hub **1907** to provide a storage mechanism for the computing system **1900**. An I/O switch **1916** can be used to provide an interface mechanism to enable connections between the I/O hub **1907** and other components, such as a network adapter **1918** and/or wireless network adapter **1919** that may be integrated into the platform, and various other devices that can be added via one or more add-in device(s) **1920**. The network adapter **1918** can be an Ethernet adapter or another wired network adapter. The wireless network adapter **1919** can include one or more of a Wi-Fi, Bluetooth, near field communication (NFC), or other network device that includes one or more wireless radios.

The computing system **1900** can include other components not explicitly shown, including USB or other port connections, optical storage drives, video capture devices, and the like, may also be connected to the I/O hub **1907**. Communication paths interconnecting the various components in FIG. **19** may be implemented using any suitable protocols, such as PCI (Peripheral Component Interconnect) based protocols (e.g., PCI-Express), or any other bus or point-to-point communication interfaces and/or protocol(s), such as the NV-Link high-speed interconnect, or interconnect protocols known in the art.

In one embodiment, the one or more parallel processor(s) **1912** incorporate circuitry optimized for graphics and video processing, including, for example, video output circuitry, and constitutes a graphics processing unit (GPU). In another embodiment, the one or more parallel processor(s) **1912** incorporate circuitry optimized for general purpose processing, while preserving the underlying computational architecture, described in greater detail herein. In yet another embodiment, components of the computing system **1900** may be integrated with one or more other system elements on a single integrated circuit. For example, the one or more parallel processor(s), **1912** memory hub **1905**, processor(s) **1902**, and I/O hub **1907** can be integrated into a system on chip (SoC) integrated circuit. Alternatively, the components of the computing system **1900** can be integrated into a single package to form a system in package (SIP) configuration. In one embodiment at least a portion of the components of the computing system **1900** can be integrated into a multi-chip module (MCM), which can be interconnected with other multi-chip modules into a modular computing system.

It will be appreciated that the computing system **1900** shown herein is illustrative and that variations and modifications are possible. The connection topology, including the number and arrangement of bridges, the number of processor(s) **1902**, and the number of parallel processor(s) **1912**, may be modified as desired. For instance, in some embodiments, system memory **1904** is connected to the processor(s) **1902** directly rather than through a bridge, while other devices communicate with system memory **1904** via the memory hub **1905** and the processor(s) **1902**. In other alternative topologies, the parallel processor(s) **1912** are connected to the I/O hub **1907** or directly to one of the one or more processor(s) **1902**, rather than to the memory hub **1905**. In other embodiments, the I/O hub **1907** and memory hub **1905** may be integrated into a single chip. Some embodiments may include two or more sets of processor(s) **1902** attached via multiple sockets, which can couple with two or more instances of the parallel processor(s) **1912**.

Some of the particular components shown herein are optional and may not be included in all implementations of the computing system **1900**. For example, any number of add-in cards or peripherals may be supported, or some components may be eliminated. Furthermore, some architectures may use different terminology for components similar to those illustrated in FIG. **19**. For example, the memory hub **1905** may be referred to as a Northbridge in some architectures, while the I/O hub **1907** may be referred to as a Southbridge.

FIG. **20**A illustrates a parallel processor **2000**, according to an embodiment. The various components of the parallel processor **2000** may be implemented using one or more integrated circuit devices, such as programmable processors, application specific integrated circuits (ASICs), or field programmable gate arrays (FPGA). The illustrated parallel processor **2000** is a variant of the one or more parallel processor(s) **1912** shown in FIG. **19**, according to an embodiment.

In one embodiment the parallel processor **2000** includes a parallel processing unit **2002**. The parallel processing unit includes an I/O unit **2004** that enables communication with other devices, including other instances of the parallel processing unit **2002**. The I/O unit **2004** may be directly connected to other devices. In one embodiment the I/O unit **2004** connects with other devices via the use of a hub or switch interface, such as memory hub **1905**. The connections between the memory hub **1905** and the I/O unit **2004** form a communication link **1913**. Within the parallel processing unit **2002**, the I/O unit **2004** connects with a host interface **2006** and a memory crossbar **2016**, where the host interface **2006** receives commands directed to performing processing operations and the memory crossbar **2016** receives commands directed to performing memory operations.

When the host interface **2006** receives a command buffer via the I/O unit **2004**, the host interface **2006** can direct work operations to perform those commands to a front end **2008**. In one embodiment the front end **2008** couples with a scheduler **2010**, which is configured to distribute commands or other work items to a processing cluster array **2012**. In one embodiment the scheduler **2010** ensures that the processing cluster array **2012** is properly configured and in a valid state before tasks are distributed to the processing clusters of the processing cluster array **2012**. In one embodiment the scheduler **2010** is implemented via firmware logic executing on a microcontroller. The microcontroller implemented scheduler **2010** is configurable to perform complex scheduling and work distribution operations at coarse and fine granularity, enabling rapid preemption and context switching of threads executing on the processing array **2012**. In one embodiment, the host software can prove workloads for scheduling on the processing array **2012** via one of multiple graphics processing doorbells. The workloads can then be automatically distributed across the processing array **2012** by the scheduler **2010** logic within the scheduler microcontroller.

The processing cluster array **2012** can include up to "N" processing clusters (e.g., cluster **2014**A, cluster **2014**B, through cluster **2014**N). Each cluster **2014**A-**2014**N of the processing cluster array **2012** can execute a large number of concurrent threads. The scheduler **2010** can allocate work to the clusters **2014**A-**2014**N of the processing cluster array **2012** using various scheduling and/or work distribution algorithms, which may vary depending on the workload arising for each type of program or computation. The scheduling can be handled dynamically by the scheduler **2010**, or can be assisted in part by compiler logic during compilation of program logic configured for execution by the processing cluster array **2012**. In one embodiment, different clusters **2014**A-**2014**N of the processing cluster array **2012** can be allocated for processing different types of programs or for performing different types of computations.

The processing cluster array **2012** can be configured to perform various types of parallel processing operations. In one embodiment the processing cluster array **2012** is configured to perform general-purpose parallel compute operations. For example, the processing cluster array **2012** can include logic to execute processing tasks including filtering of video and/or audio data, performing modeling operations, including physics operations, and performing data transformations.

In one embodiment the processing cluster array **2012** is configured to perform parallel graphics processing operations. In embodiments in which the parallel processor **2000** is configured to perform graphics processing operations, the processing cluster array **2012** can include additional logic to support the execution of such graphics processing operations, including, but not limited to texture sampling logic to perform texture operations, as well as tessellation logic and other vertex processing logic. Additionally, the processing cluster array **2012** can be configured to execute graphics processing related shader programs such as, but not limited to vertex shaders, tessellation shaders, geometry shaders, and pixel shaders. The parallel processing unit **2002** can transfer data from system memory via the I/O unit **2004** for processing. During processing the transferred data can be stored to on-chip memory (e.g., parallel processor memory **2022**) during processing, then written back to system memory.

In one embodiment, when the parallel processing unit **2002** is used to perform graphics processing, the scheduler **2010** can be configured to divide the processing workload into approximately equal sized tasks, to better enable distribution of the graphics processing operations to multiple clusters **2014**A-**2014**N of the processing cluster array **2012**. In some embodiments, portions of the processing cluster array **2012** can be configured to perform different types of processing. For example a first portion may be configured to perform vertex shading and topology generation, a second portion may be configured to perform tessellation and geometry shading, and a third portion may be configured to perform pixel shading or other screen space operations, to produce a rendered image for display. Intermediate data produced by one or more of the clusters **2014**A-**2014**N may be stored in buffers to allow the intermediate data to be transmitted between clusters **2014**A-**2014**N for further processing.

