US 20170257585A1

(19) United States

(12) Patent Application Publication

Desai et al.

(10) Pub. No.: US 2017/0257585 A1

(43) Pub. Date: Sep. 7, 2017

(54) LINE BUFFER UNIT FOR IMAGE PROCESSOR

(71) Applicant: **Google Inc.**, Mountain View, CA (US)

(72) Inventors: **Neeti Desai**, Sunnyvale, CA (US); **Albert Meixner**, Mountain View, CA (US); **Qiuling Zhu**, San Jose, CA (US); **Jason Rupert Redgrave**, Mountain View, CA (US); **Ofer Shacham**, Palo Alto, CA (US); **Daniel Frederic Finchelstein**, Redwood City, CA (US)

(73) Assignee: **Google Inc.**, Mountain View, CA (US)

(21) Appl. No.: **15/598,027**

(22) Filed: **May 17, 2017**

Related U.S. Application Data

(63) Continuation of application No. 14/694,712, filed on Apr. 23, 2015.

Publication Classification

(51) **Int. Cl.**

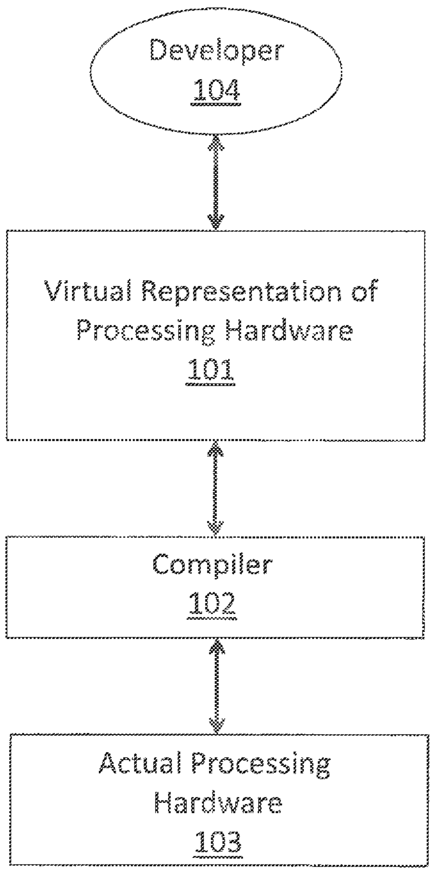
H04N 5/369	(2006.01)
H04N 5/91	(2006.01)
G06T 1/60	(2006.01)

(52) **U.S. Cl.**

CPC ***H04N 5/3692*** (2013.01); ***G06T 1/60*** (2013.01); ***H04N 5/91*** (2013.01)

(57) **ABSTRACT**

An apparatus is described that include a line buffer unit composed of a plurality of a line buffer interface units. Each line buffer interface unit is to handle one or more requests by a respective producer to store a respective line group in a memory and handle one or more requests by a respective consumer to fetch and provide the respective line group from memory. The line buffer unit has programmable storage space whose information establishes line group size so that different line group sizes for different image sizes are storable in memory.


Developer
104
Virtual Representation of Processing Hardware
101
Compiler
102
Actual Processing Hardware
103

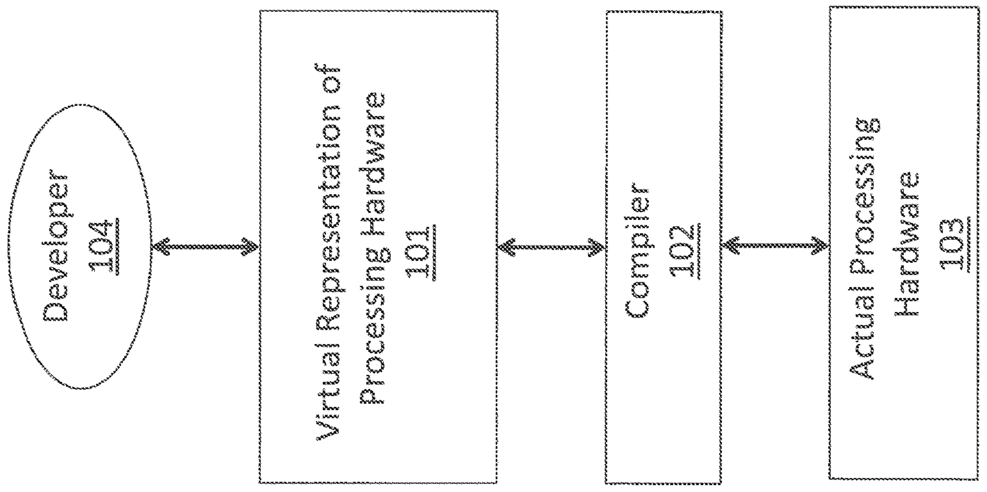

Developer 104
Virtual Representation of Processing Hardware 101
Compiler 102
Actual Processing Hardware 103

Fig. 1

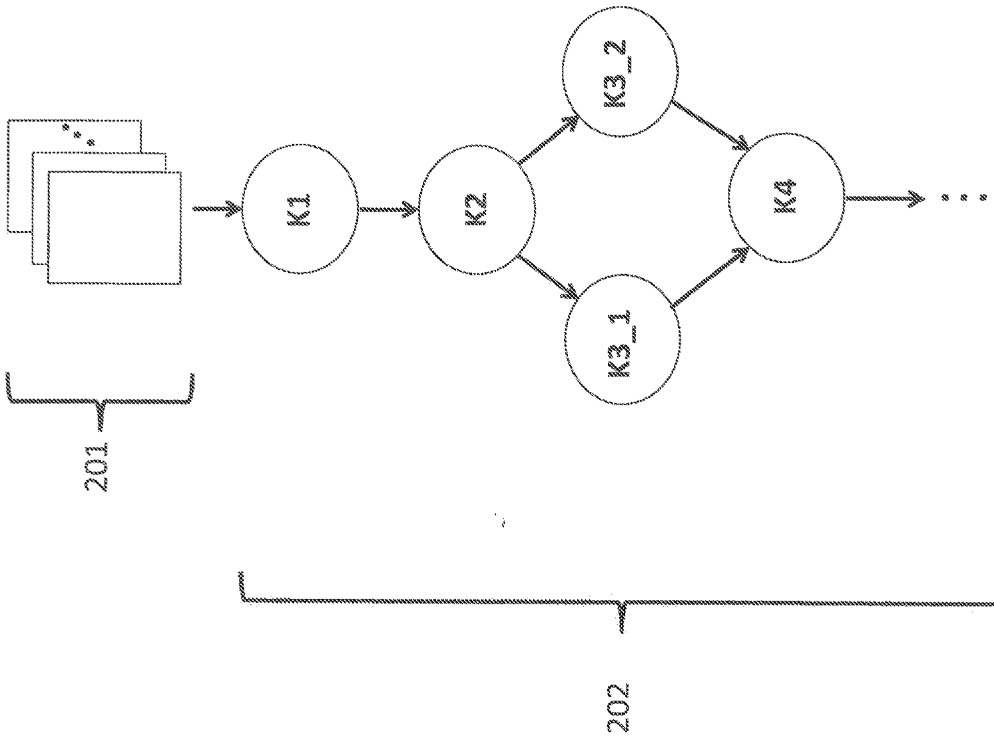


Fig. 2a

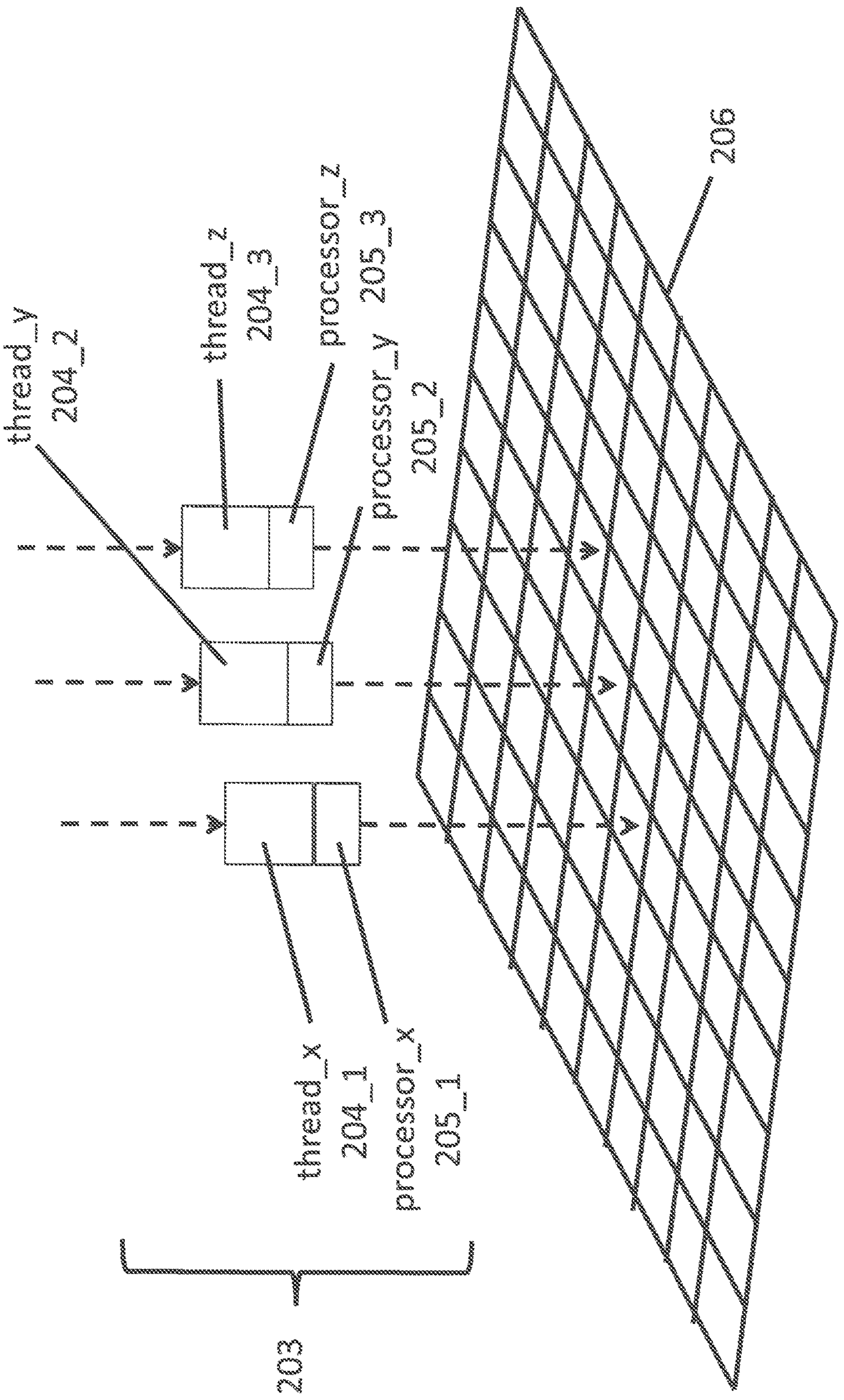

thread_y
204_2
thread_z
204_3
processor_z
205_3
processor_y
205_2
thread_x
204_1
processor_x
205_1
203
206

Fig. 2b

301

```
//thread X1: process stencil 1//
R1<= LOAD A   - - - - - - - - - - - -
R2<= LOAD B
R2 <= ADD R1, R2
R1<= LOAD C
R2<= ADD R1, R2
R1<= LOAD D
R2<= ADD R1, R2
R1<= LOAD E
R2<= ADD R1, R2
R1<= LOAD F
R2<= ADD R1, R2
R1<= LOAD G
R2<= ADD R1, R2
R1<= LOAD H
R2<= ADD R1, R2
R1<= LOAD I
R2<= ADD R1, R2
R2<= DIV R2, 9
X1<= STORE R2 - - - - - - - - - - - -
```

302

```
//thread X2: process stencil 2//
R1<= LOAD D
R2<= LOAD E
R2 <= ADD R1, R2
R1<= LOAD F
R2<= ADD R1, R2
R1<= LOAD G
R2<= ADD R1, R2
R1<= LOAD H
R2<= ADD R1, R2
R1<= LOAD I
R2<= ADD R1, R2
R1<= LOAD J
R2<= ADD R1, R2
R1<= LOAD K
R2<= ADD R1, R2
R1<= LOAD L
R2<= ADD R1, R2
R2<= DIV R2, 9
X2<= STORE R2
```

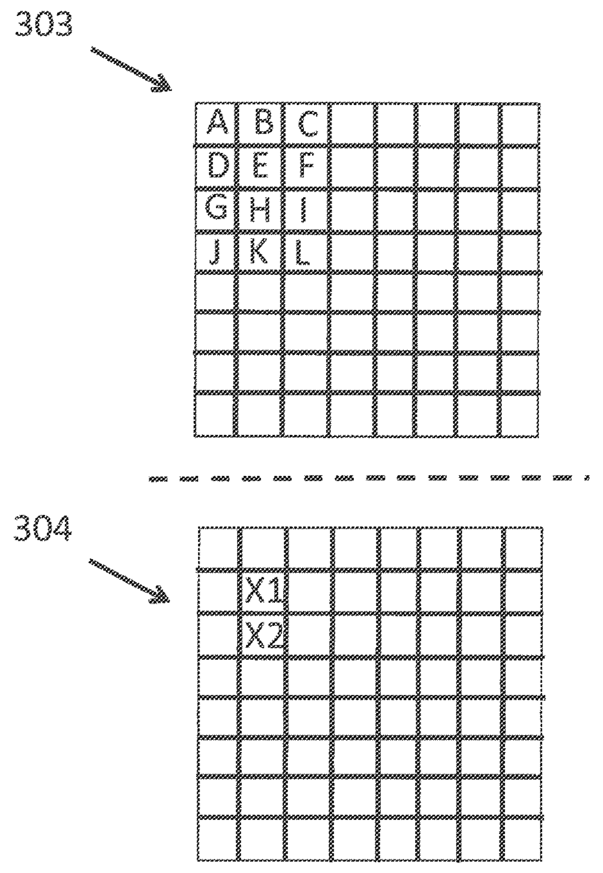


Fig. 3

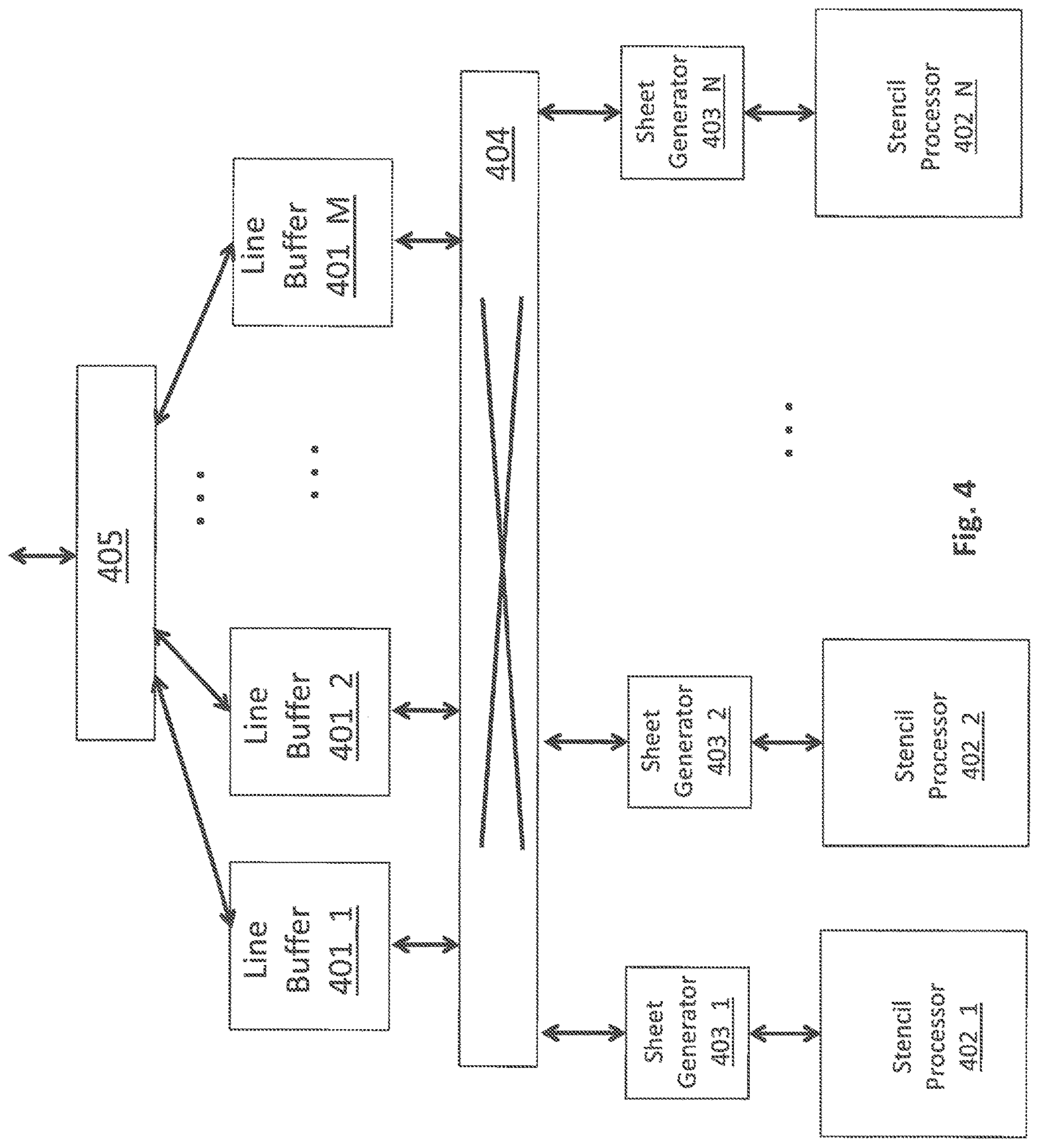

405
Line Buffer 401_1
Line Buffer 401_2
Line Buffer 401_M
404
Sheet Generator 403_1
Sheet Generator 403_2
Sheet Generator 403_N
Stencil Processor 402_1
Stencil Processor 402_2
Stencil Processor 402_N

