(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
24 September 2020 (24.09.2020)

WIPO | PCT

(10) International Publication Number
WO 2020/190809 A1

(51) International Patent Classification:
G06F 9/30 (2018.01) *G06F 9/38* (2018.01)

(21) International Application Number:
PCT/US2020/022847

(22) International Filing Date:
14 March 2020 (14.03.2020)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

62/819,337	15 March 2019 (15.03.2019)	US
62/819,361	15 March 2019 (15.03.2019)	US
62/819,435	15 March 2019 (15.03.2019)	US
62/935,670	15 November 2019 (15.11.2019)	US

(71) Applicant: **INTEL CORPORATION** [US/US]; 2200 Mission College Blvd., Santa Clara, California 95054 (US).

(72) Inventors: **APPU, Abhishek**; 8032 Murcia Way, El Dorado Hills, California 95762 (US). **MAIYURAN, Subramaniam**; 1915 Black Slate Court, Gold River, California 95670 (US). **MACPHERSON, Mike**; 12476 NW Hibbard Drive, Portland, Oregon 97229 (US). **FU, Fangwen**; 1900 Prairie City Road, Folsom, California 95630 (US). **CHEN, Jiasheng**; 4328 Suffolk Way, El Dorado Hills, California 95762 (US). **GEORGE, Varghese**; 460 Tobrurry Way, Folsom, California 95630 (US). **RANGANATHAN, Vasanth**; 3620 RoseCrest Circle, El Dorado Hills, California 95762 (US). **GARG, Ashutosh**; 1900 Prairie City Road, Folsom, California 95630 (US). **RAY, Joydeep**; 727 Misty Ridge Cir, Folsom, California 95630 (US).

(74) Agent: **HAMILTON, Howard S.**; Jaffery Watson Mendonsa & Hamilton LLP, 7501 Village Square Drive, Suite 206, Castle Pines, Colorado 80108 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA,

(54) Title: ARCHITECTURE FOR BLOCK SPARSE OPERATIONS ON A SYSTOLIC ARRAY

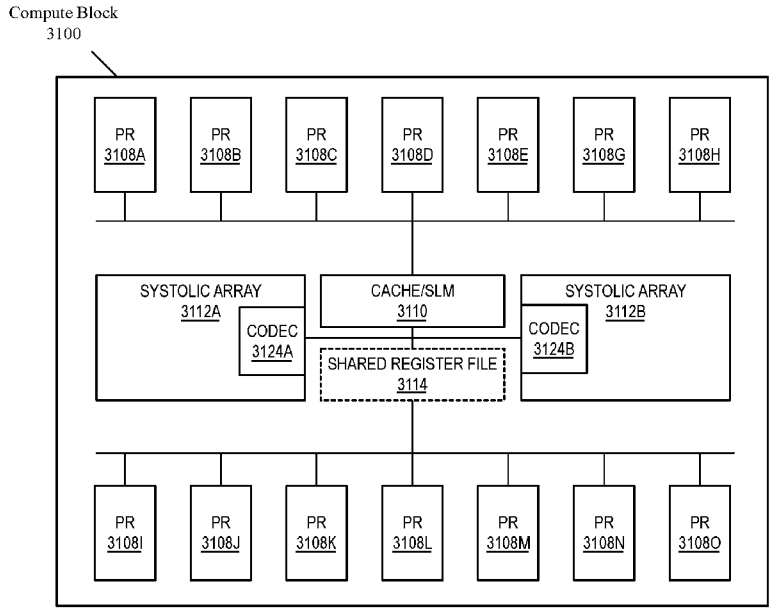


FIG. 31A

(57) Abstract: Embodiments described herein include, software, firmware, and hardware logic that provides techniques to perform arithmetic on sparse data via a systolic processing unit. One embodiment provides for data aware sparsity via compressed bitstreams. One embodiment provides for block sparse dot product instructions. One embodiment provides for a depth-wise adapter for a systolic array.

WO 2020/190809 A1

[Continued on next page]

SC, SD, SE, SG, SK, SL, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

ARCHITECTURE FOR BLOCK SPARSE OPERATIONS ON A SYSTOLIC ARRAY

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] The present application is related to and, under 35 U.S.C. 119(e), claims the benefit of and priority to U.S. Provisional Applications 62/819,337, entitled GRAPHICS PROCESSING, by Abhishek Appu, et al., filed March 15, 2019 (Attorney Docket No. AC0271-Z), 62/819,435, entitled GRAPHICS DATA PROCESSING, by Lakshminarayanan Striramassarma, et al., filed March 15, 2019 (Attorney Docket No. AC0285-Z), 62/819,361, entitled SYSTEMS AND METHODS FOR PARTITIONING CACHE TO REDUCE CACHE ACCESS LATENCY, by Subramaniam Maiyuran, et al., filed March 15, 2019 (Attorney Docket No. AC0286-Z), and 62/935,670, entitled SYSTOLIC ARITHMETIC ON SPARSE DATA, by Abhishek Appu, et al., filed November 15, 2019 (Attorney Docket No. AC5197-Z), the contents of all are incorporated herein by reference.

FIELD

[0002] This disclosure relates generally to data processing and more particularly to sparse data processing via a matrix accelerator of a general-purpose graphics processing unit.

BACKGROUND OF THE DISCLOSURE

[0003] Current parallel graphics data processing includes systems and methods developed to perform specific operations on graphics data such as, for example, linear interpolation, tessellation, rasterization, texture mapping, depth testing, etc. Traditionally, graphics processors used fixed function computational units to process graphics data; however, more recently, portions of graphics processors have been made programmable, enabling such processors to support a wider variety of operations for processing vertex and fragment data.

[0004] To further increase performance, graphics processors typically implement processing techniques such as pipelining that attempt to process, in parallel, as much graphics data as possible throughout the different parts of the graphics pipeline. Parallel graphics processors with single instruction, multiple thread (SIMT) architectures are designed to maximize the amount of parallel processing in the graphics pipeline. In an SIMT architecture, groups of parallel threads attempt to execute program instructions synchronously together as often as possible to increase processing efficiency. A general overview of software and hardware for SIMT architectures can be found in Shane Cook, *CUDA Programming* Chapter 3, pages 37-51 (2013).

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] So that the manner in which the above recited features of the present embodiments can be understood in detail, a more particular description of the embodiments, briefly

summarized above, may be had by reference to embodiments, some of which are illustrated in the appended drawings. It is to be noted, however, that the appended drawings illustrate only typical embodiments and are therefore not to be considered limiting of its scope.

[0006] **FIG. 1** is a block diagram illustrating a computer system configured to implement one or more aspects of the embodiments described herein;

[0007] **FIG. 2A-2D** illustrate parallel processor components;

[0008] **FIG. 3A-3C** are block diagrams of graphics multiprocessors and multiprocessor-based GPUs;

[0009] **FIG. 4A-4F** illustrate an exemplary architecture in which a plurality of GPUs is communicatively coupled to a plurality of multi-core processors;

[0010] **FIG. 5** illustrates a graphics processing pipeline;

[0011] **FIG. 6** illustrates a machine learning software stack;

[0012] **FIG. 7** illustrates a general-purpose graphics processing unit;

[0013] **FIG. 8** illustrates a multi-GPU computing system;

[0014] **FIG. 9A-9B** illustrate layers of exemplary deep neural networks;

[0015] **FIG. 10** illustrates an exemplary recurrent neural network;

[0016] **FIG. 11** illustrates training and deployment of a deep neural network;

[0017] **FIG. 12** is a block diagram illustrating distributed learning;

[0018] **FIG. 13** illustrates an exemplary inferencing system on a chip (SOC) suitable for performing inferencing using a trained model;

[0019] **FIG. 14** is a block diagram of a processing system;

[0020] **FIG. 15A-15C** illustrate computing systems and graphics processors;

[0021] **FIG. 16A-16C** illustrate block diagrams of additional graphics processor and compute accelerator architectures;

[0022] **FIG. 17** is a block diagram of a graphics processing engine of a graphics processor;

[0023] **FIG. 18A-18B** illustrate thread execution logic including an array of processing elements employed in a graphics processor core;

[0024] **FIG. 19** illustrates an additional execution unit;

[0025] **FIG. 20** is a block diagram illustrating a graphics processor instruction formats;

[0026] **FIG. 21** is a block diagram of an additional graphics processor architecture;

[0027] **FIG. 22A-22B** illustrate a graphics processor command format and command sequence;

[0028] **FIG. 23** illustrates exemplary graphics software architecture for a data processing system;

[0029] **FIG. 24A** is a block diagram illustrating an IP core development system;

[0030] **FIG. 24B** illustrates a cross-section side view of an integrated circuit package assembly;

[0031] **FIG. 24C** illustrates a package assembly that includes multiple units of hardware logic chiplets connected to a substrate (e.g., base die);

[0032] **FIG. 24D** illustrates a package assembly including interchangeable chiplets;

[0033] **FIG. 25** is a block diagram illustrating an exemplary system on a chip integrated circuit;

[0034] **FIG. 26A-26B** are block diagrams illustrating exemplary graphics processors for use within an SoC;

[0035] **FIG. 27** illustrates an additional execution unit, according to an embodiment;

[0036] **FIG. 28** illustrates a matrix operation performed by an instruction pipeline, according to an embodiment;

[0037] **FIG. 29A-29B** illustrate details of hardware-based systolic array, according to some embodiments;

[0038] **FIG. 30** illustrates a systolic array including partial sum loopback and circuitry to accelerate sparse matrix multiply;

[0039] **FIG. 31A-31B** illustrate matrix acceleration logic including codecs to enable the reading of sparse data in a compressed format;

[0040] **FIG. 32A-32B** illustrates zero-based SM encoding for sparse neural network data, according to embodiments;

[0041] **FIG. 33A-33B** illustrates specified-value SM encoding for sparse neural network data, according to embodiments;

[0042] **FIG. 34A-34B** illustrates comparisons between unstructured sparsity and block sparsity within training data for a neural network;

[0043] **FIG. 35A-35B** illustrate a four element dot product instruction executable by dot product logic;

[0044] **FIG. 36A-36B** illustrate a two element dot product instruction executable by dot product logic;

[0045] **FIG. 37A-37C** illustrate an instruction and associated hardware logic to enable a multiply add vector operation with zero skipping for sparse input;

[0046] **FIG. 38A-38B** contrasts 3D convolution with depthwise convolution;

[0047] **FIG. 39A-39C** illustrate a system in including a systolic tensor array and an image to column adapter for use during depthwise convolution;

[0048] **FIG. 40** illustrates a method of performing depthwise convolution on a systolic array; and

[0049] **FIG. 41** is a block diagram of a computing device including a graphics processor, according to an embodiment.

DETAILED DESCRIPTION

[0050] A graphics processing unit (GPU) is communicatively coupled to host/processor cores to accelerate, for example, graphics operations, machine-learning operations, pattern analysis operations, and/or various general-purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or another interconnect (e.g., a high-speed interconnect such as PCIe or NVLink). Alternatively, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

[0051] Embodiments described herein include, software, firmware, and hardware logic that provides techniques to perform arithmetic on sparse data via a systolic processing unit. One embodiment provides for data aware sparsity via compressed bitstreams. One embodiment provides for block sparse dot product instructions (e.g., dp4a, dp2a). One embodiment provides for a depth-wise adapter for a systolic array.

[0052] In the following description, numerous specific details are set forth to provide a more thorough understanding. However, it will be apparent to one of skill in the art that the embodiments described herein may be practiced without one or more of these specific details. In other instances, well-known features have not been described to avoid obscuring the details of the present embodiments.

System Overview

[0053] **FIG. 1** is a block diagram illustrating a computing system 100 configured to implement one or more aspects of the embodiments described herein. The computing system 100 includes a processing subsystem 101 having one or more processor(s) 102 and a system memory 104 communicating via an interconnection path that may include a memory hub 105. The memory hub 105 may be a separate component within a chipset component or may be integrated within the one or more processor(s) 102. The memory hub 105 couples with an I/O subsystem 111 via a communication link 106. The I/O subsystem 111 includes an I/O hub 107 that can enable the computing system 100 to receive input from one or more input device(s) 108. Additionally, the I/O hub 107 can enable a display controller, which may be included in the one or more processor(s) 102, to provide outputs to one or more display device(s) 110A. In one

embodiment the one or more display device(s) 110A coupled with the I/O hub 107 can include a local, internal, or embedded display device.

[0054] The processing subsystem 101, for example, includes one or more parallel processor(s) 112 coupled to memory hub 105 via a bus or other communication link 113. The communication link 113 may be one of any number of standards-based communication link technologies or protocols, such as, but not limited to PCI Express, or may be a vendor specific communications interface or communications fabric. The one or more parallel processor(s) 112 may form a computationally focused parallel or vector processing system that can include a large number of processing cores and/or processing clusters, such as a many integrated core (MIC) processor. For example, the one or more parallel processor(s) 112 form a graphics processing subsystem that can output pixels to one of the one or more display device(s) 110A coupled via the I/O Hub 107. The one or more parallel processor(s) 112 can also include a display controller and display interface (not shown) to enable a direct connection to one or more display device(s) 110B.

[0055] Within the I/O subsystem 111, a system storage unit 114 can connect to the I/O hub 107 to provide a storage mechanism for the computing system 100. An I/O switch 116 can be used to provide an interface mechanism to enable connections between the I/O hub 107 and other components, such as a network adapter 118 and/or wireless network adapter 119 that may be integrated into the platform, and various other devices that can be added via one or more add-in device(s) 120. The add-in device(s) 120 may also include, for example, one or more external graphics processor devices and/or compute accelerators. The network adapter 118 can be an Ethernet adapter or another wired network adapter. The wireless network adapter 119 can include one or more of a Wi-Fi, Bluetooth, near field communication (NFC), or other network device that includes one or more wireless radios.

[0056] The computing system 100 can include other components not explicitly shown, including USB or other port connections, optical storage drives, video capture devices, and the like, may also be connected to the I/O hub 107. Communication paths interconnecting the various components in FIG. 1 may be implemented using any suitable protocols, such as PCI (Peripheral Component Interconnect) based protocols (e.g., PCI-Express), or any other bus or point-to-point communication interfaces and/or protocol(s), such as the NV-Link high-speed interconnect, or interconnect protocols known in the art.

[0057] The one or more parallel processor(s) 112 may incorporate circuitry optimized for graphics and video processing, including, for example, video output circuitry, and constitutes a graphics processing unit (GPU). Alternatively or additionally, the one or more parallel processor(s) 112 can incorporate circuitry optimized for general purpose processing, while

preserving the underlying computational architecture, described in greater detail herein. Components of the computing system 100 may be integrated with one or more other system elements on a single integrated circuit. For example, the one or more parallel processor(s) 112, memory hub 105, processor(s) 102, and I/O hub 107 can be integrated into a system on chip (SoC) integrated circuit. Alternatively, the components of the computing system 100 can be integrated into a single package to form a system in package (SIP) configuration. In one embodiment at least a portion of the components of the computing system 100 can be integrated into a multi-chip module (MCM), which can be interconnected with other multi-chip modules into a modular computing system.

[0058] It will be appreciated that the computing system 100 shown herein is illustrative and that variations and modifications are possible. The connection topology, including the number and arrangement of bridges, the number of processor(s) 102, and the number of parallel processor(s) 112, may be modified as desired. For instance, system memory 104 can be connected to the processor(s) 102 directly rather than through a bridge, while other devices communicate with system memory 104 via the memory hub 105 and the processor(s) 102. In other alternative topologies, the parallel processor(s) 112 are connected to the I/O hub 107 or directly to one of the one or more processor(s) 102, rather than to the memory hub 105. In other embodiments, the I/O hub 107 and memory hub 105 may be integrated into a single chip. It is also possible that two or more sets of processor(s) 102 are attached via multiple sockets, which can couple with two or more instances of the parallel processor(s) 112.

[0059] Some of the particular components shown herein are optional and may not be included in all implementations of the computing system 100. For example, any number of add-in cards or peripherals may be supported, or some components may be eliminated. Furthermore, some architectures may use different terminology for components similar to those illustrated in FIG. 1. For example, the memory hub 105 may be referred to as a Northbridge in some architectures, while the I/O hub 107 may be referred to as a Southbridge.

[0060] **FIG. 2A** illustrates a parallel processor 200. The parallel processor 200 may be a GPU, GPGPU or the like as described herein. The various components of the parallel processor 200 may be implemented using one or more integrated circuit devices, such as programmable processors, application specific integrated circuits (ASICs), or field programmable gate arrays (FPGA). The illustrated parallel processor 200 may be the, or one of the parallel processor(s) 112 shown in FIG. 1.

[0061] The parallel processor 200 includes a parallel processing unit 202. The parallel processing unit includes an I/O unit 204 that enables communication with other devices, including other instances of the parallel processing unit 202. The I/O unit 204 may be directly

connected to other devices. For instance, the I/O unit 204 connects with other devices via the use of a hub or switch interface, such as memory hub 105. The connections between the memory hub 105 and the I/O unit 204 form a communication link 113. Within the parallel processing unit 202, the I/O unit 204 connects with a host interface 206 and a memory crossbar 216, where the host interface 206 receives commands directed to performing processing operations and the memory crossbar 216 receives commands directed to performing memory operations.

[0062] When the host interface 206 receives a command buffer via the I/O unit 204, the host interface 206 can direct work operations to perform those commands to a front end 208. In one embodiment the front end 208 couples with a scheduler 210, which is configured to distribute commands or other work items to a processing cluster array 212. The scheduler 210 ensures that the processing cluster array 212 is properly configured and in a valid state before tasks are distributed to the processing clusters of the processing cluster array 212. The scheduler 210 may be implemented via firmware logic executing on a microcontroller. The microcontroller implemented scheduler 210 is configurable to perform complex scheduling and work distribution operations at coarse and fine granularity, enabling rapid preemption and context switching of threads executing on the processing array 212. Preferably, the host software can prove workloads for scheduling on the processing array 212 via one of multiple graphics processing doorbells. The workloads can then be automatically distributed across the processing array 212 by the scheduler 210 logic within the scheduler microcontroller.

[0063] The processing cluster array 212 can include up to "N" processing clusters (e.g., cluster 214A, cluster 214B, through cluster 214N). Each cluster 214A-214N of the processing cluster array 212 can execute a large number of concurrent threads. The scheduler 210 can allocate work to the clusters 214A-214N of the processing cluster array 212 using various scheduling and/or work distribution algorithms, which may vary depending on the workload arising for each type of program or computation. The scheduling can be handled dynamically by the scheduler 210, or can be assisted in part by compiler logic during compilation of program logic configured for execution by the processing cluster array 212. Optionally, different clusters 214A-214N of the processing cluster array 212 can be allocated for processing different types of programs or for performing different types of computations.

[0064] The processing cluster array 212 can be configured to perform various types of parallel processing operations. For example, the cluster array 212 is configured to perform general-purpose parallel compute operations. For example, the processing cluster array 212 can include logic to execute processing tasks including filtering of video and/or audio data,

performing modeling operations, including physics operations, and performing data transformations.

[0065] The processing cluster array 212 is configured to perform parallel graphics processing operations. In such embodiments in which the parallel processor 200 is configured to perform graphics processing operations, the processing cluster array 212 can include additional logic to support the execution of such graphics processing operations, including, but not limited to texture sampling logic to perform texture operations, as well as tessellation logic and other vertex processing logic. Additionally, the processing cluster array 212 can be configured to execute graphics processing related shader programs such as, but not limited to vertex shaders, tessellation shaders, geometry shaders, and pixel shaders. The parallel processing unit 202 can transfer data from system memory via the I/O unit 204 for processing. During processing the transferred data can be stored to on-chip memory (e.g., parallel processor memory 222) during processing, then written back to system memory.

[0066] In embodiments in which the parallel processing unit 202 is used to perform graphics processing, the scheduler 210 may be configured to divide the processing workload into approximately equal sized tasks, to better enable distribution of the graphics processing operations to multiple clusters 214A-214N of the processing cluster array 212. In some of these embodiments, portions of the processing cluster array 212 can be configured to perform different types of processing. For example a first portion may be configured to perform vertex shading and topology generation, a second portion may be configured to perform tessellation and geometry shading, and a third portion may be configured to perform pixel shading or other screen space operations, to produce a rendered image for display. Intermediate data produced by one or more of the clusters 214A-214N may be stored in buffers to allow the intermediate data to be transmitted between clusters 214A-214N for further processing.

[0067] During operation, the processing cluster array 212 can receive processing tasks to be executed via the scheduler 210, which receives commands defining processing tasks from front end 208. For graphics processing operations, processing tasks can include indices of data to be processed, e.g., surface (patch) data, primitive data, vertex data, and/or pixel data, as well as state parameters and commands defining how the data is to be processed (e.g., what program is to be executed). The scheduler 210 may be configured to fetch the indices corresponding to the tasks or may receive the indices from the front end 208. The front end 208 can be configured to ensure the processing cluster array 212 is configured to a valid state before the workload specified by incoming command buffers (e.g., batch-buffers, push buffers, etc.) is initiated.

[0068] Each of the one or more instances of the parallel processing unit 202 can couple with parallel processor memory 222. The parallel processor memory 222 can be accessed via the

memory crossbar 216, which can receive memory requests from the processing cluster array 212 as well as the I/O unit 204. The memory crossbar 216 can access the parallel processor memory 222 via a memory interface 218. The memory interface 218 can include multiple partition units (e.g., partition unit 220A, partition unit 220B, through partition unit 220N) that can each couple to a portion (e.g., memory unit) of parallel processor memory 222. The number of partition units 220A-220N may be configured to be equal to the number of memory units, such that a first partition unit 220A has a corresponding first memory unit 224A, a second partition unit 220B has a corresponding memory unit 224B, and an Nth partition unit 220N has a corresponding Nth memory unit 224N. In other embodiments, the number of partition units 220A-220N may not be equal to the number of memory devices.

[0069] The memory units 224A-224N can include various types of memory devices, including dynamic random-access memory (DRAM) or graphics random access memory, such as synchronous graphics random access memory (SGRAM), including graphics double data rate (GDDR) memory. Optionally, the memory units 224A-224N may also include 3D stacked memory, including but not limited to high bandwidth memory (HBM). Persons skilled in the art will appreciate that the specific implementation of the memory units 224A-224N can vary, and can be selected from one of various conventional designs. Render targets, such as frame buffers or texture maps may be stored across the memory units 224A-224N, allowing partition units 220A-220N to write portions of each render target in parallel to efficiently use the available bandwidth of parallel processor memory 222. In some embodiments, a local instance of the parallel processor memory 222 may be excluded in favor of a unified memory design that utilizes system memory in conjunction with local cache memory.

[0070] Optionally, any one of the clusters 214A-214N of the processing cluster array 212 has the ability to process data that will be written to any of the memory units 224A-224N within parallel processor memory 222. The memory crossbar 216 can be configured to transfer the output of each cluster 214A-214N to any partition unit 220A-220N or to another cluster 214A-214N, which can perform additional processing operations on the output. Each cluster 214A-214N can communicate with the memory interface 218 through the memory crossbar 216 to read from or write to various external memory devices. In one of the embodiments with the memory crossbar 216 the memory crossbar 216 has a connection to the memory interface 218 to communicate with the I/O unit 204, as well as a connection to a local instance of the parallel processor memory 222, enabling the processing units within the different processing clusters 214A-214N to communicate with system memory or other memory that is not local to the parallel processing unit 202. Generally, the memory crossbar 216 may, for example, by able to

use virtual channels to separate traffic streams between the clusters 214A-214N and the partition units 220A-220N.

[0071] While a single instance of the parallel processing unit 202 is illustrated within the parallel processor 200, any number of instances of the parallel processing unit 202 can be included. For example, multiple instances of the parallel processing unit 202 can be provided on a single add-in card, or multiple add-in cards can be interconnected. The different instances of the parallel processing unit 202 can be configured to inter-operate even if the different instances have different numbers of processing cores, different amounts of local parallel processor memory, and/or other configuration differences. Optionally, some instances of the parallel processing unit 202 can include higher precision floating point units relative to other instances. Systems incorporating one or more instances of the parallel processing unit 202 or the parallel processor 200 can be implemented in a variety of configurations and form factors, including but not limited to desktop, laptop, or handheld personal computers, servers, workstations, game consoles, and/or embedded systems.

[0072] **FIG. 2B** is a block diagram of a partition unit 220. The partition unit 220 may be an instance of one of the partition units 220A-220N of FIG. 2A. As illustrated, the partition unit 220 includes an L2 cache 221, a frame buffer interface 225, and a ROP 226 (raster operations unit). The L2 cache 221 is a read/write cache that is configured to perform load and store operations received from the memory crossbar 216 and ROP 226. Read misses and urgent write-back requests are output by L2 cache 221 to frame buffer interface 225 for processing. Updates can also be sent to the frame buffer via the frame buffer interface 225 for processing. In one embodiment the frame buffer interface 225 interfaces with one of the memory units in parallel processor memory, such as the memory units 224A-224N of FIG. 2A (e.g., within parallel processor memory 222). The partition unit 220 may additionally or alternatively also interface with one of the memory units in parallel processor memory via a memory controller (not shown).

[0073] In graphics applications, the ROP 226 is a processing unit that performs raster operations such as stencil, z test, blending, and the like. The ROP 226 then outputs processed graphics data that is stored in graphics memory. In some embodiments the ROP 226 includes compression logic to compress depth or color data that is written to memory and decompress depth or color data that is read from memory. The compression logic can be lossless compression logic that makes use of one or more of multiple compression algorithms. The type of compression that is performed by the ROP 226 can vary based on the statistical characteristics of the data to be compressed. For example, in one embodiment, delta color compression is performed on depth and color data on a per-tile basis.

[0074] The ROP 226 may be included within each processing cluster (e.g., cluster 214A-214N of FIG. 2A) instead of within the partition unit 220. In such embodiment, read and write requests for pixel data are transmitted over the memory crossbar 216 instead of pixel fragment data. The processed graphics data may be displayed on a display device, such as one of the one or more display device(s) 110 of FIG. 1, routed for further processing by the processor(s) 102, or routed for further processing by one of the processing entities within the parallel processor 200 of FIG. 2A.

[0075] **FIG. 2C** is a block diagram of a processing cluster 214 within a parallel processing unit. For example, the processing cluster is an instance of one of the processing clusters 214A-214N of FIG. 2A. The processing cluster 214 can be configured to execute many threads in parallel, where the term "thread" refers to an instance of a particular program executing on a particular set of input data. Optionally, single-instruction, multiple-data (SIMD) instruction issue techniques may be used to support parallel execution of a large number of threads without providing multiple independent instruction units. Alternatively, single-instruction, multiple-thread (SIMT) techniques may be used to support parallel execution of a large number of generally synchronized threads, using a common instruction unit configured to issue instructions to a set of processing engines within each one of the processing clusters. Unlike a SIMD execution regime, where all processing engines typically execute identical instructions, SIMT execution allows different threads to more readily follow divergent execution paths through a given thread program. Persons skilled in the art will understand that a SIMD processing regime represents a functional subset of a SIMT processing regime.

[0076] Operation of the processing cluster 214 can be controlled via a pipeline manager 232 that distributes processing tasks to SIMT parallel processors. The pipeline manager 232 receives instructions from the scheduler 210 of FIG. 2A and manages execution of those instructions via a graphics multiprocessor 234 and/or a texture unit 236. The illustrated graphics multiprocessor 234 is an exemplary instance of a SIMT parallel processor. However, various types of SIMT parallel processors of differing architectures may be included within the processing cluster 214. One or more instances of the graphics multiprocessor 234 can be included within a processing cluster 214. The graphics multiprocessor 234 can process data and a data crossbar 240 can be used to distribute the processed data to one of multiple possible destinations, including other shader units. The pipeline manager 232 can facilitate the distribution of processed data by specifying destinations for processed data to be distributed via the data crossbar 240.