During operation, the processing cluster array **2012** can receive processing tasks to be executed via the scheduler **2010**, which receives commands defining processing tasks from front end **2008**. For graphics processing operations, processing tasks can include indices of data to be processed, e.g., surface (patch) data, primitive data, vertex data, and/or pixel data, as well as state parameters and commands defining how the data is to be processed (e.g., what program is to be executed). The scheduler **2010** may be configured to fetch the indices corresponding to the tasks or may receive the indices from the front end **2008**. The front end **2008** can be configured to ensure the processing cluster array **2012** is configured to a valid state before the workload specified by incoming command buffers (e.g., batch-buffers, push buffers, etc.) is initiated.

Each of the one or more instances of the parallel processing unit **2002** can couple with parallel processor memory **2022**. The parallel processor memory **2022** can be accessed via the memory crossbar **2016**, which can receive memory requests from the processing cluster array **2012** as well as the I/O unit **2004**. The memory crossbar **2016** can access the parallel processor memory **2022** via a memory interface **2018**. The memory interface **2018** can include multiple partition units (e.g., partition unit **2020**A, partition unit **2020**B, through partition unit **2020**N) that can each couple to a portion (e.g., memory unit) of parallel processor memory **2022**. In one implementation the number of partition units **2020**A-**2020**N is configured to be equal to the number of memory units, such that a first partition unit **2020**A has a corresponding first memory unit **2024**A, a second partition unit **2020**B has a corresponding memory unit **2024**B, and an Nth partition unit **2020**N has a corresponding Nth memory unit **2024**N. In other embodiments, the number of partition units **2020**A-**2020**N may not be equal to the number of memory devices.

In various embodiments, the memory units **2024**A-**2024**N can include various types of memory devices, including dynamic random access memory (DRAM) or graphics random access memory, such as synchronous graphics random access memory (SGRAM), including graphics double data rate (GDDR) memory. In one embodiment, the memory units **2024**A-**2024**N may also include 3D stacked memory, including but not limited to high bandwidth memory (HBM). Persons skilled in the art will appreciate that the specific implementation of the memory units **2024**A-**2024**N can vary, and can be selected from one of various conventional designs. Render targets, such as frame buffers or texture maps may be stored across the memory units **2024**A-**2024**N, allowing partition units **2020**A-**2020**N to write portions of each render target in parallel to efficiently use the available bandwidth of parallel processor memory **2022**. In some embodiments, a local instance of the parallel processor memory **2022** may be excluded in favor of a unified memory design that utilizes system memory in conjunction with local cache memory.

In one embodiment, any one of the clusters **2014**A-**2014**N of the processing cluster array **2012** can process data that will be written to any of the memory units **2024**A-**2024**N within parallel processor memory **2022**. The memory crossbar **2016** can be configured to transfer the output of each cluster **2014**A-**2014**N to any partition unit **2020**A-**2020**N or to another cluster **2014**A-**2014**N, which can perform additional processing operations on the output. Each cluster **2014**A-**2014**N can communicate with the memory interface **2018** through the memory crossbar **2016** to read from or write to various external memory devices. In one embodiment the memory crossbar **2016** has a connection to the memory interface **2018** to communicate with the I/O unit **2004**, as well as a connection to a local instance of the parallel processor memory **2022**, enabling the processing units within the different processing clusters **2014**A-**2014**N to communicate with system memory or other memory that is not local to the parallel processing unit **2002**. In one embodiment the memory crossbar **2016** can use virtual channels to separate traffic streams between the clusters **2014**A-**2014**N and the partition units **2020**A-**2020**N.

While a single instance of the parallel processing unit **2002** is illustrated within the parallel processor **2000**, any number of instances of the parallel processing unit **2002** can be included. For example, multiple instances of the parallel processing unit **2002** can be provided on a single add-in card, or multiple add-in cards can be interconnected. The different instances of the parallel processing unit **2002** can be configured to inter-operate even if the different instances have different numbers of processing cores, different amounts of local parallel processor memory, and/or other configuration differences. For example and in one embodiment, some instances of the parallel processing unit **2002** can include higher precision floating point units relative to other instances. Systems incorporating one or more instances of the parallel processing unit **2002** or the parallel processor **2000** can be implemented in a variety of configurations and form factors, including but not limited to desktop, laptop, or handheld personal computers, servers, workstations, game consoles, and/or embedded systems.

FIG. **20**B is a block diagram of a partition unit **2020**, according to an embodiment. In one embodiment the partition unit **2020** is an instance of one of the partition units **2020**A-**2020**N of FIG. **20**A. As illustrated, the partition unit **2020** includes an L2 cache **2021**, a frame buffer interface **2025**, and a ROP **2026** (raster operations unit). The L2 cache **2021** is a read/write cache that is configured to perform load and store operations received from the memory crossbar **2016** and ROP **2026**. Read misses and urgent write-back requests are output by L2 cache **2021** to frame buffer interface **2025** for processing. Updates can also be sent to the frame buffer via the frame buffer interface **2025** for processing. In one embodiment the frame buffer interface **2025** interfaces with one of the memory units in parallel processor memory, such as the memory units **2024**A-**2024**N of FIG. **20** (e.g., within parallel processor memory **2022**).

In graphics applications, the ROP **2026** is a processing unit that performs raster operations such as stencil, z test, blending, and the like. The ROP **2026** then outputs processed graphics data that is stored in graphics memory. In some embodiments the ROP **2026** includes compression logic to compress depth or color data that is written to memory and decompress depth or color data that is read from memory. The compression logic can be lossless compression logic that makes use of one or more of multiple compression algorithms. The type of compression that is performed by the ROP **2026** can vary based on the statistical characteristics of the data to be compressed. For example, in one embodiment, delta color compression is performed on depth and color data on a per-tile basis.

In some embodiments, the ROP **2026** is included within each processing cluster (e.g., cluster **2014**A-**2014**N of FIG. **20**) instead of within the partition unit **2020**. In such embodiment, read and write requests for pixel data are transmitted over the memory crossbar **2016** instead of pixel fragment data. The processed graphics data may be displayed on a display device, such as one of the one or more display device(s) **1910** of FIG. **19**, routed for further processing by the processor(s) **1902**, or routed for further processing by one of the processing entities within the parallel processor **2000** of FIG. **20**A.

FIG. **20**C is a block diagram of a processing cluster **2014** within a parallel processing unit, according to an embodiment. In one embodiment the processing cluster is an instance of one of the processing clusters **2014**A-**2014**N of FIG. **20**. The processing cluster **2014** can be configured to execute many threads in parallel, where the term "thread" refers to an instance of a particular program executing on a particular set of input data. In some embodiments, single-instruction, multiple-data (SIMD) instruction issue techniques are used to support parallel execution of a large number of threads without providing multiple independent instruction units. In other embodiments, single-instruction, multiple-thread (SIMT) techniques are used to support parallel execution of a large number of generally synchronized threads, using a common instruction unit configured to issue instructions to a set of processing engines within each one of the processing clusters. Unlike a SIMD execution regime, where all processing engines typically execute identical instructions, SIMT execution allows different threads to more readily follow divergent execution paths through a given thread program. Persons skilled in the art will understand that a SIMD processing regime represents a functional subset of a SIMT processing regime.

Operation of the processing cluster **2014** can be controlled via a pipeline manager **2032** that distributes processing tasks to SIMT parallel processors. The pipeline manager **2032** receives instructions from the scheduler **2010** of FIG. **20** and manages execution of those instructions via a graphics multiprocessor **2034** and/or a texture unit **2036**. The illustrated graphics multiprocessor **2034** is an exemplary instance of a SIMT parallel processor. However, various types of SIMT parallel processors of differing architectures may be included within the processing cluster **2014**. One or more instances of the graphics multiprocessor **2034** can be included within a processing cluster **2014**. The graphics multiprocessor **2034** can process data and a data crossbar **2040** can be used to distribute the processed data to one of multiple possible destinations, including other shader units. The pipeline manager **2032** can facilitate the distribution of processed data by specifying destinations for processed data to be distributed vis the data crossbar **2040**.