Fig. 4

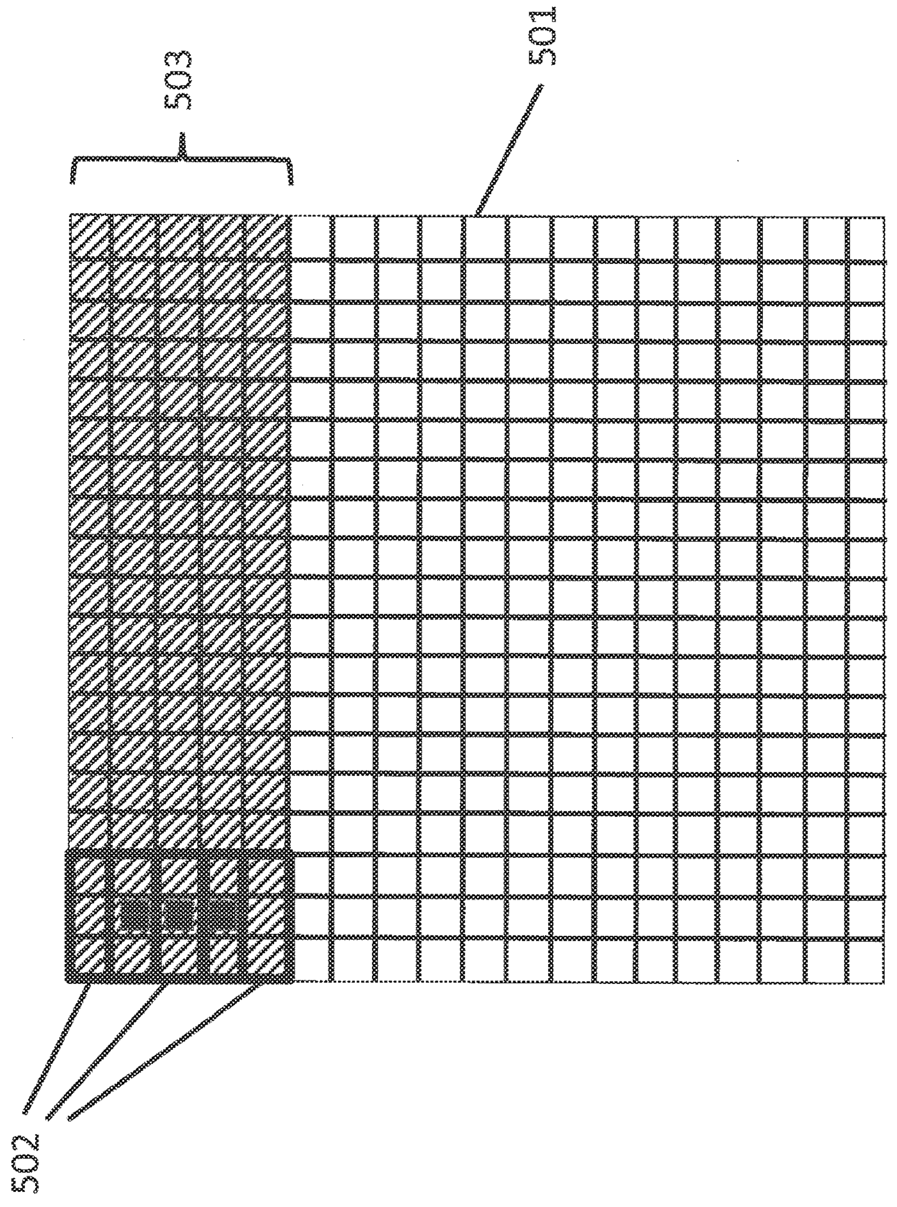


Fig. 5a

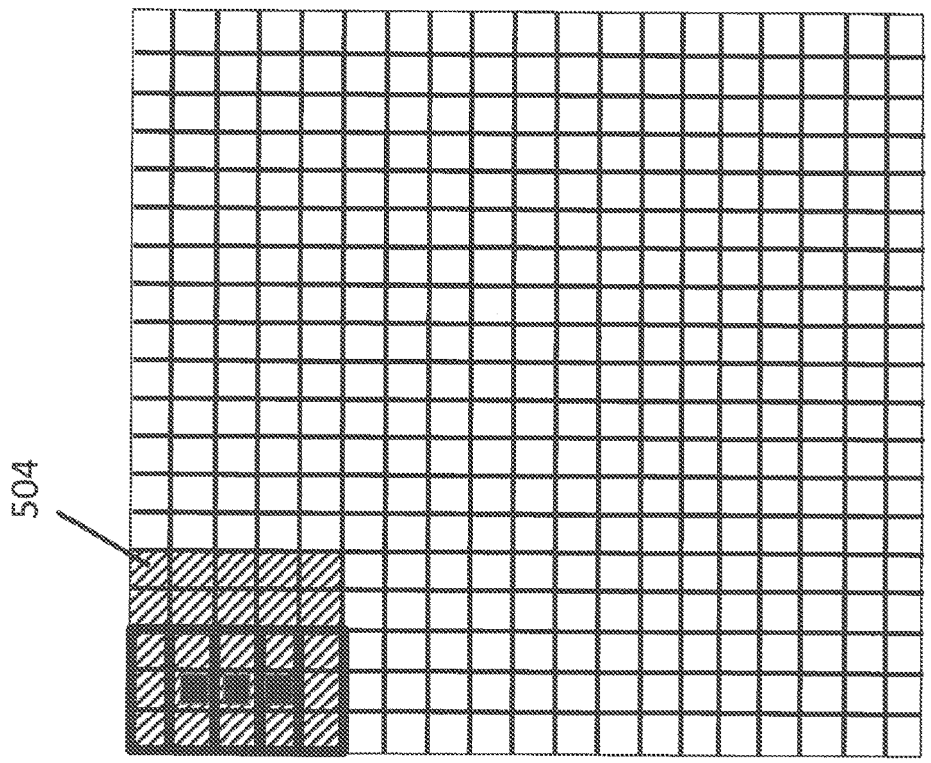


Fig. 5b

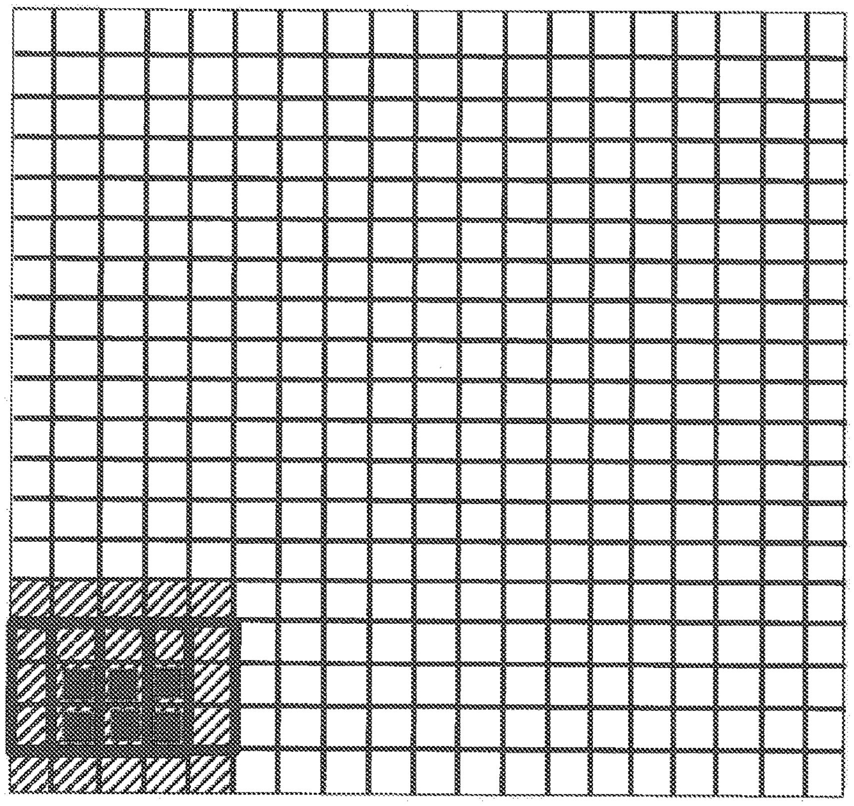


Fig. 5c

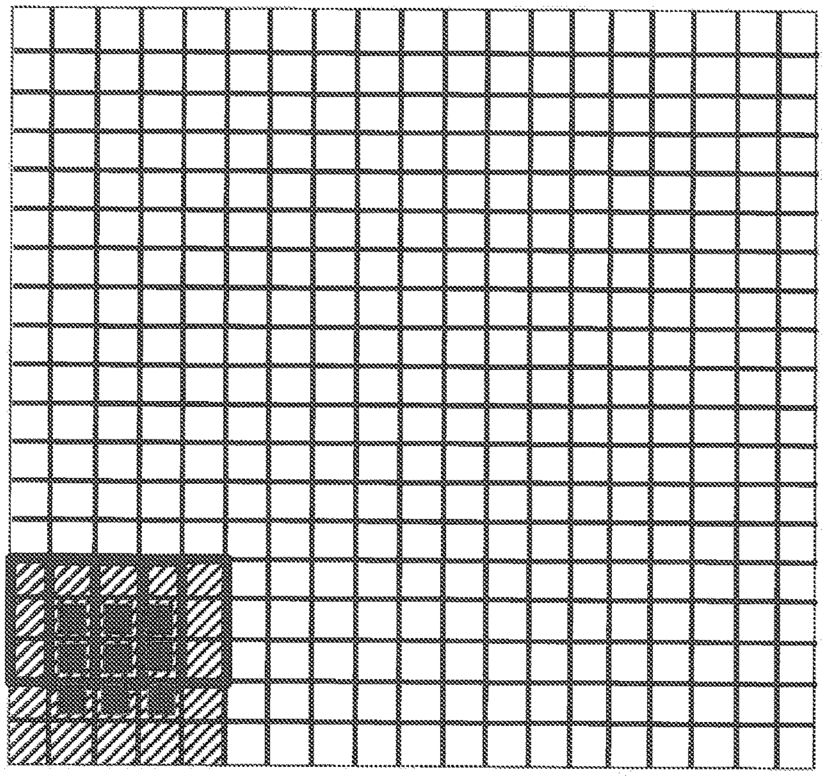


Fig. 5d

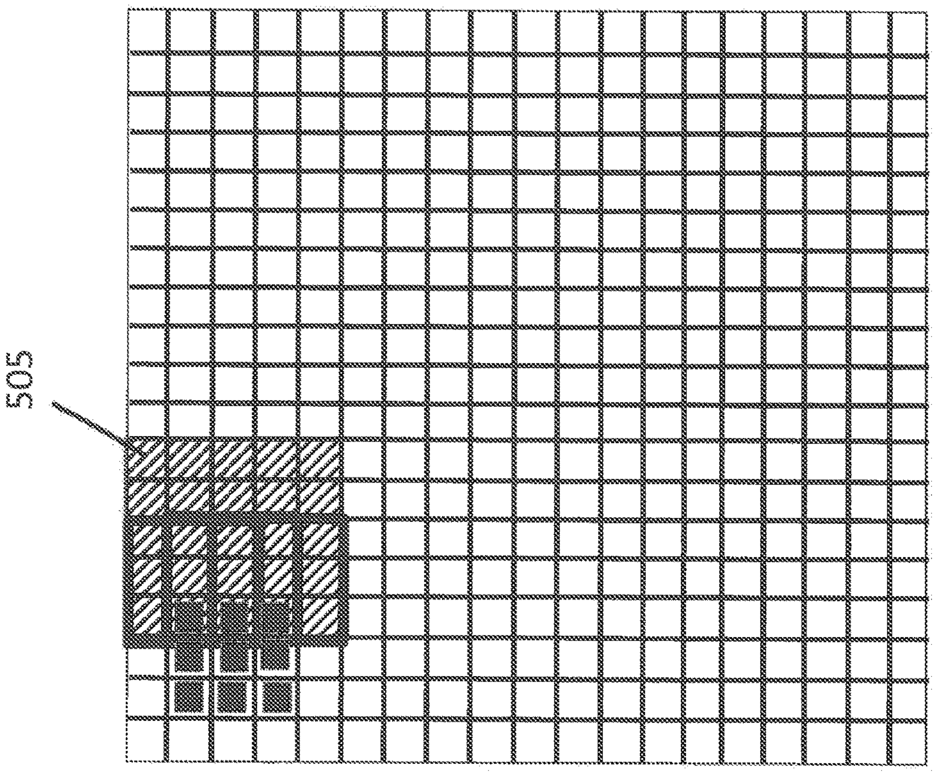


Fig. 5e

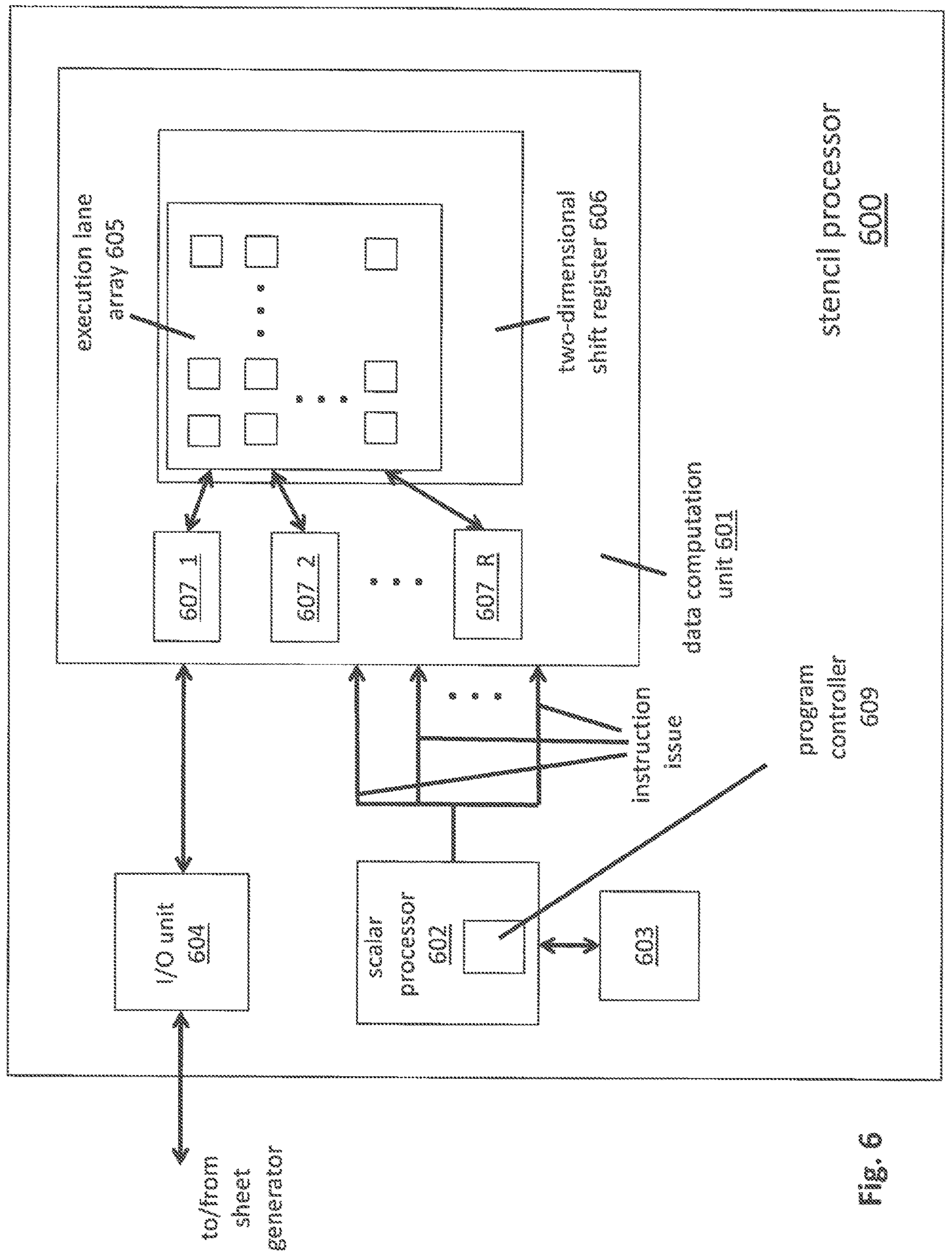

stencil processor 600
execution lane array 605
two-dimensional shift register 606
607_1
607_2
607_R
data computation unit 601
I/O unit 604
scalar processor 602
603
instruction issue
program controller 609
to/from sheet generator