[0077] Each graphics multiprocessor 234 within the processing cluster 214 can include an identical set of functional execution logic (e.g., arithmetic logic units, load-store units, etc.). The functional execution logic can be configured in a pipelined manner in which new instructions

can be issued before previous instructions are complete. The functional execution logic supports a variety of operations including integer and floating-point arithmetic, comparison operations, Boolean operations, bit-shifting, and computation of various algebraic functions. The same functional-unit hardware could be leveraged to perform different operations and any combination of functional units may be present.

[0078] The instructions transmitted to the processing cluster 214 constitutes a thread. A set of threads executing across the set of parallel processing engines is a thread group. A thread group executes the same program on different input data. Each thread within a thread group can be assigned to a different processing engine within a graphics multiprocessor 234. A thread group may include fewer threads than the number of processing engines within the graphics multiprocessor 234. When a thread group includes fewer threads than the number of processing engines, one or more of the processing engines may be idle during cycles in which that thread group is being processed. A thread group may also include more threads than the number of processing engines within the graphics multiprocessor 234. When the thread group includes more threads than the number of processing engines within the graphics multiprocessor 234, processing can be performed over consecutive clock cycles. Optionally, multiple thread groups can be executed concurrently on the graphics multiprocessor 234.

[0079] The graphics multiprocessor 234 may include an internal cache memory to perform load and store operations. Optionally, the graphics multiprocessor 234 can forego an internal cache and use a cache memory (e.g., L1 cache 248) within the processing cluster 214. Each graphics multiprocessor 234 also has access to L2 caches within the partition units (e.g., partition units 220A-220N of FIG. 2A) that are shared among all processing clusters 214 and may be used to transfer data between threads. The graphics multiprocessor 234 may also access off-chip global memory, which can include one or more of local parallel processor memory and/or system memory. Any memory external to the parallel processing unit 202 may be used as global memory. Embodiments in which the processing cluster 214 includes multiple instances of the graphics multiprocessor 234 can share common instructions and data, which may be stored in the L1 cache 248.

[0080] Each processing cluster 214 may include an MMU 245 (memory management unit) that is configured to map virtual addresses into physical addresses. In other embodiments, one or more instances of the MMU 245 may reside within the memory interface 218 of FIG. 2A. The MMU 245 includes a set of page table entries (PTEs) used to map a virtual address to a physical address of a tile and optionally a cache line index. The MMU 245 may include address translation lookaside buffers (TLB) or caches that may reside within the graphics multiprocessor 234 or the L1 cache or processing cluster 214. The physical address is processed to distribute

surface data access locality to allow efficient request interleaving among partition units. The cache line index may be used to determine whether a request for a cache line is a hit or miss.

[0081] In graphics and computing applications, a processing cluster 214 may be configured such that each graphics multiprocessor 234 is coupled to a texture unit 236 for performing texture mapping operations, e.g., determining texture sample positions, reading texture data, and filtering the texture data. Texture data is read from an internal texture L1 cache (not shown) or in some embodiments from the L1 cache within graphics multiprocessor 234 and is fetched from an L2 cache, local parallel processor memory, or system memory, as needed. Each graphics multiprocessor 234 outputs processed tasks to the data crossbar 240 to provide the processed task to another processing cluster 214 for further processing or to store the processed task in an L2 cache, local parallel processor memory, or system memory via the memory crossbar 216. A preROP 242 (pre-raster operations unit) is configured to receive data from graphics multiprocessor 234, direct data to ROP units, which may be located with partition units as described herein (e.g., partition units 220A-220N of FIG. 2A). The preROP 242 unit can perform optimizations for color blending, organize pixel color data, and perform address translations.

[0082] It will be appreciated that the core architecture described herein is illustrative and that variations and modifications are possible. Any number of processing units, e.g., graphics multiprocessor 234, texture units 236, preROPs 242, etc., may be included within a processing cluster 214. Further, while only one processing cluster 214 is shown, a parallel processing unit as described herein may include any number of instances of the processing cluster 214. Optionally, each processing cluster 214 can be configured to operate independently of other processing clusters 214 using separate and distinct processing units, L1 caches, etc.

[0083] FIG. 2D shows an example of the graphics multiprocessor 234 in which the graphics multiprocessor 234 couples with the pipeline manager 232 of the processing cluster 214. The graphics multiprocessor 234 has an execution pipeline including but not limited to an instruction cache 252, an instruction unit 254, an address mapping unit 256, a register file 258, one or more general purpose graphics processing unit (GPGPU) cores 262, and one or more load/store units 266. The GPGPU cores 262 and load/store units 266 are coupled with cache memory 272 and shared memory 270 via a memory and cache interconnect 268. The graphics multiprocessor 234 may additionally include tensor and/or ray-tracing cores 263 that include hardware logic to accelerate matrix and/or ray-tracing operations.

[0084] The instruction cache 252 may receive a stream of instructions to execute from the pipeline manager 232. The instructions are cached in the instruction cache 252 and dispatched for execution by the instruction unit 254. The instruction unit 254 can dispatch instructions as

thread groups (e.g., warps), with each thread of the thread group assigned to a different execution unit within GPGPU core 262. An instruction can access any of a local, shared, or global address space by specifying an address within a unified address space. The address mapping unit 256 can be used to translate addresses in the unified address space into a distinct memory address that can be accessed by the load/store units 266.

[0085] The register file 258 provides a set of registers for the functional units of the graphics multiprocessor 234. The register file 258 provides temporary storage for operands connected to the data paths of the functional units (e.g., GPGPU cores 262, load/store units 266) of the graphics multiprocessor 234. The register file 258 may be divided between each of the functional units such that each functional unit is allocated a dedicated portion of the register file 258. For example, the register file 258 may be divided between the different warps being executed by the graphics multiprocessor 234.

[0086] The GPGPU cores 262 can each include floating point units (FPUs) and/or integer arithmetic logic units (ALUs) that are used to execute instructions of the graphics multiprocessor 234. In some implementations, the GPGPU cores 262 can include hardware logic that may otherwise reside within the tensor and/or ray-tracing cores 263. The GPGPU cores 262 can be similar in architecture or can differ in architecture. For example and in one embodiment, a first portion of the GPGPU cores 262 include a single precision FPU and an integer ALU while a second portion of the GPGPU cores include a double precision FPU. Optionally, the FPUs can implement the IEEE 754-2008 standard for floating point arithmetic or enable variable precision floating point arithmetic. The graphics multiprocessor 234 can additionally include one or more fixed function or special function units to perform specific functions such as copy rectangle or pixel blending operations. One or more of the GPGPU cores can also include fixed or special function logic.

[0087] The GPGPU cores 262 may include SIMD logic capable of performing a single instruction on multiple sets of data. Optionally, GPGPU cores 262 can physically execute SIMD4, SIMD8, and SIMD16 instructions and logically execute SIMD1, SIMD2, and SIMD32 instructions. The SIMD instructions for the GPGPU cores can be generated at compile time by a shader compiler or automatically generated when executing programs written and compiled for single program multiple data (SPMD) or SIMT architectures. Multiple threads of a program configured for the SIMT execution model can be executed via a single SIMD instruction. For example and in one embodiment, eight SIMT threads that perform the same or similar operations can be executed in parallel via a single SIMD8 logic unit.

[0088] The memory and cache interconnect 268 is an interconnect network that connects each of the functional units of the graphics multiprocessor 234 to the register file 258 and to the

shared memory 270. For example, the memory and cache interconnect 268 is a crossbar interconnect that allows the load/store unit 266 to implement load and store operations between the shared memory 270 and the register file 258. The register file 258 can operate at the same frequency as the GPGPU cores 262, thus data transfer between the GPGPU cores 262 and the register file 258 is very low latency. The shared memory 270 can be used to enable communication between threads that execute on the functional units within the graphics multiprocessor 234. The cache memory 272 can be used as a data cache for example, to cache texture data communicated between the functional units and the texture unit 236. The shared memory 270 can also be used as a program managed cached. Threads executing on the GPGPU cores 262 can programmatically store data within the shared memory in addition to the automatically cached data that is stored within the cache memory 272.

[0089] **FIG. 3A-3C** illustrate additional graphics multiprocessors, according to embodiments. FIG. 3A-3B illustrate graphics multiprocessors 325, 350, which are related to the graphics multiprocessor 234 of FIG. 2C and may be used in place of one of those. Therefore, the disclosure of any features in combination with the graphics multiprocessor 234 herein also discloses a corresponding combination with the graphics multiprocessor(s) 325, 350, but is not limited to such. FIG. 3C illustrates a graphics processing unit (GPU) 380 which includes dedicated sets of graphics processing resources arranged into multi-core groups 365A-365N, which correspond to the graphics multiprocessors 325, 350. The illustrated graphics multiprocessors 325, 350 and the multi-core groups 365A-365N can be streaming multiprocessors (SM) capable of simultaneous execution of a large number of execution threads.

[0090] The graphics multiprocessor 325 of FIG. 3A includes multiple additional instances of execution resource units relative to the graphics multiprocessor 234 of FIG. 2D. For example, the graphics multiprocessor 325 can include multiple instances of the instruction unit 332A-332B, register file 334A-334B, and texture unit(s) 344A-344B. The graphics multiprocessor 325 also includes multiple sets of graphics or compute execution units (e.g., GPGPU core 336A-336B, tensor core 337A-337B, ray-tracing core 338A-338B) and multiple sets of load/store units 340A-340B. The execution resource units have a common instruction cache 330, texture and/or data cache memory 342, and shared memory 346.

[0091] The various components can communicate via an interconnect fabric 327. The interconnect fabric 327 may include one or more crossbar switches to enable communication between the various components of the graphics multiprocessor 325. The interconnect fabric 327 may be a separate, high-speed network fabric layer upon which each component of the graphics multiprocessor 325 is stacked. The components of the graphics multiprocessor 325 communicate with remote components via the interconnect fabric 327. For example, the

GPGPU cores 336A-336B, 337A-337B, and 3378A-338B can each communicate with shared memory 346 via the interconnect fabric 327. The interconnect fabric 327 can arbitrate communication within the graphics multiprocessor 325 to ensure a fair bandwidth allocation between components.

[0092] The graphics multiprocessor 350 of FIG. 3B includes multiple sets of execution resources 356A-356D, where each set of execution resource includes multiple instruction units, register files, GPGPU cores, and load store units, as illustrated in FIG. 2D and FIG. 3A. The execution resources 356A-356D can work in concert with texture unit(s) 360A-360D for texture operations, while sharing an instruction cache 354, and shared memory 353. For example, the execution resources 356A-356D can share an instruction cache 354 and shared memory 353, as well as multiple instances of a texture and/or data cache memory 358A-358B. The various components can communicate via an interconnect fabric 352 similar to the interconnect fabric 327 of FIG. 3A.

[0093] Persons skilled in the art will understand that the architecture described in FIG. 1, 2A-2D, and 3A-3B are descriptive and not limiting as to the scope of the present embodiments. Thus, the techniques described herein may be implemented on any properly configured processing unit, including, without limitation, one or more mobile application processors, one or more desktop or server central processing units (CPUs) including multi-core CPUs, one or more parallel processing units, such as the parallel processing unit 202 of FIG. 2A, as well as one or more graphics processors or special purpose processing units, without departure from the scope of the embodiments described herein.

[0094] The parallel processor or GPGPU as described herein may be communicatively coupled to host/processor cores to accelerate graphics operations, machine-learning operations, pattern analysis operations, and various general-purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or other interconnect (e.g., a high-speed interconnect such as PCIe or NVLink). In other embodiments, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

[0095] **FIG. 3C** illustrates a graphics processing unit (GPU) 380 which includes dedicated sets of graphics processing resources arranged into multi-core groups 365A-365N. While the details of only a single multi-core group 365A are provided, it will be appreciated that the other multi-core groups 365B-365N may be equipped with the same or similar sets of graphics

processing resources. Details described with respect to the multi-core groups 365A-365N may also apply to any graphics multiprocessor 234, 325, 350 described herein.

[0096] As illustrated, a multi-core group 365A may include a set of graphics cores 370, a set of tensor cores 371, and a set of ray tracing cores 372. A scheduler/dispatcher 368 schedules and dispatches the graphics threads for execution on the various cores 370, 371, 372. A set of register files 369 store operand values used by the cores 370, 371, 372 when executing the graphics threads. These may include, for example, integer registers for storing integer values, floating point registers for storing floating point values, vector registers for storing packed data elements (integer and/or floating-point data elements) and tile registers for storing tensor/matrix values. The tile registers may be implemented as combined sets of vector registers.

[0097] One or more combined level 1 (L1) caches and shared memory units 373 store graphics data such as texture data, vertex data, pixel data, ray data, bounding volume data, etc., locally within each multi-core group 365A. One or more texture units 374 can also be used to perform texturing operations, such as texture mapping and sampling. A Level 2 (L2) cache 375 shared by all or a subset of the multi-core groups 365A-365N stores graphics data and/or instructions for multiple concurrent graphics threads. As illustrated, the L2 cache 375 may be shared across a plurality of multi-core groups 365A-365N. One or more memory controllers 367 couple the GPU 380 to a memory 366 which may be a system memory (e.g., DRAM) and/or a dedicated graphics memory (e.g., GDDR6 memory).

[0098] Input/output (I/O) circuitry 363 couples the GPU 380 to one or more I/O devices 362 such as digital signal processors (DSPs), network controllers, or user input devices. An on-chip interconnect may be used to couple the I/O devices 362 to the GPU 380 and memory 366. One or more I/O memory management units (IOMMUs) 364 of the I/O circuitry 363 couple the I/O devices 362 directly to the system memory 366. Optionally, the IOMMU 364 manages multiple sets of page tables to map virtual addresses to physical addresses in system memory 366. The I/O devices 362, CPU(s) 361, and GPU(s) 380 may then share the same virtual address space.

[0099] In one implementation of the IOMMU 364, the IOMMU 364 supports virtualization. In this case, it may manage a first set of page tables to map guest/graphics virtual addresses to guest/graphics physical addresses and a second set of page tables to map the guest/graphics physical addresses to system/host physical addresses (e.g., within system memory 366). The base addresses of each of the first and second sets of page tables may be stored in control registers and swapped out on a context switch (e.g., so that the new context is provided with access to the relevant set of page tables). While not illustrated in FIG. 3C, each of the cores 370, 371, 372 and/or multi-core groups 365A-365N may include translation lookaside buffers (TLBs)

to cache guest virtual to guest physical translations, guest physical to host physical translations, and guest virtual to host physical translations.

[0100] The CPUs 361, GPUs 380, and I/O devices 362 may be integrated on a single semiconductor chip and/or chip package. The illustrated memory 366 may be integrated on the same chip or may be coupled to the memory controllers 367 via an off-chip interface. In one implementation, the memory 366 comprises GDDR6 memory which shares the same virtual address space as other physical system-level memories, although the underlying principles described herein are not limited to this specific implementation.

[0101] The tensor cores 371 may include a plurality of execution units specifically designed to perform matrix operations, which are the fundamental compute operation used to perform deep learning operations. For example, simultaneous matrix multiplication operations may be used for neural network training and inferencing. The tensor cores 371 may perform matrix processing using a variety of operand precisions including single precision floating-point (e.g., 32 bits), half-precision floating point (e.g., 16 bits), integer words (16 bits), bytes (8 bits), and half-bytes (4 bits). For example, a neural network implementation extracts features of each rendered scene, potentially combining details from multiple frames, to construct a high-quality final image.

[0102] In deep learning implementations, parallel matrix multiplication work may be scheduled for execution on the tensor cores 371. The training of neural networks, in particular, requires a significant number matrix dot product operations. In order to process an inner-product formulation of an N x N x N matrix multiply, the tensor cores 371 may include at least N dot-product processing elements. Before the matrix multiply begins, one entire matrix is loaded into tile registers and at least one column of a second matrix is loaded each cycle for N cycles. Each cycle, there are N dot products that are processed.

[0103] Matrix elements may be stored at different precisions depending on the particular implementation, including 16-bit words, 8-bit bytes (e.g., INT8) and 4-bit half-bytes (e.g., INT4). Different precision modes may be specified for the tensor cores 371 to ensure that the most efficient precision is used for different workloads (e.g., such as inferencing workloads which can tolerate quantization to bytes and half-bytes).

[0104] The ray tracing cores 372 may accelerate ray tracing operations for both real-time ray tracing and non-real-time ray tracing implementations. In particular, the ray tracing cores 372 may include ray traversal/intersection circuitry for performing ray traversal using bounding volume hierarchies (BVHs) and identifying intersections between rays and primitives enclosed within the BVH volumes. The ray tracing cores 372 may also include circuitry for performing depth testing and culling (e.g., using a Z buffer or similar arrangement). In one implementation,

the ray tracing cores 372 perform traversal and intersection operations in concert with the image denoising techniques described herein, at least a portion of which may be executed on the tensor cores 371. For example, the tensor cores 371 may implement a deep learning neural network to perform denoising of frames generated by the ray tracing cores 372. However, the CPU(s) 361, graphics cores 370, and/or ray tracing cores 372 may also implement all or a portion of the denoising and/or deep learning algorithms.

[0105] In addition, as described above, a distributed approach to denoising may be employed in which the GPU 380 is in a computing device coupled to other computing devices over a network or high-speed interconnect. In this distributed approach, the interconnected computing devices may share neural network learning/training data to improve the speed with which the overall system learns to perform denoising for different types of image frames and/or different graphics applications.

[0106] The ray tracing cores 372 may process all BVH traversal and/or ray-primitive intersections, saving the graphics cores 370 from being overloaded with thousands of instructions per ray. For example, each ray tracing core 372 includes a first set of specialized circuitry for performing bounding box tests (e.g., for traversal operations) and/or a second set of specialized circuitry for performing the ray-triangle intersection tests (e.g., intersecting rays which have been traversed). Thus, for example, the multi-core group 365A can simply launch a ray probe, and the ray tracing cores 372 independently perform ray traversal and intersection and return hit data (e.g., a hit, no hit, multiple hits, etc.) to the thread context. The other cores 370, 371 are freed to perform other graphics or compute work while the ray tracing cores 372 perform the traversal and intersection operations.

[0107] Optionally, each ray tracing core 372 may include a traversal unit to perform BVH testing operations and/or an intersection unit which performs ray-primitive intersection tests. The intersection unit generates a "hit", "no hit", or "multiple hit" response, which it provides to the appropriate thread. During the traversal and intersection operations, the execution resources of the other cores (e.g., graphics cores 370 and tensor cores 371) are freed to perform other forms of graphics work.

[0108] In one optional embodiment described below, a hybrid rasterization/ray tracing approach is used in which work is distributed between the graphics cores 370 and ray tracing cores 372.

[0109] The ray tracing cores 372 (and/or other cores 370, 371) may include hardware support for a ray tracing instruction set such as Microsoft's DirectX Ray Tracing (DXR) which includes a DispatchRays command, as well as ray-generation, closest-hit, any-hit, and miss shaders, which enable the assignment of unique sets of shaders and textures for each object.

Another ray tracing platform which may be supported by the ray tracing cores 372, graphics cores 370 and tensor cores 371 is Vulkan 1.1.85. Note, however, that the underlying principles described herein are not limited to any particular ray tracing ISA.

[0110] In general, the various cores 372, 371, 370 may support a ray tracing instruction set that includes instructions/functions for one or more of ray generation, closest hit, any hit, ray-primitive intersection, per-primitive and hierarchical bounding box construction, miss, visit, and exceptions. More specifically, a preferred embodiment includes ray tracing instructions to perform one or more of the following functions:

[0111] Ray Generation – Ray generation instructions may be executed for each pixel, sample, or other user-defined work assignment.

[0112] Closest Hit – A closest hit instruction may be executed to locate the closest intersection point of a ray with primitives within a scene.

[0113] Any Hit – An any hit instruction identifies multiple intersections between a ray and primitives within a scene, potentially to identify a new closest intersection point.

[0114] Intersection – An intersection instruction performs a ray-primitive intersection test and outputs a result.

[0115] Per-primitive Bounding box Construction – This instruction builds a bounding box around a given primitive or group of primitives (e.g., when building a new BVH or other acceleration data structure).

[0116] Miss – Indicates that a ray misses all geometry within a scene, or specified region of a scene.

[0117] Visit – Indicates the children volumes a ray will traverse.

[0118] Exceptions – Includes various types of exception handlers (e.g., invoked for various error conditions).

Techniques for GPU to Host Processor Interconnection

[0119] **FIG. 4A** illustrates an exemplary architecture in which a plurality of GPUs 410-413, e.g. such as the parallel processors 200 shown in FIG. 2A, are communicatively coupled to a plurality of multi-core processors 405-406 over high-speed links 440A-440D (e.g., buses, point-to-point interconnects, etc.). The high-speed links 440A-440D may support a communication throughput of 4GB/s, 30GB/s, 80GB/s or higher, depending on the implementation. Various interconnect protocols may be used including, but not limited to, PCIe 4.0 or 5.0 and NVLink 2.0. However, the underlying principles described herein are not limited to any particular communication protocol or throughput.

[0120] Two or more of the GPUs 410-413 may be interconnected over high-speed links 442A-442B, which may be implemented using the same or different protocols/links than those

used for high-speed links 440A-440D. Similarly, two or more of the multi-core processors 405-406 may be connected over high speed link 443 which may be symmetric multi-processor (SMP) buses operating at 20GB/s, 30GB/s, 120GB/s or higher. Alternatively, all communication between the various system components shown in **FIG. 4A** may be accomplished using the same protocols/links (e.g., over a common interconnection fabric). As mentioned, however, the underlying principles described herein are not limited to any particular type of interconnect technology.

[0121] Each multi-core processor 405-406 may be communicatively coupled to a processor memory 401-402, via memory interconnects 430A-430B, respectively, and each GPU 410-413 is communicatively coupled to GPU memory 420-423 over GPU memory interconnects 450A-450D, respectively. The memory interconnects 430A-430B and 450A-450D may utilize the same or different memory access technologies. By way of example, and not limitation, the processor memories 401-402 and GPU memories 420-423 may be volatile memories such as dynamic random-access memories (DRAMs) (including stacked DRAMs), Graphics DDR SDRAM (GDDR) (e.g., GDDR5, GDDR6), or High Bandwidth Memory (HBM) and/or may be non-volatile memories such as 3D XPoint/Optane or Nano-Ram. For example, some portion of the memories may be volatile memory and another portion may be non-volatile memory (e.g., using a two-level memory (2LM) hierarchy).

[0122] As described below, although the various processors 405-406 and GPUs 410-413 may be physically coupled to a particular memory 401-402, 420-423, respectively, a unified memory architecture may be implemented in which the same virtual system address space (also referred to as the "effective address" space) is distributed among all of the various physical memories. For example, processor memories 401-402 may each comprise 64GB of the system memory address space and GPU memories 420-423 may each comprise 32GB of the system memory address space (resulting in a total of 256GB addressable memory in this example).

[0123] **FIG. 4B** illustrates additional optional details for an interconnection between a multi-core processor 407 and a graphics acceleration module 446. The graphics acceleration module 446 may include one or more GPU chips integrated on a line card which is coupled to the processor 407 via the high-speed link 440. Alternatively, the graphics acceleration module 446 may be integrated on the same package or chip as the processor 407.

[0124] The illustrated processor 407 includes a plurality of cores 460A-460D, each with a translation lookaside buffer 461A-461D and one or more caches 462A-462D. The cores may include various other components for executing instructions and processing data which are not illustrated to avoid obscuring the underlying principles of the components described herein (e.g., instruction fetch units, branch prediction units, decoders, execution units, reorder buffers, etc.).

The caches 462A-462D may comprise level 1 (L1) and level 2 (L2) caches. In addition, one or more shared caches 456 may be included in the caching hierarchy and shared by sets of the cores 460A-460D. For example, one embodiment of the processor 407 includes 24 cores, each with its own L1 cache, twelve shared L2 caches, and twelve shared L3 caches. In this embodiment, one of the L2 and L3 caches are shared by two adjacent cores. The processor 407 and the graphics accelerator integration module 446 connect with system memory 441, which may include processor memories 401-402.

[0125] Coherency is maintained for data and instructions stored in the various caches 462A-462D, 456 and system memory 441 via inter-core communication over a coherence bus 464. For example, each cache may have cache coherency logic/circuitry associated therewith to communicate to over the coherence bus 464 in response to detected reads or writes to particular cache lines. In one implementation, a cache snooping protocol is implemented over the coherence bus 464 to snoop cache accesses. Cache snooping/coherency techniques are well understood by those of skill in the art and will not be described in detail here to avoid obscuring the underlying principles described herein.

[0126] A proxy circuit 425 may be provided that communicatively couples the graphics acceleration module 446 to the coherence bus 464, allowing the graphics acceleration module 446 to participate in the cache coherence protocol as a peer of the cores. In particular, an interface 435 provides connectivity to the proxy circuit 425 over high-speed link 440 (e.g., a PCIe bus, NVLink, etc.) and an interface 437 connects the graphics acceleration module 446 to the high-speed link 440.

[0127] In one implementation, an accelerator integration circuit 436 provides cache management, memory access, context management, and interrupt management services on behalf of a plurality of graphics processing engines 431, 432, N of the graphics acceleration module 446. The graphics processing engines 431, 432, N may each comprise a separate graphics processing unit (GPU). Alternatively, the graphics processing engines 431, 432, N may comprise different types of graphics processing engines within a GPU such as graphics execution units, media processing engines (e.g., video encoders/decoders), samplers, and blit engines. In other words, the graphics acceleration module may be a GPU with a plurality of graphics processing engines 431-432, N or the graphics processing engines 431-432, N may be individual GPUs integrated on a common package, line card, or chip.

[0128] The accelerator integration circuit 436 may include a memory management unit (MMU) 439 for performing various memory management functions such as virtual-to-physical memory translations (also referred to as effective-to-real memory translations) and memory access protocols for accessing system memory 441. The MMU 439 may also include a

translation lookaside buffer (TLB) (not shown) for caching the virtual/effective to physical/real address translations. In one implementation, a cache 438 stores commands and data for efficient access by the graphics processing engines 431-432, N. The data stored in cache 438 and graphics memories 433-434, M may be kept coherent with the core caches 462A-462D, 456 and system memory 411. As mentioned, this may be accomplished via proxy circuit 425 which takes part in the cache coherency mechanism on behalf of cache 438 and memories 433-434, M (e.g., sending updates to the cache 438 related to modifications/accesses of cache lines on processor caches 462A-462D, 456 and receiving updates from the cache 438).

[0129] A set of registers 445 store context data for threads executed by the graphics processing engines 431-432, N and a context management circuit 448 manages the thread contexts. For example, the context management circuit 448 may perform save and restore operations to save and restore contexts of the various threads during contexts switches (e.g., where a first thread is saved and a second thread is stored so that the second thread can be execute by a graphics processing engine). For example, on a context switch, the context management circuit 448 may store current register values to a designated region in memory (e.g., identified by a context pointer). It may then restore the register values when returning to the context. An interrupt management circuit 447, for example, may receive and processes interrupts received from system devices.