Each graphics multiprocessor **2034** within the processing cluster **2014** can include an identical set of functional execution logic (e.g., arithmetic logic units, load-store units, etc.). The functional execution logic can be configured in a pipelined manner in which new instructions can be issued before previous instructions are complete. The functional execution logic supports a variety of operations including integer and floating point arithmetic, comparison operations, Boolean operations, bit-shifting, and computation of various algebraic functions. In one embodiment the same functional-unit hardware can be leveraged to perform different operations and any combination of functional units may be present.

The instructions transmitted to the processing cluster **2014** constitutes a thread. A set of threads executing across the set of parallel processing engines is a thread group. A thread group executes the same program on different input data. Each thread within a thread group can be assigned to a different processing engine within a graphics multiprocessor **2034**. A thread group may include fewer threads than the number of processing engines within the graphics multiprocessor **2034**. When a thread group includes fewer threads than the number of processing engines, one or more of the processing engines may be idle during cycles in which that thread group is being processed. A thread group may also include more threads than the number of processing engines within the graphics multiprocessor **2034**. When the thread group includes more threads than the number of processing engines within the graphics multiprocessor **2034**, processing can be performed over consecutive clock cycles. In one embodiment multiple thread groups can be executed concurrently on a graphics multiprocessor **2034**.

In one embodiment the graphics multiprocessor **2034** includes an internal cache memory to perform load and store operations. In one embodiment, the graphics multiprocessor **2034** can forego an internal cache and use a cache memory (e.g., L1 cache **2108**) within the processing cluster **2014**. Each graphics multiprocessor **2034** also has access to L2 caches within the partition units (e.g., partition units **2020**A-**2020**N of FIG. **20**) that are shared among all processing clusters **2014** and may be used to transfer data between threads. The graphics multiprocessor **2034** may also access off-chip global memory, which can include one or more of local parallel processor memory and/or system memory. Any memory external to the parallel processing unit **2002** may be used as global memory. Embodiments in which the processing cluster **2014** includes multiple instances of the graphics multiprocessor **2034** can share common instructions and data, which may be stored in the L1 cache **2108**.

Each processing cluster **2014** may include an MMU **2045** (memory management unit) that is configured to map virtual addresses into physical addresses. In other embodiments, one or more instances of the MMU **2045** may reside within the memory interface **2018** of FIG. **20**. The MMU **2045** includes a set of page table entries (PTEs) used to map a virtual address to a physical address of a tile (talk more about tiling) and optionally a cache line index. The MMU **2045** may include address translation lookaside buffers (TLB) or caches that may reside within the graphics multiprocessor **2034** or the L1 cache or processing cluster **2014**. The physical address is processed to distribute surface data access locality to allow efficient request interleaving among partition units. The cache line index may be used to determine whether a request for a cache line is a hit or miss.

In graphics and computing applications, a processing cluster **2014** may be configured such that each graphics multiprocessor **2034** is coupled to a texture unit **2036** for performing texture mapping operations, e.g., determining texture sample positions, reading texture data, and filtering the texture data. Texture data is read from an internal texture L1 cache (not shown) or in some embodiments from the L1 cache within graphics multiprocessor **2034** and is fetched from an L2 cache, local parallel processor memory, or system memory, as needed. Each graphics multiprocessor **2034** outputs processed tasks to the data crossbar **2040** to provide the processed task to another processing cluster **2014** for further processing or to store the processed task in an L2 cache, local parallel processor memory, or system memory via the memory crossbar **2016**. A preROP **2042** (pre-raster operations unit) is configured to receive data from graphics multiprocessor **2034**, direct data to ROP units, which may be located with partition units as described herein (e.g., partition units **2020**A-**2020**N of FIG. **20**). The preROP **2042** unit can perform optimizations for color blending, organize pixel color data, and perform address translations.

It will be appreciated that the core architecture described herein is illustrative and that variations and modifications are possible. Any number of processing units, e.g., graphics multiprocessor **2034**, texture units **2036**, preROPs **2042**, etc., may be included within a processing cluster **2014**. Further, while only one processing cluster **2014** is shown, a parallel processing unit as described herein may include any number of instances of the processing cluster **2014**. In one embodiment, each processing cluster **2014** can be configured to operate independently of other processing clusters **2014** using separate and distinct processing units, L1 caches, etc.

FIG. **20**D shows a graphics multiprocessor **2034**, according to one embodiment. In such embodiment the graphics multiprocessor **2034** couples with the pipeline manager **2032** of the processing cluster **2014**. The graphics multiprocessor **2034** has an execution pipeline including but not limited to an instruction cache **2052**, an instruction unit **2054**, an address mapping unit **2056**, a register file **2058**, one or more general purpose graphics processing unit (GPGPU) cores **2062**, and one or more load/store units **2066**. The GPGPU cores **2062** and load/store units **2066** are coupled with cache memory **2072** and shared memory **2070** via a memory and cache interconnect **2068**.

In one embodiment, the instruction cache **2052** receives a stream of instructions to execute from the pipeline manager **2032**. The instructions are cached in the instruction cache **2052** and dispatched for execution by the instruction unit **2054**. The instruction unit **2054** can dispatch instructions as thread groups (e.g., warps), with each thread of the thread group assigned to a different execution unit within GPGPU core **2062**. An instruction can access any of a local, shared, or global address space by specifying an address within a unified address space. The address mapping unit **2056** can be used to translate addresses in the unified address space into a distinct memory address that can be accessed by the load/store units **2066**.

The register file **2058** provides a set of registers for the functional units of the graphics multiprocessor **2124**. The register file **2058** provides temporary storage for operands connected to the data paths of the functional units (e.g., GPGPU cores **2062**, load/store units **2066**) of the graphics multiprocessor **2124**. In one embodiment, the register file **2058** is divided between each of the functional units such that each functional unit is allocated a dedicated portion of the register file **2058**. In one embodiment, the register file **2058** is divided between the different warps being executed by the graphics multiprocessor **2124**.

The GPGPU cores **2062** can each include floating point units (FPUs) and/or integer arithmetic logic units (ALUs) that are used to execute instructions of the graphics multiprocessor **2124**. The GPGPU cores **2062** can be similar in architecture or can differ in architecture, according to embodiments. For example and in one embodiment, a first portion of the GPGPU cores **2062** include a single precision FPU and an integer ALU while a second portion of the GPGPU cores include a double precision FPU. In one embodiment the FPUs can implement the IEEE 754-2008 standard for floating point arithmetic or enable variable precision floating point arithmetic. The graphics multiprocessor **2124** can additionally include one or more fixed function or special function units to perform specific functions such as copy rectangle or pixel blending operations. In one embodiment one or more of the GPGPU cores can also include fixed or special function logic.

In one embodiment the GPGPU cores **2062** include SIMD logic capable of performing a single instruction on multiple sets of data. In one embodiment GPGPU cores **2062** can physically execute SIMD4, SIMD8, and SIMD16 instructions and logically execute SIMD1, SIMD2, and SIMD32 instructions. The SIMD instructions for the GPGPU cores can be generated at compile time by a shader compiler or automatically generated when executing programs written and compiled for single program multiple data (SPMD) or SIMT architectures. Multiple threads of a program configured for the SIMT execution model can executed via a single

SIMD instruction. For example and in one embodiment, eight SIMT threads that perform the same or similar operations can be executed in parallel via a single SIMD8 logic unit.

The memory and cache interconnect **2068** is an interconnect network that connects each of the functional units of the graphics multiprocessor **2124** to the register file **2058** and to the shared memory **2070**. In one embodiment, the memory and cache interconnect **2068** is a crossbar interconnect that allows the load/store unit **2066** to implement load and store operations between the shared memory **2070** and the register file **2058**. The register file **2058** can operate at the same frequency as the GPGPU cores **2062**, thus data transfer between the GPGPU cores **2062** and the register file **2058** is very low latency. The shared memory **2070** can be used to enable communication between threads that execute on the functional units within the graphics multiprocessor **2034**. The cache memory **2072** can be used as a data cache for example, to cache texture data communicated between the functional units and the texture unit **2036**. The shared memory **2070** can also be used as a program managed cached. Threads executing on the GPGPU cores **2062** can programmatically store data within the shared memory in addition to the automatically cached data that is stored within the cache memory **2072**.