Fig. 6

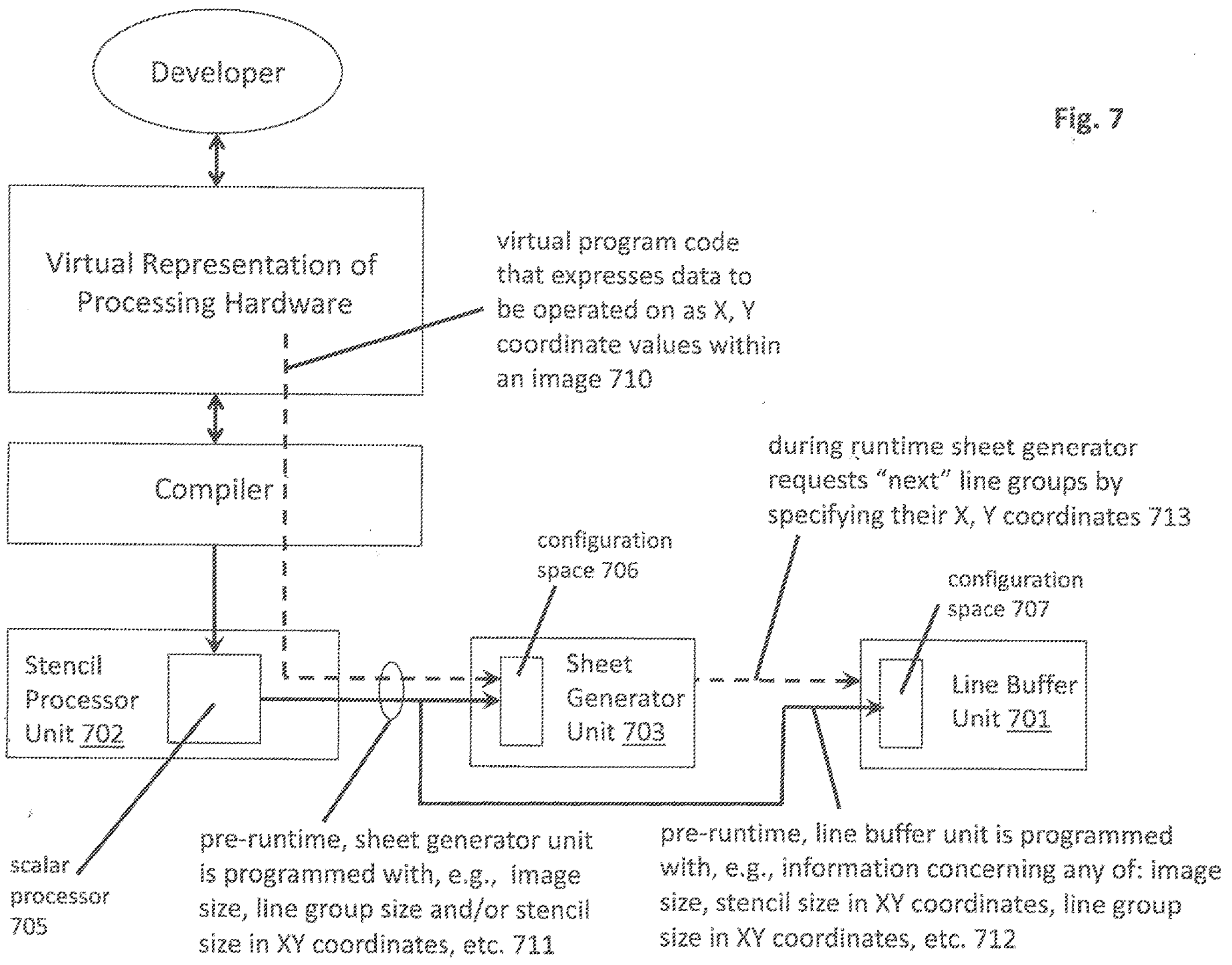

Developer
Virtual Representation of Processing Hardware
virtual program code that expresses data to be operated on as X, Y coordinate values within an image 710
Compiler
during runtime sheet generator requests "next" line groups by specifying their X, Y coordinates 713
configuration space 706
configuration space 707
Stencil Processor Unit 702
Sheet Generator Unit 703
Line Buffer Unit 701
scalar processor 705
pre-runtime, sheet generator unit is programmed with, e.g., image size, line group size and/or stencil size in XY coordinates, etc. 711
pre-runtime, line buffer unit is programmed with, e.g., information concerning any of: image size, stencil size in XY coordinates, line group size in XY coordinates, etc. 712

Fig. 7

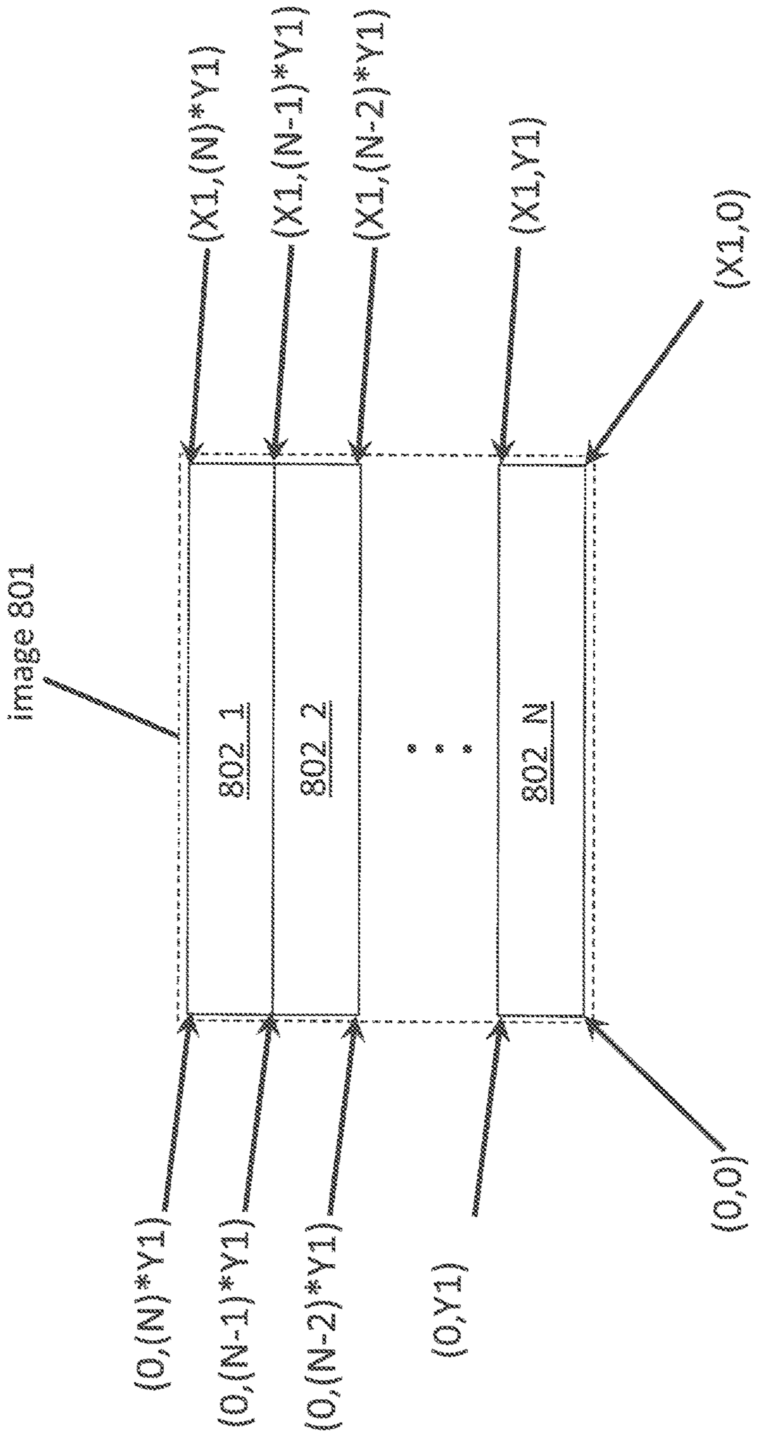


image 801
802_1
802_2
802_N
(X1,(N)*Y1)
(X1,(N-1)*Y1)
(X1,(N-2)*Y1)
(X1,Y1)
(X1,0)
(0,(N)*Y1)
(0,(N-1)*Y1)
(0,(N-2)*Y1)
(0Y1)
(0,0)

Fig. 8

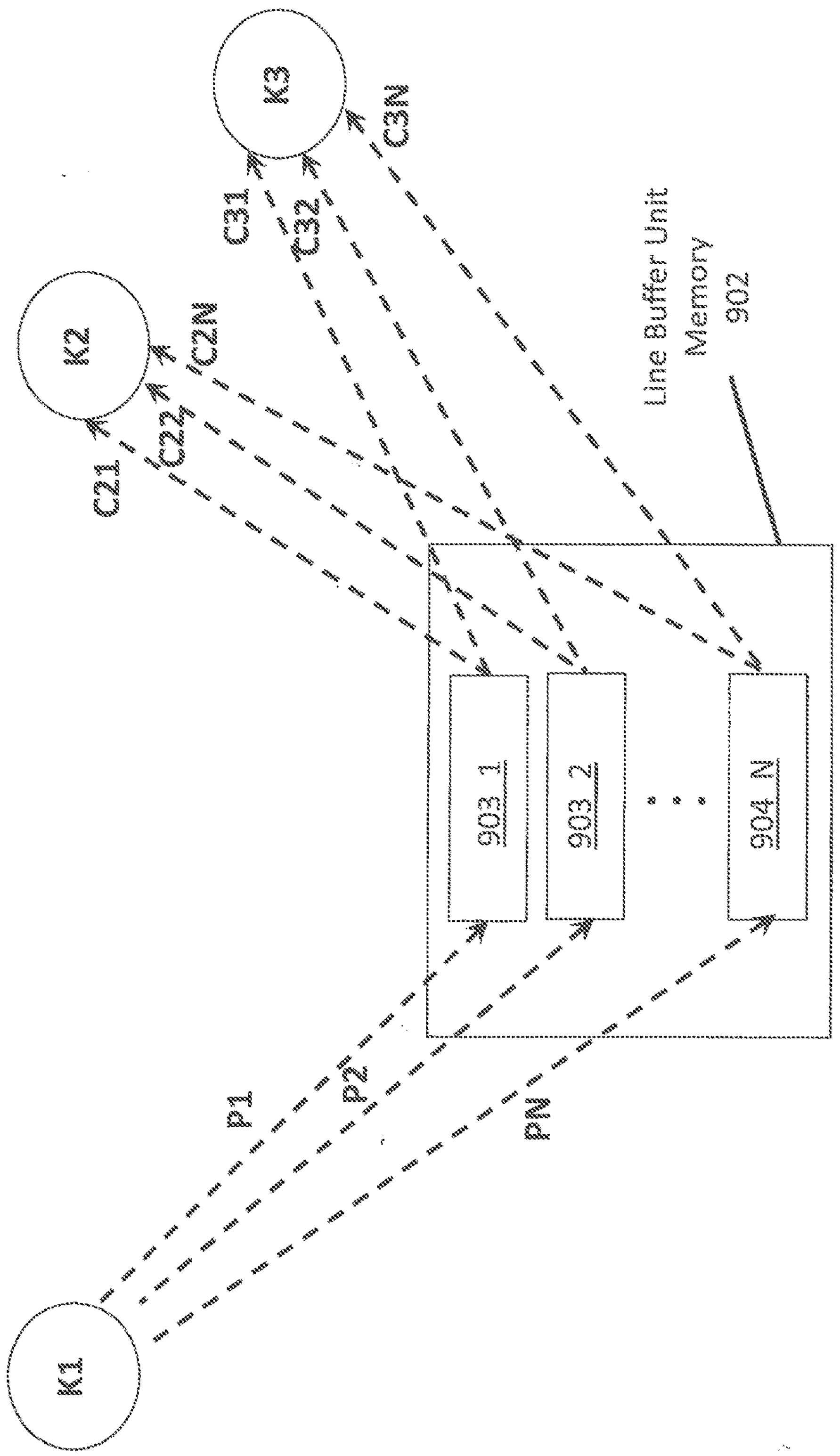

K1
K2
K3
P1
P2
PN
C21
C22
C2N
C31
C32
C3N
Line Buffer Unit Memory 902
903_1
903_2
904_N

Fig. 9a

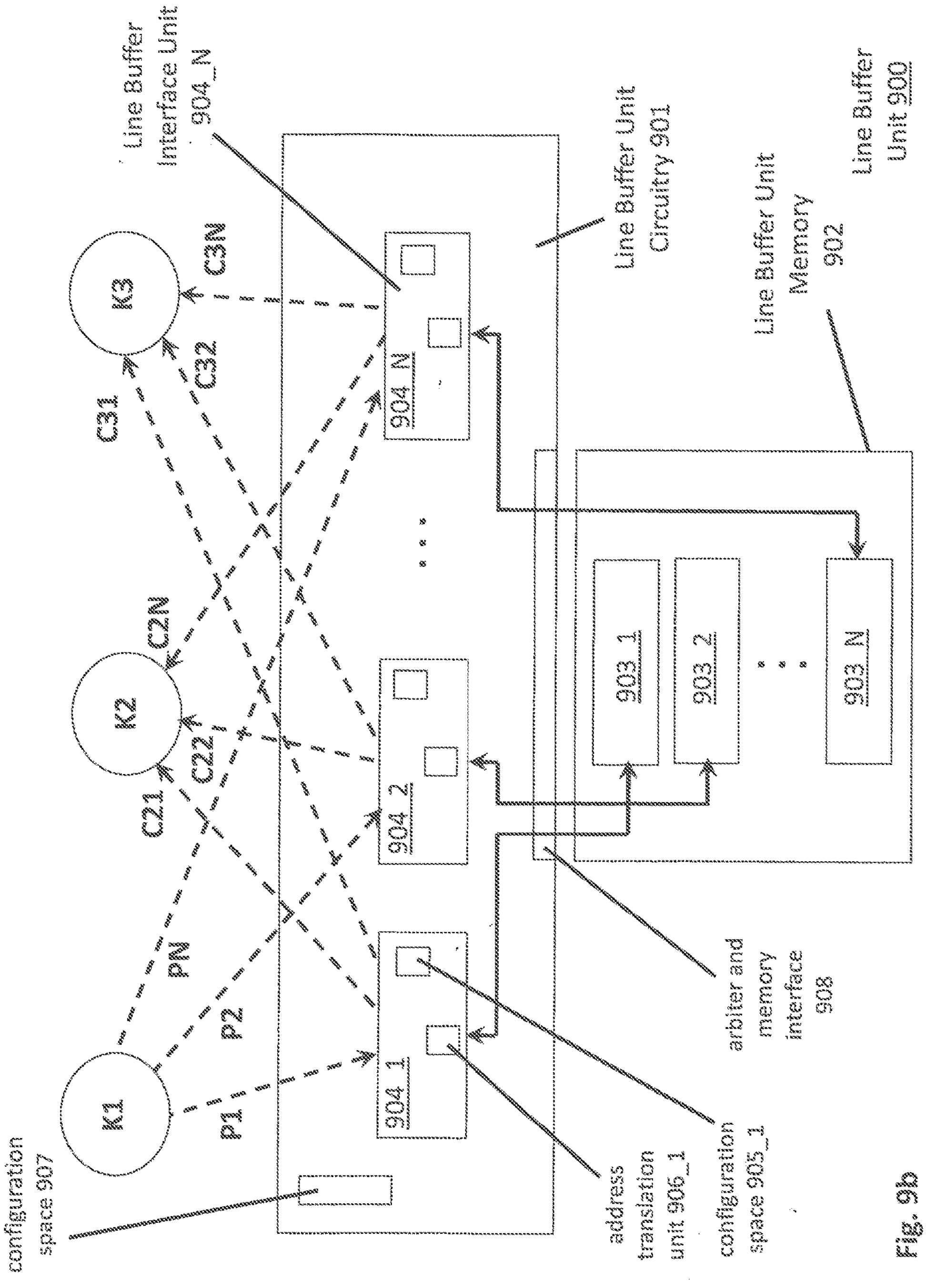

Line Buffer Interface Unit 904_N
Line Buffer Unit Circuitry 901
Line Buffer Unit Memory 902
Line Buffer Unit 900
K1
K2
K3
P1
P2
PN
C21
C22
C2N
C31
C32
C3N
904_1
904_2
904_N
903_1
903_2
903_N
configuration space 907
address translation unit 906_1
configuration space 905_1
arbiter and memory interface 908