[0130] In one implementation, virtual/effective addresses from a graphics processing engine 431 are translated to real/physical addresses in system memory 411 by the MMU 439. Optionally, the accelerator integration circuit 436 supports multiple (e.g., 4, 8, 16) graphics accelerator modules 446 and/or other accelerator devices. The graphics accelerator module 446 may be dedicated to a single application executed on the processor 407 or may be shared between multiple applications. Optionally, a virtualized graphics execution environment is provided in which the resources of the graphics processing engines 431-432, N are shared with multiple applications or virtual machines (VMs). The resources may be subdivided into "slices" which are allocated to different VMs and/or applications based on the processing requirements and priorities associated with the VMs and/or applications.

[0131] Thus, the accelerator integration circuit 436 acts as a bridge to the system for the graphics acceleration module 446 and provides address translation and system memory cache services. In one embodiment, to facilitate the bridging functionality, the accelerator integration circuit 436 may also include shared I/O 497 (e.g., PCIe, USB) and hardware to enable system control of voltage, clocking, performance, thermals, and security. The shared I/O 497 may utilize separate physical connections or may traverse the high-speed link 440. In addition, the

accelerator integration circuit 436 may provide virtualization facilities for the host processor to manage virtualization of the graphics processing engines, interrupts, and memory management.

[0132] Because hardware resources of the graphics processing engines 431-432, N are mapped explicitly to the real address space seen by the host processor 407, any host processor can address these resources directly using an effective address value. One optional function of the accelerator integration circuit 436 is the physical separation of the graphics processing engines 431-432, N so that they appear to the system as independent units.

[0133] One or more graphics memories 433-434, M may be coupled to each of the graphics processing engines 431-432, N, respectively. The graphics memories 433-434, M store instructions and data being processed by each of the graphics processing engines 431-432, N. The graphics memories 433-434, M may be volatile memories such as DRAMs (including stacked DRAMs), GDDR memory (e.g., GDDR5, GDDR6), or HBM, and/or may be non-volatile memories such as 3D XPoint/Optane or Nano-Ram.

[0134] To reduce data traffic over the high-speed link 440, biasing techniques may be used to ensure that the data stored in graphics memories 433-434, M is data which will be used most frequently by the graphics processing engines 431-432, N and preferably not used by the cores 460A-460D (at least not frequently). Similarly, the biasing mechanism attempts to keep data needed by the cores (and preferably not the graphics processing engines 431-432, N) within the caches 462A-462D, 456 of the cores and system memory 411.

[0135] According to a variant shown in **FIG. 4C** the accelerator integration circuit 436 is integrated within the processor 407. The graphics processing engines 431-432, N communicate directly over the high-speed link 440 to the accelerator integration circuit 436 via interface 437 and interface 435 (which, again, may be utilize any form of bus or interface protocol). The accelerator integration circuit 436 may perform the same operations as those described with respect to **FIG. 4B**, but potentially at a higher throughput given its close proximity to the coherency bus 464 and caches 462A-462D, 456.

[0136] The embodiments described may support different programming models including a dedicated-process programming model (no graphics acceleration module virtualization) and shared programming models (with virtualization). The latter may include programming models which are controlled by the accelerator integration circuit 436 and programming models which are controlled by the graphics acceleration module 446.

[0137] In the embodiments of the dedicated process model, graphics processing engines 431-432, N may be dedicated to a single application or process under a single operating system. The single application can funnel other application requests to the graphics engines 431-432, N, providing virtualization within a VM/partition.

[0138] In the dedicated-process programming models, the graphics processing engines 431-432, N, may be shared by multiple VM/application partitions. The shared models require a system hypervisor to virtualize the graphics processing engines 431-432, N to allow access by each operating system. For single-partition systems without a hypervisor, the graphics processing engines 431-432, N are owned by the operating system. In both cases, the operating system can virtualize the graphics processing engines 431-432, N to provide access to each process or application.

[0139] For the shared programming model, the graphics acceleration module 446 or an individual graphics processing engine 431-432, N selects a process element using a process handle. The process elements may be stored in system memory 411 and be addressable using the effective address to real address translation techniques described herein. The process handle may be an implementation-specific value provided to the host process when registering its context with the graphics processing engine 431-432, N (that is, calling system software to add the process element to the process element linked list). The lower 16-bits of the process handle may be the offset of the process element within the process element linked list.

[0140] **FIG. 4D** illustrates an exemplary accelerator integration slice 490. As used herein, a "slice" comprises a specified portion of the processing resources of the accelerator integration circuit 436. Application effective address space 482 within system memory 411 stores process elements 483. The process elements 483 may be stored in response to GPU invocations 481 from applications 480 executed on the processor 407. A process element 483 contains the process state for the corresponding application 480. A work descriptor (WD) 484 contained in the process element 483 can be a single job requested by an application or may contain a pointer to a queue of jobs. In the latter case, the WD 484 is a pointer to the job request queue in the application's address space 482.

[0141] The graphics acceleration module 446 and/or the individual graphics processing engines 431-432, N can be shared by all or a subset of the processes in the system. For example, the technologies described herein may include an infrastructure for setting up the process state and sending a WD 484 to a graphics acceleration module 446 to start a job in a virtualized environment.

[0142] In one implementation, the dedicated-process programming model is implementation-specific. In this model, a single process owns the graphics acceleration module 446 or an individual graphics processing engine 431. Because the graphics acceleration module 446 is owned by a single process, the hypervisor initializes the accelerator integration circuit 436 for the owning partition and the operating system initializes the accelerator integration circuit 436 for the owning process at the time when the graphics acceleration module 446 is assigned.

[0143] In operation, a WD fetch unit 491 in the accelerator integration slice 490 fetches the next WD 484 which includes an indication of the work to be done by one of the graphics processing engines of the graphics acceleration module 446. Data from the WD 484 may be stored in registers 445 and used by the MMU 439, interrupt management circuit 447 and/or context management circuit 448 as illustrated. For example, the MMU 439 may include segment/page walk circuitry for accessing segment/page tables 486 within the OS virtual address space 485. The interrupt management circuit 447 may process interrupt events 492 received from the graphics acceleration module 446. When performing graphics operations, an effective address 493 generated by a graphics processing engine 431-432, N is translated to a real address by the MMU 439.

[0144] The same set of registers 445 may be duplicated for each graphics processing engine 431-432, N and/or graphics acceleration module 446 and may be initialized by the hypervisor or operating system. Each of these duplicated registers may be included in an accelerator integration slice 490. Exemplary registers that may be initialized by the hypervisor are shown in **Table 1**.

Table 1 – Hypervisor Initialized Registers

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

[0145] Exemplary registers that may be initialized by the operating system are shown in **Table 2**.

Table 2 – Operating System Initialized Registers

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

[0146] Each WD 484 may be specific to a particular graphics acceleration module 446 and/or graphics processing engine 431-432, N. It contains all the information a graphics processing engine 431-432, N requires to do its work or it can be a pointer to a memory location where the application has set up a command queue of work to be completed.

[0147] **FIG. 4E** illustrates additional optional details of a shared model. It includes a hypervisor real address space 498 in which a process element list 499 is stored. The hypervisor real address space 498 is accessible via a hypervisor 496 which virtualizes the graphics acceleration module engines for the operating system 495.

[0148] The shared programming models allow for all or a subset of processes from all or a subset of partitions in the system to use a graphics acceleration module 446. There are two programming models where the graphics acceleration module 446 is shared by multiple processes and partitions: time-sliced shared and graphics directed shared.

[0149] In this model, the system hypervisor 496 owns the graphics acceleration module 446 and makes its function available to all operating systems 495. For a graphics acceleration module 446 to support virtualization by the system hypervisor 496, the graphics acceleration module 446 may adhere to the following requirements: 1) An application's job request must be autonomous (that is, the state does not need to be maintained between jobs), or the graphics acceleration module 446 must provide a context save and restore mechanism. 2) An application's job request is guaranteed by the graphics acceleration module 446 to complete in a specified amount of time, including any translation faults, or the graphics acceleration module 446 provides the ability to preempt the processing of the job. 3) The graphics acceleration module 446 must be guaranteed fairness between processes when operating in the directed shared programming model.

[0150] For the shared model, the application 480 may be required to make an operating system 495 system call with a graphics acceleration module 446 type, a work descriptor (WD), an authority mask register (AMR) value, and a context save/restore area pointer (CSRP). The graphics acceleration module 446 type describes the targeted acceleration function for the system

call. The graphics acceleration module 446 type may be a system-specific value. The WD is formatted specifically for the graphics acceleration module 446 and can be in the form of a graphics acceleration module 446 command, an effective address pointer to a user-defined structure, an effective address pointer to a queue of commands, or any other data structure to describe the work to be done by the graphics acceleration module 446. In one embodiment, the AMR value is the AMR state to use for the current process. The value passed to the operating system is similar to an application setting the AMR. If the accelerator integration circuit 436 and graphics acceleration module 446 implementations do not support a User Authority Mask Override Register (UAMOR), the operating system may apply the current UAMOR value to the AMR value before passing the AMR in the hypervisor call. The hypervisor 496 may optionally apply the current Authority Mask Override Register (AMOR) value before placing the AMR into the process element 483. The CSRP may be one of the registers 445 containing the effective address of an area in the application's address space 482 for the graphics acceleration module 446 to save and restore the context state. This pointer is optional if no state is required to be saved between jobs or when a job is preempted. The context save/restore area may be pinned system memory.

[0151] Upon receiving the system call, the operating system 495 may verify that the application 480 has registered and been given the authority to use the graphics acceleration module 446. The operating system 495 then calls the hypervisor 496 with the information shown in **Table 3.**

Table 3 – OS to Hypervisor Call Parameters

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

[0152] Upon receiving the hypervisor call, the hypervisor 496 verifies that the operating system 495 has registered and been given the authority to use the graphics acceleration module

446. The hypervisor 496 then puts the process element 483 into the process element linked list for the corresponding graphics acceleration module 446 type. The process element may include the information shown in **Table 4**.

Table 4 – Process Element Information

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

[0153] The hypervisor may initialize a plurality of accelerator integration slice 490 registers 445.

[0154] As illustrated in **FIG. 4F**, in one optional implementation a unified memory addressable via a common virtual memory address space used to access the physical processor memories 401-402 and GPU memories 420-423 is employed. In this implementation, operations executed on the GPUs 410-413 utilize the same virtual/effective memory address space to access the processors memories 401-402 and vice versa, thereby simplifying programmability. A first portion of the virtual/effective address space may be allocated to the processor memory 401, a second portion to the second processor memory 402, a third portion to the GPU memory 420, and so on. The entire virtual/effective memory space (sometimes referred to as the effective address space) may thereby be distributed across each of the processor memories 401-402 and GPU memories 420-423, allowing any processor or GPU to access any physical memory with a virtual address mapped to that memory.

[0155] Bas/coherence management circuitry 494A-494E within one or more of the MMUs 439A-439E may be provided that ensures cache coherence between the caches of the host processors (e.g., 405) and the GPUs 410-413 and implements biasing techniques indicating the physical memories in which certain types of data should be stored. While multiple instances of bias/coherence management circuitry 494A-494E are illustrated in **FIG. 4F**, the bias/coherence circuitry may be implemented within the MMU of one or more host processors 405 and/or within the accelerator integration circuit 436.

[0156] The GPU-attached memory 420-423 may be mapped as part of system memory, and accessed using shared virtual memory (SVM) technology, but without suffering the typical performance drawbacks associated with full system cache coherence. The ability to GPU-attached memory 420-423 to be accessed as system memory without onerous cache coherence overhead provides a beneficial operating environment for GPU offload. This arrangement allows the host processor 405 software to setup operands and access computation results, without the overhead of tradition I/O DMA data copies. Such traditional copies involve driver calls, interrupts and memory mapped I/O (MMIO) accesses that are all inefficient relative to simple memory accesses. At the same time, the ability to access GPU attached memory 420-423 without cache coherence overheads can be critical to the execution time of an offloaded computation. In cases with substantial streaming write memory traffic, for example, cache coherence overhead can significantly reduce the effective write bandwidth seen by a GPU 410-413. The efficiency of operand setup, the efficiency of results access, and the efficiency of GPU computation all play a role in determining the effectiveness of GPU offload.

[0157] A selection of between GPU bias and host processor bias may be driven by a bias tracker data structure. A bias table may be used, for example, which may be a page-granular structure (i.e., controlled at the granularity of a memory page) that includes 1 or 2 bits per GPU-attached memory page. The bias table may be implemented in a stolen memory range of one or more GPU-attached memories 420-423, with or without a bias cache in the GPU 410-413 (e.g., to cache frequently/recently used entries of the bias table). Alternatively, the entire bias table may be maintained within the GPU.

[0158] In one implementation, the bias table entry associated with each access to the GPU-attached memory 420-423 is accessed prior the actual access to the GPU memory, causing the following operations. First, local requests from the GPU 410-413 that find their page in GPU bias are forwarded directly to a corresponding GPU memory 420-423. Local requests from the GPU that find their page in host bias are forwarded to the processor 405 (e.g., over a high-speed link as discussed above). Optionally, requests from the processor 405 that find the requested page in host processor bias complete the request like a normal memory read. Alternatively,

requests directed to a GPU-biased page may be forwarded to the GPU 410-413. The GPU may then transition the page to a host processor bias if it is not currently using the page.

[0159] The bias state of a page can be changed either by a software-based mechanism, a hardware-assisted software-based mechanism, or, for a limited set of cases, a purely hardware-based mechanism.

[0160] One mechanism for changing the bias state employs an API call (e.g. OpenCL), which, in turn, calls the GPU's device driver which, in turn, sends a message (or enqueues a command descriptor) to the GPU directing it to change the bias state and, for some transitions, perform a cache flushing operation in the host. The cache flushing operation is required for a transition from host processor 405 bias to GPU bias, but is not required for the opposite transition.

[0161] Cache coherency may be maintained by temporarily rendering GPU-biased pages uncacheable by the host processor 405. To access these pages, the processor 405 may request access from the GPU 410 which may or may not grant access right away, depending on the implementation. Thus, to reduce communication between the host processor 405 and GPU 410 it is beneficial to ensure that GPU-biased pages are those which are required by the GPU but not the host processor 405 and vice versa.

Graphics Processing Pipeline

[0162] **FIG. 5** illustrates a graphics processing pipeline 500. A graphics multiprocessor, such as graphics multiprocessor 234 as in FIG. 2D, graphics multiprocessor 325 of FIG. 3A, graphics multiprocessor 350 of FIG. 3B can implement the illustrated graphics processing pipeline 500. The graphics multiprocessor can be included within the parallel processing subsystems as described herein, such as the parallel processor 200 of FIG. 2A, which may be related to the parallel processor(s) 112 of FIG. 1 and may be used in place of one of those. The various parallel processing systems can implement the graphics processing pipeline 500 via one or more instances of the parallel processing unit (e.g., parallel processing unit 202 of FIG. 2A) as described herein. For example, a shader unit (e.g., graphics multiprocessor 234 of FIG. 2C) may be configured to perform the functions of one or more of a vertex processing unit 504, a tessellation control processing unit 508, a tessellation evaluation processing unit 512, a geometry processing unit 516, and a fragment/pixel processing unit 524. The functions of data assembler 502, primitive assemblers 506, 514, 518, tessellation unit 510, rasterizer 522, and raster operations unit 526 may also be performed by other processing engines within a processing cluster (e.g., processing cluster 214 of FIG. 2A) and a corresponding partition unit (e.g., partition unit 220A-220N of FIG. 2A). The graphics processing pipeline 500 may also be implemented using dedicated processing units for one or more functions. It is also possible that one or more

portions of the graphics processing pipeline 500 are performed by parallel processing logic within a general-purpose processor (e.g., CPU). Optionally, one or more portions of the graphics processing pipeline 500 can access on-chip memory (e.g., parallel processor memory 222 as in FIG. 2A) via a memory interface 528, which may be an instance of the memory interface 218 of FIG. 2A. The graphics processor pipeline 500 may also be implemented via a multi-core group 365A as in FIG. 3C.

[0163] The data assembler 502 is a processing unit that may collect vertex data for surfaces and primitives. The data assembler 502 then outputs the vertex data, including the vertex attributes, to the vertex processing unit 504. The vertex processing unit 504 is a programmable execution unit that executes vertex shader programs, lighting and transforming vertex data as specified by the vertex shader programs. The vertex processing unit 504 reads data that is stored in cache, local or system memory for use in processing the vertex data and may be programmed to transform the vertex data from an object-based coordinate representation to a world space coordinate space or a normalized device coordinate space.

[0164] A first instance of a primitive assembler 506 receives vertex attributes from the vertex processing unit 504. The primitive assembler 506 readings stored vertex attributes as needed and constructs graphics primitives for processing by tessellation control processing unit 508. The graphics primitives include triangles, line segments, points, patches, and so forth, as supported by various graphics processing application programming interfaces (APIs).

[0165] The tessellation control processing unit 508 treats the input vertices as control points for a geometric patch. The control points are transformed from an input representation from the patch (e.g., the patch's bases) to a representation that is suitable for use in surface evaluation by the tessellation evaluation processing unit 512. The tessellation control processing unit 508 can also compute tessellation factors for edges of geometric patches. A tessellation factor applies to a single edge and quantifies a view-dependent level of detail associated with the edge. A tessellation unit 510 is configured to receive the tessellation factors for edges of a patch and to tessellate the patch into multiple geometric primitives such as line, triangle, or quadrilateral primitives, which are transmitted to a tessellation evaluation processing unit 512. The tessellation evaluation processing unit 512 operates on parameterized coordinates of the subdivided patch to generate a surface representation and vertex attributes for each vertex associated with the geometric primitives.

[0166] A second instance of a primitive assembler 514 receives vertex attributes from the tessellation evaluation processing unit 512, reading stored vertex attributes as needed, and constructs graphics primitives for processing by the geometry processing unit 516. The geometry processing unit 516 is a programmable execution unit that executes geometry shader

programs to transform graphics primitives received from primitive assembler 514 as specified by the geometry shader programs. The geometry processing unit 516 may be programmed to subdivide the graphics primitives into one or more new graphics primitives and calculate parameters used to rasterize the new graphics primitives.

[0167] The geometry processing unit 516 may be able to add or delete elements in the geometry stream. The geometry processing unit 516 outputs the parameters and vertices specifying new graphics primitives to primitive assembler 518. The primitive assembler 518 receives the parameters and vertices from the geometry processing unit 516 and constructs graphics primitives for processing by a viewport scale, cull, and clip unit 520. The geometry processing unit 516 reads data that is stored in parallel processor memory or system memory for use in processing the geometry data. The viewport scale, cull, and clip unit 520 performs clipping, culling, and viewport scaling and outputs processed graphics primitives to a rasterizer 522.

[0168] The rasterizer 522 can perform depth culling and other depth-based optimizations. The rasterizer 522 also performs scan conversion on the new graphics primitives to generate fragments and output those fragments and associated coverage data to the fragment/pixel processing unit 524. The fragment/pixel processing unit 524 is a programmable execution unit that is configured to execute fragment shader programs or pixel shader programs. The fragment/pixel processing unit 524 transforming fragments or pixels received from rasterizer 522, as specified by the fragment or pixel shader programs. For example, the fragment/pixel processing unit 524 may be programmed to perform operations included but not limited to texture mapping, shading, blending, texture correction and perspective correction to produce shaded fragments or pixels that are output to a raster operations unit 526. The fragment/pixel processing unit 524 can read data that is stored in either the parallel processor memory or the system memory for use when processing the fragment data. Fragment or pixel shader programs may be configured to shade at sample, pixel, tile, or other granularities depending on the sampling rate configured for the processing units.

[0169] The raster operations unit 526 is a processing unit that performs raster operations including, but not limited to stencil, z-test, blending, and the like, and outputs pixel data as processed graphics data to be stored in graphics memory (e.g., parallel processor memory 222 as in FIG. 2A, and/or system memory 104 as in FIG 1), to be displayed on the one or more display device(s) 110 or for further processing by one of the one or more processor(s) 102 or parallel processor(s) 112. The raster operations unit 526 may be configured to compress z or color data that is written to memory and decompress z or color data that is read from memory.

Machine Learning Overview

[0170] The architecture described above can be applied to perform training and inference operations using machine learning models. Machine learning has been successful at solving many kinds of tasks. The computations that arise when training and using machine learning algorithms (e.g., neural networks) lend themselves naturally to efficient parallel implementations. Accordingly, parallel processors such as general-purpose graphic processing units (GPGPUs) have played a significant role in the practical implementation of deep neural networks. Parallel graphics processors with single instruction, multiple thread (SIMT) architectures are designed to maximize the amount of parallel processing in the graphics pipeline. In an SIMT architecture, groups of parallel threads attempt to execute program instructions synchronously together as often as possible to increase processing efficiency. The efficiency provided by parallel machine learning algorithm implementations allows the use of high capacity networks and enables those networks to be trained on larger datasets.

[0171] A machine learning algorithm is an algorithm that can learn based on a set of data. For example, machine learning algorithms can be designed to model high-level abstractions within a data set. For example, image recognition algorithms can be used to determine which of several categories to which a given input belong; regression algorithms can output a numerical value given an input; and pattern recognition algorithms can be used to generate translated text or perform text to speech and/or speech recognition.

[0172] An exemplary type of machine learning algorithm is a neural network. There are many types of neural networks; a simple type of neural network is a feedforward network. A feedforward network may be implemented as an acyclic graph in which the nodes are arranged in layers. Typically, a feedforward network topology includes an input layer and an output layer that are separated by at least one hidden layer. The hidden layer transforms input received by the input layer into a representation that is useful for generating output in the output layer. The network nodes are fully connected via edges to the nodes in adjacent layers, but there are no edges between nodes within each layer. Data received at the nodes of an input layer of a feedforward network are propagated (i.e., "fed forward") to the nodes of the output layer via an activation function that calculates the states of the nodes of each successive layer in the network based on coefficients ("weights") respectively associated with each of the edges connecting the layers. Depending on the specific model being represented by the algorithm being executed, the output from the neural network algorithm can take various forms.

[0173] Before a machine learning algorithm can be used to model a particular problem, the algorithm is trained using a training data set. Training a neural network involves selecting a network topology, using a set of training data representing a problem being modeled by the

network, and adjusting the weights until the network model performs with a minimal error for all instances of the training data set. For example, during a supervised learning training process for a neural network, the output produced by the network in response to the input representing an instance in a training data set is compared to the "correct" labeled output for that instance, an error signal representing the difference between the output and the labeled output is calculated, and the weights associated with the connections are adjusted to minimize that error as the error signal is backward propagated through the layers of the network. The network is considered "trained" when the errors for each of the outputs generated from the instances of the training data set are minimized.

[0174] The accuracy of a machine learning algorithm can be affected significantly by the quality of the data set used to train the algorithm. The training process can be computationally intensive and may require a significant amount of time on a conventional general-purpose processor. Accordingly, parallel processing hardware is used to train many types of machine learning algorithms. This is particularly useful for optimizing the training of neural networks, as the computations performed in adjusting the coefficients in neural networks lend themselves naturally to parallel implementations. Specifically, many machine learning algorithms and software applications have been adapted to make use of the parallel processing hardware within general-purpose graphics processing devices.

[0175] **FIG. 6** is a generalized diagram of a machine learning software stack 600. A machine learning application 602 can be configured to train a neural network using a training dataset or to use a trained deep neural network to implement machine intelligence. The machine learning application 602 can include training and inference functionality for a neural network and/or specialized software that can be used to train a neural network before deployment. The machine learning application 602 can implement any type of machine intelligence including but not limited to image recognition, mapping and localization, autonomous navigation, speech synthesis, medical imaging, or language translation.

[0176] Hardware acceleration for the machine learning application 602 can be enabled via a machine learning framework 604. The machine learning framework 604 can provide a library of machine learning primitives. Machine learning primitives are basic operations that are commonly performed by machine learning algorithms. Without the machine learning framework 604, developers of machine learning algorithms would be required to create and optimize the main computational logic associated with the machine learning algorithm, then re-optimize the computational logic as new parallel processors are developed. Instead, the machine learning application can be configured to perform the necessary computations using the primitives provided by the machine learning framework 604. Exemplary primitives include tensor

convolutions, activation functions, and pooling, which are computational operations that are performed while training a convolutional neural network (CNN). The machine learning framework 604 can also provide primitives to implement basic linear algebra subprograms performed by many machine-learning algorithms, such as matrix and vector operations.

[0177] The machine learning framework 604 can process input data received from the machine learning application 602 and generate the appropriate input to a compute framework 606. The compute framework 606 can abstract the underlying instructions provided to the GPGPU driver 608 to enable the machine learning framework 604 to take advantage of hardware acceleration via the GPGPU hardware 610 without requiring the machine learning framework 604 to have intimate knowledge of the architecture of the GPGPU hardware 610. Additionally, the compute framework 606 can enable hardware acceleration for the machine learning framework 604 across a variety of types and generations of the GPGPU hardware 610.

GPGPU Machine Learning Acceleration

[0178] **FIG. 7** illustrates a general-purpose graphics processing unit 700, , which may be the parallel processor 200 of FIG. 2A or the parallel processor(s) 112 of FIG. 1. The general-purpose processing unit (GPGPU) 700 may be configured to be particularly efficient in processing the type of computational workloads associated with training deep neural networks. Additionally, the GPGPU 700 can be linked directly to other instances of the GPGPU to create a multi-GPU cluster to improve training speed for particularly deep neural networks.

[0179] The GPGPU 700 includes a host interface 702 to enable a connection with a host processor. The host interface 702 may be a PCI Express interface. However, the host interface can also be a vendor specific communications interface or communications fabric. The GPGPU 700 receives commands from the host processor and uses a global scheduler 704 to distribute execution threads associated with those commands to a set of processing clusters 706A-706H. The processing clusters 706A-706H share a cache memory 708. The cache memory 708 can serve as a higher-level cache for cache memories within the processing clusters 706A-706H. The illustrated processing clusters 706A-706H may correspond with processing clusters 214A-214N as in FIG. 2A.

[0180] The GPGPU 700 includes memory 714A-714B coupled with the processing clusters 706A-H via a set of memory controllers 712A-712B. The memory 714A-714B can include various types of memory devices including dynamic random-access memory (DRAM) or graphics random access memory, such as synchronous graphics random access memory (SGRAM), including graphics double data rate (GDDR) memory. The memory 714A-714B may also include 3D stacked memory, including but not limited to high bandwidth memory (HBM).

[0181] Each of the processing clusters 706A-706H may include a set of graphics multiprocessors, such as the graphics multiprocessor 234 of FIG. 2D, graphics multiprocessor 325 of FIG. 3A, graphics multiprocessor 350 of FIG. 3B, or may include a multi-core group 365A-365N as in FIG. 3C. The graphics multiprocessors of the compute cluster include multiple types of integer and floating-point logic units that can perform computational operations at a range of precisions including suited for machine learning computations. For example, at least a subset of the floating-point units in each of the processing clusters 706A-706H can be configured to perform 16-bit or 32-bit floating point operations, while a different subset of the floating-point units can be configured to perform 64-bit floating point operations.

[0182] Multiple instances of the GPGPU 700 can be configured to operate as a compute cluster. The communication mechanism used by the compute cluster for synchronization and data exchange varies across embodiments. For example, the multiple instances of the GPGPU 700 communicate over the host interface 702. In one embodiment the GPGPU 700 includes an I/O hub 709 that couples the GPGPU 700 with a GPU link 710 that enables a direct connection to other instances of the GPGPU. The GPU link 710 may be coupled to a dedicated GPU-to-GPU bridge that enables communication and synchronization between multiple instances of the GPGPU 700. Optionally, the GPU link 710 couples with a high-speed interconnect to transmit and receive data to other GPGPUs or parallel processors. The multiple instances of the GPGPU 700 may be located in separate data processing systems and communicate via a network device that is accessible via the host interface 702. The GPU link 710 may be configured to enable a connection to a host processor in addition to or as an alternative to the host interface 702.