FIGS. **21**A-**21**B illustrate additional graphics multiprocessors, according to embodiments. The illustrated graphics multiprocessors **2125**, **2150** are variants of the graphics multiprocessor **2034** of FIG. **20**C. The illustrated graphics multiprocessors **2125**, **2150** can be configured as a streaming multiprocessor (SM) capable of simultaneous execution of a large number of execution threads.

FIG. **21**A shows a graphics multiprocessor **2125** according to an additional embodiment. The graphics multiprocessor **2125** includes multiple additional instances of execution resource units relative to the graphics multiprocessor **2034** of FIG. **20**D. For example, the graphics multiprocessor **2125** can include multiple instances of the instruction unit **2132**A-**2132**B, register file **2134**A-**2134**B, and texture unit(s) **2144**A-**2144**B. The graphics multiprocessor **2125** also includes multiple sets of graphics or compute execution units (e.g., GPGPU core **2136**A-**2136**B, GPGPU core **2137**A-**2137**B, GPGPU core **2138**A-**2138**B) and multiple sets of load/store units **2140**A-**2140**B. In one embodiment the execution resource units have a common instruction cache **2130**, texture and/or data cache memory **2142**, and shared memory **2146**.

The various components can communicate via an interconnect fabric **2127**. In one embodiment the interconnect fabric **2127** includes one or more crossbar switches to enable communication between the various components of the graphics multiprocessor **2125**. In one embodiment the interconnect fabric **2127** is a separate, high-speed network fabric layer upon which each component of the graphics multiprocessor **2125** is stacked. The components of the graphics multiprocessor **2125** communicate with remote components via the interconnect fabric **2127**. For example, the GPGPU cores **2136**A-**2136**B, **2137**A-**2137**B, and **21378**A-**2138**B can each communicate with shared memory **2146** via the interconnect fabric **2127**. The interconnect fabric **2127** can arbitrate communication within the graphics multiprocessor **2125** to ensure a fair bandwidth allocation between components.

FIG. **21**B shows a graphics multiprocessor **2150** according to an additional embodiment. The graphics processor includes multiple sets of execution resources **2156**A-**2156**D, where each set of execution resource includes multiple instruction units, register files, GPGPU cores, and load store units, as illustrated in FIG. **20**D and FIG. **21**A. The execution resources **2156**A-**2156**D can work in concert with texture unit(s) **2160**A-**2160**D for texture operations, while sharing an instruction cache **2154**, and shared memory **2162**. In one embodiment the execution resources **2156**A-**2156**D can share an instruction cache **2154** and shared memory **2162**, as well as multiple instances of a texture and/or data cache memory **2158**A-**2158**B. The various components can communicate via an interconnect fabric **2152** similar to the interconnect fabric **2127** of FIG. **21**A.

Persons skilled in the art will understand that the architecture described in FIGS. **19**, **20**A-**20**D, and **21**A-**21**B are descriptive and not limiting as to the scope of the present embodiments. Thus, the techniques described herein may be implemented on any properly configured processing unit, including, without limitation, one or more mobile application processors, one or more desktop or server central processing units (CPUs) including multi-core CPUs, one or more parallel processing units, such as the parallel processing unit **202** of FIG. **2**, as well as one or more graphics processors or special purpose processing units, without departure from the scope of the embodiments described herein.

In some embodiments a parallel processor or GPGPU as described herein is communicatively coupled to host/processor cores to accelerate graphics operations, machine-learning operations, pattern analysis operations, and various general purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or other interconnect (e.g., a high speed interconnect such as PCIe or NVLink). In other embodiments, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

Techniques for GPU to Host Processor Interconnection

FIG. **22**A illustrates an exemplary architecture in which a plurality of GPUs **2210**-**2213** are communicatively coupled to a plurality of multi-core processors **2205**-**2206** over high-speed links **2240**-**2243** (e.g., buses, point-to-point interconnects, etc.). In one embodiment, the high-speed links **2240**-**2243** support a communication throughput of 4 GB/s, 30 GB/s, 80 GB/s or higher, depending on the implementation. Various interconnect protocols may be used including, but not limited to, PCIe 4.0 or 5.0 and NVLink 2.0. However, the underlying principles of the invention are not limited to any particular communication protocol or throughput.

In addition, in one embodiment, two or more of the GPUs **2210**-**2213** are interconnected over high-speed links **2244**-**2245**, which may be implemented using the same or different protocols/links than those used for high-speed links **2240**-**2243**. Similarly, two or more of the multi-core processors **2205**-**2206** may be connected over high speed link **2233** which may be symmetric multi-processor (SMP) buses operating at 20 GB/s, 30 GB/s, 120 GB/s or higher. Alternatively, all communication between the various system components shown in FIG. **22**A may be accomplished using the same protocols/links (e.g., over a common interconnection fabric). As mentioned, however, the underlying principles of the invention are not limited to any particular type of interconnect technology.

In one embodiment, each multi-core processor **2205-2206** is communicatively coupled to a processor memory **2201-2202**, via memory interconnects **2230-2231**, respectively, and each GPU **2210-2213** is communicatively coupled to GPU memory **2220-2223** over GPU memory interconnects **2250-2253**, respectively. The memory interconnects **2230-2231** and **2250-2253** may utilize the same or different memory access technologies. By way of example, and not limitation, the processor memories **2201-2202** and GPU memories **2220-2223** may be volatile memories such as dynamic random access memories (DRAMs) (including stacked DRAMs), Graphics DDR SDRAM (GDDR) (e.g., GDDR5, GDDR6), or High Bandwidth Memory (HBM) and/or may be non-volatile memories such as 3D XPoint or Nano-Ram. In one embodiment, some portion of the memories may be volatile memory and another portion may be non-volatile memory (e.g., using a two-level memory (2LM) hierarchy).

As described below, although the various processors **2205-2206** and GPUs **2210-2213** may be physically coupled to a particular memory **2201-2202**, **2220-2223**, respectively, a unified memory architecture may be implemented in which the same virtual system address space (also referred to as the "effective address" space) is distributed among all of the various physical memories. For example, processor memories **2201-2202** may each comprise 64 GB of the system memory address space and GPU memories **2220-2223** may each comprise 32 GB of the system memory address space (resulting in a total of 256 GB addressable memory in this example).

FIG. **22**B illustrates additional details for an interconnection between a multi-core processor **2207** and a graphics acceleration module **2246** in accordance with one embodiment. The graphics acceleration module **2246** may include one or more GPU chips integrated on a line card which is coupled to the processor **2207** via the high-speed link **2240**. Alternatively, the graphics acceleration module **2246** may be integrated on the same package or chip as the processor **2207**.

The illustrated processor **2207** includes a plurality of cores **2260**A-**2260**D, each with a translation lookaside buffer **2261**A-**2261**D and one or more caches **2262**A-**2262**D. The cores may include various other components for executing instructions and processing data which are not illustrated to avoid obscuring the underlying principles of the invention (e.g., instruction fetch units, branch prediction units, decoders, execution units, reorder buffers, etc.). The caches **2262**A-**2262**D may comprise level 1 (L1) and level 2 (L2) caches. In addition, one or more shared caches **2226** may be included in the caching hierarchy and shared by sets of the cores **2260**A-**2260**D. For example, one embodiment of the processor **2207** includes 24 cores, each with its own L1 cache, twelve shared L2 caches, and twelve shared L3 caches. In this embodiment, one of the L2 and L3 caches are shared by two adjacent cores. The processor **2207** and the graphics accelerator integration module **2246** connect with system memory **2241**, which may include processor memories **2201-2202**

Coherency is maintained for data and instructions stored in the various caches **2262**A-**2262**D, **2256** and system memory **2241** via inter-core communication over a coherence bus **2264**. For example, each cache may have cache coherency logic/circuitry associated therewith to communicate to over the coherence bus **2264** in response to detected reads or writes to particular cache lines. In one implementation, a cache snooping protocol is implemented over the coherence bus **2264** to snoop cache accesses. Cache snooping/coherency techniques are well understood by those of skill in the art and will not be described in detail here to avoid obscuring the underlying principles of the invention.