Fig. 9b

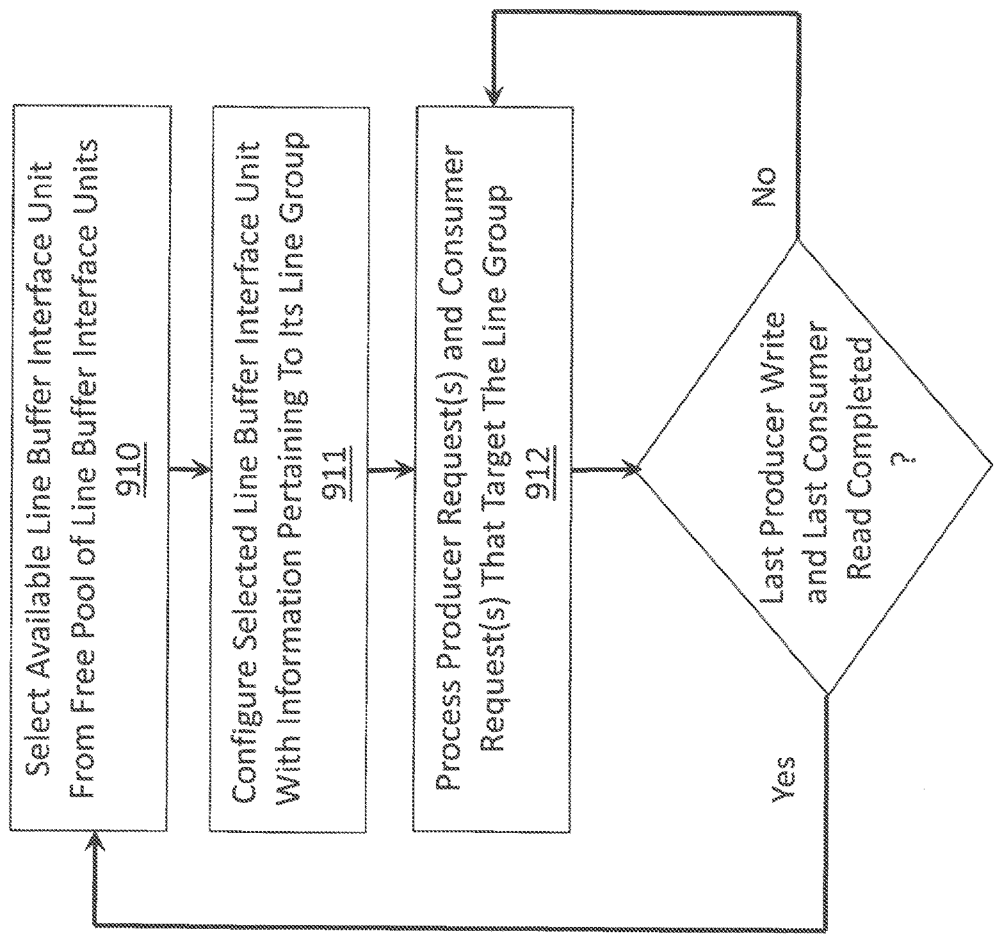

Select Available Line Buffer Interface Unit From Free Pool of Line Buffer Interface Units 910
Configure Selected Line Buffer Interface Unit With Information Pertaining To Its Line Group 911
Process Producer Request(s) and Consumer Request(s) That Target The Line Group 912
Last Producer Write and Last Consumer Read Completed ?
No
Yes

Fig. 9c

	Register	Description
921	LB_ENABLE	enable the line buffer interface unit
922	NUM_CHANNELS	number of channels in the line group being processed by the line buffer interface unit
923	NUM_CONSUMERS	number of consumer kernels that will access the line group being processed by the line buffer interface unit
924	ROW_WIDTH	number of pixels per row of the line group being processed by the line buffer interface unit
925	VB_ENABLE	enable virtually tall line buffer mode for the line group being processed by the line buffer interface unit
926	FB_ROWS	number of rows of a full line group for the line group being processed by the buffer interface unit
927	VB_ROWS	width of lower portions to be transferred in virtually tall line group mode
928	VB_COLS	height of lower portions to be transferred in virtually tall line group mode
929	NUM_REUSE_ROWS	number of reuse rows that have to be copied from virtually tall line group to full width buffer if virtually tall line group mode is enabled
930	FB_BASE_ADDRESS	base address in linear memory where the full width line group is stored
931	VB_BASE_ADDRESS	base address in linear memory where the virtually tall line group is stored

Fig. 9d

	Register	Description
932	NUM_CHANNELS	number of channels in the line group being processed by the line buffer interface unit
933	NUM_CONSUMERS	number of consumer kernels that will access the line group being processed by the line buffer interface unit
934	IMAGE_SIZE	the size of the image that line groups are constituent parts of
935	STENCIL_DIMENSION	the dimensions of the stencil that will process the line group

Fig. 9e

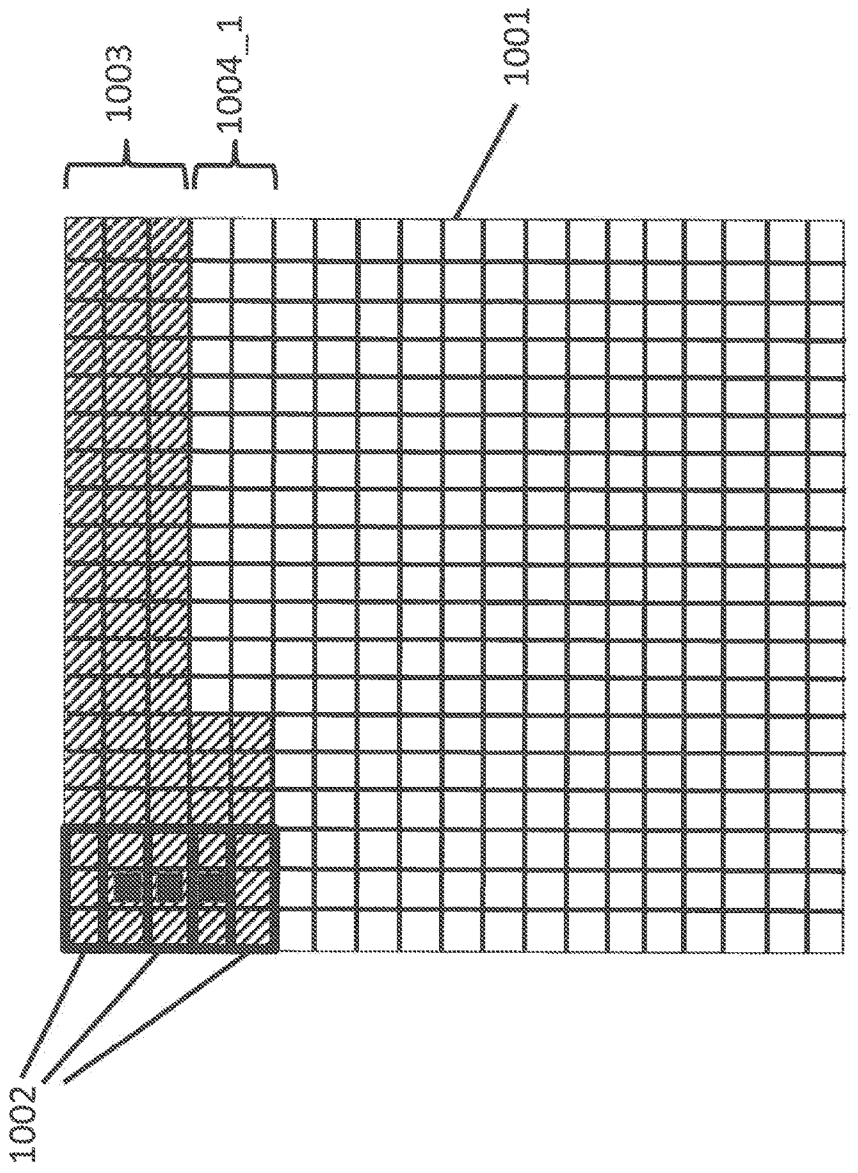


Fig. 10a

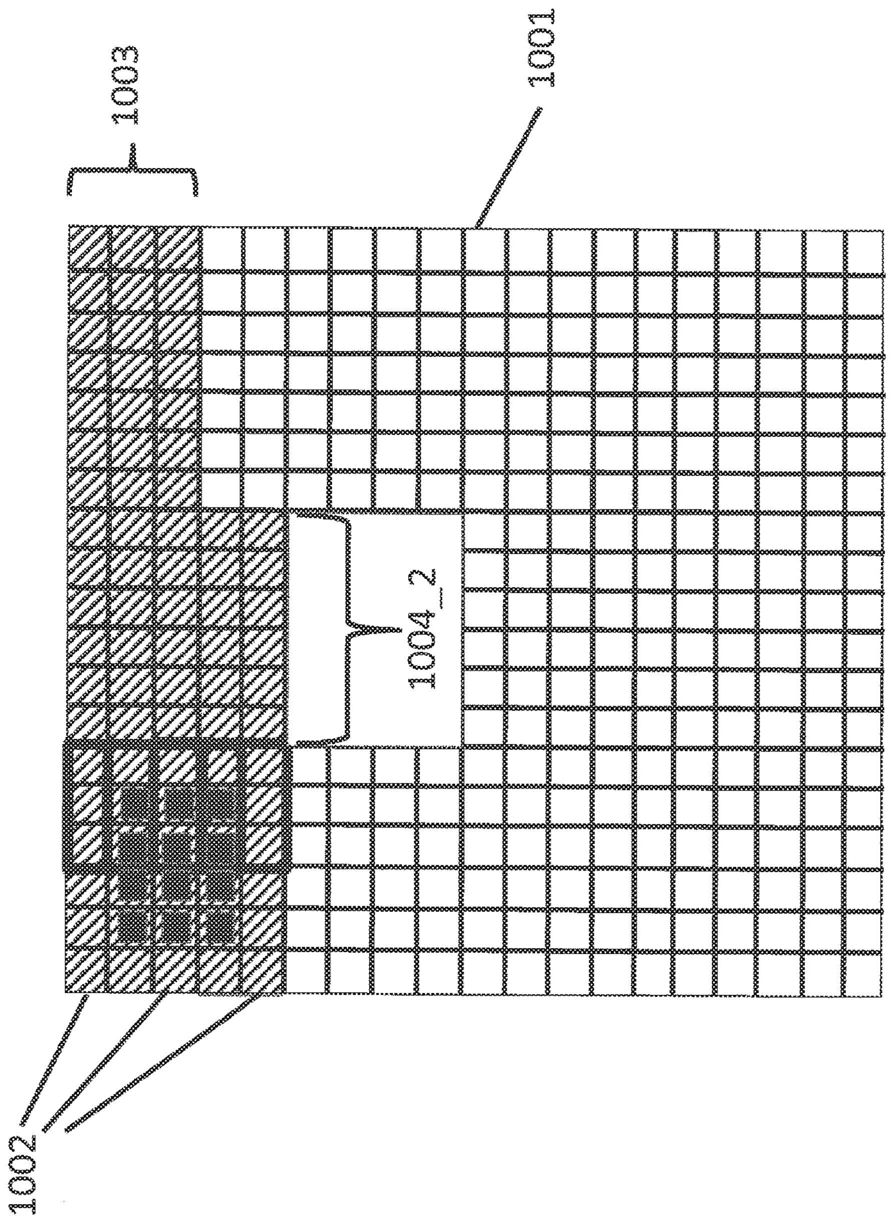


Fig. 10b

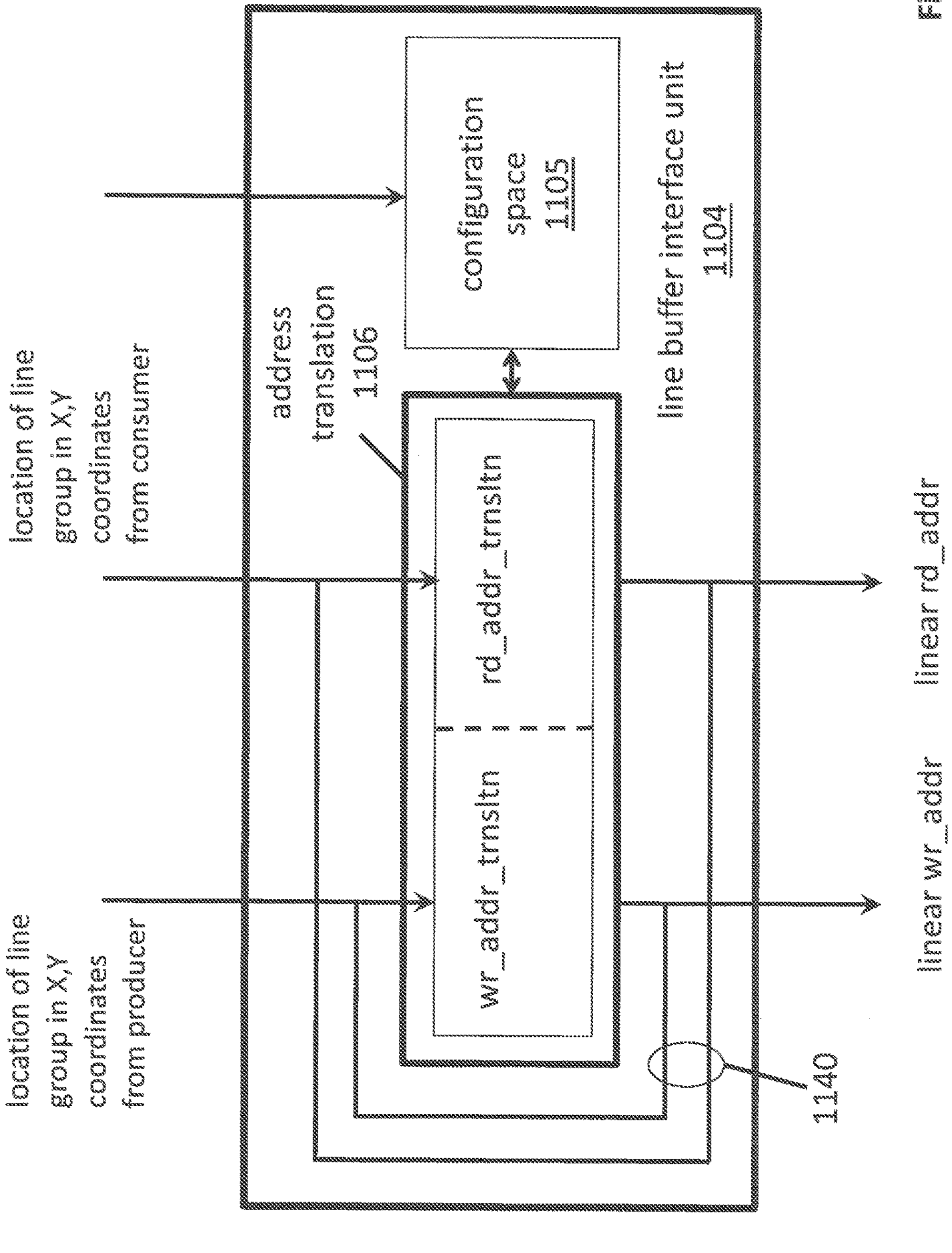

location of line group in X,Y coordinates from producer
location of line group in X,Y coordinates from consumer
address translation 1106
configuration space 1105
line buffer interface unit 1104
wr_addr_trnsltn
rd_addr_trnsltn
1140
linear wr_addr
linear rd_addr