[0183] While the illustrated configuration of the GPGPU 700 can be configured to train neural networks, an alternate configuration of the GPGPU 700 can be configured for deployment within a high performance or low power inferencing platform. In an inferencing configuration, the GPGPU 700 includes fewer of the processing clusters 706A-706H relative to the training configuration. Additionally, memory technology associated with the memory 714A-714B may differ between inferencing and training configurations. In one embodiment, the inferencing configuration of the GPGPU 700 can support inferencing specific instructions. For example, an inferencing configuration can provide support for one or more 8-bit integer dot product instructions, which are commonly used during inferencing operations for deployed neural networks.

[0184] **FIG. 8** illustrates a multi-GPU computing system 800. The multi-GPU computing system 800 can include a processor 802 coupled to multiple GPGPUs 806A-806D via a host interface switch 804. The host interface switch 804 may be a PCI express switch device that couples the processor 802 to a PCI express bus over which the processor 802 can communicate

with the set of GPGPUs 806A-806D. Each of the multiple GPGPUs 806A-806D can be an instance of the GPGPU 700 of FIG. 7. The GPGPUs 806A-806D can interconnect via a set of high-speed point to point GPU to GPU links 816. The high-speed GPU to GPU links can connect to each of the GPGPUs 806A-806D via a dedicated GPU link, such as the GPU link 710 as in FIG. 7. The P2P GPU links 816 enable direct communication between each of the GPGPUs 806A-806D without requiring communication over the host interface bus to which the processor 802 is connected. With GPU-to-GPU traffic directed to the P2P GPU links, the host interface bus remains available for system memory access or to communicate with other instances of the multi-GPU computing system 800, for example, via one or more network devices. While in FIG. 8 the GPGPUs 806A-806D connect to the processor 802 via the host interface switch 804, the processor 802 may alternatively include direct support for the P2P GPU links 816 and connect directly to the GPGPUs 806A-806D.

Machine Learning Neural Network Implementations

[0185] The computing architecture described herein can be configured to perform the types of parallel processing that is particularly suited for training and deploying neural networks for machine learning. A neural network can be generalized as a network of functions having a graph relationship. As is well-known in the art, there are a variety of types of neural network implementations used in machine learning. One exemplary type of neural network is the feedforward network, as previously described.

[0186] A second exemplary type of neural network is the Convolutional Neural Network (CNN). A CNN is a specialized feedforward neural network for processing data having a known, grid-like topology, such as image data. Accordingly, CNNs are commonly used for compute vision and image recognition applications, but they also may be used for other types of pattern recognition such as speech and language processing. The nodes in the CNN input layer are organized into a set of "filters" (feature detectors inspired by the receptive fields found in the retina), and the output of each set of filters is propagated to nodes in successive layers of the network. The computations for a CNN include applying the convolution mathematical operation to each filter to produce the output of that filter. Convolution is a specialized kind of mathematical operation performed by two functions to produce a third function that is a modified version of one of the two original functions. In convolutional network terminology, the first function to the convolution can be referred to as the input, while the second function can be referred to as the convolution kernel. The output may be referred to as the feature map. For example, the input to a convolution layer can be a multidimensional array of data that defines the various color components of an input image. The convolution kernel can be a multidimensional

array of parameters, where the parameters are adapted by the training process for the neural network.

[0187] Recurrent neural networks (RNNs) are a family of feedforward neural networks that include feedback connections between layers. RNNs enable modeling of sequential data by sharing parameter data across different parts of the neural network. The architecture for an RNN includes cycles. The cycles represent the influence of a present value of a variable on its own value at a future time, as at least a portion of the output data from the RNN is used as feedback for processing subsequent input in a sequence. This feature makes RNNs particularly useful for language processing due to the variable nature in which language data can be composed.

[0188] The figures described below present exemplary feedforward, CNN, and RNN networks, as well as describe a general process for respectively training and deploying each of those types of networks. It will be understood that these descriptions are exemplary and non-limiting as to any specific embodiment described herein and the concepts illustrated can be applied generally to deep neural networks and machine learning techniques in general.

[0189] The exemplary neural networks described above can be used to perform deep learning. Deep learning is machine learning using deep neural networks. The deep neural networks used in deep learning are artificial neural networks composed of multiple hidden layers, as opposed to shallow neural networks that include only a single hidden layer. Deeper neural networks are generally more computationally intensive to train. However, the additional hidden layers of the network enable multistep pattern recognition that results in reduced output error relative to shallow machine learning techniques.

[0190] Deep neural networks used in deep learning typically include a front-end network to perform feature recognition coupled to a back-end network which represents a mathematical model that can perform operations (e.g., object classification, speech recognition, etc.) based on the feature representation provided to the model. Deep learning enables machine learning to be performed without requiring hand crafted feature engineering to be performed for the model. Instead, deep neural networks can learn features based on statistical structure or correlation within the input data. The learned features can be provided to a mathematical model that can map detected features to an output. The mathematical model used by the network is generally specialized for the specific task to be performed, and different models will be used to perform different task.

[0191] Once the neural network is structured, a learning model can be applied to the network to train the network to perform specific tasks. The learning model describes how to adjust the weights within the model to reduce the output error of the network. Backpropagation of errors is a common method used to train neural networks. An input vector is presented to the network for

processing. The output of the network is compared to the desired output using a loss function and an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards until each neuron has an associated error value which roughly represents its contribution to the original output. The network can then learn from those errors using an algorithm, such as the stochastic gradient descent algorithm, to update the weights of the of the neural network.

[0192] **FIG. 9A-9B** illustrate an exemplary convolutional neural network. FIG. 9A illustrates various layers within a CNN. As shown in FIG. 9A, an exemplary CNN used to model image processing can receive input 902 describing the red, green, and blue (RGB) components of an input image. The input 902 can be processed by multiple convolutional layers (e.g., convolutional layer 904, convolutional layer 906). The output from the multiple convolutional layers may optionally be processed by a set of fully connected layers 908. Neurons in a fully connected layer have full connections to all activations in the previous layer, as previously described for a feedforward network. The output from the fully connected layers 908 can be used to generate an output result from the network. The activations within the fully connected layers 908 can be computed using matrix multiplication instead of convolution. Not all CNN implementations make use of fully connected layers 908. For example, in some implementations the convolutional layer 906 can generate output for the CNN.

[0193] The convolutional layers are sparsely connected, which differs from traditional neural network configuration found in the fully connected layers 908. Traditional neural network layers are fully connected, such that every output unit interacts with every input unit. However, the convolutional layers are sparsely connected because the output of the convolution of a field is input (instead of the respective state value of each of the nodes in the field) to the nodes of the subsequent layer, as illustrated. The kernels associated with the convolutional layers perform convolution operations, the output of which is sent to the next layer. The dimensionality reduction performed within the convolutional layers is one aspect that enables the CNN to scale to process large images.

[0194] FIG. 9B illustrates exemplary computation stages within a convolutional layer of a CNN. Input to a convolutional layer 912 of a CNN can be processed in three stages of a convolutional layer 914. The three stages can include a convolution stage 916, a detector stage 918, and a pooling stage 920. The convolution layer 914 can then output data to a successive convolutional layer. The final convolutional layer of the network can generate output feature map data or provide input to a fully connected layer, for example, to generate a classification value for the input to the CNN.

[0195] In the convolution stage 916 performs several convolutions in parallel to produce a set of linear activations. The convolution stage 916 can include an affine transformation, which is any transformation that can be specified as a linear transformation plus a translation. Affine transformations include rotations, translations, scaling, and combinations of these transformations. The convolution stage computes the output of functions (e.g., neurons) that are connected to specific regions in the input, which can be determined as the local region associated with the neuron. The neurons compute a dot product between the weights of the neurons and the region in the local input to which the neurons are connected. The output from the convolution stage 916 defines a set of linear activations that are processed by successive stages of the convolutional layer 914.

[0196] The linear activations can be processed by a detector stage 918. In the detector stage 918, each linear activation is processed by a non-linear activation function. The non-linear activation function increases the nonlinear properties of the overall network without affecting the receptive fields of the convolution layer. Several types of non-linear activation functions may be used. One particular type is the rectified linear unit (ReLU), which uses an activation function defined as $f(x) = \max(0, x)$, such that the activation is thresholded at zero.

[0197] The pooling stage 920 uses a pooling function that replaces the output of the convolutional layer 906 with a summary statistic of the nearby outputs. The pooling function can be used to introduce translation invariance into the neural network, such that small translations to the input do not change the pooled outputs. Invariance to local translation can be useful in scenarios where the presence of a feature in the input data is more important than the precise location of the feature. Various types of pooling functions can be used during the pooling stage 920, including max pooling, average pooling, and l2-norm pooling. Additionally, some CNN implementations do not include a pooling stage. Instead, such implementations substitute and additional convolution stage having an increased stride relative to previous convolution stages.

[0198] The output from the convolutional layer 914 can then be processed by the next layer 922. The next layer 922 can be an additional convolutional layer or one of the fully connected layers 908. For example, the first convolutional layer 904 of FIG. 9A can output to the second convolutional layer 906, while the second convolutional layer can output to a first layer of the fully connected layers 908.

[0199] **FIG. 10** illustrates an exemplary recurrent neural network 1000. In a recurrent neural network (RNN), the previous state of the network influences the output of the current state of the network. RNNs can be built in a variety of ways using a variety of functions. The use of RNNs generally revolves around using mathematical models to predict the future based on a prior

sequence of inputs. For example, an RNN may be used to perform statistical language modeling to predict an upcoming word given a previous sequence of words. The illustrated RNN 1000 can be described has having an input layer 1002 that receives an input vector, hidden layers 1004 to implement a recurrent function, a feedback mechanism 1005 to enable a 'memory' of previous states, and an output layer 1006 to output a result. The RNN 1000 operates based on time-steps. The state of the RNN at a given time step is influenced based on the previous time step via the feedback mechanism 1005. For a given time step, the state of the hidden layers 1004 is defined by the previous state and the input at the current time step. An initial input (x_1) at a first time step can be processed by the hidden layer 1004. A second input (x_2) can be processed by the hidden layer 1004 using state information that is determined during the processing of the initial input (x_1). A given state can be computed as $s_t = f(Ux_t + Ws_{t-1})$, where U and W are parameter matrices. The function f is generally a nonlinearity, such as the hyperbolic tangent function (Tanh) or a variant of the rectifier function $f(x) = \max(0, x)$. However, the specific mathematical function used in the hidden layers 1004 can vary depending on the specific implementation details of the RNN 1000.

[0200] In addition to the basic CNN and RNN networks described, variations on those networks may be enabled. One example RNN variant is the long short term memory (LSTM) RNN. LSTM RNNs are capable of learning long-term dependencies that may be necessary for processing longer sequences of language. A variant on the CNN is a convolutional deep belief network, which has a structure similar to a CNN and is trained in a manner similar to a deep belief network. A deep belief network (DBN) is a generative neural network that is composed of multiple layers of stochastic (random) variables. DBNs can be trained layer-by-layer using greedy unsupervised learning. The learned weights of the DBN can then be used to provide pre-train neural networks by determining an optimal initial set of weights for the neural network.

[0201] **FIG. 11** illustrates training and deployment of a deep neural network. Once a given network has been structured for a task the neural network is trained using a training dataset 1102. Various training frameworks 1104 have been developed to enable hardware acceleration of the training process. For example, the machine learning framework 604 of FIG. 6 may be configured as a training framework 604. The training framework 604 can hook into an untrained neural network 1106 and enable the untrained neural net to be trained using the parallel processing resources described herein to generate a trained neural net 1108.

[0202] To start the training process the initial weights may be chosen randomly or by pre-training using a deep belief network. The training cycle then be performed in either a supervised or unsupervised manner.

[0203] Supervised learning is a learning method in which training is performed as a mediated operation, such as when the training dataset 1102 includes input paired with the desired output for the input, or where the training dataset includes input having known output and the output of the neural network is manually graded. The network processes the inputs and compares the resulting outputs against a set of expected or desired outputs. Errors are then propagated back through the system. The training framework 1104 can adjust to adjust the weights that control the untrained neural network 1106. The training framework 1104 can provide tools to monitor how well the untrained neural network 1106 is converging towards a model suitable to generating correct answers based on known input data. The training process occurs repeatedly as the weights of the network are adjusted to refine the output generated by the neural network. The training process can continue until the neural network reaches a statistically desired accuracy associated with a trained neural net 1108. The trained neural network 1108 can then be deployed to implement any number of machine learning operations to generate an inference result 1114 based on input of new data 1112.

[0204] Unsupervised learning is a learning method in which the network attempts to train itself using unlabeled data. Thus, for unsupervised learning the training dataset 1102 will include input data without any associated output data. The untrained neural network 1106 can learn groupings within the unlabeled input and can determine how individual inputs are related to the overall dataset. Unsupervised training can be used to generate a self-organizing map, which is a type of trained neural network 1108 capable of performing operations useful in reducing the dimensionality of data. Unsupervised training can also be used to perform anomaly detection, which allows the identification of data points in an input dataset that deviate from the normal patterns of the data.

[0205] Variations on supervised and unsupervised training may also be employed. Semi-supervised learning is a technique in which in the training dataset 1102 includes a mix of labeled and unlabeled data of the same distribution. Incremental learning is a variant of supervised learning in which input data is continuously used to further train the model. Incremental learning enables the trained neural network 1108 to adapt to the new data 1112 without forgetting the knowledge instilled within the network during initial training.

[0206] Whether supervised or unsupervised, the training process for particularly deep neural networks may be too computationally intensive for a single compute node. Instead of using a single compute node, a distributed network of computational nodes can be used to accelerate the training process.

[0207] **FIG. 12** is a block diagram illustrating distributed learning. Distributed learning is a training model that uses multiple distributed computing nodes to perform supervised or

unsupervised training of a neural network. The distributed computational nodes can each include one or more host processors and one or more of the general-purpose processing nodes, such as the highly parallel general-purpose graphics processing unit 700 as in FIG. 7. As illustrated, distributed learning can be performed model parallelism 1202, data parallelism 1204, or a combination of model and data parallelism 1204.

[0208] In model parallelism 1202, different computational nodes in a distributed system can perform training computations for different parts of a single network. For example, each layer of a neural network can be trained by a different processing node of the distributed system. The benefits of model parallelism include the ability to scale to particularly large models. Splitting the computations associated with different layers of the neural network enables the training of very large neural networks in which the weights of all layers would not fit into the memory of a single computational node. In some instances, model parallelism can be particularly useful in performing unsupervised training of large neural networks.

[0209] In data parallelism 1204, the different nodes of the distributed network have a complete instance of the model and each node receives a different portion of the data. The results from the different nodes are then combined. While different approaches to data parallelism are possible, data parallel training approaches all require a technique of combining results and synchronizing the model parameters between each node. Exemplary approaches to combining data include parameter averaging and update based data parallelism. Parameter averaging trains each node on a subset of the training data and sets the global parameters (e.g., weights, biases) to the average of the parameters from each node. Parameter averaging uses a central parameter server that maintains the parameter data. Update based data parallelism is similar to parameter averaging except
that instead of transferring parameters from the nodes to the parameter server, the updates to the model are transferred. Additionally, update based data parallelism can be performed in a decentralized manner, where the updates are compressed and transferred between nodes.

[0210] Combined model and data parallelism 1206 can be implemented, for example, in a distributed system in which each computational node includes multiple GPUs. Each node can have a complete instance of the model with separate GPUs within each node are used to train different portions of the model.

[0211] Distributed training has increased overhead relative to training on a single machine. However, the parallel processors and GPGPUs described herein can each implement various techniques to reduce the overhead of distributed training, including techniques to enable high bandwidth GPU-to-GPU data transfer and accelerated remote data synchronization.

Exemplary Machine Learning Applications

[0212] Machine learning can be applied to solve a variety of technological problems, including but not limited to computer vision, autonomous driving and navigation, speech recognition, and language processing. Computer vision has traditionally been one of the most active research areas for machine learning applications. Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating new categories of visual abilities. For example, computer vision applications can be configured to recognize sound waves from the vibrations induced in objects visible in a video. Parallel processor accelerated machine learning enables computer vision applications to be trained using significantly larger training dataset than previously feasible and enables inferencing systems to be deployed using low power parallel processors.

[0213] Parallel processor accelerated machine learning has autonomous driving applications including lane and road sign recognition, obstacle avoidance, navigation, and driving control. Accelerated machine learning techniques can be used to train driving models based on datasets that define the appropriate responses to specific training input. The parallel processors described herein can enable rapid training of the increasingly complex neural networks used for autonomous driving solutions and enables the deployment of low power inferencing processors in a mobile platform suitable for integration into autonomous vehicles.

[0214] Parallel processor accelerated deep neural networks have enabled machine learning approaches to automatic speech recognition (ASR). ASR includes the creation of a function that computes the most probable linguistic sequence given an input acoustic sequence. Accelerated machine learning using deep neural networks have enabled the replacement of the hidden Markov models (HMMs) and Gaussian mixture models (GMMs) previously used for ASR.

[0215] Parallel processor accelerated machine learning can also be used to accelerate natural language processing. Automatic learning procedures can make use of statistical inference algorithms to produce models that are robust to erroneous or unfamiliar input. Exemplary natural language processor applications include automatic machine translation between human languages.

[0216] The parallel processing platforms used for machine learning can be divided into training platforms and deployment platforms. Training platforms are generally highly parallel and include optimizations to accelerate multi-GPU single node training and multi-node, multi-GPU training. Exemplary parallel processors suited for training include the general-purpose graphics processing unit 700 of FIG. 7 and the multi-GPU computing system 800 of FIG. 8. On the contrary, deployed machine learning platforms generally include lower power parallel

processors suitable for use in products such as cameras, autonomous robots, and autonomous vehicles.

[0217] **FIG. 13** illustrates an exemplary inferencing system on a chip (SOC) 1300 suitable for performing inferencing using a trained model. The SOC 1300 can integrate processing components including a media processor 1302, a vision processor 1304, a GPGPU 1306 and a multi-core processor 1308. The GPGPU 1306 may be a GPGPU as described herein, such as the GPGPU 700, and the multi-core processor 1308 may be a multi-core processor described herein, such as the multi-core processors 405-406. The SOC 1300 can additionally include on-chip memory 1305 that can enable a shared on-chip data pool that is accessible by each of the processing components. The processing components can be optimized for low power operation to enable deployment to a variety of machine learning platforms, including autonomous vehicles and autonomous robots. For example, one implementation of the SOC 1300 can be used as a portion of the main control system for an autonomous vehicle. Where the SOC 1300 is configured for use in autonomous vehicles the SOC is designed and configured for compliance with the relevant functional safety standards of the deployment jurisdiction.

[0218] During operation, the media processor 1302 and vision processor 1304 can work in concert to accelerate computer vision operations. The media processor 1302 can enable low latency decode of multiple high-resolution (e.g., 4K, 8K) video streams. The decoded video streams can be written to a buffer in the on-chip-memory 1305. The vision processor 1304 can then parse the decoded video and perform preliminary processing operations on the frames of the decoded video in preparation of processing the frames using a trained image recognition model. For example, the vision processor 1304 can accelerate convolution operations for a CNN that is used to perform image recognition on the high-resolution video data, while back end model computations are performed by the GPGPU 1306.

[0219] The multi-core processor 1308 can include control logic to assist with sequencing and synchronization of data transfers and shared memory operations performed by the media processor 1302 and the vision processor 1304. The multi-core processor 1308 can also function as an application processor to execute software applications that can make use of the inferencing compute capability of the GPGPU 1306. For example, at least a portion of the navigation and driving logic can be implemented in software executing on the multi-core processor 1308. Such software can directly issue computational workloads to the GPGPU 1306 or the computational workloads can be issued to the multi-core processor 1308, which can offload at least a portion of those operations to the GPGPU 1306.

[0220] The GPGPU 1306 can include compute clusters such as a low power configuration of the processing clusters 706A-706H within general-purpose graphics processing unit 700. The

compute clusters within the GPGPU 1306 can support instruction that are specifically optimized to perform inferencing computations on a trained neural network. For example, the GPGPU 1306 can support instructions to perform low precision computations such as 8-bit and 4-bit integer vector operations.

Additional System Overview

[0221] **FIG. 14** is a block diagram of a processing system 1400. The elements of FIG. 14 having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. System 1400 may be used in a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 1402 or processor cores 1407. The system 1400 may be a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices such as within Internet-of-things (IoT) devices with wired or wireless connectivity to a local or wide area network.

[0222] The system 1400 may be a processing system having components that correspond with those of FIG. 1. For example, in different configurations, processor(s) 1402 or processor core(s) 1407 may correspond with processor(s) 102 of FIG. 1. Graphics processor(s) 1408 may correspond with parallel processor(s) 112 of FIG. 1. External graphics processor 1418 may be one of the add-in device(s) 120 of FIG. 1.

[0223] The system 1400 can include, couple with, or be integrated within: a server-based gaming platform; a game console, including a game and media console; a mobile gaming console, a handheld game console, or an online game console. The system 1400 may be part of a mobile phone, smart phone, tablet computing device or mobile Internet-connected device such as a laptop with low internal storage capacity. Processing system 1400 can also include, couple with, or be integrated within: a wearable device, such as a smart watch wearable device; smart eyewear or clothing enhanced with augmented reality (AR) or virtual reality (VR) features to provide visual, audio or tactile outputs to supplement real world visual, audio or tactile experiences or otherwise provide text, audio, graphics, video, holographic images or video, or tactile feedback; other augmented reality (AR) device; or other virtual reality (VR) device. The processing system 1400 may include or be part of a television or set top box device. The system 1400 can include, couple with, or be integrated within a self-driving vehicle such as a bus, tractor trailer, car, motor or electric power cycle, plane or glider (or any combination thereof). The self-driving vehicle may use system 1400 to process the environment sensed around the vehicle.

[0224] The one or more processors 1402 may include one or more processor cores 1407 to process instructions which, when executed, perform operations for system or user software. The least one of the one or more processor cores 1407 may be configured to process a specific instruction set 1409. The instruction set 1409 may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a Very Long Instruction Word (VLIW). One or more processor cores 1407 may process a different instruction set 1409, which may include instructions to facilitate the emulation of other instruction sets. Processor core 1407 may also include other processing devices, such as a Digital Signal Processor (DSP).

[0225] The processor 1402 may include cache memory 1404. Depending on the architecture, the processor 1402 can have a single internal cache or multiple levels of internal cache. In some embodiments, the cache memory is shared among various components of the processor 1402. In some embodiments, the processor 1402 also uses an external cache (e.g., a Level-3 (L3) cache or Last Level Cache (LLC)) (not shown), which may be shared among processor cores 1407 using known cache coherency techniques. A register file 1406 can be additionally included in processor 1402 and may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor 1402.

[0226] The one or more processor(s) 1402 may be coupled with one or more interface bus(es) 1410 to transmit communication signals such as address, data, or control signals between processor 1402 and other components in the system 1400. The interface bus 1410, in one of these embodiments, can be a processor bus, such as a version of the Direct Media Interface (DMI) bus. However, processor busses are not limited to the DMI bus, and may include one or more Peripheral Component Interconnect buses (e.g., PCI, PCI express), memory busses, or other types of interface busses. For example, the processor(s) 1402 may include an integrated memory controller 1416 and a platform controller hub 1430. The memory controller 1416 facilitates communication between a memory device and other components of the system 1400, while the platform controller hub (PCH) 1430 provides connections to I/O devices via a local I/O bus.

[0227] The memory device 1420 can be a dynamic random-access memory (DRAM) device, a static random-access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. The memory device 1420 can, for example, operate as system memory for the system 1400, to store data 1422 and instructions 1421 for use when the one or more processors 1402 executes an application or process. Memory controller 1416 also couples with an optional external graphics

processor 1418, which may communicate with the one or more graphics processors 1408 in processors 1402 to perform graphics and media operations. In some embodiments, graphics, media, and or compute operations may be assisted by an accelerator 1412 which is a coprocessor that can be configured to perform a specialized set of graphics, media, or compute operations. For example, the accelerator 1412 may be a matrix multiplication accelerator used to optimize machine learning or compute operations. The accelerator 1412 can be a ray-tracing accelerator that can be used to perform ray-tracing operations in concert with the graphics processor 1408. In one embodiment, an external accelerator 1419 may be used in place of or in concert with the accelerator 1412.

[0228] A display device 1411 may be provided that can connect to the processor(s) 1402. The display device 1411 can be one or more of an internal display device, as in a mobile electronic device or a laptop device or an external display device attached via a display interface (e.g., DisplayPort, etc.). The display device 1411 can be a head mounted display (HMD) such as a stereoscopic display device for use in virtual reality (VR) applications or augmented reality (AR) applications.

[0229] The platform controller hub 1430 may enable peripherals to connect to memory device 1420 and processor 1402 via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller 1446, a network controller 1434, a firmware interface 1428, a wireless transceiver 1426, touch sensors 1425, a data storage device 1424 (e.g., non-volatile memory, volatile memory, hard disk drive, flash memory, NAND, 3D NAND, 3D XPoint/Optane, etc.). The data storage device 1424 can connect via a storage interface (e.g., SATA) or via a peripheral bus, such as a Peripheral Component Interconnect bus (e.g., PCI, PCI express). The touch sensors 1425 can include touch screen sensors, pressure sensors, or fingerprint sensors. The wireless transceiver 1426 can be a Wi-Fi transceiver, a Bluetooth transceiver, or a mobile network transceiver such as a 3G, 4G, 5G, or Long-Term Evolution (LTE) transceiver. The firmware interface 1428 enables communication with system firmware, and can be, for example, a unified extensible firmware interface (UEFI). The network controller 1434 can enable a network connection to a wired network. In some embodiments, a high-performance network controller (not shown) couples with the interface bus 1410. The audio controller 1446 may be a multi-channel high definition audio controller. In some of these embodiments the system 1400 includes an optional legacy I/O controller 1440 for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. The platform controller hub 1430 can also connect to one or more Universal Serial Bus (USB) controllers 1442 connect input devices, such as keyboard and mouse 1443 combinations, a camera 1444, or other USB input devices.

[0230] It will be appreciated that the system 1400 shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, an instance of the memory controller 1416 and platform controller hub 1430 may be integrated into a discreet external graphics processor, such as the external graphics processor 1418. The platform controller hub 1430 and/or memory controller 1416 may be external to the one or more processor(s) 1402. For example, the system 1400 can include an external memory controller 1416 and platform controller hub 1430, which may be configured as a memory controller hub and peripheral controller hub within a system chipset that is in communication with the processor(s) 1402.

[0231] For example, circuit boards ("sleds") can be used on which components such as CPUs, memory, and other components are placed are designed for increased thermal performance. Processing components such as the processors may be located on a top side of a sled while near memory, such as DIMMs, are located on a bottom side of the sled. As a result of the enhanced airflow provided by this design, the components may operate at higher frequencies and power levels than in typical systems, thereby increasing performance. Furthermore, the sleds are configured to blindly mate with power and data communication cables in a rack, thereby enhancing their ability to be quickly removed, upgraded, reinstalled, and/or replaced. Similarly, individual components located on the sleds, such as processors, accelerators, memory, and data storage drives, are configured to be easily upgraded due to their increased spacing from each other. In the illustrative embodiment, the components additionally include hardware attestation features to prove their authenticity.