In one embodiment, a proxy circuit **2225** communicatively couples the graphics acceleration module **2246** to the coherence bus **2264**, allowing the graphics acceleration module **2246** to participate in the cache coherence protocol as a peer of the cores. In particular, an interface **2235** provides connectivity to the proxy circuit **2225** over high-speed link **2240** (e.g., a PCIe bus, NVLink, etc.) and an interface **2237** connects the graphics acceleration module **2246** to the link **2240**.

In one implementation, an accelerator integration circuit **2236** provides cache management, memory access, context management, and interrupt management services on behalf of a plurality of graphics processing engines **2231**, **2232**, N of the graphics acceleration module **2246**. The graphics processing engines **2231**, **2232**, N may each comprise a separate graphics processing unit (GPU). Alternatively, the graphics processing engines **2231**, **2232**, N may comprise different types of graphics processing engines within a GPU such as graphics execution units, media processing engines (e.g., video encoders/decoders), samplers, and blit engines. In other words, the graphics acceleration module may be a GPU with a plurality of graphics processing engines **2231-2232**, N or the graphics processing engines **2231-2232**, N may be individual GPUs integrated on a common package, line card, or chip.

In one embodiment, the accelerator integration circuit **2236** includes a memory management unit (MMU) **2239** for performing various memory management functions such as virtual-to-physical memory translations (also referred to as effective-to-real memory translations) and memory access protocols for accessing system memory **2241**. The MMU **2239** may also include a translation lookaside buffer (TLB) (not shown) for caching the virtual/effective to physical/real address translations. In one implementation, a cache **2238** stores commands and data for efficient access by the graphics processing engines **2231-2232**, N. In one embodiment, the data stored in cache **2238** and graphics memories **2233-2234**, N is kept coherent with the core caches **2262**A-**2262**D, **2256** and system memory **2211**. As mentioned, this may be accomplished via proxy circuit **2225** which takes part in the cache coherency mechanism on behalf of cache **2238** and memories **2233-2234**, N (e.g., sending updates to the cache **2238** related to modifications/accesses of cache lines on processor caches **2262**A-**2262**D, **2256** and receiving updates from the cache **2238**).

A set of registers **2245** store context data for threads executed by the graphics processing engines **2231-2232**, N and a context management circuit **2248** manages the thread contexts. For example, the context management circuit **2248** may perform save and restore operations to save and restore contexts of the various threads during contexts switches (e.g., where a first thread is saved and a second thread is stored so that the second thread can be execute by a graphics processing engine). For example, on a context switch, the context management circuit **2248** may store current register values to a designated region in memory (e.g., identified by a context pointer). It may then restore the register values when returning to the context. In one embodiment, an interrupt management circuit **2247** receives and processes interrupts received from system devices.

In one implementation, virtual/effective addresses from a graphics processing engine **2231** are translated to real/physical addresses in system memory **2211** by the MMU **2239**. One embodiment of the accelerator integration circuit **2236** supports multiple (e.g., 4, 8, 16) graphics accelerator modules **2246** and/or other accelerator devices. The graphics accelerator module **2246** may be dedicated to a single application executed on the processor **2207** or may be shared between multiple applications. In one embodiment, a virtualized graphics execution environment is presented in which the resources of the graphics processing engines **2231-2232**, N are shared with multiple applications or virtual machines (VMs). The resources may be subdivided into "slices" which are allocated to different VMs and/or applications based on the processing requirements and priorities associated with the VMs and/or applications.

Thus, the accelerator integration circuit acts as a bridge to the system for the graphics acceleration module **2246** and provides address translation and system memory cache services. In addition, the accelerator integration circuit **2236** may provide virtualization facilities for the host processor to manage virtualization of the graphics processing engines, interrupts, and memory management.

Because hardware resources of the graphics processing engines **2231-2232**, N are mapped explicitly to the real address space seen by the host processor **2207**, any host processor can address these resources directly using an effective address value. One function of the accelerator integration circuit **2236**, in one embodiment, is the physical separation of the graphics processing engines **2231-2232**, N so that they appear to the system as independent units.

As mentioned, in the illustrated embodiment, one or more graphics memories **2233-2234**, M are coupled to each of the graphics processing engines **2231-2232**, N, respectively. The graphics memories **2233-2234**, M store instructions and data being processed by each of the graphics processing engines **2231-2232**, N. The graphics memories **2233-2234**, M may be volatile memories such as DRAMs (including stacked DRAMs), GDDR memory (e.g., GDDR5, GDDR6), or HBM, and/or may be non-volatile memories such as 3D XPoint or Nano-Ram.

In one embodiment, to reduce data traffic over link **2240**, biasing techniques are used to ensure that the data stored in graphics memories **2233-2234**, M is data which will be used most frequently by the graphics processing engines **2231-2232**, N and preferably not used by the cores **2260**A-**2260**D (at least not frequently). Similarly, the biasing mechanism attempts to keep data needed by the cores (and preferably not the graphics processing engines **2231-2232**, N) within the caches **2262**A-**2262**D, **2256** of the cores and system memory **2211**.

FIG. **22**C illustrates another embodiment in which the accelerator integration circuit **2236** is integrated within the processor **2207**. In this embodiment, the graphics processing engines **2231-2232**, N communicate directly over the high-speed link **2240** to the accelerator integration circuit **2236** via interface **2237** and interface **2235** (which, again, may be utilize any form of bus or interface protocol). The accelerator integration circuit **2236** may perform the same operations as those described with respect to FIG. **22**B, but potentially at a higher throughput given its close proximity to the coherency bus **2262** and caches **2262**A-**2262**D, **2226**.

One embodiment supports different programming models including a dedicated-process programming model (no graphics acceleration module virtualization) and shared programming models (with virtualization). The latter may include programming models which are controlled by the accelerator integration circuit **2236** and programming models which are controlled by the graphics acceleration module **2246**.

In one embodiment of the dedicated process model, graphics processing engines **2231-2232**, N are dedicated to a single application or process under a single operating system. The single application can funnel other application requests to the graphics engines **2231-2232**, N, providing virtualization within a VM/partition.

In the dedicated-process programming models, the graphics processing engines **2231-2232**, N, may be shared by multiple VM/application partitions. The shared models require a system hypervisor to virtualize the graphics processing engines **2231-2232**, N to allow access by each operating system. For single-partition systems without a hypervisor, the graphics processing engines **2231-2232**, N are owned by the operating system. In both cases, the operating system can virtualize the graphics processing engines **2231-2232**, N to provide access to each process or application.

For the shared programming model, the graphics acceleration module **2246** or an individual graphics processing engine **2231-2232**, N selects a process element using a process handle. In one embodiment, process elements are stored in system memory **2211** and are addressable using the effective address to real address translation techniques described herein. The process handle may be an implementation-specific value provided to the host process when registering its context with the graphics processing engine **2231-2232**, N (that is, calling system software to add the process element to the process element linked list). The lower 16-bits of the process handle may be the offset of the process element within the process element linked list.

FIG. **22**D illustrates an exemplary accelerator integration slice **2290**. As used herein, a "slice" comprises a specified portion of the processing resources of the accelerator integration circuit **2236**. Application effective address space **2282** within system memory **2211** stores process elements **2283**. In one embodiment, the process elements **2283** are stored in response to GPU invocations **2281** from applications **2280** executed on the processor **2207**. A process element **2283** contains the process state for the corresponding application **2280**. A work descriptor (WD) **2284** contained in the process element **2283** can be a single job requested by an application or may contain a pointer to a queue of jobs. In the latter case, the WD **2284** is a pointer to the job request queue in the application's address space **2282**.