Fig. 11a

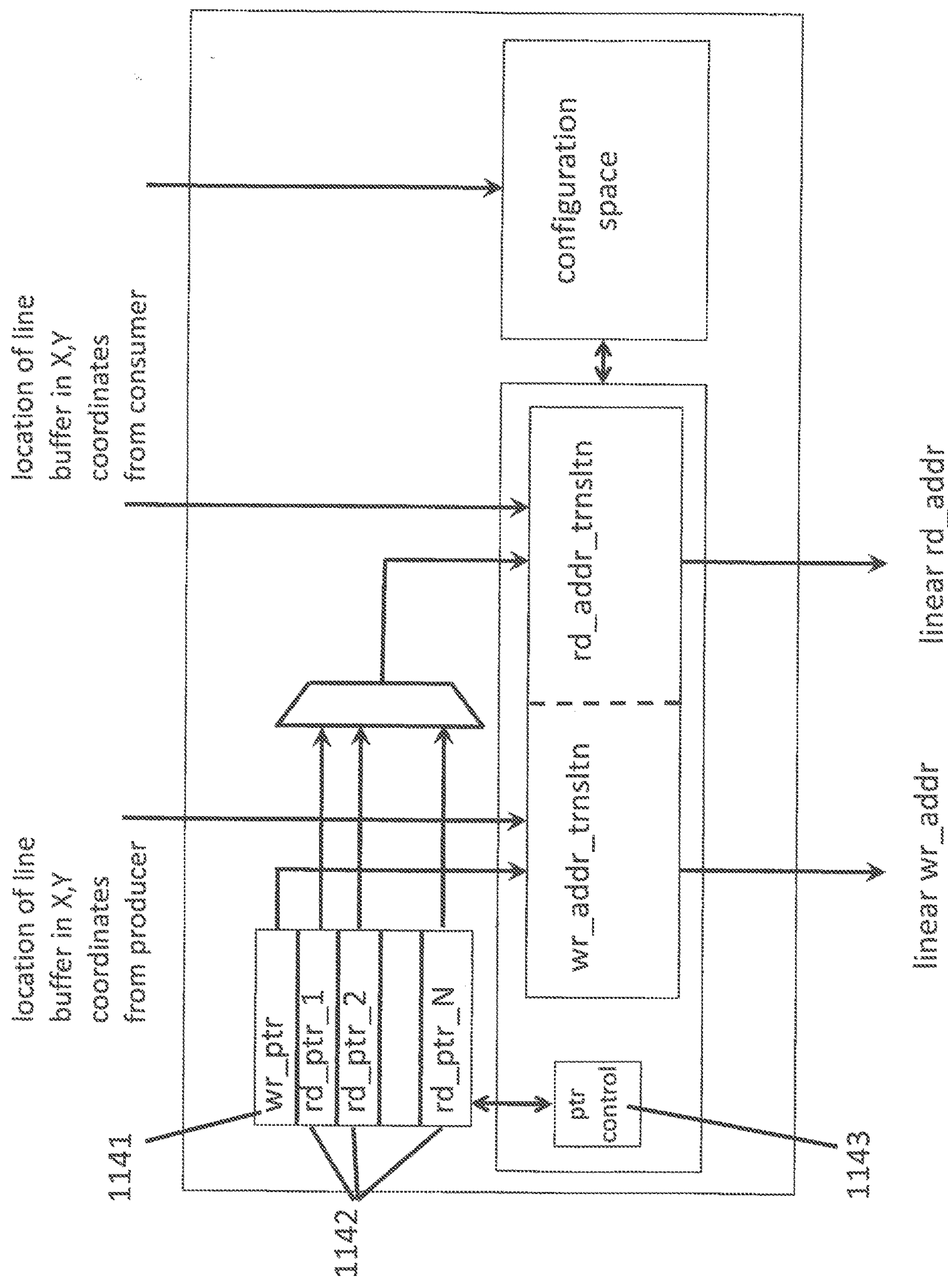

location of line buffer in X,Y coordinates from consumer
location of line buffer in X,Y coordinates from producer
configuration space
rd_addr_trnsltn
wr_addr_trnsltn
linear rd_addr
linear wr_addr
wr_ptr
rd_ptr_1
rd_ptr_2
rd_ptr_N
ptr control
1141
1142
1143

Fig. 11b

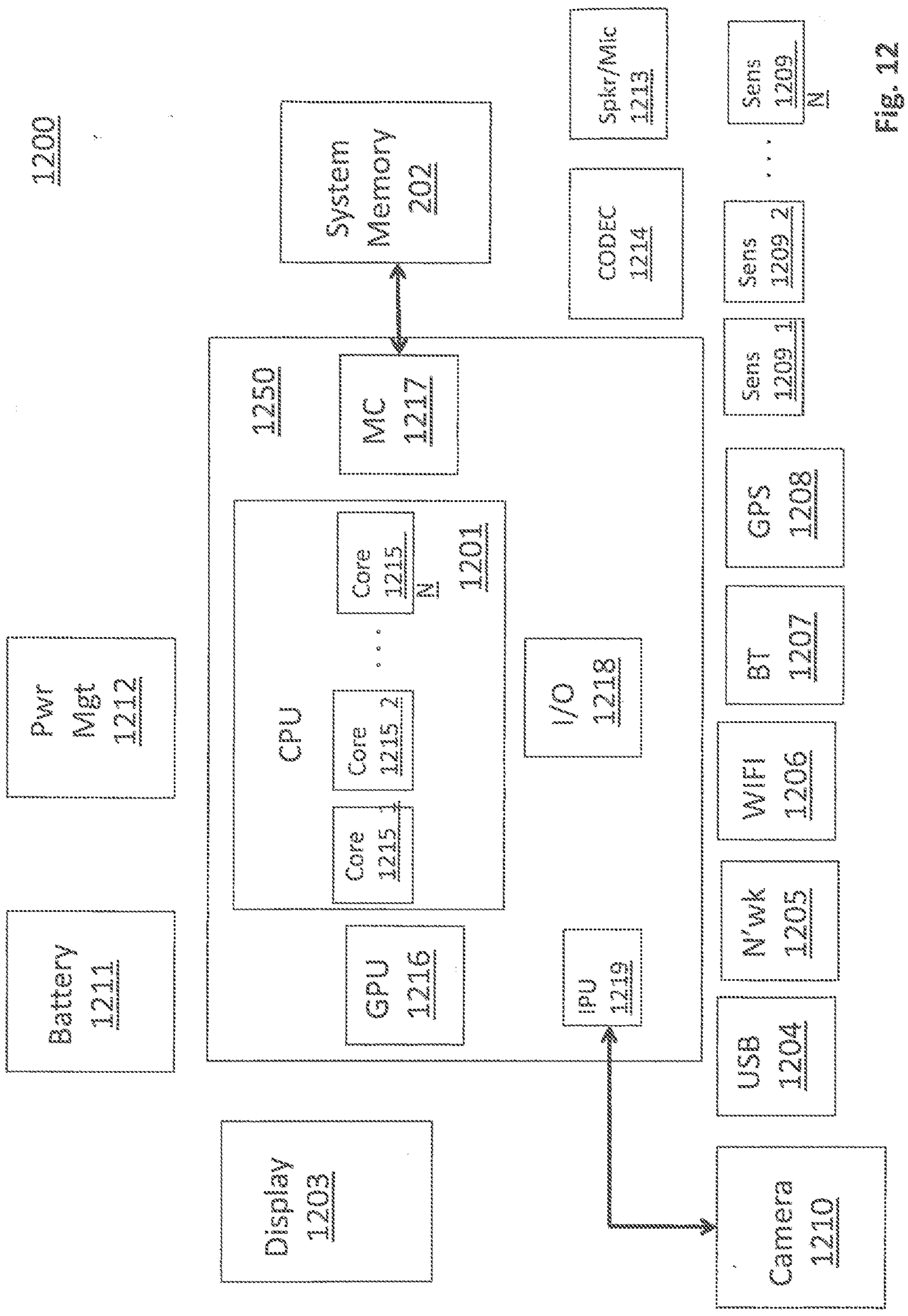

1200
System Memory 202
Spkr/Mic 1213
CODEC 1214
Sens 1209_1
Sens 1209_2
Sens 1209_N
1250
MC 1217
Pwr Mgt 1212
CPU
Core 1215_1
Core 1215_2
Core 1215_N
1201
I/O 1218
GPS 1208
BT 1207
WIFI 1206
N'wk 1205
USB 1204
Battery 1211
GPU 1216
IPU 1219
Display 1203
Camera 1210

Fig. 12

LINE BUFFER UNIT FOR IMAGE PROCESSOR

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application is a continuation of and claims priority to U.S. patent application Ser. No. 14/694,712, filed on Apr. 23, 2015, the entire contents of which are hereby incorporated by reference.

FIELD OF INVENTION

[0002] The field of invention pertains generally to image processing, and, more specifically, to a line buffer unit for an image processor.

BACKGROUND

[0003] Image processing typically involves the processing of pixel values that are organized into an array. Here, a spatially organized two dimensional array captures the two dimensional nature of images (additional dimensions may include time (e.g., a sequence of two dimensional images) and data type (e.g., colors). In a typical scenario, the arrayed pixel values are provided by a camera that has generated a still image or a sequence of frames to capture images of motion. Traditional image processors typically fall on either side of two extremes.

[0004] A first extreme performs image processing tasks as software programs executing on a general purpose processor or general purpose-like processor (e.g., a general purpose processor with vector instruction enhancements). Although the first extreme typically provides a highly versatile application software development platform, its use of finer grained data structures combined with the associated overhead (e.g., instruction fetch and decode, handling of on-chip and off-chip data, speculative execution) ultimately results in larger amounts of energy being consumed per unit of data during execution of the program code.

[0005] A second, opposite extreme applies fixed function hardwired circuitry to much larger blocks of data. The use of larger (as opposed to finer grained) blocks of data applied directly to custom designed circuits greatly reduces power consumption per unit of data. However, the use of custom designed fixed function circuitry generally results in a limited set of tasks that the processor is able to perform. As such, the widely versatile programming environment (that is associated with the first extreme) is lacking in the second extreme.

[0006] A technology platform that provides for both highly versatile application software development opportunities combined with improved power efficiency per unit of data remains a desirable yet missing solution.

SUMMARY

[0007] An apparatus is described that include a line buffer unit composed of a plurality of a line buffer interface units. Each line buffer interface unit is to handle one or more requests by a respective producer to store a respective line group in a memory and handle one or more requests by a respective consumer to fetch and provide the respective line group from memory. The line buffer unit has programmable storage space whose information establishes line group size so that different line group sizes for different image sizes are storable in memory.

LIST OF FIGURES

[0008] The following description and accompanying drawings are used to illustrate embodiments of the invention. In the drawings:

[0009] FIG. **1** shows various components of a technology platform;

[0010] FIG. **2***a* shows an embodiment of application software built with kernels;

[0011] FIG. **2***b* shows an embodiment of the structure of a kernel;

[0012] FIG. **3** shows an embodiment of the operation of a kernel;

[0013] FIG. **4** shows an embodiment of an image processor hardware architecture;

[0014] FIGS. **5***a*, **5***b*, **5***c*, **5***d* and **5***e* depict the parsing of image data into a line group, the parsing of a line group into a sheet and the operation performed on a sheet with overlapping stencils;

[0015] FIG. **6** shows an embodiment of a stencil processor;

[0016] FIG. **7** shows an embodiment of the configuration and programming of an image processor

[0017] FIG. **8** shows an image frame composed of line groups;

[0018] FIGS. **9***a*, **9***b* and **9***c* depict design and operational embodiments of a line buffer unit;

[0019] FIGS. **9***d* and **9***e* depict embodiments of programmable register space of an image processor;

[0020] FIGS. **10***a* and **10***b* depict a virtually tall mode of operation;

[0021] FIGS. **11***a* and **11***b* show line buffer interface unit embodiments;

[0022] FIG. **12** shows an embodiment of a computing system.

DETAILED DESCRIPTION

[0023] i. Introduction

[0024] The description below describes numerous embodiments concerning a new image processing technology platform that provides a widely versatile application software development environment that uses larger blocks of data (e.g., line groups and sheets as described further below) to provide for improved power efficiency.

1.0 Application Software Development Environment

[0025] a. Application and Structure of Kernels

[0026] FIG. **1** shows a high level view of an image processor technology platform that includes a virtual image processing environment **101**, the actual image processing hardware **103** and a compiler **102** for translating higher level code written for the virtual processing environment **101** to object code that the actual hardware **103** physically executes. As described in more detail below, the virtual processing environment **101** is widely versatile in terms of the applications that can be developed and is tailored for easy visualization of an application's constituent processes. Upon completion of the program code development effort by the developer **104**, the compiler **102** translates the code that was written within the virtual processing environment **101** into object code that is targeted for the actual hardware **103**.

[0027] FIG. **2***a* shows an example of the structure and form that application software written within the virtual environment may take. As observed in FIG. **2***a*, the program code may be expected to process one or more frames of input image data **201** to effect some overall transformation on the input image data **201**. The transformation is realized with the operation of one or more kernels of program code **202** that operate on the input image data in an orchestrated sequence articulated by the developer.

[0028] For example, as observed in FIG. **2***a*, the overall transformation is effected by first processing each input image with a first kernel K**1**. The output images produced by kernel K**1** are then operated on by kernel K**2**. Each of the output images produced by kernel K**2** are then operated on by kernel K**3_1** or K**3_2**, The output images produced by kernel(s) K**3_1**/K**3_2** are then operated on by kernel K**4**. Kernels K**3_1** and K**3_2** may be identical kernels designed to speed-up the overall processing by imposing parallel processing at the K**3** stage, or, may be different kernels (e.g., kernel K**3_1** operates on input images of a first specific type and kernel K**3_2** operates on input images of a second, different type).

[0029] As such, the larger overall image processing sequence may take the form of a image processing pipeline or a directed acyclic graph (DAG) and the development environment may be equipped to actually present the developer with a representation of the program code being developed as such. Kernels may be developed by a developer individually and/or may be provided by an entity that supplies any underlying technology (such as the actual signal processor hardware and/or a design thereof) and/or by a third party (e.g., a vendor of kernel software written for the development environment). As such, it is expected that a nominal development environment will include a "library" of kernels that developers are free to "hook-up" in various ways to effect the overall flow of their larger development effort. Some basic kernels that are expected to be part of such a library may include kernels to provide any one or more of the following basic image processing tasks: convolutions, denoising, color space conversions, edge and corner detection, sharpening, white balance, gamma correction, tone mapping, matrix multiply, image registration, pyramid construction, wavelet transformation, block-wise discrete cosine and Fourier transformations.

[0030] FIG. **2***b* shows an exemplary depiction of the structure of a kernel **203** as may be envisioned by a developer. As observed in FIG. **2***b*, the kernel **203** can be viewed as a number of parallel threads of program code ("threads") **204** that are each operating on a respective underlying processor **205** where each processor **205** is directed to a particular location in an output array **206** (such as a specific pixel location in the output image that the kernel is generating). For simplicity only three processors and corresponding threads are shown in FIG. **2***b*. In various embodiments, every depicted output array location would have its own dedicated processor and corresponding thread. That is, a separate processor and thread can be allocated for each pixel in the output array.

[0031] As will be described in more detail below, in various embodiments, in the actual underlying hardware an array of execution lanes and corresponding threads operate in unison (e.g., in a Single Instruction Multiple Data(s) like fashion) to generate output image data for a portion of a "line group" of the frame currently being processed. A line group is a contiguous, sizable section of an image frame. In various embodiments, the developer may be conscious the hardware operates on line groups, or, the development environment may present an abstraction in which there is a separate processor and thread for, e.g., every pixel in the output frame (e.g., every pixel in an output frame generated by its own dedicated processor and thread). Regardless, in various embodiment, the developer understands the kernel to include an individual thread for each output pixel (whether the output array is visualized is an entire output frame or a section thereof).

[0032] As will be described in more detail below, in an embodiment the processors **205** that are presented to the developer in the virtual environment have an instruction set architecture (ISA) that, not only supports standard (e.g., RISC) opcodes, but also include specially formatted data access instructions that permit the developer to easily visualize the pixel by pixel processing that is being performed. The ability to easily define/visualize any input array location in combination with an entire ISA of traditional mathematical and program control opcodes allows for an extremely versatile programming environment that essentially permits an application program developer to define, ideally, any desired function to be performed on any sized image surface. For example, ideally, any mathematical operation can be readily programmed to be applied to any stencil size.

[0033] With respect to the data access instructions, in an embodiment the ISA of the virtual processors ("virtual ISA") include a special data load instruction and a special data store instruction. The data load instruction is able to read from any location within an input array of image data. The data store instruction is able to write to any location within the output array of image data. The latter instruction allows for easily dedicating multiple instances of the same processor to different output pixel locations (each processor writes to a different pixel in the output array). As such, for example, stencil size itself (e.g., expressed as a width of pixels and a height of pixels) can be made an easily programmable feature. Visualization of the processing operations is further simplified with each of the special load and store instructions having a special instruction format whereby target array locations are specified simplistically as X and Y coordinates.

[0034] Regardless, by instantiating a separate processor for each of multiples locations in the output array, the processors can execute their respective threads in parallel so that, e.g., the respective values for all locations in the output array are produced concurrently. It is noteworthy that many image processing routines typically perform the same operations on different pixels of the same output image. As such, in one embodiment of the development environment, each processor is presumed to be identical and executes the same thread program code. Thus, the virtualized environment can be viewed as a type of two-dimensional (2D), SIMD processor composed of a 2D array of, e.g., identical processors each executing identical code in lock-step.

[0035] FIG. **3** shows a more detailed example of the processing environment for two virtual processors that are processing identical code for two different pixel locations in an output array. FIG. **3** shows an output array **304** that corresponds to an output image being generated. Here, a first virtual processor is processing the code of thread **301** to generate an output value at location X**1** of the output array **304** and a second virtual processor is processing the code of thread **302** to generate an output value at location X**2** of the output array **304**. Again, in various embodiments, the developer would understand there is a separate processor and thread for each pixel location in the output array **304** (for simplicity FIG. **3** only shows two of them). However, the developer in various embodiments need only develop code for one processor and thread (because of the SIMD like nature of the machine).