[0232] A data center can utilize a single network architecture ("fabric") that supports multiple other network architectures including Ethernet and Omni-Path. The sleds can be coupled to switches via optical fibers, which provide higher bandwidth and lower latency than typical twisted pair cabling (e.g., Category 5, Category 5e, Category 6, etc.). Due to the high bandwidth, low latency interconnections and network architecture, the data center may, in use, pool resources, such as memory, accelerators (e.g., GPUs, graphics accelerators, FPGAs, ASICs, neural network and/or artificial intelligence accelerators, etc.), and data storage drives that are physically disaggregated, and provide them to compute resources (e.g., processors) on an as needed basis, enabling the compute resources to access the pooled resources as if they were local.

[0233] A power supply or source can provide voltage and/or current to system 1400 or any component or system described herein. In one example, the power supply includes an AC to DC (alternating current to direct current) adapter to plug into a wall outlet. Such AC power can be renewable energy (e.g., solar power) power source. In one example, the power source includes a

DC power source, such as an external AC to DC converter. A power source or power supply may also include wireless charging hardware to charge via proximity to a charging field. The power source can include an internal battery, alternating current supply, motion-based power supply, solar power supply, or fuel cell source.

[0234] **FIG. 15A-15C** illustrate computing systems and graphics processors. The elements of FIG. 15A-15C having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such.

[0235] **FIG. 15A** is a block diagram of a processor 1500, which may be a variant of one of the processors 1402 and may be used in place of one of those. Therefore, the disclosure of any features in combination with the processor 1500 herein also discloses a corresponding combination with the processor(s) 1402, but is not limited to such. The processor 1500 may have one or more processor cores 1502A-1502N, an integrated memory controller 1514, and an integrated graphics processor 1508. Where an integrated graphics processor 1508 is excluded, the system that includes the processor will include a graphics processor device within a system chipset or coupled via a system bus. Processor 1500 can include additional cores up to and including additional core 1502N represented by the dashed lined boxes. Each of processor cores 1502A-1502N includes one or more internal cache units 1504A-1504N. In some embodiments each processor core 1502A-1502N also has access to one or more shared cache units 1506. The internal cache units 1504A-1504N and shared cache units 1506 represent a cache memory hierarchy within the processor 1500. The cache memory hierarchy may include at least one level of instruction and data cache within each processor core and one or more levels of shared mid-level cache, such as a Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the LLC. In some embodiments, cache coherency logic maintains coherency between the various cache units 1506 and 1504A-1504N.

[0236] The processor 1500 may also include a set of one or more bus controller units 1516 and a system agent core 1510. The one or more bus controller units 1516 manage a set of peripheral buses, such as one or more PCI or PCI express busses. System agent core 1510 provides management functionality for the various processor components. The system agent core 1510 may include one or more integrated memory controllers 1514 to manage access to various external memory devices (not shown).

[0237] For example, one or more of the processor cores 1502A-1502N may include support for simultaneous multi-threading. The system agent core 1510 includes components for

coordinating and operating cores 1502A-1502N during multi-threaded processing. System agent core 1510 may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of processor cores 1502A-1502N and graphics processor 1508.

[0238] The processor 1500 may additionally include graphics processor 1508 to execute graphics processing operations. In some of these embodiments, the graphics processor 1508 couples with the set of shared cache units 1506, and the system agent core 1510, including the one or more integrated memory controllers 1514. The system agent core 1510 may also include a display controller 1511 to drive graphics processor output to one or more coupled displays. The display controller 1511 may also be a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor 1508.

[0239] A ring-based interconnect unit 1512 may be used to couple the internal components of the processor 1500. However, an alternative interconnect unit may be used, such as a point-to-point interconnect, a switched interconnect, or other techniques, including techniques well known in the art. In some of these embodiments with a ring-based interconnect 1512, the graphics processor 1508 couples with the ring-based interconnect 1512 via an I/O link 1513.

[0240] The exemplary I/O link 1513 represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module 1518, such as an eDRAM module. Optionally, each of the processor cores 1502A-1502N and graphics processor 1508 can use embedded memory modules 1518 as a shared Last Level Cache.

[0241] The processor cores 1502A-1502N may, for example, be homogenous cores executing the same instruction set architecture. Alternatively, the processor cores 1502A-1502N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores 1502A-1502N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set. The processor cores 1502A-1502N may be heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. As another example, the processor cores 1502A-1502N are heterogeneous in terms of computational capability. Additionally, processor 1500 can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.

[0242] **FIG. 15B** is a block diagram of hardware logic of a graphics processor core 1519, according to some embodiments described herein. The graphics processor core 1519, sometimes referred to as a core slice, can be one or multiple graphics cores within a modular graphics

processor. The graphics processor core 1519 is exemplary of one graphics core slice, and a graphics processor as described herein may include multiple graphics core slices based on target power and performance envelopes. Each graphics processor core 1519 can include a fixed function block 1530 coupled with multiple sub-cores 1521A-1521F, also referred to as sub-slices, that include modular blocks of general-purpose and fixed function logic.

[0243] The fixed function block 1530 may include a geometry/fixed function pipeline 1531 that can be shared by all sub-cores in the graphics processor core 1519, for example, in lower performance and/or lower power graphics processor implementations. The geometry/fixed function pipeline 1531 may include a 3D fixed function pipeline (e.g., 3D pipeline 1612 as in FIG. 16A described below) a video front-end unit, a thread spawner and thread dispatcher, and a unified return buffer manager, which manages unified return buffers (e.g., unified return buffer 1718 in FIG. 17, as described below).

[0244] The fixed function block 1530 may also include a graphics SoC interface 1532, a graphics microcontroller 1533, and a media pipeline 1534. The graphics SoC interface 1532 provides an interface between the graphics processor core 1519 and other processor cores within a system on a chip integrated circuit. The graphics microcontroller 1533 is a programmable sub-processor that is configurable to manage various functions of the graphics processor core 1519, including thread dispatch, scheduling, and pre-emption. The media pipeline 1534 (e.g., media pipeline 1616 of FIG. 16A and FIG. 17) includes logic to facilitate the decoding, encoding, pre-processing, and/or post-processing of multimedia data, including image and video data. The media pipeline 1534 implement media operations via requests to compute or sampling logic within the sub-cores 1521-1521F.

[0245] The SoC interface 1532 may enable the graphics processor core 1519 to communicate with general-purpose application processor cores (e.g., CPUs) and/or other components within an SoC, including memory hierarchy elements such as a shared last level cache memory, the system RAM, and/or embedded on-chip or on-package DRAM. The SoC interface 1532 can also enable communication with fixed function devices within the SoC, such as camera imaging pipelines, and enables the use of and/or implements global memory atomics that may be shared between the graphics processor core 1519 and CPUs within the SoC. The SoC interface 1532 can also implement power management controls for the graphics processor core 1519 and enable an interface between a clock domain of the graphic core 1519 and other clock domains within the SoC. Optionally, the SoC interface 1532 enables receipt of command buffers from a command streamer and global thread dispatcher that are configured to provide commands and instructions to each of one or more graphics cores within a graphics processor. The commands and instructions can be dispatched to the media pipeline 1534, when media operations are to be

performed, or a geometry and fixed function pipeline (e.g., geometry and fixed function pipeline 1531, geometry and fixed function pipeline 1537) when graphics processing operations are to be performed.

[0246] The graphics microcontroller 1533 can be configured to perform various scheduling and management tasks for the graphics processor core 1519. In one configuration the graphics microcontroller 1533 can, for example, perform graphics and/or compute workload scheduling on the various graphics parallel engines within execution unit (EU) arrays 1522A-1522F, 1524A-1524F within the sub-cores 1521A-1521F. In this workload scheduling, host software executing on a CPU core of an SoC including the graphics processor core 1519 can submit workloads to one of multiple graphic processor doorbells, which invokes a scheduling operation on the appropriate graphics engine. Scheduling operations include determining which workload to run next, submitting a workload to a command streamer, pre-empting existing workloads running on an engine, monitoring progress of a workload, and notifying host software when a workload is complete. Optionally, the graphics microcontroller 1533 can also facilitate low-power or idle states for the graphics processor core 1519, providing the graphics processor core 1519 with the ability to save and restore registers within the graphics processor core 1519 across low-power state transitions independently from the operating system and/or graphics driver software on the system.

[0247] The graphics processor core 1519 may have more than or fewer than the illustrated sub-cores 1521A-1521F, up to *N* modular sub-cores. For each set of *N* sub-cores, the graphics processor core 1519 can also include shared function logic 1535, shared and/or cache memory 1536, a geometry/fixed function pipeline 1537, as well as additional fixed function logic 1538 to accelerate various graphics and compute processing operations. The shared function logic 1535 can include logic units associated with the shared function logic 1720 of FIG. 17 (e.g., sampler, math, and/or inter-thread communication logic) that can be shared by each *N* sub-cores within the graphics processor core 1519. The shared and/or cache memory 1536 can be a last-level cache for the set of *N* sub-cores 1521A-1521F within the graphics processor core 1519, and can also serve as shared memory that is accessible by multiple sub-cores. The geometry/fixed function pipeline 1537 can be included instead of the geometry/fixed function pipeline 1531 within the fixed function block 1530 and can include the same or similar logic units.

[0248] The graphics processor core 1519 may include additional fixed function logic 1538 that can include various fixed function acceleration logic for use by the graphics processor core 1519. Optionally, the additional fixed function logic 1538 includes an additional geometry pipeline for use in position only shading. In position-only shading, two geometry pipelines exist, the full geometry pipeline within the geometry/fixed function pipeline 1538, 1531, and a cull

pipeline, which is an additional geometry pipeline which may be included within the additional fixed function logic 1538. For example, the cull pipeline may be a trimmed down version of the full geometry pipeline. The full pipeline and the cull pipeline can execute different instances of the same application, each instance having a separate context. Position only shading can hide long cull runs of discarded triangles, enabling shading to be completed earlier in some instances. For example, the cull pipeline logic within the additional fixed function logic 1538 can execute position shaders in parallel with the main application and generally generates critical results faster than the full pipeline, as the cull pipeline fetches and shades only the position attribute of the vertices, without performing rasterization and rendering of the pixels to the frame buffer. The cull pipeline can use the generated critical results to compute visibility information for all the triangles without regard to whether those triangles are culled. The full pipeline (which in this instance may be referred to as a replay pipeline) can consume the visibility information to skip the culled triangles to shade only the visible triangles that are finally passed to the rasterization phase.

[0249] Optionally, the additional fixed function logic 1538 can also include machine-learning acceleration logic, such as fixed function matrix multiplication logic, for implementations including optimizations for machine learning training or inferencing.

[0250] Within each graphics sub-core 1521A-1521F a set of execution resources is included that may be used to perform graphics, media, and compute operations in response to requests by graphics pipeline, media pipeline, or shader programs. The graphics sub-cores 1521A-1521F include multiple EU arrays 1522A-1522F, 1524A-1524F, thread dispatch and inter-thread communication (TD/IC) logic 1523A-1523F, a 3D (e.g., texture) sampler 1525A-1525F, a media sampler 1506A-1506F, a shader processor 1527A-1527F, and shared local memory (SLM) 1528A-1528F. The EU arrays 1522A-1522F, 1524A-1524F each include multiple execution units, which are general-purpose graphics processing units capable of performing floating-point and integer/fixed-point logic operations in service of a graphics, media, or compute operation, including graphics, media, or compute shader programs. The TD/IC logic 1523A-1523F performs local thread dispatch and thread control operations for the execution units within a sub-core and facilitate communication between threads executing on the execution units of the sub-core. The 3D sampler 1525A-1525F can read texture or other 3D graphics related data into memory. The 3D sampler can read texture data differently based on a configured sample state and the texture format associated with a given texture. The media sampler 1506A-1506F can perform similar read operations based on the type and format associated with media data. For example, each graphics sub-core 1521A-1521F can alternately include a unified 3D and media sampler. Threads executing on the execution units within each of the sub-cores 1521A-1521F

can make use of shared local memory 1528A-1528F within each sub-core, to enable threads executing within a thread group to execute using a common pool of on-chip memory.

[0251] **FIG. 15C** is a block diagram of general-purpose graphics processing unit (GPGPU) 1570 that can be configured as a graphics processor, e.g. the graphics processor 1508, and/or compute accelerator, according to embodiments described herein. The GPGPU 1570 can interconnect with host processors (e.g., one or more CPU(s) 1546) and memory 1571, 1572 via one or more system and/or memory busses. Memory 1571 may be system memory that can be shared with the one or more CPU(s) 1546, while memory 1572 is device memory that is dedicated to the GPGPU 1570. For example, components within the GPGPU 1570 and device memory 1572 may be mapped into memory addresses that are accessible to the one or more CPU(s) 1546. Access to memory 1571 and 1572 may be facilitated via a memory controller 1568. The memory controller 1568 may include an internal direct memory access (DMA) controller 1569 or can include logic to perform operations that would otherwise be performed by a DMA controller.

[0252] The GPGPU 1570 includes multiple cache memories, including an L2 cache 1553, L1 cache 1554, an instruction cache 1555, and shared memory 1556, at least a portion of which may also be partitioned as a cache memory. The GPGPU 1570 also includes multiple compute units 1560A-1560N. Each compute unit 1560A-1560N includes a set of vector registers 1561, scalar registers 1562, vector logic units 1563, and scalar logic units 1564. The compute units 1560A-1560N can also include local shared memory 1565 and a program counter 1566. The compute units 1560A-1560N can couple with a constant cache 1567, which can be used to store constant data, which is data that will not change during the run of kernel or shader program that executes on the GPGPU 1570. The constant cache 1567 may be a scalar data cache and cached data can be fetched directly into the scalar registers 1562.

[0253] During operation, the one or more CPU(s) 1546 can write commands into registers or memory in the GPGPU 1570 that has been mapped into an accessible address space. The command processors 1557 can read the commands from registers or memory and determine how those commands will be processed within the GPGPU 1570. A thread dispatcher 1558 can then be used to dispatch threads to the compute units 1560A-1560N to perform those commands. Each compute unit 1560A-1560N can execute threads independently of the other compute units. Additionally, each compute unit 1560A-1560N can be independently configured for conditional computation and can conditionally output the results of computation to memory. The command processors 1557 can interrupt the one or more CPU(s) 1546 when the submitted commands are complete.

[0254] **FIG. 16A-16C** illustrate block diagrams of additional graphics processor and compute accelerator architectures provided by embodiments described herein, e.g. in accordance with FIG. 15A-15C. The elements of FIG. 16A-16C having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such.

[0255] **FIG. 16A** is a block diagram of a graphics processor 1600, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores, or other semiconductor devices such as, but not limited to, memory devices or network interfaces. The graphics processor 1600 may be a variant of the graphics processor 1508 and may be used in place of the graphics processor 1508. Therefore, the disclosure of any features in combination with the graphics processor 1508 herein also discloses a corresponding combination with the graphics processor 1600, but is not limited to such. The graphics processor may communicate via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor memory. Graphics processor 1600 may include a memory interface 1614 to access memory. Memory interface 1614 can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

[0256] Optionally, graphics processor 1600 also includes a display controller 1602 to drive display output data to a display device 1618. Display controller 1602 includes hardware for one or more overlay planes for the display and composition of multiple layers of video or user interface elements. The display device 1618 can be an internal or external display device. In one embodiment the display device 1618 is a head mounted display device, such as a virtual reality (VR) display device or an augmented reality (AR) display device. Graphics processor 1600 may include a video codec engine 1606 to encode, decode, or transcode media to, from, or between one or more media encoding formats, including, but not limited to Moving Picture Experts Group (MPEG) formats such as MPEG-2, Advanced Video Coding (AVC) formats such as H.264/MPEG-4 AVC, H.265/HEVC, Alliance for Open Media (AOMedia) VP8, VP9, as well as the Society of Motion Picture & Television Engineers (SMPTE) 421M/VC-1, and Joint Photographic Experts Group (JPEG) formats such as JPEG, and Motion JPEG (MJPEG) formats.

[0257] Graphics processor 1600 may include a block image transfer (BLIT) engine 1604 to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, alternatively, 2D graphics operations may be performed using one or more components of graphics processing engine (GPE) 1610. In some embodiments, GPE 1610 is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

[0258] GPE 1610 may include a 3D pipeline 1612 for performing 3D operations, such as rendering three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline 1612 includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media subsystem 1615. While 3D pipeline 1612 can be used to perform media operations, an embodiment of GPE 1610 also includes a media pipeline 1616 that is specifically used to perform media operations, such as video post-processing and image enhancement.

[0259] Media pipeline 1616 may include fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of video codec engine 1606. Media pipeline 1616 may additionally include a thread spawning unit to spawn threads for execution on 3D/Media subsystem 1615. The spawned threads perform computations for the media operations on one or more graphics execution units included in 3D/Media subsystem 1615.

[0260] The 3D/Media subsystem 1615 may include logic for executing threads spawned by 3D pipeline 1612 and media pipeline 1616. The pipelines may send thread execution requests to 3D/Media subsystem 1615, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. The 3D/Media subsystem 1615 may include one or more internal caches for thread instructions and data. Additionally, the 3D/Media subsystem 1615 may also include shared memory, including registers and addressable memory, to share data between threads and to store output data.

[0261] **FIG. 16B** illustrates a graphics processor 1620, being a variant of the graphics processor 1600 and may be used in place of the graphics processor 1600 and vice versa. Therefore, the disclosure of any features in combination with the graphics processor 1600 herein also discloses a corresponding combination with the graphics processor 1620, but is not limited to such. The graphics processor 1620 has a tiled architecture, according to embodiments described herein. The graphics processor 1620 may include a graphics processing engine cluster 1622 having multiple instances of the graphics processing engine 1610 of FIG. 16A within a graphics engine tile 1610A-1610D. Each graphics engine tile 1610A-1610D can be interconnected via a set of tile interconnects 1623A-1623F. Each graphics engine tile 1610A-1610D can also be connected to a memory module or memory device 1626A-1626D via memory interconnects 1625A-1625D. The memory devices 1626A-1626D can use any graphics memory technology. For example, the memory devices 1626A-1626D may be graphics double data rate

(GDDR) memory. The memory devices 1626A-1626D may be high-bandwidth memory (HBM) modules that can be on-die with their respective graphics engine tile 1610A-1610D. The memory devices 1626A-1626D may be stacked memory devices that can be stacked on top of their respective graphics engine tile 1610A-1610D. Each graphics engine tile 1610A-1610D and associated memory devices 1626A-1626D may reside on separate chiplets, which are bonded to a base die or base substrate, as described in further detail in FIG. 24B-24D.

[0262] The graphics processor 1620 may be configured with a non-uniform memory access (NUMA) systemin which memory devices 1626A-1626D are coupled with associated graphics engine tiles 1610A-1610D. A given memory device may be accessed by graphics engine tiles other than the tile to which it is directly connected. However, access latency to the memory devices 1626A-1626D may be lowest when accessing a local tile. In one embodiment, a cache coherent NUMA (ccNUMA) system is enabled that uses the tile interconnects 1623A-1623F to enable communication between cache controllers within the graphics engine tiles 1610A-1610D to keep a consistent memory image when more than one cache stores the same memory location.

[0263] The graphics processing engine cluster 1622 can connect with an on-chip or on-package fabric interconnect 1624. The fabric interconnect 1624 can enable communication between graphics engine tiles 1610A-1610D and components such as the video codec 1606 and one or more copy engines 1604. The copy engines 1604 can be used to move data out of, into, and between the memory devices 1626A-1626D and memory that is external to the graphics processor 1620 (e.g., system memory). The fabric interconnect 1624 can also be used to interconnect the graphics engine tiles 1610A-1610D. The graphics processor 1620 may optionally include a display controller 1602 to enable a connection with an external display device 1618. The graphics processor may also be configured as a graphics or compute accelerator. In the accelerator configuration, the display controller 1602 and display device 1618 may be omitted.

[0264] The graphics processor 1620 can connect to a host system via a host interface 1628. The host interface 1628 can enable communication between the graphics processor 1620, system memory, and/or other system components. The host interface 1628 can be, for example, a PCI express bus or another type of host system interface.

[0265] **FIG. 16C** illustrates a compute accelerator 1630, according to embodiments described herein. The compute accelerator 1630 can include architectural similarities with the graphics processor 1620 of FIG. 16B and is optimized for compute acceleration. A compute engine cluster 1632 can include a set of compute engine tiles 1640A-1640D that include execution logic that is optimized for parallel or vector-based general-purpose compute operations. The compute engine tiles 1640A-1640D may not include fixed function graphics

processing logic, although in some embodiments one or more of the compute engine tiles 1640A-1640D can include logic to perform media acceleration. The compute engine tiles 1640A-1640D can connect to memory devices 1626A-1626D via memory interconnects 1625A-1625D. The memory devices 1626A-1626D and memory interconnects 1625A-1625D may be similar technology as in graphics processor 1620, or can be different. The graphics compute engine tiles 1640A-1640D can also be interconnected via a set of tile interconnects 1623A-1623F and may be connected with and/or interconnected by a fabric interconnect 1624. In one embodiment the compute accelerator 1630 includes a large L3 cache 1636 that can be configured as a device-wide cache. The compute accelerator 1630 can also connect to a host processor and memory via a host interface 1628 in a similar manner as the graphics processor 1620 of FIG. 16B.

Graphics Processing Engine

[0266] **FIG. 17** is a block diagram of a graphics processing engine 1710 of a graphics processor in accordance with some embodiments. The graphics processing engine (GPE) 1710 may be a version of the GPE 1610 shown in **FIG. 16A**, and may also represent a graphics engine tile 1610A-1610D of **FIG. 16B**. The elements of FIG. 17 having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. For example, the 3D pipeline 1612 and media pipeline 1616 of **FIG. 16A** are also illustrated in **FIG. 17**. The media pipeline 1616 is optional in some embodiments of the GPE 1710 and may not be explicitly included within the GPE 1710. For example and in at least one embodiment, a separate media and/or image processor is coupled to the GPE 1710.

[0267] GPE 1710 may couple with or include a command streamer 1703, which provides a command stream to the 3D pipeline 1612 and/or media pipelines 1616. Alternatively or additionally, the command streamer 1703 may be directly coupled to a unified return buffer 1718. The unified return buffer 1718 may be communicatively coupled to a graphics core array 1714. Optionally, the command streamer 1703 is coupled with memory, which can be system memory, or one or more of internal cache memory and shared cache memory. The command streamer 1703 may receive commands from the memory and sends the commands to 3D pipeline 1612 and/or media pipeline 1616. The commands are directives fetched from a ring buffer, which stores commands for the 3D pipeline 1612 and media pipeline 1616. The ring buffer can additionally include batch command buffers storing batches of multiple commands. The commands for the 3D pipeline 1612 can also include references to data stored in memory, such as but not limited to vertex and geometry data for the 3D pipeline 1612 and/or image data and

memory objects for the media pipeline 316. The 3D pipeline 1612 and media pipeline 1616 process the commands and data by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to the graphics core array 1714. The graphics core array 1714 may include one or more blocks of graphics cores (e.g., graphics core(s) 1715A, graphics core(s) 1715B), each block including one or more graphics cores. Each graphics core includes a set of graphics execution resources that includes general-purpose and graphics specific execution logic to perform graphics and compute operations, as well as fixed function texture processing and/or machine learning and artificial intelligence acceleration logic.

[0268] In various embodiments the 3D pipeline 1612 can include fixed function and programmable logic to process one or more shader programs, such as vertex shaders, geometry shaders, pixel shaders, fragment shaders, compute shaders, or other shader programs, by processing the instructions and dispatching execution threads to the graphics core array 1714. The graphics core array 1714 provides a unified block of execution resources for use in processing these shader programs. Multi-purpose execution logic (e.g., execution units) within the graphics core(s) 1715A-1714B of the graphic core array 1714 includes support for various 3D API shader languages and can execute multiple simultaneous execution threads associated with multiple shaders.

[0269] The graphics core array 1714 may include execution logic to perform media functions, such as video and/or image processing. The execution units may include general-purpose logic that is programmable to perform parallel general-purpose computational operations, in addition to graphics processing operations. The general-purpose logic can perform processing operations in parallel or in conjunction with general-purpose logic within the processor core(s) 1407 of FIG. 14 or core 1502A-1502N as in FIG. 15A.

[0270] Output data generated by threads executing on the graphics core array 1714 can output data to memory in a unified return buffer (URB) 1718. The URB 1718 can store data for multiple threads. The URB 1718 may be used to send data between different threads executing on the graphics core array 1714. The URB 1718 may additionally be used for synchronization between threads on the graphics core array 1714 and fixed function logic within the shared function logic 1720.

[0271] Optionally, the graphics core array 1714 may be scalable, such that the array includes a variable number of graphics cores, each having a variable number of execution units based on the target power and performance level of GPE 1710. The execution resources may be dynamically scalable, such that execution resources may be enabled or disabled as needed.

[0272] The graphics core array 1714 couples with shared function logic 1720 that includes multiple resources that are shared between the graphics cores in the graphics core array. The

shared functions within the shared function logic 1720 are hardware logic units that provide specialized supplemental functionality to the graphics core array 1714. In various embodiments, shared function logic 1720 includes but is not limited to sampler 1721, math 1722, and inter-thread communication (ITC) 1723 logic. Additionally, one or more cache(s) 1725 within the shared function logic 1720 may be implemented.

[0273] A shared function is implemented at least in a case where the demand for a given specialized function is insufficient for inclusion within the graphics core array 1714. Instead a single instantiation of that specialized function is implemented as a stand-alone entity in the shared function logic 1720 and shared among the execution resources within the graphics core array 1714. The precise set of functions that are shared between the graphics core array 1714 and included within the graphics core array 1714 varies across embodiments. Specific shared functions within the shared function logic 1720 that are used extensively by the graphics core array 1714 may be included within shared function logic 1716 within the graphics core array 1714. Optionally, the shared function logic 1716 within the graphics core array 1714 can include some or all logic within the shared function logic 1720. All logic elements within the shared function logic 1720 may be duplicated within the shared function logic 1716 of the graphics core array 1714. Alternatively, the shared function logic 1720 is excluded in favor of the shared function logic 1716 within the graphics core array 1714.

Execution Units

[0274] **FIG. 18A-18B** illustrate thread execution logic 1800 including an array of processing elements employed in a graphics processor core according to embodiments described herein. The elements of FIG. 18A-18B having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. FIG. 18A-18B illustrates an overview of thread execution logic 1800, which may be representative of hardware logic illustrated with each sub-core 1521A-1521F of FIG. 15B. FIG. 18A is representative of an execution unit within a general-purpose graphics processor, while FIG. 18B is representative of an execution unit that may be used within a compute accelerator.

[0275] As illustrated in FIG. 18A, thread execution logic 1800 may include a shader processor 1802, a thread dispatcher 1804, instruction cache 1806, a scalable execution unit array including a plurality of execution units 1808A-1808N, a sampler 1810, shared local memory 1811, a data cache 1812, and a data port 1814. Optionally, the scalable execution unit array can dynamically scale by enabling or disabling one or more execution units (e.g., any of execution units 1808A, 1808B, 1808C, 1808D, through 1808N-1 and 1808N) based on the computational

requirements of a workload. The included components may be interconnected via an interconnect fabric that links to each of the components. Thread execution logic 1800 may include one or more connections to memory, such as system memory or cache memory, through one or more of instruction cache 1806, data port 1814, sampler 1810, and execution units 1808A-1808N. Each execution unit (e.g. 1808A) may be a stand-alone programmable general-purpose computational unit that is capable of executing multiple simultaneous hardware threads while processing multiple data elements in parallel for each thread. In various embodiments, the array of execution units 1808A-1808N is scalable to include any number individual execution units.