The graphics acceleration module **2246** and/or the individual graphics processing engines **2231-2232**, N can be shared by all or a subset of the processes in the system. Embodiments of the invention include an infrastructure for setting up the process state and sending a WD **2284** to a graphics acceleration module **2246** to start a job in a virtualized environment.

In one implementation, the dedicated-process programming model is implementation-specific. In this model, a single process owns the graphics acceleration module **2246** or an individual graphics processing engine **2231**. Because the graphics acceleration module **2246** is owned by a single process, the hypervisor initializes the accelerator integration circuit **2236** for the owning partition and the operating system initializes the accelerator integration circuit **2236** for the owning process at the time when the graphics acceleration module **2246** is assigned.

In operation, a WD fetch unit **2291** in the accelerator integration slice **2290** fetches the next WD **2284** which includes an indication of the work to be done by one of the graphics processing engines of the graphics acceleration module **2246**. Data from the WD **2284** may be stored in registers **2245** and used by the MMU **2239**, interrupt management circuit **2247** and/or context management circuit **2246** as illustrated. For example, one embodiment of the MMU **2239** includes segment/page walk circuitry for accessing segment/page tables **2286** within the OS virtual address space **2285**. The interrupt management circuit **2247** may process interrupt events **2292** received from the graphics acceleration module **2246**. When performing graphics operations, an effective address **2293** generated by a graphics processing engine **2231-2232**, N is translated to a real address by the MMU **2239**.

In one embodiment, the same set of registers **2245** are duplicated for each graphics processing engine **2231-2232**, N and/or graphics acceleration module **2246** and may be initialized by the hypervisor or operating system. Each of these duplicated registers may be included in an accelerator integration slice **2290**. Exemplary registers that may be initialized by the hypervisor are shown in Table 1.

TABLE 1

	Hypervisor Initialized Registers
1	Slice Control Register
2	Real Address (RA) Scheduled Processes Area Pointer
3	Authority Mask Override Register
4	Interrupt Vector Table Entry Offset
5	Interrupt Vector Table Entry Limit
6	State Register
7	Logical Partition ID
8	Real address (RA) Hypervisor Accelerator Utilization Record Pointer
9	Storage Description Register

Exemplary registers that may be initialized by the operating system are shown in Table 2.

TABLE 2

	Operating System Initialized Registers
1	Process and Thread Identification
2	Effective Address (EA) Context Save/Restore Pointer
3	Virtual Address (VA) Accelerator Utilization Record Pointer
4	Virtual Address (VA) Storage Segment Table Pointer
5	Authority Mask
6	Work descriptor

In one embodiment, each WD **2284** is specific to a particular graphics acceleration module **2246** and/or graphics processing engine **2231-2232**, N. It contains all the information a graphics processing engine **2231-2232**, N requires to do its work or it can be a pointer to a memory location where the application has set up a command queue of work to be completed.

FIG. **22**E illustrates additional details for one embodiment of a shared model. This embodiment includes a hypervisor real address space **2298** in which a process element list **2299** is stored. The hypervisor real address space **2298** is accessible via a hypervisor **2296** which virtualizes the graphics acceleration module engines for the operating system **2295**.

The shared programming models allow for all or a subset of processes from all or a subset of partitions in the system to use a graphics acceleration module **2246**. There are two programming models where the graphics acceleration module **2246** is shared by multiple processes and partitions: time-sliced shared and graphics directed shared.

In this model, the system hypervisor **2296** owns the graphics acceleration module **2246** and makes its function available to all operating systems **2295**. For a graphics acceleration module **2246** to support virtualization by the system hypervisor **2296**, the graphics acceleration module **2246** may adhere to the following requirements: 1) An application's job request must be autonomous (that is, the state does not need to be maintained between jobs), or the graphics acceleration module **2246** must provide a context save and restore mechanism. 2) An application's job request is guaranteed by the graphics acceleration module **2246** to complete in a specified amount of time, including any translation faults, or the graphics acceleration module **2246** provides the ability to preempt the processing of the job. 3) The graphics acceleration module **2246** must be guaranteed fairness between processes when operating in the directed shared programming model.

In one embodiment, for the shared model, the application **2280** is required to make an operating system **2295** system call with a graphics acceleration module **2246** type, a work descriptor (WD), an authority mask register (AMR) value, and a context save/restore area pointer (CSRP). The graphics acceleration module **2246** type describes the targeted acceleration function for the system call. The graphics acceleration module **2246** type may be a system-specific value. The WD is formatted specifically for the graphics acceleration module **2246** and can be in the form of a graphics acceleration module **2246** command, an effective address pointer to a user-defined structure, an effective address pointer to a queue of commands, or any other data structure to describe the work to be done by the graphics acceleration module **2246**. In one embodiment, the AMR value is the AMR state to use for the current process. The value passed to the operating system is similar to an application setting the AMR. If the accelerator integration circuit **2236** and graphics acceleration module **2246** implementations do not support a User Authority Mask Override Register (UAMOR), the operating system may apply the current UAMOR value to the AMR value before passing the AMR in the hypervisor call. The hypervisor **2296** may optionally apply the current Authority Mask Override Register (AMOR) value before placing the AMR into the process element **2283**. In one embodiment, the CSRP is one of the registers **2245** containing the effective address of an area in the application's address space **2282** for the graphics acceleration module **2246** to save and restore the context state. This pointer is optional if no state is required to be saved between jobs or when a job is preempted. The context save/restore area may be pinned system memory.

Upon receiving the system call, the operating system **2295** may verify that the application **2280** has registered and been given the authority to use the graphics acceleration module **2246**. The operating system **2295** then calls the hypervisor **2296** with the information shown in Table 3.

TABLE 3

	OS to Hypervisor Call Parameters
1	A work descriptor (WD)
2	An Authority Mask Register (AMR) value (potentially masked).
3	An effective address (EA) Context Save/Restore Area Pointer (CSRP)
4	A process ID (PID) and optional thread ID (TID)
5	A virtual address (VA) accelerator utilization record pointer (AURP)
6	The virtual address of the storage segment table pointer (SSTP)
7	A logical interrupt service number (LISN)

Upon receiving the hypervisor call, the hypervisor **2296** verifies that the operating system **2295** has registered and been given the authority to use the graphics acceleration module **2246**. The hypervisor **2296** then puts the process element **2283** into the process element linked list for the corresponding graphics acceleration module **2246** type. The process element may include the information shown in Table 4.

TABLE 4

Process Element Information	
1	A work descriptor (WD)
2	An Authority Mask Register (AMR) value (potentially masked).
3	An effective address (EA) Context Save/Restore Area Pointer (CSRP)
4	A process ID (PID) and optional thread ID (TID)
5	A virtual address (VA) accelerator utilization record pointer (AURP)
6	The virtual address of the storage segment table pointer (SSTP)
7	A logical interrupt service number (LISN)
8	Interrupt vector table, derived from the hypervisor call parameters.
9	A state register (SR) value
10	A logical partition ID (LPID)
11	A real address (RA) hypervisor accelerator utilization record pointer
12	The Storage Descriptor Register (SDR)

In one embodiment, the hypervisor initializes a plurality of accelerator integration slice **2290** registers **2245**.

As illustrated in FIG. **22**F, one embodiment of the invention employs a unified memory addressable via a common virtual memory address space used to access the physical processor memories **2201**-**2202** and GPU memories **2220**-**2223**. In this implementation, operations executed on the GPUs **2210**-**2213** utilize the same virtual/effective memory address space to access the processors memories **2201**-**2202** and vice versa, thereby simplifying programmability. In one embodiment, a first portion of the virtual/effective address space is allocated to the processor memory **2201**, a second portion to the second processor memory **2202**, a third portion to the GPU memory **2220**, and so on. The entire virtual/effective memory space (sometimes referred to as the effective address space) is thereby distributed across each of the processor memories **2201**-**2202** and GPU memories **2220**-**2223**, allowing any processor or GPU to access any physical memory with a virtual address mapped to that memory.