[0036] As is known in the art, an output pixel value is often determined by processing the pixels of an input array that include and surround the corresponding output pixel location. For example, as can be seen from FIG. **3**, position X**1** of the output array **304** corresponds to position E of the input array **303**. The stencil of input array **303** pixel values that would be processed to determine output value X**1** would therefore corresponds to input values ABCDEFGHI. Similarly, the stencil of input array pixels that would be processed to determine output value X**2** would corresponds to input values DEFGHIJKL.

[0037] FIG. **3** shows an example of corresponding virtual environment program code for a pair of threads **301**, **302** that could be used to calculate output values X**1** and X**2**, respectively. In the example of FIG. **3** both pairs of code are identical and average a stencil of nine input array values to determine a corresponding output value. The only difference between the two threads is the variables that are called up from the input array and the location of the output array that is written to. Specifically, the thread that writes to output location X**1** operates on stencil ABCDEFGHI and the thread that writes to output location X**2** operates on stencil DEFGHIJKL.

[0038] As can be seen from the respective program code from the pair of threads **301**, **302**, each virtual processor at least includes internal registers R**1** and R**2** and at least supports the following instructions: 1) a LOAD instruction from the input array into R**1**; 2) a LOAD instruction from the input array into R**2**; 3) an ADD instruction that adds the contents of R**1** and R**2** and places the resultant in R**2**; 4) a DIV instruction that divides the value within R**2** by immediate operand **9**; and, 5) a STORE instruction the stores the contents of R**2** into the output array location that the thread is dedicated to. Again, although only two output array locations and only two threads and corresponding processors are depicted in FIG. **3**, conceivably, every location in the output array could be assigned a virtual processor and corresponding thread that performs these functions. In various embodiments, in keeping with the SIMD-like nature of the processing environment, the multiple threads execute in isolation of one another. That is, there is no thread-to-thread communication between virtual processors (one SIMD channel is preventing from crossing into another SIMD channel).

b. Virtual Processor Memory Model

[0039] In various embodiments, a pertinent feature of the virtual processors is their memory model. As is understood in the art, a processor reads data from memory, operates on that data and writes new data back into memory. A memory model is the perspective or view that a processor has of the manner in which data is organized in memory. In an embodiment, the memory model of the virtual processors includes both input and output array regions. Input pixel values for threads are stored in the input array region and output pixel values generated by threads are stored in the output array region.

[0040] In an embodiment, a novel memory addressing scheme is used to define which particular input values are to be called in from an input array portion of the virtual processor's memory model. Specifically, a "position relative" addressing scheme is used that defines the desired input data with X, Y coordinates rather than a traditional linear memory address. As such, the load instruction of the virtual processors' ISA includes an instruction format that identifies a specific memory location within the input array with an X component and a Y component. As such, a two-dimensional coordinate system is used to address memory for input values read from the input array.

[0041] The use of a position relative memory addressing approach permits the region of an image that a virtual processor is operating on to be more readily identifiable to a developer. As mentioned above, the ability to easily define/visualize any input array location in combination with an entire ISA of traditional mathematical and program control opcodes allows for an extremely versatile programming environment that essentially permits an application program developer to readily define, ideally, any desired function to be performed on any sized image surface. Various instruction format embodiments for instructions that adopt a position relative addressing scheme, as well as embodiments of other features of the supported ISA, are described in more detail further below.

[0042] The output array contains the output image data that the threads are responsible for generating. The output image data may be final image data such as the actual image data that is presented on a display that follows the overall image processing sequence, or, may be intermediate image data that a subsequent kernel of the overall image processing sequence uses as its input image data information. Again, typically virtual processors do not compete for same output data items because they write to different pixel locations of the output image data during a same cycle.

[0043] In an embodiment, the position relative addressing scheme is also used for writes to the output array. As such, the ISA for each virtual processor includes a store instruction whose instruction format defines a targeted write location in memory as a two-dimensional X, Y coordinate rather than a traditional random access memory address.

2.0 Hardware Architecture Embodiments

[0044] a. Image Processor Hardware Architecture and Operation

[0045] FIG. **4** shows an embodiment of an architecture **400** for an image processor implemented in hardware. The image processor may be targeted, for example, by a compiler that converts program code written for a virtual processor within a simulated environment into program code that is actually executed by the hardware processor. As observed in FIG. **4**, the architecture **400** includes a plurality of line buffer units **401_1** through **401**_M interconnected to a plurality of stencil processor units **402_1** through **402**_N and corresponding sheet generator units **403_1** through **403**_N through a network **404** (e.g., a network on chip (NOC) including an on chip switch network, an on chip ring network or other kind of network). In an embodiment, any line buffer unit may connect to any sheet generator and corresponding stencil processor through the network **404**.

[0046] In an embodiment, program code is compiled and loaded onto a corresponding stencil processor **402** to perform the image processing operations earlier defined by a software developer (program code may also be loaded onto the stencil processor's associated sheet generator **403**, e.g., depending on design and implementation). In at least some instances an image processing pipeline may be realized by loading a first kernel program for a first pipeline stage into a first stencil processor **402_1**, loading a second kernel program for a second pipeline stage into a second stencil processor **402_2**, etc. where the first kernel performs the functions of the first stage of the pipeline, the second kernel performs the functions of the second stage of the pipeline, etc. and additional control flow methods are installed to pass output image data from one stage of the pipeline to the next stage of the pipeline.

[0047] In other configurations, the image processor may be realized as a parallel machine having two or more stencil processors **402_1**, **402_2** operating the same kernel program code. For example, a highly dense and high data rate stream of image data may be processed by spreading frames across multiple stencil processors each of which perform the same function.

[0048] In yet other configurations, essentially any DAG of kernels may be loaded onto the hardware processor by configuring respective stencil processors with their own respective kernel of program code and configuring appropriate control flow hooks into the hardware to direct output images from one kernel to the input of a next kernel in the DAG design.

[0049] As a general flow, frames of image data are received by a macro I/O unit **405** and passed to one or more of the line buffer units **401** on a frame by frame basis. A particular line buffer unit parses its frame of image data into a smaller region of image data, referred to as a "a line group", and then passes the line group through the network **404** to a particular sheet generator. A complete or "full" singular line group may be composed, for example, with the data of multiple contiguous complete rows or columns of a frame (for simplicity the present specification will mainly refer to contiguous rows). The sheet generator further parses the line group of image data into a smaller region of image data, referred to as a "sheet", and presents the sheet to its corresponding stencil processor.

[0050] In the case of an image processing pipeline or a DAG flow having a single input, generally, input frames are directed to the same line buffer unit **401_1** which parses the image data into line groups and directs the line groups to the sheet generator **403_1** whose corresponding stencil processor **402_1** is executing the code of the first kernel in the pipeline/DAG. Upon completion of operations by the stencil processor **402_1** on the line groups it processes, the sheet generator **403_1** sends output line groups to a "downstream" line buffer unit **401_2** (in some use cases the output line group may be sent_back to the same line buffer unit **401_1** that earlier had sent the input line groups).

[0051] One or more "consumer" kernels that represent the next stage/operation in the pipeline/DAG executing on their own respective other sheet generator and stencil processor (e.g., sheet generator **403_2** and stencil processor **402_2**) then receive from the downstream line buffer unit **401_2** the image data generated by the first stencil processor **402_1**. In this manner, a "producer" kernel operating on a first stencil processor has its output data forwarded to a "consumer" kernel operating on a second stencil processor where the consumer kernel performs the next set of tasks after the producer kernel consistent with the design of the overall pipeline or DAG.

[0052] A stencil processor **402** is designed to simultaneously operate on multiple overlapping stencils of image data. The multiple overlapping stencils and internal hardware processing capacity of the stencil processor effectively determines the size of a sheet. Here, within a stencil processor **402**, arrays of execution lanes operate in unison to simultaneously process the image data surface area covered by the multiple overlapping stencils.

[0053] As will be described in more detail below, in various embodiments, sheets of image data are loaded into a two-dimensional register array structure within the stencil processor **402**. The use of sheets and the two-dimensional register array structure is believed to effectively provide for power consumption improvements by moving a large amount of data into a large amount of register space as, e.g., a single load operation with processing tasks performed directly on the data immediately thereafter by an execution lane array. Additionally, the use of an execution lane array and corresponding register array provide for different stencil sizes that are easily programmable/configurable.

[0054] FIGS. **5*a*** through **5*e*** illustrate at a high level embodiments of both the parsing activity of a line buffer unit **401**, the finer grained parsing activity of a sheet generator unit **403** as well as the stencil processing activity of the stencil processor **402** that is coupled to the sheet generator unit **403**.

[0055] FIG. **5*a*** depicts an embodiment of an input frame of image data **501**. FIG. **5*a*** also depicts an outline of three overlapping stencils **502** (each having a dimension of **3** pixels x **3** pixels) that a stencil processor is designed to operate over. The output pixel that each stencil respectively generates output image data for is highlighted in solid black. For simplicity, the three overlapping stencils **502** are depicted as overlapping only in the vertical direction. It is pertinent to recognize that in actuality a stencil processor may be designed to have overlapping stencils in both the vertical and horizontal directions.

[0056] Because of the vertical overlapping stencils **502** within the stencil processor, as observed in FIG. **5*a***, there exists a wide band of image data within the frame that a single stencil processor can operate over. As will be discussed in more detail below, in an embodiment, the stencil processors process data within their overlapping stencils in a left to right fashion across the image data (and then repeat for the next set of lines, in top to bottom order). Thus, as the stencil processors continue forward with their operation, the number of solid black output pixel blocks will grow right-wise horizontally. As discussed above, a line buffer unit **401** is responsible for parsing a line group of input image data from an incoming frame that is sufficient for the stencil processors to operate over for an extended number of upcoming cycles. An exemplary depiction of a line group is illustrated as a shaded region **503**. In an embodiment, as described further below, the line buffer unit **401** can comprehend different dynamics for sending/receiving a line group to/from a sheet generator. For example, according to one mode, referred to as "full group", the complete full width lines of image data are passed between a line buffer unit and a sheet generator. According to a second mode, referred to as "virtually tall", a line group is passed initially with a subset of full width rows. The remaining rows are then passed sequentially in smaller (less than full width) pieces.

[0057] With the line group **503** of the input image data having been defined by the line buffer unit and passed to the sheet generator unit, the sheet generator unit further parses the line group into finer sheets that are more precisely fitted to the hardware limitations of the stencil processor. More specifically, as will be described in more detail further below, in an embodiment, each stencil processor consists of a two dimensional shift register array. The two dimensional shift register array essentially shifts image data "beneath" an array of execution lanes where the pattern of the shifting causes each execution lane to operate on data within its own respective stencil (that is, each execution lane processes on its own stencil of information to generate an output for that stencil). In an embodiment, sheets are surface areas of input image data that "fill" or are otherwise loaded into the two dimensional shift register array.

[0058] Thus, as observed in FIG. **5***b*, the sheet generator parses an initial sheet **504** from the line group **503** and provides it to the stencil processor (here, the sheet of data corresponds to the shaded region that is generally identified by reference number **504**). As observed in FIGS. **5***c* and **5***d*, the stencil processor operates on the sheet of input image data by effectively moving the overlapping stencils **502** in a left to right fashion over the sheet. As of FIG. **5***d*, the number of pixels for which an output value could be calculated from the data within the sheet is exhausted (no other pixel positions can have an output value determined from the information within the sheet). For simplicity the border regions of the image have been ignored.

[0059] As observed in FIG. **5***e* the sheet generator then provides a next sheet **505** for the stencil processor to continue operations on. Note that the initial positions of the stencils as they begin operation on the next sheet is the next progression to the right from the point of exhaustion on the first sheet (as depicted previously in FIG. **5***d*). With the new sheet **505**, the stencils will simply continue moving to the right as the stencil processor operates on the new sheet in the same manner as with the processing of the first sheet.

[0060] Note that there is some overlap between the data of the first sheet **504** and the data of the second sheet **505** owing to the border regions of stencils that surround an output pixel location. The overlap could be handled simply by the sheet generator re-transmitting the overlapping data twice. In alternate implementations, to feed a next sheet to the stencil processor, the sheet generator may proceed to only send new data to the stencil processor and the stencil processor reuses the overlapping data from the previous sheet.

b. Stencil Processor Design and Operation

[0061] FIG. **6** shows an embodiment of a stencil processor architecture **600**. As observed in FIG. **6**, the stencil processor includes a data computation unit **601**, a scalar processor **602** and associated memory **603** and an I/O unit **604**. The data computation unit **601** includes an array of execution lanes **605**, a two-dimensional shift array structure **606** and separate random access memories **607** associated with specific rows or columns of the array.

[0062] The I/O unit **604** is responsible for loading "input" sheets of data received from the sheet generator into the data computation unit **601** and storing "output" sheets of data from the stencil processor into the sheet generator. In an embodiment the loading of sheet data into the data computation unit **601** entails parsing a received sheet into rows/columns of image data and loading the rows/columns of image data into the two dimensional shift register structure **606** or respective random access memories **607** of the rows/columns of the execution lane array (described in more detail below). If the sheet is initially loaded into memories **607**, the individual execution lanes within the execution lane array **605** may then load sheet data into the two-dimensional shift register structure **606** from the random access memories **607** when appropriate (e.g., as a load instruction just prior to operation on the sheet's data). Upon completion of the loading of a sheet of data into the register structure **606** (whether directly from a sheet generator or from memories **607**), the execution lanes of the execution lane array **605** operate on the data and eventually "write back" finished data as a sheet directly back to the sheet generator, or, into the random access memories **607**. If the later the I/O unit **604** fetches the data from the random access memories **607** to form an output sheet which is then forwarded to the sheet generator.

[0063] The scalar processor **602** includes a program controller **609** that reads the instructions of the stencil processor's program code from scalar memory **603** and issues the instructions to the execution lanes in the execution lane array **605**. In an embodiment, a single same instruction is broadcast to all execution lanes within the array **605** to effect a SIMD-like behavior from the data computation unit **601**. In an embodiment, the instruction format of the instructions read from scalar memory **603** and issued to the execution lanes of the execution lane array **605** includes a very-long-instruction-word (VLIW) type format that includes more than one opcode per instruction. In a further embodiment, the VLIW format includes both an ALU opcode that directs a mathematical function performed by each execution lane's ALU (which, as described below, in an embodiment may specify more than one traditional ALU operation) and a memory opcode (that directs a memory operation for a specific execution lane or set of execution lanes).

[0064] The term "execution lane" refers to a set of one or more execution units capable of executing an instruction (e.g., logic circuitry that can execute an instruction). An execution lane can, in various embodiments, include more processor-like functionality beyond just execution units, however. For example, besides one or more execution units, an execution lane may also include logic circuitry that decodes a received instruction, or, in the case of more MIMD-like designs, logic circuitry that fetches and decodes an instruction. With respect to MIMD-like approaches, although a centralized program control approach has largely been described herein, a more distributed approach may be implemented in various alternative embodiments (e.g., including program code and a program controller within each execution lane of the array **605**).

[0065] The combination of an execution lane array **605**, program controller **609** and two dimensional shift register structure **606** provides a widely adaptable/configurable hardware platform for a broad range of programmable functions. For example, application software developers are able to program kernels having a wide range of different functional capability as well as dimension (e.g., stencil size) given that the individual execution lanes are able to perform a wide variety of functions and are able to readily access input image data proximate to any output array location.

[0066] Apart from acting as a data store for image data being operated on by the execution lane array **605**, the random access memories **607** may also keep one or more look-up tables. In various embodiments one or more scalar look-up tables may also be instantiated within the scalar memory **603**.

[0067] A scalar look-up involves passing the same data value from the same look-up table from the same index to each of the execution lanes within the execution lane array **605**. In various embodiments, the VLIW instruction format described above is expanded to also include a scalar opcode that directs a look-up operation performed by the scalar processor into a scalar look-up table. The index that is specified for use with the opcode may be an immediate operand or fetched from some other data storage location. Regardless, in an embodiment, a look-up from a scalar look-up table within scalar memory essentially involves broadcasting the same data value to all execution lanes within the execution lane array **605** during a the same clock cycle.

3.0 Line Buffer Unit Embodiments

[0068] a. Line Buffer Unit Overview

[0069] Recall from the discussion above in Section **1.0** that in various embodiments, program code that is written for the hardware platform is written with a unique virtual code that includes an instruction set having load and store instructions whose instruction format identifies input and output array locations as, e.g., X,Y coordinates. In various implementations, the X,Y coordinate information may actually be programmed into the hardware platform and recognized/understood by various ones of its components. This stands apart from, for example, translating the X,Y coordination (e.g., within the compiler) into different information. For example, in the case of the two-dimensional shift register structure within the stencil processor, the X,Y coordinate information is translated into register shift movements. By contrast, other parts of the hardware platform may specifically receive and comprehend the X,Y coordinate information originally expressed at the higher, virtual code level.