[0276] The execution units 1808A-1808N may be primarily used to execute shader programs. A shader processor 1802 can process the various shader programs and dispatch execution threads associated with the shader programs via a thread dispatcher 1804. The thread dispatcher may include logic to arbitrate thread initiation requests from the graphics and media pipelines and instantiate the requested threads on one or more execution units 1808A-1808N. For example, a geometry pipeline can dispatch vertex, tessellation, or geometry shaders to the thread execution logic for processing. Optionally, the thread dispatcher 1804 can also process runtime thread spawning requests from the executing shader programs.

[0277] The execution units 1808A-1808N may support an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders). Each of the execution units 1808A-1808N is capable of multi-issue single instruction multiple data (SIMD) execution and multi-threaded operation enables an efficient execution environment in the face of higher latency memory accesses. Each hardware thread within each execution unit has a dedicated high-bandwidth register file and associated independent thread-state. Execution is multi-issue per clock to pipelines capable of integer, single and double precision floating point operations, SIMD branch capability, logical operations, transcendental operations, and other miscellaneous operations. While waiting for data from memory or one of the shared functions, dependency logic within the execution units 1808A-1808N causes a waiting thread to sleep until the requested data has been returned. While the waiting thread is sleeping, hardware resources may be devoted to processing other threads. For example, during a delay associated with a vertex shader operation, an execution unit can perform operations for a pixel shader, fragment shader, or another type of shader program, including a different vertex shader, such as vertex shader

2107 illustrated in **FIG. 21**. Various embodiments can apply to use execution by use of Single Instruction Multiple Thread (SIMT) as an alternate to use of SIMD or in addition to use of SIMD. Reference to a SIMD core or operation can apply also to SIMT or apply to SIMD in combination with SIMT.

[0278] Each execution unit in execution units 1808A-1808N operates on arrays of data elements. The number of data elements is the "execution size," or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical Arithmetic Logic Units (ALUs), Floating-Point Units (FPUs), or other logic units (e.g., tensor cores, ray tracing cores, etc.) for a particular graphics processor. Additionally, the execution units 1808A-1808N may support integer and floating-point data types.

[0279] The execution unit instruction set includes SIMD instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 184-bit packed data elements (Quad-Word (QW) size data elements), eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

[0280] Optionally, one or more execution units can be combined into a fused execution unit 1809A-1809N having thread control logic (1807A-1807N) that is common to the fused EUs. Multiple EUs can be fused into an EU group. Each EU in the fused EU group can be configured to execute a separate SIMD hardware thread. The number of EUs in a fused EU group can vary according to embodiments. Additionally, various SIMD widths can be performed per-EU, including but not limited to SIMD8, SIMD16, and SIMD32. Each fused graphics execution unit 1809A-1809N includes at least two execution units. For example, fused execution unit 1809A includes a first EU 1808A, second EU 1808B, and thread control logic 1807A that is common to the first EU 1808A and the second EU 1808B. The thread control logic 1807A controls threads executed on the fused graphics execution unit 1809A, allowing each EU within the fused execution units 1809A-1809N to execute using a common instruction pointer register.

[0281] One or more internal instruction caches (e.g., 1806) are included in the thread execution logic 1800 to cache thread instructions for the execution units. One or more data caches (e.g., 1812) may be included in the thread execution logic 1800 to cache thread data

during thread execution. Threads executing on the execution logic 1800 can also store explicitly managed data in the shared local memory 1811. A sampler 1810 may be included to provide texture sampling for 3D operations and media sampling for media operations. Sampler 1810 may include specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

[0282] During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 1800 via thread spawning and dispatch logic. Once a group of geometric objects has been processed and rasterized into pixel data, pixel processor logic (e.g., pixel shader logic, fragment shader logic, etc.) within the shader processor 1802 is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). A pixel shader or fragment shader may calculate the values of the various vertex attributes that are to be interpolated across the rasterized object. The pixel processor logic within the shader processor 1802 may then execute an application programming interface (API)-supplied pixel or fragment shader program. To execute the shader program, the shader processor 1802 dispatches threads to an execution unit (e.g., 1808A) via thread dispatcher 1804. Shader processor 1802 may use texture sampling logic in the sampler 1810 to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

[0283] In addition, the data port 1814 may provide a memory access mechanism for the thread execution logic 1800 to output processed data to memory for further processing on a graphics processor output pipeline. The data port 1814 may include or couple to one or more cache memories (e.g., data cache 1812) to cache data for memory access via the data port 1814.

[0284] Optionally, the execution logic 1800 can also include a ray tracer 1805 that can provide ray tracing acceleration functionality. The ray tracer 1805 can support a ray tracing instruction set that includes instructions/functions for ray generation. The ray tracing instruction set can be similar to or different from the ray-tracing instruction set supported by the ray tracing cores 372 in FIG. 3C.

[0285] FIG. 18B illustrates exemplary internal details of an execution unit 1808. A graphics execution unit 1808 can include an instruction fetch unit 1837, a general register file array (GRF) 1824, an architectural register file array (ARF) 1826, a thread arbiter 1822, a send unit 1830, a branch unit 1832, a set of SIMD floating point units (FPUs) 1834, and optionally a set of dedicated integer SIMD ALUs 1835. The GRF 1824 and ARF 1826 includes the set of general register files and architecture register files associated with each simultaneous hardware thread that may be active in the graphics execution unit 1808. Per thread architectural state may be

maintained in the ARF 1826, while data used during thread execution is stored in the GRF 1824. The execution state of each thread, including the instruction pointers for each thread, can be held in thread-specific registers in the ARF 1826.

[0286] The graphics execution unit 1808 may have an architecture that is a combination of Simultaneous Multi-Threading (SMT) and fine-grained Interleaved Multi-Threading (IMT). The architecture may have a modular configuration that can be fine-tuned at design time based on a target number of simultaneous threads and number of registers per execution unit, where execution unit resources are divided across logic used to execute multiple simultaneous threads. The number of logical threads that may be executed by the graphics execution unit 1808 is not limited to the number of hardware threads, and multiple logical threads can be assigned to each hardware thread.

[0287] Optionally, the graphics execution unit 1808 can co-issue multiple instructions, which may each be different instructions. The thread arbiter 1822 of the graphics execution unit thread 1808 can dispatch the instructions to one of the send unit 1830, branch unit 1832, or SIMD FPU(s) 1834 for execution. Each execution thread can access 128 general-purpose registers within the GRF 1824, where each register can store 32 bytes, accessible as a SIMD 8-element vector of 32-bit data elements. Each execution unit thread may have access to 4 Kbytes within the GRF 1824, although embodiments are not so limited, and greater or fewer register resources may be provided in other embodiments. The graphics execution unit 1808 may be partitioned into seven hardware threads that can independently perform computational operations, although the number of threads per execution unit can also vary according to embodiments, for example, up to 16 hardware threads may be supported. In an exemplary embodiment, in which seven threads may access 4 Kbytes, the GRF 1824 can store a total of 28 Kbytes. In another exemplary embodiment, where 16 threads may access 4Kbytes, the GRF 1824 can store a total of 64Kbytes. The number of threads per execution unit are, however, not limited to those examples and may be more or less than the given numbers. Flexible addressing modes can permit registers to be addressed together to build effectively wider registers or to represent strided rectangular block data structures.

[0288] Additionally or alternatively, memory operations, sampler operations, and other longer-latency system communications may be dispatched via "send" instructions that are executed by the message passing send unit 1830. Branch instructions may be dispatched to a dedicated branch unit 1832 to facilitate SIMD divergence and eventual convergence.

[0289] The graphics execution unit 1808 may include one or more SIMD floating point units (FPU(s)) 1834 to perform floating-point operations. The FPU(s) 1834 may also support integer computation. In some instances, the FPU(s) 1834 can SIMD execute up to *M* number of 32-bit

floating-point (or integer) operations, or SIMD execute up to $2M$ 16-bit integer or 16-bit floating-point operations. Optionally, at least one of the FPU(s) provides extended math capability to support high-throughput transcendental math functions and double precision 184-bit floating-point. A set of 8-bit integer SIMD ALUs 1835 may also be present, and may be specifically optimized to perform operations associated with machine learning computations.

[0290] Optionally, arrays of multiple instances of the graphics execution unit 1808 can be instantiated in a graphics sub-core grouping (e.g., a sub-slice). For scalability, product architects can choose the exact number of execution units per sub-core grouping. The execution unit 1808 may execute instructions across a plurality of execution channels. In addition, each thread executed on the graphics execution unit 1808 may be executed on a different channel.

[0291] **FIG. 19** illustrates a further exemplary execution unit 1900. The elements of FIG. 19 having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. The execution unit 1900 may be a compute-optimized execution unit for use in, for example, a compute engine tile 1640A-1640D as in FIG. 16C, but is not limited as such. The execution unit 1900 may also be used in a graphics engine tile 1610A-1610D as in FIG. 16B. The execution unit 1900 may include a thread control unit 1901, a thread state unit 1902, an instruction fetch/prefetch unit 1903, and an instruction decode unit 1904. The execution unit 1900 may additionally include a register file 1906 that stores registers that can be assigned to hardware threads within the execution unit. The execution unit 1900 may additionally include a send unit 1907 and a branch unit 1908. The send unit 1907 and branch unit 1908 may operate similarly as the send unit 1830 and a branch unit 1832 of the graphics execution unit 1808 of FIG. 18B.

[0292] The execution unit 1900 can also include a compute unit 1910 that includes multiple different types of functional units. The compute unit 1910 may also include an ALU unit 1911 that includes an array of arithmetic logic units. The ALU unit 1911 can be configured to perform 64-bit, 32-bit, and 16-bit integer and floating-point operations. Integer and floating-point operations may be performed simultaneously. The compute unit 1910 can also include a systolic array 1912, and a math unit 1913. The systolic array 1912 includes a W wide and D deep network of data processing units that can be used to perform vector or other data-parallel operations in a systolic manner. The systolic array 1912 can be configured to perform matrix operations, such as matrix dot product operations. The systolic array 1912 may support 16-bit floating point operations, as well as 8-bit and 4-bit integer operations. The systolic array 1912 may be configured to accelerate machine learning operations. The systolic array 1912 can be

configured with support for the bfloat16, a 16-bit floating point format. A math unit 1913 can be included to perform a specific subset of mathematical operations in an efficient and lower-power manner than then ALU unit 1911. The math unit 1913 can include math logic found in shared function logic of a graphics processing engine provided by other embodiments described, e.g., the math logic 1722 of the shared function logic 1720 of FIG. 17. The math unit 1913 can be configured to perform 32-bit and 64-bit floating point operations.

[0293] The thread control unit 1901 includes logic to control the execution of threads within the execution unit. The thread control unit 1901 can include thread arbitration logic to start, stop, and preempt execution of threads within the execution unit 1900. The thread state unit 1902 can be used to store thread state for threads assigned to execute on the execution unit 1900. Storing the thread state within the execution unit 1900 enables the rapid pre-emption of threads when those threads become blocked or idle. The instruction fetch/prefetch unit 1903 can fetch instructions from an instruction cache of higher-level execution logic (e.g., instruction cache 1806 as in FIG. 18A). The instruction fetch/prefetch unit 1903 can also issue prefetch requests for instructions to be loaded into the instruction cache based on an analysis of currently executing threads. The instruction decode unit 1904 can be used to decode instructions to be executed by the compute units. The instruction decode unit 1904 can be used as a secondary decoder to decode complex instructions into constituent micro-operations.

[0294] The execution unit 1900 additionally includes a register file 1906 that can be used by hardware threads executing on the execution unit 1900. Registers in the register file 1906 can be divided across the logic used to execute multiple simultaneous threads within the compute unit 1910 of the execution unit 1900. The number of logical threads that may be executed by the graphics execution unit 1900is not limited to the number of hardware threads, and multiple logical threads can be assigned to each hardware thread. The size of the register file 1906 can vary across embodiments based on the number of supported hardware threads. Register renaming may be used to dynamically allocate registers to hardware threads.

[0295] **FIG. 20** is a block diagram illustrating a graphics processor instruction format 2000. The graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. The instruction formats 2000 described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

[0296] The graphics processor execution units as described herein may natively support instructions in a 128-bit instruction format 2010. A 64-bit compacted instruction format 2030 is

available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit instruction format 2010 provides access to all instruction options, while some options and operations are restricted in the 64-bit format 2030. The native instructions available in the 64-bit format 2030 vary by embodiment. The instruction is compacted in part using a set of index values in an index field 2013. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit instruction format 2010. Other sizes and formats of instruction can be used.

[0297] For each format, instruction opcode 2012 defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. Instruction control field 2014 may enable control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For instructions in the 128-bit instruction format 2010 an exec-size field 2016 limits the number of data channels that will be executed in parallel. An exec-size field 2016 may not be available for use in the 64-bit compact instruction format 2030.

[0298] Some execution unit instructions have up to three operands including two source operands, src0 2020, src1 2022, and one destination 2018. The execution units may support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 2024), where the instruction opcode 2012 determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

[0299] The 128-bit instruction format 2010 may include an access/address mode field 2026 specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction.

[0300] The 128-bit instruction format 2010 may also include an access/address mode field 2026, which specifies an address mode and/or an access mode for the instruction. The access mode may be used to define a data access alignment for the instruction. Access modes including a 16-byte aligned access mode and a 1-byte aligned access mode may be supported, where the byte alignment of the access mode determines the access alignment of the instruction operands. For example, when in a first mode, the instruction may use byte-aligned addressing for source

and destination operands and when in a second mode, the instruction may use 16-byte-aligned addressing for all source and destination operands.

[0301] The address mode portion of the access/address mode field 2026 may determine whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed based on an address register value and an address immediate field in the instruction.

[0302] Instructions may be grouped based on opcode 2012 bit-fields to simplify Opcode decode 2040. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. A move and logic opcode group 2042 may include data movement and logic instructions (e.g., move (mov), compare (cmp)). Move and logic group 2042 may share the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group 2044 (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group 2046 includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group 2048 includes component-wise arithmetic instructions (e.g., add, multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group 2048 performs the arithmetic operations in parallel across data channels. The vector math group 2050 includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands. The illustrated opcode decode 2040, in one embodiment, can be used to determine which portion of an execution unit will be used to execute a decoded instruction. For example, some instructions may be designated as systolic instructions that will be performed by a systolic array. Other instructions, such as ray-tracing instructions (not shown) can be routed to a ray-tracing core or ray-tracing logic within a slice or partition of execution logic.

Graphics Pipeline

[0303] **FIG. 21** is a block diagram of graphics processor 2100, according to another embodiment. The elements of FIG. 21 having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such.

[0304] The graphics processor 2100 may include different types of graphics processing pipelines, such as a geometry pipeline 2120, a media pipeline 2130, a display engine 2140, thread execution logic 2150, and a render output pipeline 2170. Graphics processor 2100 may be a graphics processor within a multi-core processing system that includes one or more general-purpose processing cores. The graphics processor may be controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor 2100 via a ring interconnect 2102. Ring interconnect 2102 may couple graphics processor 2100 to other processing components, such as other graphics processors or general-purpose processors. Commands from ring interconnect 2102 are interpreted by a command streamer 2103, which supplies instructions to individual components of the geometry pipeline 2120 or the media pipeline 2130.

[0305] Command streamer 2103 may direct the operation of a vertex fetcher 2105 that reads vertex data from memory and executes vertex-processing commands provided by command streamer 2103. The vertex fetcher 2105 may provide vertex data to a vertex shader 2107, which performs coordinate space transformation and lighting operations to each vertex. Vertex fetcher 2105 and vertex shader 2107 may execute vertex-processing instructions by dispatching execution threads to execution units 2152A-2152B via a thread dispatcher 2131.

[0306] The execution units 2152A-2152B may be an array of vector processors having an instruction set for performing graphics and media operations. The execution units 2152A-2152B may have an attached L1 cache 2151 that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

[0307] A geometry pipeline 2120 may include tessellation components to perform hardware-accelerated tessellation of 3D objects. A programmable hull shader 2111 may configure the tessellation operations. A programmable domain shader 2117 may provide back-end evaluation of tessellation output. A tessellator 2113 may operate at the direction of hull shader 2111 and contain special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to geometry pipeline 2120. In addition, if tessellation is not used, tessellation components (e.g., hull shader 2111, tessellator 2113, and domain shader 2117) can be bypassed.

[0308] Complete geometric objects may be processed by a geometry shader 2119 via one or more threads dispatched to execution units 2152A-2152B, or can proceed directly to the clipper 2129. The geometry shader may operate on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader 2119 receives input from the vertex shader 2107. The geometry shader

2119 may be programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

[0309] Before rasterization, a clipper 2129 processes vertex data. The clipper 2129 may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. A rasterizer and depth test component 2173 in the render output pipeline 2170 may dispatch pixel shaders to convert the geometric objects into per pixel representations. The pixel shader logic may be included in thread execution logic 2150. Optionally, an application can bypass the rasterizer and depth test component 2173 and access un-rasterized vertex data via a stream out unit 2123.

[0310] The graphics processor 2100 has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units 2152A-2152B and associated logic units (e.g., L1 cache 2151, sampler 2154, texture cache 2158, etc.) interconnect via a data port 2156 to perform memory access and communicate with render output pipeline components of the processor. A sampler 2154, caches 2151, 2158 and execution units 2152A-2152B each may have separate memory access paths. Optionally, the texture cache 2158 can also be configured as a sampler cache.

[0311] The render output pipeline 2170 may contain a rasterizer and depth test component 2173 that converts vertex-based objects into an associated pixel-based representation. The rasterizer logic may include a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache 2178 and depth cache 2179 are also available in some embodiments. A pixel operations component 2177 performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine 2141, or substituted at display time by the display controller 2143 using overlay display planes. A shared L3 cache 2175 may be available to all graphics components, allowing the sharing of data without the use of main system memory.

[0312] The graphics processor media pipeline 2130 may include a media engine 2137 and a video front-end 2134. Video front-end 2134 may receive pipeline commands from the command streamer 2103. The media pipeline 2130 may include a separate command streamer. Video front-end 2134 may process media commands before sending the command to the media engine 2137. Media engine 2137 may include thread spawning functionality to spawn threads for dispatch to thread execution logic 2150 via thread dispatcher 2131.

[0313] The graphics processor 2100 may include a display engine 2140. This display engine 2140 may be external to processor 2100 and may couple with the graphics processor via the ring

interconnect 2102, or some other interconnect bus or fabric. Display engine 2140 may include a 2D engine 2141 and a display controller 2143. Display engine 2140 may contain special purpose logic capable of operating independently of the 3D pipeline. Display controller 2143 may couple with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

[0314] The geometry pipeline 2120 and media pipeline 2130 maybe configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). A driver software for the graphics processor may translate API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. Support may be provided for the Open Graphics Library (OpenGL), Open Computing Language (OpenCL), and/or Vulkan graphics and compute API, all from the Khronos Group. Support may also be provided for the Direct3D library from the Microsoft Corporation. A combination of these libraries may be supported. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

Graphics Pipeline Programming

[0315] **FIG. 22A** is a block diagram illustrating a graphics processor command format 2200 used for programming graphics processing pipelines, such as, for example, the pipelines described herein in conjunction with FIG. 16A, 17, 21. **FIG. 22B** is a block diagram illustrating a graphics processor command sequence 2210 according to an embodiment. The solid lined boxes in **FIG. 22A** illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format 2200 of **FIG. 22A** includes data fields to identify a client 2202, a command operation code (opcode) 2204, and data 2206 for the command. A sub-opcode 2205 and a command size 2208 are also included in some commands.

[0316] Client 2202 may specify the client unit of the graphics device that processes the command data. A graphics processor command parser may examine the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. The graphics processor client units may include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit may have a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads the opcode 2204 and, if present, sub-opcode 2205 to determine the operation to perform. The client unit performs the command using information in data field

2206. For some commands an explicit command size 2208 is expected to specify the size of the command. The command parser may automatically determine the size of at least some of the commands based on the command opcode. Commands may be aligned via multiples of a double word. Other command formats can also be used.

[0317] The flow diagram in **FIG. 22B** illustrates an exemplary graphics processor command sequence 2210. Software or firmware of a data processing system that features an exemplary graphics processor may use a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only and is not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.

[0318] The graphics processor command sequence 2210 may begin with a pipeline flush command 2212 to cause any active graphics pipeline to complete the currently pending commands for the pipeline. Optionally, the 3D pipeline 2222 and the media pipeline 2224 may not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. Pipeline flush command 2212 can be used for pipeline synchronization or before placing the graphics processor into a low power state.

[0319] A pipeline select command 2213 may be used when a command sequence requires the graphics processor to explicitly switch between pipelines. A pipeline select command 2213 may be required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. A pipeline flush command 2212 may be required immediately before a pipeline switch via the pipeline select command 2213.

[0320] A pipeline control command 2214 may configure a graphics pipeline for operation and may be used to program the 3D pipeline 2222 and the media pipeline 2224. The pipeline control command 2214 may configure the pipeline state for the active pipeline. The pipeline control command 2214 may be used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.

[0321] Return buffer state commands 2216 may be used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write

intermediate data during processing. The graphics processor may also use one or more return buffers to store output data and to perform cross thread communication. The return buffer state 2216 may include selecting the size and number of return buffers to use for a set of pipeline operations.

[0322] The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination 2220, the command sequence is tailored to the 3D pipeline 2222 beginning with the 3D pipeline state 2230 or the media pipeline 2224 beginning at the media pipeline state 2240.

[0323] The commands to configure the 3D pipeline state 2230 include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based on the particular 3D API in use. The 3D pipeline state 2230 commands may also be able to selectively disable or bypass certain pipeline elements if those elements will not be used.

[0324] A 3D primitive 2232 command may be used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive 2232 command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive 2232 command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. The 3D primitive 2232 command may be used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, 3D pipeline 2222 dispatches shader execution threads to graphics processor execution units.

[0325] The 3D pipeline 2222 may be triggered via an execute 2234 command or event. A register may write trigger command executions. An execution may be triggered via a 'go' or 'kick' command in the command sequence. Command execution may be triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

[0326] The graphics processor command sequence 2210 may follow the media pipeline 2224 path when performing media operations. In general, the specific use and manner of programming for the media pipeline 2224 depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. The media pipeline can also be bypassed and media decode can be performed in

whole or in part using resources provided by one or more general-purpose processing cores. The media pipeline may also include elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

[0327] Media pipeline 2224 may be configured in a similar manner as the 3D pipeline 2222. A set of commands to configure the media pipeline state 2240 are dispatched or placed into a command queue before the media object commands 2242. Commands for the media pipeline state 2240 may include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. Commands for the media pipeline state 2240 may also support the use of one or more pointers to "indirect" state elements that contain a batch of state settings.

[0328] Media object commands 2242 may supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. Optionally, all media pipeline states must be valid before issuing a media object command 2242. Once the pipeline state is configured and media object commands 2242 are queued, the media pipeline 2224 is triggered via an execute command 2244 or an equivalent execute event (e.g., register write). Output from media pipeline 2224 may then be post processed by operations provided by the 3D pipeline 2222 or the media pipeline 2224. GPGPU operations may be configured and executed in a similar manner as media operations.

Graphics Software Architecture

[0329] **FIG. 23** illustrates an exemplary graphics software architecture for a data processing system 2300. Such a software architecture may include a 3D graphics application 2310, an operating system 2320, and at least one processor 2330. Processor 2330 may include a graphics processor 2332 and one or more general-purpose processor core(s) 2334. The processor 2330 may be a variant of the processor 1402 or any other of the processors described herein. The processor 2330 may be used in place of the processor 1402 or any other of the processors described herein. Therefore, the disclosure of any features in combination with the processor 1402 or any other of the processors described herein also discloses a corresponding combination with the graphics processor 2330, but is not limited to such. Moreover, the elements of FIG. 23 having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein,

but are not limited to such. The graphics application 2310 and operating system 2320 are each executed in the system memory 2350 of the data processing system.

[0330] 3D graphics application 2310 may contain one or more shader programs including shader instructions 2312. The shader language instructions may be in a high-level shader language, such as the High-Level Shader Language (HLSL) of Direct3D, the OpenGL Shader Language (GLSL), and so forth. The application may also include executable instructions 2314 in a machine language suitable for execution by the general-purpose processor core 2334. The application may also include graphics objects 2316 defined by vertex data.

[0331] The operating system 2320 may be a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. The operating system 2320 can support a graphics API 2322 such as the Direct3D API, the OpenGL API, or the Vulkan API. When the Direct3D API is in use, the operating system 2320 uses a front-end shader compiler 2324 to compile any shader instructions 2312 in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. High-level shaders may be compiled into low-level shaders during the compilation of the 3D graphics application 2310. The shader instructions 2312 may be provided in an intermediate form, such as a version of the Standard Portable Intermediate Representation (SPIR) used by the Vulkan API.

[0332] User mode graphics driver 2326 may contain a back-end shader compiler 2327 to convert the shader instructions 2312 into a hardware specific representation. When the OpenGL API is in use, shader instructions 2312 in the GLSL high-level language are passed to a user mode graphics driver 2326 for compilation. The user mode graphics driver 2326 may use operating system kernel mode functions 2328 to communicate with a kernel mode graphics driver 2329. The kernel mode graphics driver 2329 may communicate with graphics processor 2332 to dispatch commands and instructions.

IP Core Implementations

[0333] One or more aspects may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as "IP cores," are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model may be supplied to various customers or

manufacturing facilities, which load the hardware model on fabrication machines that manufacture the integrated circuit. The integrated circuit may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

[0334] **FIG. 24A** is a block diagram illustrating an IP core development system 2400 that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system 2400 may be used to generate modular, re-usable designs that can be incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility 2430 can generate a software simulation 2410 of an IP core design in a high-level programming language (e.g., C/C++). The software simulation 2410 can be used to design, test, and verify the behavior of the IP core using a simulation model 2412. The simulation model 2412 may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design 2415 can then be created or synthesized from the simulation model 2412. The RTL design 2415 is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design 2415, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

[0335] The RTL design 2415 or equivalent may be further synthesized by the design facility into a hardware model 2420, which may be in a hardware description language (HDL), or some other representation of physical design data. The HDL may be further simulated or tested to verify the IP core design. The IP core design can be stored for delivery to a 3rd party fabrication facility 2465 using non-volatile memory 2440 (e.g., hard disk, flash memory, or any non-volatile storage medium). Alternatively, the IP core design may be transmitted (e.g., via the Internet) over a wired connection 2450 or wireless connection 2460. The fabrication facility 2465 may then fabricate an integrated circuit that is based at least in part on the IP core design. The fabricated integrated circuit can be configured to perform operations in accordance with at least one embodiment described herein.