In one embodiment, bias/coherence management circuitry **2294**A-**2294**E within one or more of the MMUs **2239**A-**2239**E ensures cache coherence between the caches of the host processors (e.g., **2205**) and the GPUs **2210**-**2213** and implements biasing techniques indicating the physical memories in which certain types of data should be stored. While multiple instances of bias/coherence management circuitry **2294**A-**2294**E are illustrated in FIG. **22**F, the bias/coherence circuitry may be implemented within the MMU of one or more host processors **2205** and/or within the accelerator integration circuit **2236**.

One embodiment allows GPU-attached memory **2220**-**2223** to be mapped as part of system memory, and accessed using shared virtual memory (SVM) technology, but without suffering the typical performance drawbacks associated with full system cache coherence. The ability to GPU-attached memory **2220**-**2223** to be accessed as system memory without onerous cache coherence overhead provides a beneficial operating environment for GPU offload. This arrangement allows the host processor **2205** software to setup operands and access computation results, without the overhead of tradition I/O DMA data copies. Such traditional copies involve driver calls, interrupts and memory mapped I/O (MMIO) accesses that are all inefficient relative to simple memory accesses. At the same time, the ability to access GPU attached memory **2220**-**2223** without cache coherence overheads can be critical to the execution time of an offloaded computation. In cases with substantial streaming write memory traffic, for example, cache coherence overhead can significantly reduce the effective write bandwidth seen by a GPU **2210**-**2213**. The efficiency of operand setup, the efficiency of results access, and the efficiency of GPU computation all play a role in determining the effectiveness of GPU offload.

In one implementation, the selection of between GPU bias and host processor bias is driven by a bias tracker data structure. A bias table may be used, for example, which may be a page-granular structure (i.e., controlled at the granularity of a memory page) that includes 1 or 2 bits per GPU-attached memory page. The bias table may be implemented in a stolen memory range of one or more GPU-attached memories **2220**-**2223**, with or without a bias cache in the GPU **2210**-**2213** (e.g., to cache frequently/recently used entries of the bias table). Alternatively, the entire bias table may be maintained within the GPU.

In one implementation, the bias table entry associated with each access to the GPU-attached memory **2220**-**2223** is accessed prior the actual access to the GPU memory, causing the following operations. First, local requests from the GPU **2210**-**2213** that find their page in GPU bias are forwarded directly to a corresponding GPU memory **2220**-**2223**. Local requests from the GPU that find their page in host bias are forwarded to the processor **2205** (e.g., over a high-speed link as discussed above). In one embodiment, requests from the processor **2205** that find the requested page in host processor bias complete the request like a normal memory read. Alternatively, requests directed to a GPU-biased page may be forwarded to the GPU **2210**-**2213**. The GPU may then transition the page to a host processor bias if it is not currently using the page.

The bias state of a page can be changed either by a software-based mechanism, a hardware-assisted software-based mechanism, or, for a limited set of cases, a purely hardware-based mechanism.

One mechanism for changing the bias state employs an API call (e.g. OpenCL), which, in turn, calls the GPU's device driver which, in turn, sends a message (or enqueues a command descriptor) to the GPU directing it to change the bias state and, for some transitions, perform a cache flushing operation in the host. The cache flushing operation is required for a transition from host processor **2205** bias to GPU bias, but is not required for the opposite transition.

In one embodiment, cache coherency is maintained by temporarily rendering GPU-biased pages uncacheable by the host processor **2205**. To access these pages, the processor **2205** may request access from the GPU **2210** which may or may not grant access right away, depending on the implementation. Thus, to reduce communication between the processor **2205** and GPU **2210** it is beneficial to ensure that GPU-biased pages are those which are required by the GPU but not the host processor **2205** and vice versa.

Graphics Processing Pipeline

FIG. **23** illustrates a graphics processing pipeline **2300**, according to an embodiment. In one embodiment a graphics processor can implement the illustrated graphics processing pipeline **2300**. The graphics processor can be included within the parallel processing subsystems as described herein, such as the parallel processor **2000** of FIG. **20**, which, in one embodiment, is a variant of the parallel processor(s) **1912** of FIG. **19**. The various parallel processing systems can implement the graphics processing pipeline **2300** via one or more instances of the parallel processing unit (e.g., parallel processing unit **2002** of FIG. **20**) as described herein. For example, a shader unit (e.g., graphics multiprocessor **2034** of FIG. **21**) may be configured to perform the functions of one or more of a vertex processing unit **2304**, a tessellation control processing unit **2308**, a tessellation evaluation processing unit **2312**, a geometry processing unit **2316**, and a fragment/pixel processing unit **2324**. The functions of data assembler **2302**, primitive assemblers **2306**, **2314**, **2318**, tessellation unit **2310**, rasterizer **2322**, and raster operations unit **2326** may also be performed by other processing engines within a processing cluster (e.g., processing cluster **2014** of FIG. **21**) and a corresponding partition unit (e.g., partition unit **2020**A-**2020**N of FIG. **20**). The graphics processing pipeline **2300** may also be implemented using dedicated processing units for one or more functions. In one embodiment, one or more portions of the graphics processing pipeline **2300** can be performed by parallel processing logic within a general purpose processor (e.g., CPU). In one embodiment, one or more portions of the graphics processing pipeline **2300** can access on-chip memory (e.g., parallel processor memory **2022** as in FIG. **20**) via a memory interface **2328**, which may be an instance of the memory interface **2018** of FIG. **20**.

In one embodiment the data assembler **2302** is a processing unit that collects vertex data for surfaces and primitives. The data assembler **2302** then outputs the vertex data, including the vertex attributes, to the vertex processing unit **2304**. The vertex processing unit **2304** is a programmable execution unit that executes vertex shader programs, lighting and transforming vertex data as specified by the vertex shader programs. The vertex processing unit **2304** reads data that is stored in cache, local or system memory for use in processing the vertex data and may be programmed to transform the vertex data from an object-based coordinate representation to a world space coordinate space or a normalized device coordinate space.

A first instance of a primitive assembler **2306** receives vertex attributes from the vertex processing unit **230**. The primitive assembler **2306** readings stored vertex attributes as needed and constructs graphics primitives for processing by tessellation control processing unit **2308**. The graphics primitives include triangles, line segments, points, patches, and so forth, as supported by various graphics processing application programming interfaces (APIs).

The tessellation control processing unit **2308** treats the input vertices as control points for a geometric patch. The control points are transformed from an input representation from the patch (e.g., the patch's bases) to a representation that is suitable for use in surface evaluation by the tessellation evaluation processing unit **2312**. The tessellation control processing unit **2308** can also compute tessellation factors for edges of geometric patches. A tessellation factor applies to a single edge and quantifies a view-dependent level of detail associated with the edge. A tessellation unit **2310** is configured to receive the tessellation factors for edges of a patch and to tessellate the patch into multiple geometric primitives such as line, triangle, or quadrilateral primitives, which are transmitted to a tessellation evaluation processing unit **2312**. The tessellation evaluation processing unit **2312** operates on parameterized coordinates of the subdivided patch to generate a surface representation and vertex attributes for each vertex associated with the geometric primitives.

A second instance of a primitive assembler **2314** receives vertex attributes from the tessellation evaluation processing unit **2312**, reading stored vertex attributes as needed, and constructs graphics primitives for processing by the geometry processing unit **2316**. The geometry processing unit **2316** is a programmable execution unit that executes geometry shader programs to transform graphics primitives received from primitive assembler **2314** as specified by the geometry shader programs. In one embodiment the geometry processing unit **2316** is programmed to subdivide the graphics primitives into one or more new graphics primitives and calculate parameters used to rasterize the new graphics primitives.