[0070] As observed in FIG. **7**, as described in Section 1.0, a program code developer expresses data locations as X,Y coordinates with the special instruction format at the virtual code level **710**. During the compilation stage, the virtual code is translated into program code that is actually processed by the hardware (object code) and corresponding configuration information that is loaded into the hardware's configuration (e.g., register) space. As observed in FIG. **7**, in an embodiment, the object code for a particular kernel is loaded into the program space of the stencil processor's scalar processor **705**.

[0071] As part of the configuration process, configuration software executing on the scalar processor **705** loads the appropriate configuration information **711**, **712** into both the sheet generator unit **703** that is coupled to the stencil processor **702**, and, the line buffer unit **701** that will generate new sheets for the stencil processor **702** to operate on, or, receive processed sheets generated by the stencil processor **702**. Here, generally, sheets can still be contemplated in terms of X,Y coordinates of an overall image. That is, once an image or frame is defined (e.g., in terms of number of pixels per row, number of rows, number of pixels per column and number of columns), any portion or position of the image can still be referred to with X,Y coordinates.

[0072] As such, in various embodiments, either or both of the sheet generator unit **703** and line buffer unit **701** are configured with information **711**, **712** within their respective configuration space **706**, **707** that establishes an informational platform from which specific locations and/or regions (e.g., line groups, sheets) of an image or frame are identified in X,Y coordinates. In various implementations/uses, the X,Y coordinates may be the same X,Y coordinates expressed at the virtual code level.

[0073] Examples of such information include, e.g., number of active line groups in the line buffer unit, image size for each line group (e.g., as a set of four X, Y coordinates (one for each corner) or a pair of X, Y coordinates (one for a lower nearer corner and one for an upper farther corner)), absolute image width and image height, stencil size (expressed as X, Y values that define the size of a single stencil and/or the area of the overlapping stencils of the stencil processor), sheet and/or line group size (e.g., specified in same terms as an image size but having smaller dimensions), etc. Additionally, the line buffer unit **701** at least may be programmed with additional configuration information such as the number of producer kernels writing and the number of consumer kernels reading the line groups that are managed by the line buffer unit **701**. The number of channels and/or the dimensions associated with the image data are also typically included as configuration information.

[0074] FIG. **8** depicts the use of X,Y coordinates to define, as just one example, line groups within an image. Here, N line groups **801_1**, **801_2**, . . . **801**_N are observable within an image **801**. As can be seen from FIG. **8**, each line group can be readily defined by reference to X, Y coordinates within the image that define, e.g., one or more of a line group's corner points. As such, in various embodiments, a line group's name or other data structure used to define a particular line group may include X, Y coordinate locations associated with the line group in order to particularly identify it.

[0075] Referring briefly back to FIG. **7**, note that FIG. **7** shows that during runtime, a sheet generator **703** may request a "next" line group (or portion of a line group) from the line buffer unit **701** by, e.g., including X, Y coordinate information that defines the desired data region. FIG. **8** shows nominal "full width" line groups composed only of complete rows of image data. In an alternative configuration referred to as "virtually-tall", described in more detail further below, the line buffer unit **701** initially passes only a first upper portion of a line group as full width rows of image data. The subsequent lower rows of the line group are then specifically requested for by the sheet generator in contiguous chunks that are less than a full width row and are separately requested for. As such, multiple requests are made by the sheet generator in order to obtain the full line group. Here, each such request may define a next lower portion by X, Y coordinates that are attributable to the next lower portion.

[0076] FIGS. **9***a* through **9***c* demonstrate various features of a line buffer unit embodiment **900**. As observed in FIG. **9***a*, a line buffer unit includes memory **902** in which line groups **903_1** through **903**_N are stored (e.g., static or dynamic random access memory (SRAM or DRAM)). FIG. **9***a* shows the activity between the various kernels that produce and consume the line groups **903_1** through **903**_N for a particular image/frame within the memory **902**.

[0077] As observed in FIG. **9***a*, a producer kernel K1 sends new line groups to the memory **902** over separate time instances P1, P2 through PN. The producer kernel K1 executes on a stencil processor that generates new sheets of data. The sheet generator that is coupled to the stencil processor accumulates sheets to form line groups and forwards the line groups to the memory **902**.

[0078] Also as depicted in FIG. **9***a*, there are two consumer kernels K**2**, K**3** that operate on the line groups **903_1** through **903**_N generated by producer kernel K**1**. Here, consumer kernels K**2** and K**3** receive the first line group **903_1** at times C**21** and C**31**, respectively. Obviously, times C**21** and C**31** occur after time P**1**. Other restrictions may not exist. For example times C**21** and/or C**31** may occur before or after any of times P**2** through PN. Here, the respective sheet generators for kernels K**2** and K**3** request a next line group at a time that is appropriate for their respective kernel. If any of kernels K**2**, K**3** request line group **903_1** before time P**1**, the request idles until after line group **903_1** is actually written into memory **902**. In many implementations, a producer kernel operates on a different stencil processor than a consumer kernel.

[0079] Conceivably, requests from either or both of kernels K**2** and K**3** for all of line groups **903_1** through **903**_N may arrive prior to time P**1**. Thus, line groups may be requested by consumer kernels at any time. The line groups are forwarded to the consumer kernels as they request them subject, however, to the rate at which the producer kernel K**1** can produce them. In various embodiments, consumer kernels request line groups in sequence and likewise receive them in sequence (kernel K**2** receives line groups **902_2** through **902**_N at times C**22** through C**2**N in sequence). For simplicity only one producer kernel is depicted for a particular line group. It is conceivable that various embodiments may be designed to permit different producers to write to a same line group (e.g., where consumers are not permitted to be serviced until after all producers have written to the line group).

[0080] In cases where there is no producer kernel (because the consumer kernel(s) is/are the first kernels in the processor's DAG processing flow), frames of image data may be transferred into memory **902** (e.g., via direct memory access (DMA) or from a camera) and parsed into line groups. In cases where there are no consumer kernel(s) (because the producer kernel is the last kernel in the processor's overall program flow), resultant line groups may be combined to form output frames.

[0081] FIG. **9***b* shows a more detailed embodiment of an entire line buffer unit **900**. For the sake of discussion, the activity of FIG. **9***a* is superimposed on the line buffer unit **900** of FIG. **9***b*. As can be seen in FIG. **9***b*, a line buffer unit **900** includes memory **902** coupled to line buffer unit circuitry **901**. Line buffer unit circuitry **901** may be constructed, for example, with dedicated logic circuitry. Within line buffer unit circuitry **901**, a line buffer interface unit **904_1** through **904**_N is reserved for each line group **903_1** through **903**_N within memory **902**. In various embodiments, there is a fixed number of line buffer interface units **904_1** through **904**_N which sets an upper limit on the number of line groups that a line buffer unit can manage at any instant of time (if fewer than N line groups are active, a corresponding smaller number of line buffer unit interfaces are activated and in use at any time).

[0082] As depicted in FIG. **9***b*, with a total number of N line buffer interface units **904** within the line buffer unit circuitry **901**, the line buffer unit **900** is handling a maximum number of line groups. Additionally, with a largest permitted line group size (where line group size is a configurable parameter) an approximate size for memory **902** can be determined (of course, to allow for hardware efficiencies a smaller memory footprint may be instantiated at the cost of not simultaneously permitting N maximum sized line groups).

[0083] Each line buffer interface unit **904_1** through **904**_N is responsible for handling the producer and consumer requests for a particular line group that it has been assigned to handle. For example, line buffer interface unit **904_1** handles the request from producer K**1** at time P**1** to store line group **903_1** as well as handles the requests from consumer kernels K**2** and K**3** for line group **903_1**. In response to the former, line buffer interface unit **904_1** writes line group **903_1** into memory **902**. In response to the latter, line buffer interface unit **904_1** performs respective reads of line group **903_1** from memory **902** and forwards line group **903_1** to consumers K**2** and K**3** at times C**21** and C**31**, respectively.

[0084] After all consumers of a line group have been forwarded their copy of the line group, the line buffer interface unit is "free" to be assigned to another line group. For example, if line group **903_1** represents the first line group within a first image frame of a sequence of frames, after line group **903_1** has been forwarded to consumers K**2** and K**3** at times C**21** and C**31**, line buffer interface unit **904_1** may next be assigned to handle the first line group within the next, second image frame of the sequence of frames. In this manner, the line buffer unit circuitry **901** can be viewed as having a "pool" of line buffer interface units **904** where each interface unit is assigned a new line group to manage after its immediately preceding line group was delivered to its last consumer. Thus, there is a rotation of interface units as they repeatedly enter and are removed from a "free pool" of line buffer interface units who have served their last consumer and are waiting for their next line group.

[0085] FIG. **9***c* illustrates an embodiment of the rotation in more detail. As observed in FIG. **9***c*, an available line buffer interface unit is selected from a free pool of line buffer interface units within the line buffer unit circuitry **910**. The line buffer interface unit is then configured with appropriate configuration information **911** (e.g., X, Y position information of the new line group or a linear memory address equivalent). Here, note in FIG. **9***b* that each line buffer interface unit may include configuration register space **905** where such configuration information is kept.

[0086] The line buffer interface unit then proceeds to handle producer and consumer requests for its newly assigned line group **912**. After the last producer has written to the line group (in various embodiments there is only one producer per line group) and after the last consumer has been provided with the version of the line group that has been written to by its producer(s), the line buffer interface unit is returned to the free pool and the process repeats **910** for a next line group. The control logic circuitry within the line buffer unit circuitry **901** that oversees the control flow of FIG. **9***c* is not depicted in FIG. **9***b* for illustrative convenience.

b. Programmable Register Space Embodiments

[0087] With respect to the updated configuration information **911** that is provided to a line buffer interface unit as part of the assignment of a next line group, in a nominal case, the line buffer unit **900** itself is handling a static arrangement of, e.g., only one fixed producer that is feeding a fixed set of one or more consumers. In this case, primary configuration information (e.g., line group size, number of consumers, etc.) is also apt to be static and will not change from line group to line group. Rather, the new configuration information that is provided to a line buffer interface unit mainly identifies the new line group (e.g., the location of the line group within memory, etc.). More complicated potential arrangements/designs are possible, however. Some of these are described in more detail immediately below.

[0088] FIG. **9***d* depicts an embodiment of the contents of a line buffer interface unit's register space (e.g., the contents of register space **905_1** of FIG. **9***b*). A description of some of the register fields immediately follows.

[0089] The LB_Enable field **921** essentially enables a line buffer interface unit and is "set" as part of the process of taking the line buffer interface unit from the free pool. The Num_Channels field **922** defines the number of channels within the line group's image data. In an embodiment, the Num_Channels field **922** can be used to determine the total amount of data per line group. For example, a video stream often includes a frame sequence of red (R) pixels, a frame sequence of blue (B) pixels and a frame sequence of green (G) pixels. Thus, for any line group, there are actually three line groups worth of information (R, G and B).

[0090] The Num_Consumers field **923** describes the number of consumers that will request the line group. In an embodiment, the line buffer interface unit will be entered to the free pool after a line group instance has been delivered a number of times equal to the value in the Num_Consumers field **923**.

[0091] The Row_Width field **924** defines the width of a full line group (e.g., in number of pixels). Note that the Row_Width **924** value can be expressed as an X coordinate value provided by the compiler. The FB_Rows field **926** defines the height of a full line group (e.g., in number of pixels). Note that the FB_Rows field **924** can be expressed as a Y coordinate value provided by the compiler.

[0092] The FB_Base_Address field **930** defines the location of the line group in the line buffer unit memory. In a first operational mode, referred to as "full" line group mode, a full sized line group is accessed in memory (line groups are received from producers and delivered to consumers as containing the full amount of their respective data). In the full line group mode, the Num_Channels field **922**, the Row_Width field **924** and the FB_Rows field **926** can be used with the FB_Address field **930** to determine the range of addresses that are to be applied to memory to completely access a full line group. Additionally, these same parameters can be used to "translate" a request from a sheet generator that has requested the line group in X, Y coordinates into a linear memory address.

[0093] The VB_Enable, VB_Rows, VB_Cols, Num_Reuse_Rows and VB_Base_Address fields **925**, **927**, **928**, **931** are used in another operational mode, referred to as the "virtually tall" line group mode, which is described in detail further below.

[0094] Whereas FIG. **9***d* displayed the configuration register space **905** for a single line buffer interface unit, by contrast, FIG. **9***e* shows an embodiment of the contents of global configuration register space **907** for the line buffer unit circuitry **901** as a whole. Whereas the per line buffer interface unit register space of FIG. **9***d* is focused on a specific line group, by contrast, the global register space **907** of FIG. **9***e* is focused on understanding the parsing of different line groups from a same image as well as other information that is specific to the producer/consumer combination that are associated with the processing of the image.

[0095] As observed in FIG. **9***e*, an embodiment of the global register space includes the number of channels **932** and the number of consumers **933** for a particular image. For simplicity, the register space of FIG. **9***e* only contemplates one image with one set of producers and consumers (e.g., only a single video stream and a single point in a DAG). Conceivably, multiple instances of the register space of FIG. **9***e* could be allocated to permit the line buffer unit circuitry to effectively multi-task.

[0096] A first form of multi-tasking is within a DAG or software pipeline that is implemented on the image processor. Here, the same line buffer unit could be configured to handle the line grouping for two different nodes within the DAG or for two different stages of the pipeline (that is, a single line buffer unit could support more than one stencil processor). The different nodes/stages could easily have different numbers of consumers but in many cases are likely to have the same image and stencil size characteristics. A second form of multi-tasking is across multiple different DAGs and/or multiple different pipelines that are implemented on the same image processor hardware. For example, an image processor having four stencil processors could concurrently execute two completely different two-stage pipelines that respectively process completely different image sizes with completely different stencil dimensions.

[0097] Returning to the particular embodiment of FIG. **9***e*, note that any particular node in a DAG or between pipeline stages can be characterized at a high level by the number of channels in the image, the image size, the dimensions of the applicable stencil and the number of consumers of the line groups (FIG. **9***e* again assumes one producer per line group but conceivably more than one producer could write to a single line group in which case the global register space of FIG. **9***e* would also include a field for the number of producers). The Num_Channels and Num_Consumers fields **932**, **933** are essentially the same as the corresponding fields **922**, **923** of FIG. **9***c*.

[0098] The Image_Size and Stencil_Dimension fields **934**, **935** essentially describe the dimensions of the image to be processed and the dimensions of the stencil that will operate on the line groups that are to be carved from the image respectively. Note that both fields **934**, **935** can be expressed in terms of X, Y coordinate values and can be provided from the compiler. Additionally, in an embodiment, control logic circuitry within the line buffer circuitry unit (not shown in FIG. **9***b*) uses the Image_Size and Stencil_Dimension fields **934**, **935** to determine the Row_Width **924**, FB_Rows **926** and FB_Base_Address values **930** that are loaded into a line buffer interface unit's register space when the line buffer interface unit is assigned to handle line groups from the producer/consumer set that the global information pertains to. In an alternate or further embodiment, image size is expressed as two separate values, image_width and image_height, which may have their own separately addressable register space. Likewise, stencil size may be expressed as two separate values, stencil_width and stencil_height, which may have their own separately addressable register space.

[0099] Row_Width **924** is directly obtainable from the Image_Size **934** information. For example, if Image_Size is expressed as the X, Y coordinate pair at the farthest pixel from the image origin (the upper right hand corner if the origin is at the lower left hand corner), Row_Width can be determined as the X coordinate value.

[0100] The FB_Rows and FB_Base_Address fields **926**, **930** can be determined from the Image_Size and Stencil_Dimension fields **934**, **935**. Here, specifically, the height of each line group (FB_Rows **926**) can be calculated from the height of the image (Y coordinate value of Image_Size **934**) and the stencil height (Y coordinate value of Stencil_Dimension **935**). Once the height of the line groups is known, the number of line groups that are to be parsed from the image and the starting linear address for each such line group in memory (FB_Base_Address **930**) can also be determined.

[0101] Thus, in an embodiment, when a line buffer unit is assigned to handle a line group for a particular producer/consumer combination whose global register space is characterized by the register fields of FIG. **9***e*, the above described determinations are calculated on the fly and each of FB_Width **924**, FB_Rows **926**, Base_Address **934** are loaded into the line buffer interface unit's specific register space along with Num_Channels **922** and Num_Consumers **923** which copy over directly. Logic circuitry and data paths may therefore exist between the global register space and each instance of line buffer interface unit register space to perform these determinations and data transfers.

[0102] In an alternate embodiment, the compiler performs each of these calculations thereby eliminating much if not all of the global register space altogether. Here, for instance, the compiler can determine the Base_Address value for each line group and load the values in a look-up table within the line buffer circuitry unit. The values are called from the look-up table and loaded into a line buffer interface unit's register space as their corresponding line groups are configured for. Different combinations between these two extremes (hardware on-the-fly vs. static compiler determined) may also be implemented.