[0336] **FIG. 24B** illustrates a cross-section side view of an integrated circuit package assembly 2470. The integrated circuit package assembly 2470 illustrates an implementation of one or more processor or accelerator devices as described herein. The package assembly 2470 includes multiple units of hardware logic 2472, 2474 connected to a substrate 2480. The logic 2472, 2474 may be implemented at least partly in configurable logic or fixed-functionality logic hardware, and can include one or more portions of any of the processor core(s), graphics processor(s), or other accelerator devices described herein. Each unit of logic 2472, 2474 can be

implemented within a semiconductor die and coupled with the substrate 2480 via an interconnect structure 2473. The interconnect structure 2473 may be configured to route electrical signals between the logic 2472, 2474 and the substrate 2480, and can include interconnects such as, but not limited to bumps or pillars. The interconnect structure 2473 may be configured to route electrical signals such as, for example, input/output (I/O) signals and/or power or ground signals associated with the operation of the logic 2472, 2474. Optionally, the substrate 2480 may be an epoxy-based laminate substrate. The substrate 2480 may also include other suitable types of substrates. The package assembly 2470 can be connected to other electrical devices via a package interconnect 2483. The package interconnect 2483 may be coupled to a surface of the substrate 2480 to route electrical signals to other electrical devices, such as a motherboard, other chipset, or multi-chip module.

[0337] The units of logic 2472, 2474 may be electrically coupled with a bridge 2482 that is configured to route electrical signals between the logic 2472, 2474. The bridge 2482 may be a dense interconnect structure that provides a route for electrical signals. The bridge 2482 may include a bridge substrate composed of glass or a suitable semiconductor material. Electrical routing features can be formed on the bridge substrate to provide a chip-to-chip connection between the logic 2472, 2474.

[0338] Although two units of logic 2472, 2474 and a bridge 2482 are illustrated, embodiments described herein may include more or fewer logic units on one or more dies. The one or more dies may be connected by zero or more bridges, as the bridge 2482 may be excluded when the logic is included on a single die. Alternatively, multiple dies or units of logic can be connected by one or more bridges. Additionally, multiple logic units, dies, and bridges can be connected together in other possible configurations, including three-dimensional configurations.

[0339] **FIG. 24C** illustrates a package assembly 2490 that includes multiple units of hardware logic chiplets connected to a substrate 2480 (e.g., base die). A graphics processing unit, parallel processor, and/or compute accelerator as described herein can be composed from diverse silicon chiplets that are separately manufactured. In this context, a chiplet is an at least partially packaged integrated circuit that includes distinct units of logic that can be assembled with other chiplets into a larger package. A diverse set of chiplets with different IP core logic can be assembled into a single device. Additionally the chiplets can be integrated into a base die or base chiplet using active interposer technology. The concepts described herein enable the interconnection and communication between the different forms of IP within the GPU. IP cores can be manufactured using different process technologies and composed during manufacturing, which avoids the complexity of converging multiple IPs, especially on a large SoC with several flavors IPs, to the same manufacturing process. Enabling the use of multiple process

technologies improves the time to market and provides a cost-effective way to create multiple product SKUs. Additionally, the disaggregated IPs are more amenable to being power gated independently, components that are not in use on a given workload can be powered off, reducing overall power consumption.

[0340] The hardware logic chiplets can include special purpose hardware logic chiplets 2472, logic or I/O chiplets 2474, and/or memory chiplets 2475. The hardware logic chiplets 2472 and logic or I/O chiplets 2474 may be implemented at least partly in configurable logic or fixed-functionality logic hardware and can include one or more portions of any of the processor core(s), graphics processor(s), parallel processors, or other accelerator devices described herein. The memory chiplets 2475 can be DRAM (e.g., GDDR, HBM) memory or cache (SRAM) memory.

[0341] Each chiplet can be fabricated as separate semiconductor die and coupled with the substrate 2480 via an interconnect structure 2473. The interconnect structure 2473 may be configured to route electrical signals between the various chiplets and logic within the substrate 2480. The interconnect structure 2473 can include interconnects such as, but not limited to bumps or pillars. In some embodiments, the interconnect structure 2473 may be configured to route electrical signals such as, for example, input/output (I/O) signals and/or power or ground signals associated with the operation of the logic, I/O and memory chiplets.

[0342] The substrate 2480 may be an epoxy-based laminate substrate, however, it is not limited to that and the substrate 2480 may also include other suitable types of substrates. The package assembly 2490 can be connected to other electrical devices via a package interconnect 2483. The package interconnect 2483 may be coupled to a surface of the substrate 2480 to route electrical signals to other electrical devices, such as a motherboard, other chipset, or multi-chip module.

[0343] A logic or I/O chiplet 2474 and a memory chiplet 2475 may be electrically coupled via a bridge 2487 that is configured to route electrical signals between the logic or I/O chiplet 2474 and a memory chiplet 2475. The bridge 2487 may be a dense interconnect structure that provides a route for electrical signals. The bridge 2487 may include a bridge substrate composed of glass or a suitable semiconductor material. Electrical routing features can be formed on the bridge substrate to provide a chip-to-chip connection between the logic or I/O chiplet 2474 and a memory chiplet 2475. The bridge 2487 may also be referred to as a silicon bridge or an interconnect bridge. For example, the bridge 2487 is an Embedded Multi-die Interconnect Bridge (EMIB). Alternatively, the bridge 2487 may simply be a direct connection from one chiplet to another chiplet.

[0344] The substrate 2480 can include hardware components for I/O 2491, cache memory 2492, and other hardware logic 2493. A fabric 2485 can be embedded in the substrate 2480 to enable communication between the various logic chiplets and the logic 2491, 2493 within the substrate 2480. Optionally, the I/O 2491, fabric 2485, cache, bridge, and other hardware logic 2493 can be integrated into a base die that is layered on top of the substrate 2480.

[0345] Furthermore, a package assembly 2490 can also include a smaller or greater number of components and chiplets that are interconnected by a fabric 2485 or one or more bridges 2487. The chiplets within the package assembly 2490 may be arranged in a 3D or 2.5D arrangement. In general, bridge structures 2487 may be used to facilitate a point to point interconnect between, for example, logic or I/O chiplets and memory chiplets. The fabric 2485 can be used to interconnect the various logic and/or I/O chiplets (e.g., chiplets 2472, 2474, 2491, 2493). with other logic and/or I/O chiplets. The cache memory 2492 within the substrate can act as a global cache for the package assembly 2490, part of a distributed global cache, or as a dedicated cache for the fabric 2485.

[0346] **FIG. 24D** illustrates a package assembly 2494 including interchangeable chiplets 2495, according to an embodiment. The interchangeable chiplets 2495 can be assembled into standardized slots on one or more base chiplets 2496, 2498. The base chiplets 2496, 2498 can be coupled via a bridge interconnect 2497, which can be similar to the other bridge interconnects described herein and may be, for example, an EMIB. Memory chiplets can also be connected to logic or I/O chiplets via a bridge interconnect. I/O and logic chiplets can communicate via an interconnect fabric. The base chiplets can each support one or more slots in a standardized format for one of logic or I/O or memory/cache.

[0347] SRAM and power delivery circuits may be fabricated into one or more of the base chiplets 2496, 2498, which can be fabricated using a different process technology relative to the interchangeable chiplets 2495 that are stacked on top of the base chiplets. For example, the base chiplets 2496, 2498 can be fabricated using a larger process technology, while the interchangeable chiplets can be manufactured using a smaller process technology. One or more of the interchangeable chiplets 2495 may be memory (e.g., DRAM) chiplets. Different memory densities can be selected for the package assembly 2494 based on the power, and/or performance targeted for the product that uses the package assembly 2494. Additionally, logic chiplets with a different number of type of functional units can be selected at time of assembly based on the power, and/or performance targeted for the product. Additionally, chiplets containing IP logic cores of differing types can be inserted into the interchangeable chiplet slots, enabling hybrid processor designs that can mix and match different technology IP blocks.

Exemplary System on a Chip Integrated Circuit

[0348] **FIG. 25-26** illustrate exemplary integrated circuits and associated graphics processors that may be fabricated using one or more IP cores. In addition to what is illustrated, other logic and circuits may be included, including additional graphics processors/cores, peripheral interface controllers, or general-purpose processor cores. The elements of FIG. 25-26 having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such.

[0349] **FIG. 25** is a block diagram illustrating an exemplary system on a chip integrated circuit 2500 that may be fabricated using one or more IP cores. Exemplary integrated circuit 2500 includes one or more application processor(s) 2505 (e.g., CPUs), at least one graphics processor 2510, which may be a variant of the graphics processor 1408, 1508, 2510, or of any graphics processor described herein and may be used in place of any graphics processor described. Therefore, the disclosure of any features in combination with a graphics processor herein also discloses a corresponding combination with the graphics processor 2510, but is not limited to such. The integrated circuit 2500 may additionally include an image processor 2515 and/or a video processor 2520, any of which may be a modular IP core from the same or multiple different design facilities. Integrated circuit 2500 may include peripheral or bus logic including a USB controller 2525, UART controller 2530, an SPI/SDIO controller 2535, and an I^2S/I^2C controller 2540. Additionally, the integrated circuit can include a display device 2545 coupled to one or more of a high-definition multimedia interface (HDMI) controller 2550 and a mobile industry processor interface (MIPI) display interface 2555. Storage may be provided by a flash memory subsystem 2560 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller 2565 for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine 2570.

[0350] **FIG. 26A-26B** are block diagrams illustrating exemplary graphics processors for use within an SoC, according to embodiments described herein. The graphics processors may be variants of the graphics processor 1408, 1508, 2510, or any other graphics processor described herein. The graphics processors may be used in place of the graphics processor 1408, 1508, 2510, or any other of the graphics processors described herein. Therefore, the disclosure of any features in combination with the graphics processor 1408, 1508, 2510, or any other of the graphics processors described herein also discloses a corresponding combination with the graphics processors of FIG. 26A-26B, but is not limited to such. **FIG. 26A** illustrates an

exemplary graphics processor 2610 of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. **FIG. 26B** illustrates an additional exemplary graphics processor 2640 of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor 2610 of FIG. 26A is an example of a low power graphics processor core. Graphics processor 2640 of FIG. 26B is an example of a higher performance graphics processor core. For example, each of the graphics processors 2610, 2640 can be a variant of the graphics processor 2510 of **FIG. 25**, as mentioned at the outset of this paragraph.

[0351] As shown in FIG. 26A, graphics processor 2610 includes a vertex processor 2605 and one or more fragment processor(s) 2615A-2615N (e.g., 2615A, 2615B, 2615C, 2615D, through 2615N-1, and 2615N). Graphics processor 2610 can execute different shader programs via separate logic, such that the vertex processor 2605 is optimized to execute operations for vertex shader programs, while the one or more fragment processor(s) 2615A-2615N execute fragment (e.g., pixel) shading operations for fragment or pixel shader programs. The vertex processor 2605 performs the vertex processing stage of the 3D graphics pipeline and generates primitives and vertex data. The fragment processor(s) 2615A-2615N use the primitive and vertex data generated by the vertex processor 2605 to produce a framebuffer that is displayed on a display device. The fragment processor(s) 2615A-2615N may be optimized to execute fragment shader programs as provided for in the OpenGL API, which may be used to perform similar operations as a pixel shader program as provided for in the Direct 3D API.

[0352] Graphics processor 2610 additionally includes one or more memory management units (MMUs) 2620A-2620B, cache(s) 2625A-2625B, and circuit interconnect(s) 2630A-2630B. The one or more MMU(s) 2620A-2620B provide for virtual to physical address mapping for the graphics processor 2610, including for the vertex processor 2605 and/or fragment processor(s) 2615A-2615N, which may reference vertex or image/texture data stored in memory, in addition to vertex or image/texture data stored in the one or more cache(s) 2625A-2625B. The one or more MMU(s) 2620A-2620B may be synchronized with other MMUs within the system, including one or more MMUs associated with the one or more application processor(s) 2505, image processor 2515, and/or video processor 2520 of **FIG. 25**, such that each processor 2505-2520 can participate in a shared or unified virtual memory system. Components of graphics processor 2610 may correspond with components of other graphics processors described herein. The one or more MMU(s) 2620A-2620B may correspond with MMU 245 of FIG. 2C. Vertex processor 2605 and fragment processor(s) 2615A-2615N may correspond with graphics multiprocessor 234. The one or more circuit interconnect(s) 2630A-2630B enable graphics processor 2610 to interface with other IP cores within the SoC, either via an internal bus of the

SoC or via a direct connection, according to embodiments. The one or more circuit interconnect(s) 2630A-2630B may correspond with the data crossbar 240 of FIG. 2C. Further correspondence may be found between analogous components of the graphics processor 2610 and the various graphics processor architectures described herein.

[0353] As shown FIG. 26B, graphics processor 2640 includes the one or more MMU(s) 2620A-2620B, cache(s) 2625A-2625B, and circuit interconnect(s) 2630A-2630B of the graphics processor 2610 of **FIG. 26A**. Graphics processor 2640 includes one or more shader cores 2655A-2655N (e.g., 2655A, 2655B, 2655C, 2655D, 2655E, 2655F, through 2655N-1, and 2655N), which provides for a unified shader core architecture in which a single core or type or core can execute all types of programmable shader code, including shader program code to implement vertex shaders, fragment shaders, and/or compute shaders. The exact number of shader cores present can vary among embodiments and implementations. Additionally, graphics processor 2640 includes an inter-core task manager 2645, which acts as a thread dispatcher to dispatch execution threads to one or more shader cores 2655A-2655N and a tiling unit 2658 to accelerate tiling operations for tile-based rendering, in which rendering operations for a scene are subdivided in image space, for example to exploit local spatial coherence within a scene or to optimize use of internal caches. Shader cores 2655A-2655N may correspond with, for example, graphics multiprocessor 234 as in FIG. 2D, or graphics multiprocessors 325, 350 of FIG. 3A and 3B respectively, or multi-core group 365A of FIG. 3C.

[0354] Embodiments described herein include, software, firmware, and hardware logic that provides techniques to perform arithmetic on sparse data via a systolic processing unit. Embodiment described herein provided techniques to skip computational operations for zero filled matrices and sub-matrices. Embodiments additionally provide techniques to maintain data compression through to a processing unit. Embodiments additionally provide an architecture for a sparse aware logic unit.

GPGPU with Tensor Acceleration Logic and Unified Memory

[0355] **FIG. 27** is a block diagram of a data processing system 2700, according to an embodiment. The data processing system 2700 is a heterogeneous processing system having a processor 2702, unified memory 2710, and a GPGPU 2720 including machine learning acceleration logic. The processor 2702 and the GPGPU 2720 can be any of the processors and GPGPU/parallel processors as described herein. The processor 2702 can execute instructions for a compiler 2715 stored in system memory 2712. The compiler 2715 executes on the processor 2702 to compile source code 2714A into compiled code 2714B. The compiled code 2714B can include instructions that may be executed by the processor 2702 and/or instructions that may be executed by the GPGPU 2720. During compilation, the compiler 2715 can perform operations

to insert metadata, including hints as to the level of data parallelism present in the compiled code 2714B and/or hints regarding the data locality associated with threads to be dispatched based on the compiled code 2714B. The compiler 2715 can include the information necessary to perform such operations or the operations can be performed with the assistance of a runtime library 2716. The runtime library 2716 can also assist the compiler 2715 in the compilation of the source code 2714A and can also include instructions that are linked at runtime with the compiled code 2714B to facilitate execution of the compiled instructions on the GPGPU 2720.

[0356] The unified memory 2710 represents a unified address space that may be accessed by the processor 2702 and the GPGPU 2720. The unified memory can include system memory 2712 as well as GPGPU memory 2718. The GPGPU memory 2718 is memory within an address pace of the GPGPU 2720 and can include some or all of system memory 2712. In one embodiment the GPGPU memory 2718 can also include at least a portion of any memory dedicated for use exclusively by the GPGPU 2720. In one embodiment, compiled code 2714B stored in system memory 2712 can be mapped into GPGPU memory 2718 for access by the GPGPU 2720.

[0357] The GPGPU 2720 includes multiple compute blocks 2724A-2724N, which can include one or more of a variety of processing resources described herein. The processing resources can be or include a variety of different computational resources such as, for example, execution units, compute units, streaming multiprocessors, graphics multiprocessors, or multi-core groups. In one embodiment the GPGPU 2720 additionally includes a tensor (e.g., matrix) accelerator 2723, which can include one or more special function compute units that are designed to accelerate a subset of matrix operations (e.g., dot product, etc.). The tensor accelerator 2723 may also be referred to as a tensor accelerator or tensor core. In one embodiment, logic components within the tensor accelerator 2723 may be distributed across the processing resources of the multiple compute blocks 2724A-2724N.

[0358] The GPGPU 2720 can also include a set of resources that can be shared by the compute blocks 2724A-2724N and the tensor accelerator 2723, including but not limited to a set of registers 2725, a power and performance module 2726, and a cache 2727. In one embodiment the registers 2725 include directly and indirectly accessible registers, where the indirectly accessible registers are optimized for use by the tensor accelerator 2723. The power and performance module 2726 can be configured to adjust power delivery and clock frequencies for the compute blocks 2724A-2724N to power gate idle components within the compute blocks 2724A-2724N. In various embodiments the cache 2727 can include an instruction cache and/or a lower level data cache.

[0359] The GPGPU 2720 can additionally include an L3 data cache 2730, which can be used to cache data accessed from the unified memory 2710 by the tensor accelerator 2723 and/or the compute elements within the compute blocks 2724A-2724N. In one embodiment the L3 data cache 2730 includes shared local memory 2732 that can be shared by the compute elements within the compute blocks 2724A-2724N and the tensor accelerator 2723.

[0360] In one embodiment the GPGPU 2720 includes instruction handling logic, such as a fetch and decode unit 2721 and a scheduler controller 2722. The fetch and decode unit 2721 includes a fetch unit and decode unit to fetch and decode instructions for execution by one or more of the compute blocks 2724A-2724N or the tensor accelerator 2723. The instructions can be scheduled to the appropriate functional unit within the compute block 2724A-2724N or the tensor accelerator via the scheduler controller 2722. In one embodiment the scheduler controller 2722 is an ASIC configurable to perform advanced scheduling operations. In one embodiment the scheduler controller 2722 is a micro-controller or a low energy-per-instruction processing core capable of executing scheduler instructions loaded from a firmware module.

[0361] In one embodiment some functions to be performed by the compute blocks 2724A-2724N can be directly scheduled to or offloaded to the tensor accelerator 2723. In various embodiments the tensor accelerator 2723 includes processing element logic configured to efficiently perform matrix compute operations, such as multiply and add operations and dot product operations used by 3D graphics or compute shader programs. In one embodiment the tensor accelerator 2723 can be configured to accelerate operations used by machine learning frameworks. In one embodiment the tensor accelerator 2723 is an application specific integrated circuit explicitly configured to perform a specific set of parallel matrix multiplication and/or addition operations. In one embodiment the tensor accelerator 2723 is a field programmable gate array (FPGA) that provides fixed function logic that can updated between workloads. The set of matrix operations that can be performed by the tensor accelerator 2723 may be limited relative to the operations that can be performed by the compute block 2724A-2724N. However, the tensor accelerator 2723 can perform those the operations at a significantly higher throughput relative to the compute block 2724A-2724N.

[0362] **FIG. 28** illustrates a matrix operation 2805 performed by an instruction pipeline 2800, according to an embodiment. The instruction pipeline 2800 can be configured to perform a matrix operation 2805, such as, but not limited to a dot product operation. The dot product of two vectors is a scalar value that is equal to sum of products of corresponding components of the vectors. The dot product can be calculated as shown in equation (1) below.

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + \ldots + a_n b_n \qquad (1)$$

[0363] The dot product can be used in a convolution operation for a convolutional neural network (CNN). FIG. 28 illustrates a two-dimensional (2D) convolution using a matrix operation 2805 including a dot product operation. While 2D convolution is illustrated, N-dimensional convolution can be performed on an N-dimensional volume using N-dimensional filters. A receptive field tile 2802 highlights a portion of an input volume in an input volume buffer 2804. The input volume buffer can be stored in memory 2830. A dot product matrix operation 2805 can be performed between the data within the receptive field tile 2802 and a convolutional filter to generate a data point within output buffer 2806, which can also be stored in memory 2830. The memory 2830 can be any of the memory described herein, including system memory 2712, GPGPU memory 2718, or one or more cache memories 2727, 2730 as in FIG 27.

[0364] The combination of the data points within the output buffer 2806 represents an activation map generated by the convolution operation. Each point within the activation map is generated by sliding the receptive field tile across the input volume buffer 2804. The activation map data can be input to an activation function to determine an output activation value. In one embodiment, convolution of the input volume buffer 2804 can be defined within a framework as high-level matrix operation 2905. The high-level matrix operations can be performed via primitive operations, such as a basic linear algebra subprogram (BLAS) operation. The primitive operations can be accelerated via hardware instructions executed by the instruction pipeline 2800.

[0365] The instruction pipeline 2800 used to accelerate hardware instructions can include the instruction fetch and decode unit 2721, which can fetch and decode hardware instructions, and the scheduler controller 2722 which can schedule decoded instructions to one or more processing resources within the compute blocks 2724A-2724N and/or the tensor accelerator 2723. In one embodiment, a hardware instruction can be scheduled to the compute blocks 2724A-2724N and offloaded to the tensor accelerator 2723. The one or more hardware instructions and associated data to perform the matrix operation 2805 can be stored in the memory 2830. Output of the hardware instruction can also be stored in the memory 2830.

[0366] In one embodiment, the tensor accelerator 2723 can execute one or more hardware instructions to perform the matrix operation 2805 using an integrated systolic array 2808 (DP logic). The systolic array 2808 can include a combination of programmable and fixed function

hardware that is configurable to perform dot product operations. While functional units within the compute blocks 2724A-2724N can also be configured to perform dot product operations, the systolic array 2808 can be configured to perform a limited subset of dot product operations at a significantly higher throughput relative to the compute block 2724A-2724N.

[0367] **FIG. 29A-29B** illustrate details of hardware-based systolic array 2808, according to some embodiments. FIG. 29A illustrates a grid of multiple functional units that are configurable to perform multiple dot product operations within a single clock cycle. FIG. 29B illustrates a single exemplary functional unit. During aa systolic matrix computation,

[0368] As shown in FIG. 29A, in one embodiment the systolic array 2808 is configurable to perform a set of parallel dot product operations using a variety of functional units. The dot products can be performed in a 'systolic' manner, in which SIMD data is pumped across multiple layers of functional units. As shown in FIG. 29A, in one embodiment the systolic array 2808 is configurable to perform a set of parallel dot product operations using a variety of functional units. The dot products can be performed in a 'systolic' manner, in which SIMD data is pumped across multiple layers of functional units. The systolic array 2808 is a collection of functional units that are arranged in a grid. The grid of functional units work in lockstep and are optimized to perform multiply-accumulate operations. Matrices to be operated on by the systolic array 2808 are divided in to sub-matrices, which are pumped across the grid of functional units.

[0369] In one embodiment the systolic array 2808 can process a configurable number of SIMD channels of data using a configurable systolic depth. For a given instruction, a SIMD width and a systolic depth can be selected to process a set of source data. The systolic depth defines the number of systolic layers of hardware logic that will be used to process an instruction. A systolic layer is a group of multiplier and adder logic units having a variable SIMD width, where the systolic layer can receive, as input, an initial accumulator value and generates a dot product value for output to a successive systolic layer or to an output register.

[0370] In some embodiments, three sources can be processed, where each source can be a vector register or an immediate. In one embodiment, source 2900 (SRC0) can be one or more initial accumulator values, which can be a single value or a vector of accumulator values. The initial accumulator value will be added to the first set of dot products computed by each functional unit within the first systolic layer. The dot product computed by a functional unit can be provided to the next systolic layer for the given SIMD channel. The dot products can be computed based on source 2901 (SRC1) and source 2902 (SRC2), which are vector registers that can contain one more channels of packed data, each channel containing a four-element vector. In one embodiment, each channel is 32-bits wide and provides four, 8-bit vector elements. Some embodiments are configurable to calculate dot products from input vectors having 8-bit

elements, 4-bit elements, and/or 2-bit elements. In one embodiment, mixed precision operations can be performed using any combination of supported element sizes (e.g., 8-bit x 2-bit, 8-bit x 4-bit, 4-bit x 4-bit, etc.). In one embodiment, the systolic array 2808 is configured for integer calculation, although automatic fixed-point operation is configurable in some embodiments. Although the instruction described herein is a four-element dot product, in some embodiments the systolic array 2808 may also be configured to support floating-point dot-product calculations on a different number of elements per vector.

[0371] In one embodiment, multiple channels of four-element vectors can be packed into a single vector register of various widths (e.g., 64-bit, 128-bit, 256-bit, 512-bit, etc.). Simultaneous dot products can be computed via the systolic array 2808 for multiple channels of vector elements provided via source 2901 and source 2902. The number of channels of vector elements to be processed can be configured based on a selected execution size and systolic depth for the dot product calculation. In one embodiment, source vectors that are wider than the specified execution size and/or systolic depth may be calculated using multiple cycles of the systolic array 2808.

[0372] The number of calculations that can be performed within a given clock cycle can vary based on the number of SIMD lanes and systolic layers. The systolic array 2808, as illustrated, can perform sixteen dot products per SIMD lane of throughput using a systolic depth of four. If configured for eight SIMD lanes, the logic can perform 128 eight-bit integer (INT8) dot products within a given cycle. If configured for eight SIMD lanes and a systolic depth of eight, each lane can perform 32 eight-bit integer (INT8) dot products and 256 dot products in total. These specific number of operations are exemplary of one embodiment, and other embodiments vary in throughput. Furthermore, if the data types are different, then the number of operations will be scaled based on the different data types.

[0373] At each functional unit, a dot product is computed via multiplier and adder logic and the dot product is added to an accumulator value. The resulting data can be output to a destination register or provide to the accumulator of the next systolic layer. Details of a functional unit 2912 are shown in FIG. 29B.

[0374] As shown in FIG. 29B a functional unit 2912 can include a set of input data buffers 2904, 2906 and an accumulator 2922, which can each accept input data. In one embodiment, input data buffer 2906 can accept source 2902, (SRC2), which can be a packed vector of input data. Input data buffer 2904 can accept a source 2901 (SRC1), which can also be a packed vector of input data. The accumulator 2922 can accept source 2900 (SRC0) that provides an initial accumulator value for the functional unit 2912. The initial accumulator value is added to the dot product computed from the elements of source 2901 and source 2902. The dot product is

computed via an element-wise multiplication of the source vectors using a set of multipliers 2923A-2923D and an adder 2924. The multipliers 2923A-2923D are used to compute a set of products. A sum of the set of products is computed by the adder 2924. The sum can be accumulated with (e.g., added to) any initial value provided via source 2900. In one embodiment, this accumulated value can be provided as an input value 2926 to the next accumulator, which can reside in a subsequent systolic layer. In one embodiment, source 2901 may include multiple channels of input data. Additional channels of source 2901 can be relayed as SRC1 input to additional SIMD lanes 2928. In one embodiment, source 2902 may include multiple channels of input data. Additional channels of source 2902 can be used as SRC2 input data to logic units within additional systolic depths. In one embodiment, source 2900 can optionally include multiple channels, with additional channels provided as input to the accumulator within additional functional units. In one embodiment, source 2900 can be a single value that is added to each accumulator in each functional unit of the initial systolic layer.

[0375] **FIG. 30** illustrates a systolic array 3000 including partial sum loopback and circuitry to accelerate sparse matrix multiply. In the systolic array 2808 described above, operands that include weight data may be stationary within the array and a partial sum is propagated throughout the array structure. While other details with respect to systolic array 2808 may be applicable, in systolic array 3000 a partial sum is recirculated instead of being propagated to a next systolic layer. In one embodiment a systolic array 3000 can be configured with M rows and N columns of processing elements (PE 3012AA-PE 3012MN). The processing elements can access registers storing input data in the form of row and column data for input matrices. The registers may be stored in a register file that is local to the systolic array 3000 or in a register file of a processing resource that is coupled with, or includes the systolic array 3000. The registers may store row elements of matrix A 3002A-3002M, which are to be multiplied by column elements of matrix B 3001A-3002N.