In some embodiments the geometry processing unit **2316** can add or delete elements in the geometry stream. The geometry processing unit **2316** outputs the parameters and vertices specifying new graphics primitives to primitive assembler **2318**. The primitive assembler **2318** receives the parameters and vertices from the geometry processing unit **2316** and constructs graphics primitives for processing by a viewport scale, cull, and clip unit **2320**. The geometry processing unit **2316** reads data that is stored in parallel processor memory or system memory for use in processing the geometry data. The viewport scale, cull, and clip unit **2320** performs clipping, culling, and viewport scaling and outputs processed graphics primitives to a rasterizer **2322**.

The rasterizer **2322** can perform depth culling and other depth-based optimizations. The rasterizer **2322** also performs scan conversion on the new graphics primitives to generate fragments and output those fragments and associated coverage data to the fragment/pixel processing unit **2324**. The fragment/pixel processing unit **2324** is a programmable execution unit that is configured to execute fragment shader programs or pixel shader programs. The fragment/pixel processing unit **2324** transforming fragments or pixels received from rasterizer **2322**, as specified by the fragment or pixel shader programs. For example, the fragment/pixel processing unit **2324** may be programmed to perform operations included but not limited to texture mapping, shading, blending, texture correction and perspective correction to produce shaded fragments or pixels that are output to a raster operations unit **2326**. The fragment/pixel processing unit **2324** can read data that is stored in either the parallel processor memory or the system memory for use when processing the fragment data. Fragment or pixel shader programs may be configured to shade at sample, pixel, tile, or other granularities depending on the sampling rate configured for the processing units.

The raster operations unit **2326** is a processing unit that performs raster operations including, but not limited to stencil, z test, blending, and the like, and outputs pixel data as processed graphics data to be stored in graphics memory (e.g., parallel processor memory **2022** as in FIG. **20**, and/or system memory **1904** as in FIG. **19**, to be displayed on the one or more display device(s) **1910** or for further processing by one of the one or more processor(s) **1902** or parallel processor(s) **1912**. In some embodiments the raster operations unit **2326** is configured to compress z or color data that is written to memory and decompress z or color data that is read from memory.

In embodiments, the term "engine" or "module" or "logic" may refer to, be part of, or include an application specific integrated circuit (ASIC), an electronic circuit, a processor (shared, dedicated, or group), and/or memory (shared, dedicated, or group) that execute one or more software or firmware programs, a combinational logic circuit, and/or other suitable components that provide the described functionality. In embodiments, an engine or a module may be implemented in firmware, hardware, software, or any combination of firmware, hardware, and software.

Embodiments of the invention may include various steps, which have been described above. The steps may be embodied in machine-executable instructions which may be used to cause a general-purpose or special-purpose processor to perform the steps. Alternatively, these steps may be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

As described herein, instructions may refer to specific configurations of hardware such as application specific integrated circuits (ASICs) configured to perform certain operations or having a predetermined functionality or software instructions stored in memory embodied in a non-transitory computer readable medium. Thus, the techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices (e.g., an end station, a network element, etc.). Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer machine-readable media, such as non-transitory computer machine-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer machine-readable communication media (e.g., electrical, optical, acoustical or other form of propagated signals—such as carrier waves, infrared signals, digital signals, etc.).

In addition, such electronic devices typically include a set of one or more processors coupled to one or more other components, such as one or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage device of a given electronic device typically stores code and/or data for execution on the set of one or more processors of that electronic device. Of course, one or more parts of an embodiment of the invention may be implemented using different combinations of software, firmware, and/or hardware. Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the invention may be practiced without some of these specific details. In certain instances, well known structures and functions were not described in elaborate detail in order to avoid obscuring the subject matter of the present invention. Accordingly, the scope and spirit of the invention should be judged in terms of the claims which follow.

What is claimed is:

1. A graphics processor comprising:

tessellation circuitry to tessellate at least a portion of an input surface based on tessellation factors associated with the input surface to generate a first set of primitives;

bounding volume generator circuitry to dynamically generate, based on the tessellation factors, two or more child nodes of a bounding volume hierarchy (BVH) to bound the first set of primitives, each of the two or more child nodes containing at least a portion of the input surface, the two or more child nodes generated concurrently with the tessellation of the portion of the input surface; and

ray traversal/intersection circuitry to perform, based on the BVH, intersection tests between the input surface and a set of rays to determine a first portion of the input surface intersected by the rays,

wherein the first set of primitives is generated by the tessellation circuitry tessellating only the first portion of the input surface.

2. The graphics processor of claim **1**, wherein the ray traversal/intersection circuitry is further to determine, from the intersection test, a second portion of the input surface not intersected by the rays.

3. The graphics processor of claim **2**, wherein the second portion of the input surface is not tessellated by the tessellation circuitry.

4. The graphics processor of claim **1**, wherein the bounding volume generator circuitry, the ray traversal/intersection circuitry, and the tessellation circuitry are all part of a tessellation hardware of the graphics processor.

5. The graphics processor of claim **4**, further comprising:

a shader to provide the input surface and the tessellation factors associated therewith to the tessellation hardware.

6. The graphics processor of claim **1**, wherein the tessellation factors specify how finely associated surfaces are to be tessellated.

7. A method comprising:

tessellating at least a portion of an input surface based on tessellation factors associated with the input surface to generate a first set of primitives;

dynamically generating, based on the tessellation factors, two or more child nodes of a bounding volume hierarchy (BVH) to bound the first set of primitives, each of the two or more child nodes containing at least a portion of the input surface, the two or more child nodes generated concurrently with the tessellation of the portion of the input surface; and

performing, based on the BVH, intersection tests between the input surface and a set of rays to determine a first portion of the input surface intersected by the rays,

wherein the first set of primitives is generated by tessellating only the first portion of the input surface.

8. The method of claim **7**, further comprising:

determining, from the intersection test, a second portion of the input surface not intersected by the rays.

9. The method of claim **8**, wherein the second portion of the input surface is not tessellated.

10. The method of claim **7**, wherein the tessellation of the portion of the input surface, the generation of the two or more child nodes of the BVH, and intersection test are all performed by a tessellation hardware of a graphics processor.

11. The method of claim **10**, further comprising:

providing the input surface and the tessellation factors associated therewith from a shader to the tessellation hardware.

12. The method of claim **7**, wherein the tessellation factors specify how finely associated surfaces are to be tessellated.

13. A non-transitory machine-readable medium having program code stored thereon which, when executed by a machine, causes the machine to perform operations of:

tessellating at least a portion of an input surface based on tessellation factors associated with the input surface to generate a first set of primitives;

dynamically generating, based on the tessellation factors, two or more child nodes of a bounding volume hierarchy (BVH) to bound the first set of primitives, each of the two or more child nodes containing at least a portion of the input surface, the two or more child nodes generated concurrently with the tessellation of the portion of the input surface; and

performing, based on the BVH, intersection tests between the input surface and a set of rays to determine a first portion of the input surface intersected by the rays,

wherein the first set of primitives is generated by tessellating only the first portion of the input surface.

14. The non-transitory machine-readable medium of claim **13**, further comprising:

determining, from the intersection test, a second portion of the input surface not intersected by the rays.

15. The non-transitory machine-readable medium of claim **14**, wherein the second portion of the input surface is not tessellated.

16. The non-transitory machine-readable medium of claim **13**, wherein the tessellation of the portion of the input surface, the generation of the two or more child nodes of the BVH, and intersection test are all performed by a tessellation hardware.

17. The non-transitory machine-readable medium of claim **16**, further comprising:

providing the input surface and the tessellation factors associated therewith from a shader to the tessellation hardware.

18. The non-transitory machine-readable medium of claim **13**, wherein the tessellation factors specify how finely associated surfaces are to be tessellated.

* * * * *