[0103] Although embodiments above emphasized the keeping of configuration information in register circuitry ("register space"), in other or combined embodiments, configuration information may be kept in memory (such as buffer unit memory) or other memory or information keeping circuitry.

c. Line Buffer Unit Embodiments & Full Line Group Mode vs. Virtually Tall Mode

[0104] The discussions above have largely been directed to "full line group" mode in which line groups are referred to and passed between the sheet generators and line buffer unit as complete, entire line groups. In another mode, referred to as "virtually tall", line groups are referred to and passed between the sheet generators as a full width upper portion and a lower portion that is completed in separate, discrete segments.

[0105] FIGS. **10***a* and **10***b* show a depiction of an exemplary virtually tall mode sequence. As observed in FIG. **10***a*, a line group is initially formed as an upper portion **1003** of full width rows and a first lower portion **1004_1** having only a first, shorter segment of width. The initial formation of a line group may be provided to a line buffer unit by a producing sheet generator, or, may be provided by a line buffer unit to a consuming sheet generator.

[0106] In the case of a producer, the line group is formed after the stencils **1002** have processed over the lower portion **1004_1** (the approximate stencil positioning is observed in FIG. **10***b*). After the producer stencil processor has processed over the lower portion **1004_1** the stencils continue forward horizontally to the right. Eventually they will process over a next lower portion **1004_2**. Upon completion of the next lower portion **1004_2**, the next lower portion **1004_2** is sent from the sheet generator to the line buffer unit which stores it in memory in the correct location, e.g., "next to" first lower portion **1004_1**. The process continues until the line group is fully written into line buffer unit memory.

[0107] In the case of consumers, the line group is initially delivered to the sheet generator as observed in FIG. **10***a*. The stencil processor operates over the first portion **1004_1** of the line group. Upon nearing the completion of the processing of the first portion **1004_1** the sheet generator will request the next lower portion **1004_2** which is fetched from memory and delivered by the line buffer unit. The process continues until the line group is completely processed.

[0108] Note that for both producers and consumers, lower portions are specifically identified by the sheet generator. That is, in both the producer case and the consumer case, lower portion **1004_2** is specifically identified by the sheet generator and the line buffer unit specifically accesses memory to store/fetch lower portion **1004_2**. In an embodiment, the sheet generator identifies lower portion **1004_2** through X, Y coordinate values that are contemplated based on information provided by the compiler (for example, any corner of lower portion **1004_2**, all four corners of lower portion **1004_2**, just an X coordinate value, etc.).

[0109] FIG. **11***a* shows a first (more simplistic) embodiment of the circuitry within a line buffer interface unit **1104**. As observed in FIG. **11***a*, the line buffer interface unit includes address translation circuitry **1106** to convert the identity of a line group or portion thereof (such as lower portion **1004_2** of FIG. **10***b*) that is identified by one or more X, Y coordinate values into a linear address for accessing line buffer unit memory. That is, line groups can be deemed to be "mapped" into line buffer unit memory. The translation circuitry **1106** essentially comprehends this mapping in X,Y terms and can convert the same to specific linear memory addresses.

[0110] The ability to comprehend the mapping is based on information within configuration register space **1105**, an embodiment of which was provided above in FIG. **9***d*. Here, with knowledge of Row_Width **924**, FB_Rows **926** and FB_Base_Address **931** the translation unit can "comprehend" the size and location of the full line group in memory. As such, for example, in the virtually tall mode, a request for a lower portion based on any X coordinate value (e.g., if the lower portion is referenced relative to the line group) or X,Y coordinate location (e.g., if the lower portion is referenced relative to the image frame) is sufficient to identify what portion is being referred to by the sheet generator. Additionally, Vb_Rows **927** and Vb_Cols **928** essentially define the dimensions of the lower portions. With knowledge of the dimensions of the upper and lower portions the amount of data to be accessed to/from buffer memory is also readily determinable. These same concepts may also to apply to full width line groups. For example, any full width line group may be identified by its X,Y location within an image. Additionally, in some embodiments, a full width line group may be passed through the network via a sequence of atomic requests/responses that reference smaller chunks of a full width line group by way of X and/or Y coordinate values.

[0111] The translation circuitry **1106** could also be used in an abstract addressing mode in which the Base_Address_

Field **931** is not populated and the sheet generators refer to line groups as X,Y coordinates within an image frame. In this case, if the translation circuitry **1006** is coupled to or otherwise apprised of some of the information in the global register space of FIG. **9***e* (e.g., Image_Size, Stencil_Size), the translation circuitry **1106** could calculate all pertinent information for the line group (its dimensions and location within the frame) and convert the same to linear addresses used for accessing line buffer unit memory. In another embodiment, the translation circuitry **1106** determines the Base_Address_Field value **931** outright (based on global type information and one or more X, Y coordinates describing the line group) and loads it into its own register space **1105**.

[0112] The line buffer interface unit embodiment of FIG. **11***a* also supports a linear addressing mode in which X,Y coordinate values are not used to refer to a line group (rather, traditional linear addresses are used). For the linear addressing mode, bypass paths **1140** circumvent the address translation circuitry **1106**. In an embodiment, regardless of which addressing mode is used at the line buffer interface unit input, the line buffer interface unit provides standard linear memory addresses for addressing line buffer unit memory. Referring briefly back to FIG. **9***b*, the linear addresses are provided to an arbiter. Memory interface **908** resolves colliding memory access requests and accesses line buffer unit memory **902**.

[0113] As discussed at length above, a sheet generator may refer to a line group with one or more X, Y coordinate values. In another embodiment, rather than the sheet generators identifying a next line group in full line group mode or a next lower portion in virtually tall mode, the sheet generators simply issue a request akin to "next" (e.g., the request only indicates a "next" full line group or "next" lower portion or "next" image data within the same full/virtually tall line group is being referred to without any coordinates).

[0114] To support this avenue of communication, the line buffer unit and/or line buffer unit interface includes state register space to comprehend what the next line group/portion is. FIG. **11***b* shows an enhanced embodiment of a line buffer interface unit that keeps pointer state information so that the sheet generators can simply refer to a "next" lower portion of a line group in virtually tall mode rather than having to specify its location with X,Y coordinates. Here, a write pointer **1141** is maintained by pointer control logic circuitry **1143** that keeps track of the lower portions that have been provided by the producing sheet generator. Essentially the write pointer **1141** stores the location of the "next" portion that the producer is scheduled to deliver. In addition, the pointer state information permits sheet generators to refer only to a "next" full width line group (in full width mode) without having to specify any X,Y coordinates (because the line buffer interface unit can determine where the next full width line group for the image is).

[0115] In an embodiment, the pointer is articulated as one or more X, Y coordinates and the translation circuitry converts the same into a linear address. When the next portion is received, the pointer **1141** is updated by pointer control logic circuitry **1143** to point to the portion that will follow the portion that has just been received. Read pointers **1142** operate similarly but a separate read pointer is kept for each consumer (again, only one producer is assumed for convenience).

[0116] In the case of full line group mode, the location of the "next" full width line group is determinable from the global register information and a similar arrangement of pointers that are kept at a global level.

d. Implementation Embodiments

[0117] It is pertinent to point out that the various image processor architecture features described above are not necessarily limited to image processing in the traditional sense and therefore may be applied to other applications that may (or may not) cause the image processor to be re-characterized. For example, if any of the various image processor architecture features described above were to be used in the creation and/or generation and/or rendering of animation as opposed to the processing of actual camera images, the image processor may be characterized as a graphics processing unit. Additionally, the image processor architectural features described above may be applied to other technical applications such as video processing, vision processing, image recognition and/or machine learning. Applied in this manner, the image processor may be integrated with (e.g., as a co-processor to) a more general purpose processor (e.g., that is or is part of a CPU of computing system), or, may be a stand alone processor within a computing system.

[0118] The hardware design embodiments discussed above may be embodied within a semiconductor chip and/or as a description of a circuit design for eventual targeting toward a semiconductor manufacturing process. In the case of the latter, such circuit descriptions may take the form of higher/behavioral level circuit descriptions (e.g., a VHDL description) or lower level circuit description (e.g., a register transfer level (RTL) description, transistor level description or mask description) or various combinations thereof. Circuit descriptions are typically embodied on a computer readable storage medium (such as a CD-ROM or other type of storage technology).

[0119] From the preceding sections is pertinent to recognize that an image processor as described above may be embodied in hardware on a computer system (e.g., as part of a handheld device's System on Chip (SOC) that processes data from the handheld device's camera). In cases where the image processor is embodied as a hardware circuit, note that the image data that is processed by the image processor may be received directly from a camera. Here, the image processor may be part of a discrete camera, or, part of a computing system having an integrated camera. In the case of the later the image data may be received directly from the camera or from the computing system's system memory (e.g., the camera sends its image data to system memory rather than the image processor). Note also that many of the features described in the preceding sections may be applicable to a graphics processor unit (which renders animation).

[0120] FIG. **12** provides an exemplary depiction of a computing system. Many of the components of the computing system described below are applicable to a computing system having an integrated camera and associated image processor (e.g., a handheld device such as a smartphone or tablet computer). Those of ordinary skill will be able to easily delineate between the two.

[0121] As observed in FIG. **12**, the basic computing system may include a central processing unit **1201** (which may include, e.g., a plurality of general purpose processing cores **1215_1** through **1215_N** and a main memory controller **1217** disposed on a multi-core processor or applications processor), system memory **1202**, a display **1203** (e.g., touchscreen, flat-panel), a local wired point-to-point link (e.g., USB) interface **1204**, various network I/O functions **1205** (such as an Ethernet interface and/or cellular modem subsystem), a wireless local area network (e.g., WiFi) interface **1206**, a wireless point-to-point link (e.g., Bluetooth) interface **1207** and a Global Positioning System interface **1208**, various sensors **1209_1** through **1209_N**, one or more cameras **1210**, a battery **1211**, a power management control unit **1212**, a speaker and microphone **1213** and an audio coder/decoder **1214**.

[0122] An applications processor or multi-core processor **1250** may include one or more general purpose processing cores **1215** within its CPU **1201**, one or more graphical processing units **1216**, a memory management function **1217** (e.g., a memory controller), an I/O control function **1218** and an image processing unit **1219**. The general purpose processing cores **1215** typically execute the operating system and application software of the computing system. The graphics processing units **1216** typically execute graphics intensive functions to, e.g., generate graphics information that is presented on the display **1203**. The memory control function **1217** interfaces with the system memory **1202** to write/read data to/from system memory **1202**. The power management control unit **1212** generally controls the power consumption of the system **1200**.

[0123] The image processing unit **1219** may be implemented according to any of the image processing unit embodiments described at length above in the preceding sections. Alternatively or in combination, the IPU **1219** may be coupled to either or both of the GPU **1216** and CPU **1201** as a co-processor thereof. Additionally, in various embodiments, the GPU **1216** may be implemented with any of the image processor features described at length above.

[0124] Each of the touchscreen display **1203**, the communication interfaces **1204-1207**, the GPS interface **1208**, the sensors **1209**, the camera **1210**, and the speaker/microphone codec **1213**, **1214** all can be viewed as various forms of I/O (input and/or output) relative to the overall computing system including, where appropriate, an integrated peripheral device as well (e.g., the one or more cameras **1210**). Depending on implementation, various ones of these I/O components may be integrated on the applications processor/multi-core processor **1250** or may be located off the die or outside the package of the applications processor/multi-core processor **1250**.

[0125] In an embodiment one or more cameras **1210** includes a depth camera capable of measuring depth between the camera and an object in its field of view. Application software, operating system software, device driver software and/or firmware executing on a general purpose CPU core (or other functional block having an instruction execution pipeline to execute program code) of an applications processor or other processor may perform any of the functions described above.

[0126] Embodiments of the invention may include various processes as set forth above. The processes may be embodied in machine-executable instructions. The instructions can be used to cause a general-purpose or special-purpose processor to perform certain processes. Alternatively, these processes may be performed by specific hardware components that contain hardwired logic for performing the processes, or by any combination of programmed computer components and custom hardware components.

[0127] Elements of the present invention may also be provided as a machine-readable medium for storing the machine-executable instructions. The machine-readable medium may include, but is not limited to, floppy diskettes, optical disks, CD-ROMs, and magneto-optical disks, FLASH memory, ROMs, RAMs, EPROMs, EEPROMs, magnetic or optical cards, propagation media or other type of media/machine-readable medium suitable for storing electronic instructions. For example, the present invention may be downloaded as a computer program which may be transferred from a remote computer (e.g., a server) to a requesting computer (e.g., a client) by way of data signals embodied in a carrier wave or other propagation medium via a communication link (e.g., a modem or network connection).

[0128] In the foregoing specification, the invention has been described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

1. A device comprising:

a plurality of line buffer interface units; and

a memory unit configured to store image data partitioned into a plurality of line groups,

wherein the device is configured to assign each line buffer interface unit to manage read and write requests for a respective line group of the plurality of line groups,

wherein each line buffer interface unit is configured to receive a write request from a producer component, to identify a write location within the memory unit corresponding to the write request, and to store data at the write location within the memory unit according to the write request,

wherein each line buffer interface unit is configured to receive a read request from one or more consumer components, to identify a read location within the memory unit corresponding to the read request, and to provide data stored at the read location within the memory unit according to the read request.

2. The device of claim **1**, wherein, upon a line buffer interface unit completing all outstanding read requests from one or more consumers components for a particular line group, the device is configured to reassign the line buffer interface unit to manage read and write requests for a different line group.

3. The device of claim **2**, wherein each line buffer interface unit has a dedicated programmable unit configuration space that is configured to store data representing properties of a line group to which the line buffer interface unit is assigned, and

wherein upon a line buffer interface unit being reassigned from a first line group to a second line group, the device is configured to update the data in the programmable unit configuration space of the line buffer interface unit to represent properties of the second line group.

4. The device of claim **3**, wherein the data in the programmable unit configuration space represents a maximum number of consumer components that the line buffer interface unit can service simultaneously, a row width of the line group, or a base linear address in the memory unit of the line group.

5. The device of claim **3**, wherein the device has a programmable global configuration space that is configured to store data representing a total size of an image having image data stored in the memory unit.

6. The device of claim **5**, wherein the programmable global configuration space is configured to store data representing a number of active line buffer interface units.

7. The device of claim **3**, wherein the device is configured to dynamically compute one or more data values in the programmable unit configuration space from one or more data values in a programmable global configuration space of the device.

8. The device of claim **7**, wherein the device is configured to compute a number of full line group rows or a line group base address in the programmable unit configuration space from image size and stencil dimension elements of the programmable global configuration space.

9. The device of claim **1**, wherein each line buffer interface unit has translation circuitry that is configured to convert a pair of values into a linear address within the memory unit.

10. The device of claim **1**, wherein each line buffer interface unit is configured to maintain a pointer to a current or next segment of a line group to be provided on a next read request.

11. A method comprising:

assigning, by a device having a plurality of line buffer interface units and a memory unit storing image data partitioned into a plurality of line groups, each line buffer interface unit to manage read and write requests for a respective line group of the plurality of line groups,

receiving, by a particular line buffer interface unit of the plurality of line buffer interface units, a write request from a producer component of the device;

identify, by the particular line buffer interface unit, a write location within the memory unit corresponding to the write request;

storing, by the particular line buffer interface unit, data at the write location within the memory unit according to the write request;

receiving, by the particular line buffer interface unit, a read request from one or more consumer components of the device;

identifying, by the particular line buffer interface unit, a read location within the memory unit corresponding to the read request; and

providing, by the particular line buffer interface unit, data stored at the read location within the memory unit according to the read request.

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

receiving, by the device, an indication that a line buffer interface unit has completed all outstanding read requests from one or more consumers components for a particular line group; and

in response, reassigning, by the device, the line buffer interface unit to manage read and write requests for a different line group.

13. The method of claim **12**, wherein each line buffer interface unit has a dedicated programmable unit configuration space that is configured to store data representing properties of a line group to which the line buffer interface unit is assigned, and

wherein reassigning the line buffer interface unit comprises updating, by the device, the data in the programmable unit configuration space of the line buffer interface unit to represent properties of the different line group.

14. The method of claim **13**, wherein the data in the programmable unit configuration space represents a maximum number of consumer components that the line buffer interface unit can service simultaneously, a row width of the line group, or a base linear address in the memory unit of the line group.

15. The method of claim **13**, wherein the device has a programmable global configuration space that is configured to store data representing a total size of an image having image data stored in the memory unit.

16. The method of claim **15**, wherein the programmable global configuration space is configured to store data representing a number of active line buffer interface units.

17. The method of claim **13**, further comprising dynamically computing, by the device, one or more data values in the programmable unit configuration space from one or more data values in a programmable global configuration space of the device.

18. The method of claim **17**, further comprising computing, by the device, a number of full line group rows or a line group base address in the programmable unit configuration space from image size and stencil dimension elements of the programmable global configuration space.

19. The method of claim **11**, wherein each line buffer interface unit has translation circuitry, and further comprising: converting, by translation circuitry of the particular line buffer interface unit, a pair of values into a linear address within the memory unit.

20. The method of claim **11**, further comprising maintaining, by the particular line buffer interface unit, a pointer to a current or next segment of a line group to be provided on a next read request.

* * * * *