[0376] In one embodiment a fused multiply-add (FMA) can be performed at each processing element PE 3012AA-PE 3012MN each clock cycle. An element of matrix A is multiplied by a corresponding element of matrix B and then added to an accumulator value or, for the first cycle, an optional initial input value (e.g., SRC0). Partial sum loopback can be configured at each processing element. After each cycle, the accumulator value may be looped back within the processing element and used as input for the next cycle. Once operations are performed for an entire row, the result may be stored to a register file. Data movement between the processing elements PE 3012AA-PE 3012MN after a set of computational cycles can vary based on the instruction or macro-operation being performed.

Data Aware Sparsity with Compression

[0377] Embodiments described herein provide an encoding layout that enables sample blocks of sparse neural network data to be encoded in a reduced-bit formal that reduces the amount of data that is required to be transmitted or stored when processing neural networks associated with the data. The number of non-zero values in a sample block is indicated in a header, followed by a significance map indicating a map of the non-zero values within the block. The non-zero values of the sample are encoded in order of appearance within the stream. In one embodiment, compression can be based on other values beyond zero values. For example, a specified value within a data set may be encoded and excluded from a compressed data stream, enabling compression based on ones, twos, or other specified values. In one embodiment compression is enabled based on near values. Values within a data set that are within a threshold of zero, or within a threshold of a specified value, may be compressed as though those values were zero or within a threshold of the specified value. Data aware sparsity with compression can be enabled via codec logic coupled with or within matrix accelerator logic.

[0378] **FIG. 31A-31B** illustrate matrix acceleration logic including codecs to enable the reading of sparse data in a compressed format. FIG. 31A illustrates a compute block 3100 including codec enabled disaggregated systolic logic. FIG. 31B illustrates processing elements within a systolic array that are coupled with codecs to decompress input data.

[0379] As shown in FIG. 31A, instead of including a systolic array 2808 in a separate tensor accelerator 2723, as in FIG. 28, or including a systolic array 1912 in each execution unit 1900 as in FIG. 19, a disaggregated set of systolic arrays 3112A-3112B can be included in a compute block 3100 that is analogous to one of the compute blocks 2724A-2724N of FIG. 27. The compute block 3100 can also include components of execution logic 1800 of FIG. 18A, including multiple interconnected processing resources (PR 3108A-3108O) that may be similar to EU 1808A-1808N or any other processing resource as described herein. In one embodiment the systolic arrays 3112A-3112B include codecs 3124A-3124B that enable the encoding and decoding of input and output data that is received for processing.

[0380] The systolic arrays 3112A-3112B include a *W* wide and *D* deep network of data processing units that can be used to perform vector or other data-parallel operations in a systolic manner, similar to other systolic arrays described herein. In one embodiment the systolic arrays 3112A-3112B can be configured to perform matrix operations, such as matrix dot product operations. In one embodiment the systolic arrays 3112A-3112B support 16-bit floating point operations, as well as 8-bit and 4-bit integer operations. In one embodiment the systolic array 3112 can be configured to accelerate machine learning operations. In such embodiments, the systolic array 3112 can be configured with support for the bfloat 16-bit floating point format. By

including systolic arrays 3112A-3112B within the compute block 3100 but outside of the PRs 3108A-3108O, the size and number of systolic arrays 3112A-3112B can be scaled independently from the number of PRs 3108A-3108O. Additionally, communication bandwidth within an PR that would otherwise be consumed by systolic array activity may be preserved. Furthermore, the systolic arrays 3112A-3112B may be clock/power gated when matrix workloads are not being performed.

[0381] Communication between the systolic arrays 3112A-3112 and the PRs 3108A-3108O may be performed via a cache or shared local memory (cache/SLM 3110) and/or a shared register file 3114. In one embodiment, instead of a distinct shared register file 3114, the cache/SLM 3110 may be partitioned for use as a shared register file. The shared register file 3114 may be structured similarly to other GPGPU register files, such as register file 1906 as in FIG. 19. The shared register file may also include a set of special purpose registers that are used to configure the interaction between the systolic arrays 3112A-3112B and the PRs 3108A-3108O. The cache/SLM 3110 may be an L1 cache, an L2 cache, and/or a block of explicitly addressable on-die memory.

[0382] Matrix data for processing by the systolic arrays 3112A-3112B may be stored in the cache/SLM 3110. Processing commands or instructions can be provided to the systolic arrays 3112A-3112B via the shared register file 3114. Processing results may be read from the cache/SLM 3110 by the PRs 3108A-3108O or from destination/output registers within the shared register file. During operation, instead of consuming bus/fabric bandwidth within the PRs 3108A-3108O, communication traffic may be localized to the systolic arrays 3112A-3112B, the cache/SLM 3110, and/or shared register file 3114. Any of the PRs 3108A-3108O within the compute block 3100 may offload a matrix workload to one or both of the systolic arrays 3112A-3112B. A message may be sent from a PR to a systolic array with a command that specifies an operation to be performed and operands for the operation. The systolic arrays 3112A-3112B can perform the requested operations (multiply/add, fused multiply/add, multiply/accumulate, dot product, etc.) and output the results to the shared register file 3114. Input, intermediate and/or output data for requested operations may be stored in the cache/SLM 3110 and multiple dependent operations may be chained. In one embodiment when processing operations for training or inference for a neural network are performed, the systolic arrays 3128A-3128B may also perform activation functions including but not limited to sigmoid, ReLU, and hyperbolic tangent (TanH) activations. In such embodiment, operations for neural networks may be offloaded to the systolic arrays 3112A-3112B at coarse granularity.

[0383] The PRs 3108A-3108O can provide input data to the systolic arrays 3112A-3112B in a compressed format and the codecs 3124A-3124B can be used to decompress the data. When

output data is ready to be provided to the PRs 3108A-3108O, the data may remain decompressed if the PRs will perform operations and the data and do not support the direct read of compressed data. If the PRs 3108A-3108O support the reading of compressed data or will not perform additional operations on the data, the output data may be re-encoded. Zero-based encoding may be used and compression may be enabled or disabled based on the degree of data sparsity. Alternatively, other forms of encoding may be used based on the distribution of the data set to be processed or output. For example, the codecs 3124A-3124B can be configured to decode sparse data that is encoded based on zero-based compression, or using another form of compression described herein (e.g., one-based, two-based, near-zero, near-one, near-two, etc.).

[0384] As shown in FIG. 31B, system 3150 illustrates processing elements of systolic array 3000, where the systolic array is configured to decode compressed sparse data. As described with respect to FIG. 30, each PE 3012AA-3013MN includes hardware logic to perform computations for matrix operations. A (A0, A1, through A_M) and B (B0, B1, through B_N) are elements of input matrices with associated with dot product, matrix multiply, multiply/add, or multiply accumulate operations. In one embodiment each PE 3012AA-3013MN is associated with codecs (3151a, 3151b, …,3151m; 3152a, 3152b, …,3152n) to decode compressed input operands associated with operations to be performed. The codecs can be configured to decode sparse data that is encoded based on zero-based compression, or using another form of compression described herein.

[0385] Sparse neural network data can be encoded (e.g., compressed) using a variety of encoding techniques, such as but not limited to unique absolute value (UAV) table encoding, significance map (SM) encoding, table encoding (TE), unique value coordinate (UVC) encoding, and mean encoding (ME). Metadata for the encoded data indicates the type of encoding format used for the data. In one embodiment, specific encoding formats can be selected for specific types of data, such as kernel data or feature data. In one embodiment, statistical analysis is performed on the data prior to encoding to enable an appropriate encoder to be selected for each block of data. The encoding may be zero-based encoding, near-zero encoding or based on other values (ones, twos, etc.).

[0386] In one embodiment data generated during SM encoding can be used to facilitate provision of compressed data to a systolic tensor array. In zero-based SM encoding mode, only non-zero values in a block are encoded. The number of non-zero values in a sample block is indicated in the header, followed by a significance map indicating a map of the non-zero values within the block. The non-zero values of the sample are then encoded in order of appearance within the stream.

[0387] **FIG. 32A-32B** illustrates zero-based SM encoding for sparse neural network data, according to embodiments. FIG. 32A illustrates an encoding layout 3220 for zero-based SM encoding. FIG. 32B illustrates decode of an exemplary SM encoded bit stream.

[0388] As shown in FIG. 32A, an encoding layout 3220 enables sample blocks of sparse neural network data to encoded in a reduced-bit formal that reduces the amount of data that is required to be transmitted or stored when processing neural networks associated with the data. In the illustrated encoding layout 3220, the number of non-zero values in a sample block is indicated in a header 3222, followed by a significance map 3224 indicating a map of the non-zero values within the block. The non-zero values 3226 of the sample are encoded in order of appearance within the stream.

[0389] As shown in FIG. 32B, encoded bitstream data can be decoded based on significance map data. In one embodiment, SM encoding mode data is presented beginning with the third byte after a two-byte header, where the presence of SM encoding is indicated by the first bit of a bit stream header (not shown) at the beginning of an encoded stream of data. The number of non-zero values in a sample block is indicated in the header 3222. In one embodiment, encoding format can change on a per-sample block basis, so the header 3222 can also include metadata that indicates SM encoding is enabled for the upcoming block of samples. A significance map 3224 indicates a map of the non-zero values within the sample block, with a one-bit entry associated with each value. The non-zero values 3226 of the sample are then encoded in order of appearance within the stream. To decode significance map data into the exemplary decoded bit stream 3230, decoder logic can initialize at least a portion of an output data buffer 3210 to zero. The decoder logic can then reference the significance map 3224 to determine which value in the bit stream are non-zero. The non-zero values can be unpacked sequentially and written to locations in the output data buffer 3210 indicated by the significance map. For example, a value of zero (0b0) in the significance map 3224 indicates that the corresponding decoded value is zero. A value of one (0b1) in the significance map 3224 indicates that the corresponding decoded value corresponds to the next successive entry in the non-zero values 3226 in the encoded bit stream.

[0390] In one embodiment, lossy near-zero encoding can be enabled that encodes values that are close to zero as though those values were zero. While some precision is lost, this loss of precision may be acceptable under some circumstances for inference data. Lossy near-zero encoding can be used as an alternative to explicit weight pruning operations that may be used to induce sparsity within post-training data.

[0391] **FIG. 33A-33B** illustrates specified-value SM encoding for sparse neural network data, according to embodiments. FIG. 33A illustrates an encoding layout 3320 for specified-

value SM encoding. FIG. 33B illustrates decode of an exemplary specified-value SM encoded bit stream.

[0392] As shown in FIG. 33A, an encoding layout 3320 similar to encoding layout 3220 may be used, with a different header 3322 that indicates specified-value significance map encoding is enabled, and a specified value 3323 that specifies the value that will be skipped when encoding the bitstream. The significance map 3324 identifies the locations within an output buffer in which the data within the residual values 3326 should be unpacked.

[0393] As shown in FIG. 33B, encoded bitstream data can be decoded based on significance map data in a similar manner as in FIG. 32B. The header 3322 includes a bitfield that specifies that specified-value significance map is enabled. The specified value 3323 identifies the value that will be skipped by the significance map 3324. The number of residual values 3326 in a sample block is indicated in the header 3322. In one embodiment, encoding format can change on a per-sample block basis, so the header 3322 can also include metadata that indicates specified-value SM encoding is enabled for the upcoming block of samples and the specified value 3323 that will be used. The significance map 3324 indicates the map of the residual values within the sample block, with a one-bit entry associated with each residual value. The residual values 3326 of the sample are then encoded in order of appearance within the stream. To decode specified-value significance map data into the exemplary decoded bit stream 3330, decoder logic can initialize at least a portion of an output data buffer 3310 to the specified value 3323. The decoder logic can then reference the significance map 3324 to determine which value in the bit stream are residual. The residual values can be unpacked sequentially and written to locations in the output data buffer 3310 indicated by the significance map. For example, a value of zero (0b0) in the significance map 3324 indicates that the corresponding decoded value is the specified value 3323. A value of one (0b1) in the significance map 3324 indicates that the corresponding decoded value corresponds to the next successive entry in the residual values 3326 in the encoded bit stream.

[0394] In one embodiment, lossy specified-value encoding can be enabled that encodes values that are close to the specified value as though those values were equal to the specified value. While some precision is lost, this loss of precision may be acceptable under some circumstances for inference data.

[0395] Using the techniques above, one skilled in the art may implement, for example, a general-purpose graphics processing unit comprising a matrix accelerator configured to accelerate sparse matrix operations. The matrix accelerator can include a hardware decoder to decode input data including multiple matrix elements. The input data may be received in a compressed format and decoded by the hardware decoder before being processed by the matrix

accelerator. In one embodiment the matrix accelerator includes multiple processing elements which, in one embodiment, are configured as a systolic array. The multiple processing elements of the systolic array may include decoders to decode input data on a per-processing element basis, or for a group of processing elements that is less than the entire systolic array.

[0396] In one embodiment the data that is received in a compressed format that is encoded based on zero-based compression. In one embodiment the compressed data is encoded in a near-zero format in which values within a threshold of zero are set to a zero value. In one embodiment the data is received in a compressed format that is encoded based on specified-value compression. In one embodiment the compressed data is encoded in a near-specified-value format in which values within a threshold of the specified value are set to the specified value.

Block Sparse Dot Product Operations

[0397] Various techniques are applied in the art of deep neural networks (DNNs) to improve computational speed and/or efficiency while maintaining acceptable accuracy of the results. It has been determined that for some types of DNN models, the computational precision of the operations may be reduced without significantly reducing the accuracy of the model. In addition to reducing computation precision, for example, by training using 16-bit floating point values instead of 36-bit floating-point values or performing inference operations using integer values, training techniques can be adapted to introduce sparsity into the weight values associated with a DNN model. Sparsity refers to the proportion of weights of a DNN that have zero values. Higher sparsity corresponds to fewer weights, which results in smaller computational and storage requirements. With zero valued weights, any multiplications performed using those weights may be bypassed and models can be stored and transmitted compactly using sparse matrix formats. Empirical results show that DNNs can tolerate high levels of sparsity minimal or acceptable reduction in accuracy. Given this trend, it would be beneficial to adapt GPGPU and parallel processor hardware that is used to perform training and inference operations for DNN models.

[0398] **FIG. 34A-34B** illustrates comparisons between unstructured sparsity and block sparsity within training data for a neural network. FIG. 34A illustrates a comparison between unstructured sparsity and block sparsity. FIG. 34B illustrates block pruning of weights for a neural network.

[0399] As shown in FIG. 34A, sparsity in DNN models may take the form of irregular sparsity, as in matrix 3402, or block sparsity, as in matrix 3404. Irregular sparsity as in matrix 3402 may occur naturally during training or pruning techniques may be applied to zero-out weight values that are below a minimum threshold. Block sparsity as in matrix 3404 may be induced by pruning on a block-by-block basis. A block size can be defined and blocks below a minimum absolute magnitude may be pruned.

[0400] As shown in FIG. 34B, weight pruning may be performed after a DNN model is trained by summing the weight values within a block and setting all weights in the block to zero if the absolute magnitude of the weights is below a threshold. Matrix 3412 may be pruned into a block sparse matrix 3414 by using, for example, a weight threshold of 0.4 and a 4x4 block. For each 4x4 block, the weights in the block may be summed and any block having an absolute magnitude that is below the threshold may have its weights set to zero. Matrix multiplication operations involving those blocks may be bypassed during inference.

[0401] In one embodiment, block sparse operations on a systolic array may be optimized by enabling a block sparse macro instructions to perform dot product operations on block sparse data. A sequence of dot product instructions (dp4a, dp2a) may be generated internally within a GPGPU in response to receipt of a block sparse macro instruction (dpXaV). In one embodiment a bitfield is generated that indicates the degree of sparsity for rows of a matrix A to be provided as input to the block sparse macro instruction. The macro instruction can use the bitfield to automatically generate a set of instructions and operands to perform successive dot product operations for elements of the matrix. The dot product instruction can be used to generate a scalar product of row elements of matrix A and column elements of matrix B. Where all input values of matrix A for an instruction will be zero, the dot product instruction for that row can be skipped. As the result will be zero. When one or more matrix A elements to be processed are zero, but not all elements, a predication mask can be used to disable multiplication for the specific zero-value elements.

[0402] **FIG. 35A-35B** illustrate a dot product instruction 3500 executable by dot product logic. FIG. 35A illustrates fields of a four-element dot product with accumulation instruction executable by systolic matrix logic provided by an embodiment. FIG. 35B illustrates element wise calculations performed during calculation of the four-element dot product with accumulation using the systolic matrix logic.

[0403] As shown in FIG. 35A, a four-element dot product with accumulate instruction 3500 can be used to cause a systolic matrix accelerator to execute a four-element dot product with accumulate. In one embodiment, the instruction 3500 includes a predicate field 3502, an opcode field 3504, a conditional modifier field 3506, an execution size field 3508, and operand fields to specify a destination 3510, zeroth source 3512 (src0), first source 3514 (src1), second source 3516 (src2), and systolic depth 3518 (sdepth). The predicate field 3502 can specify a predicate mask 3562, as in FIG. 35B. Turning to FIG. 35B, the predicate mask can be used to enable or disable an execution channel of the instruction, such output generation for a specific element is disabled. In various embodiments, a disabled channel may bypass execution, or may execute while preventing the execution logic from modifying the architectural state associated with the

disabled channel. In one embodiment, output of the execution may be discarded without being committed. When all elements of the predicate mask 3562 are zero, all operations for the instruction may be bypassed. When a dp4aV vector instruction is being used to generate multiple dp4as instructions, generation of a dp4as instruction can be bypassed when the predicate mask 3562 is zero.

[0404] Returning to FIG. 35A, the opcode field 3504 can specify an opcode that identifies the instruction 3500 to execution logic. In one embodiment the opcode field 3504 includes one or more bits that, when enabled, indicate that the instruction is to be executed by a matrix accelerator as described herein (e.g., tensor accelerator 2723). In one embodiment, the opcode field 3504 can also include one or more bits that specify that the instruction 3500 is to be executed by special purpose dot product logic, such as dot product logic (e.g., dot product logic 1608) within a matrix accelerator.

[0405] The conditional modifier field 3506 can be used to set a flag register based on the internal conditional signals output from the execution pipe, such as sign, zero, overflow, not-a-number (NaN), etc. If this field is set to 0000, no flag registers are updated. In one embodiment, supported conditionals include, but are not limited to, Zero or Equal, Not Zero or Not Equal, Greater-than, Greater-than-or-equal, Less-than, Less-than-or-equal, Overflow, and Unordered with Computed NaN. Furthermore, in some configurations, a source modifier field may be enabled.

[0406] The execution size 3508 (exec_size) specifies the number of execution channels for the instruction 3500. In the case of a four-element dot product instruction, in one embodiment the number of execution channels can be used to specify a number of pairs of four element vectors over which the calculation will be performed.

[0407] The destination 3510, zeroth source 3512 (src0), first source 3514 (src1), second source 3516 (src2) can be used to specify a destination to which a calculation is written and a location from which source data can be retrieved. In one embodiment the destination 3510 can specify a register to which data is to be written. In one embodiment the destination 3510 can be a scalar register, although in some embodiments the destination can also be a vector register that stores output from multiple channels. For example, where a single, four-element channel is specified, a single dot product can be written to a single output channel at the destination. Where multiple four-element channels are specified, multiple dot products can be written to a vector destination register. The zeroth source 3512, first source 3514, and second source 3516 can be register or immediate values that include one or more channels of source data, each channel having four elements to be processed by dot product logic, such as with source 2900, source 2901, and source 2902 as in FIG. 29A-29B. In one embodiment the four elements for each

source may be provided to a processing element of systolic array 3000 as in FIG. 30. The systolic depth 3518 (sdepth) can be used to specify the number systolic layers to use to process the input data. In one embodiment the systolic depth 3518 can be provided as an immediate value.

[0408] In some embodiments, additional fields other than those illustrated may be present. For example, in one embodiment a source modifier field is present which specifies the numeric modification of a source operand. The value of each data element of a source operand can optionally have its absolute value taken and/or its sign inverted prior to delivery to the execution pipeline. The absolute value modifier can be applied prior to the negate modifier, such that a guaranteed negative value can be produced. In one embodiment, a saturation field is present, which can be used to control destination saturation. When saturation is enabled, output data to the destination register is saturated. The specific saturation operation depends on the destination data type. Saturation is an operation that converts any data that is outside the saturation target range for the data type to the closest represented value with the target range.

[0409] Element wise operations of the instruction 3500 are shown in FIG. 35B. In one embodiment, a hardware buffer, temporary register, or internal register of a functional unit within dot product logic of systolic matrix accelerator can store elements A3-A0 of a channel 3524 of the first source 3514. An additional hardware buffer or temporary register can store elements B3-B0 of a channel 3526 of the second source 3516. The elements can be loaded into a functional unit (e.g., functional unit 2912 of FIG. 29A) of dot product logic within a systolic tensor array described herein. A set of multipliers (e.g., multipliers 2923A-2923D of FIG. 29B) can generate a set of products 3534. The set of products can be loaded into an adder (e.g., adder 2924 of FIG. 29B).

[0410] The adder can generate a sum of the set of products 3535, which can be accumulated with an existing accumulator value 3522 (e.g., C0). The existing accumulator value 3522 can be an initial accumulator value provided via the instruction 3500 or can be an accumulator value provided by a previous systolic layer. The accumulated sum of the set of products 3535 can then be output to a destination 3538, which can be the destination 3510 specified by the instruction 3500 or can be loaded as the initial accumulator value 3536 of the next systolic layer. The destination for the accumulated sum output by a systolic layer can be determined based on whether the given systolic layer is the last layer in the chain, where the number of layers is specified by the systolic depth.

[0411] **FIG. 36A-36B** illustrate an instruction 3600 executable by dot product logic, according to embodiments described herein. FIG. 36A illustrates fields of a two-element dot product with accumulation instruction executable by systolic matrix logic provided by an

embodiment. FIG. 36B illustrates element wise calculations performed during calculation of the two-element dot product with accumulation using the systolic matrix logic.

[0412] FIG. 36A illustrates fields of a two-element dot product with accumulate instruction 3600, which, when executed, causes a systolic matrix accelerator to execute a two-element dot product with accumulate. In one embodiment, the instruction 3600 includes a predicate field 3602, an opcode field 3604, a conditional modifier field 3606, an execution size field 3608, and operand fields to specify a destination 3610, zeroth source 3612 (src0), first source 3614 (src1), second source 3616 (src2), and systolic depth 3618 (sdepth). Each of the illustrated fields may be similar to those of instruction 3500, excepting that the opcode field 3604 specifies a two-element dot product and each operand specifies a vector having two packed elements instead of four.

[0413] The predicate field 3602 can specify a predicate mask 3662 of FIG. 36B. Turning to FIG. 36B, the predicate mask 3662 can be used to enable or disable an execution channel of the instruction, such output generation for a specific element is disabled, as with predicate field 3502 and predicate mask 3562. The illustrated predicate mask 3662 indicates that element A0 is non-zero, while element A1 is zero. For this set of matrix A inputs, an instruction can be generated to perform the dot product operation and multiply A0 of input channel 3624 times B0 of channel 3626. A set of products 3634 including only A0*B0 can be generated, which may then be multiplied by an existing accumulator value 3622 (e.g., C0). The accumulated sum of the set of products 3635 can then be output to a destination 3638, which can be the destination 3610 specified by the instruction 3600 or can be loaded as the initial accumulator value 3636 of the next systolic layer. The destination for the accumulated sum output by a systolic layer can be determined based on whether the given systolic layer is the last layer in the chain, where the number of layers is specified by the systolic depth.

[0414] **FIG. 37A-37D** illustrate an instruction and associated hardware logic to enable a sparse dot product operation with zero skipping for sparse input. Fig. 37A illustrates a format for a sparse dot product instruction 3700. Fig. 37B illustrates hardware decode logic 3710 to enable decode of the sparse dot product instruction 3700. Fig. 37C illustrates execution logic 3720 configured to enable decode and execution of the sparse dot product instruction 3700 described herein. FIG. 37D illustrates a method 3750 performed by hardware configured to execute a sparse dot product instruction.

[0415] As shown in Fig. 37A, one embodiment provides hardware support for a sparse dot product instruction (dpXaV instruction 3700) performs a sequence of dot product instructions with automatic zero skipping for one or more input vectors. The dpXaV instruction is representative of a four element sparse dot product instruction (dp4aV) and a two element sparse

dot product instruction (dp2aV). The format for the dpXaV instruction 3700 includes a predicate (pred 3702), the instruction opcode with an appended repeat count 3704 (dpXaV.rcount), a destination operand (dst 3706), and multiple source operands (src0 3707, src1 3708, src2 3709). In one embodiment, the predicate value is determined based on the flag register, where the bits of the predicate mask to apply predication over an array of individual dot product instructions, rather than an array of channels. Accordingly, the predicate mask can be used to enable or disable the generation of specific instructions in the set of instructions represented by the dpXaV instruction 3700. For a given generated instruction, element wise predication can be used to disable operations for a specific matrix A element.

[0416] In one embodiment the dpXaV instruction 3700 is implemented as a hardware macro instruction that is expanded into multiple additional instructions during decode. In such embodiment, the repeat count 3704 value indicates the number of instructions to be expanded into once inside the execution resource. For example, the dp4aV.8 instruction shown in **Table 5** below may be expanded within an execution resource into multiple dot product operations.

Table 5 – Exemplary Sparse Dot Product instruction

dp4aV.8 r34.0<1>:f r34.0<1,0>:f r10.0<1,0>:f r76.0<1>:f

[0417] For the instruction in Table 5 above, the repeat count 3704 is eight. The supplied operands represent the initial operands for the first instruction in the set of expanded instructions. The operands for the subsequent instructions are generated during expansion. The operand expansion makes use of a block of consecutive registers for the <dst> 3706 and <src0> 3707 and consecutive subregisters for src2 3709, with the number of consecutive registers/subregisters determined based on the value of the repeat count 3704. Based on the value of the flag register, expansion for specific instructions in the set of instructions can be bypassed, for example, in the instance that zero skipping is performed for sparse input.

[0418] As shown in Fig. 37B, instruction expansion is performed during decode of the dpXaV instruction in one embodiment, although other implementation techniques may be used in other embodiments, such as via direct execution of the instruction by the execution stage of the instruction execution pipeline. In one embodiment, instruction decode logic 3710 within a processing resource can include, in addition to conventional instruction decode logic, a counter/state machine 3711, a priority encoder 3712, an instruction generator 3713, and an operand generator 3714. The counter/state machine 3711 stores counter values and decode state for use during expansion of a dpXaV instruction. The priority encode 3712 is used to scan the

predicate to determine the first activated bit, which indicates the first dot product instruction to be generated.

[0419] The proposed dpXaV instruction can be implemented in the execution unit by adding a counter/statemachine in the Decode stage of ALU pipe and generating a sequence of dot product instructions. A priority encoder 3712 scans the predicate mask to find the first set bit. Instructions are produced by the instruction generator 3713 if the corresponding bit for that instruction is set in the in the predicate bitfield. The operand generator 3714 then generates the corresponding operand values based on the initial operand register and subregister values and on the bit positions that are set within the predicate mask.

[0420] As shown in Fig. 37C, execution logic 3720 within a multi-threaded processing resource can be configured to expand and execute a dpXaV instruction as described herein. The execution logic 3720 can be found within a multi-threaded processing resource, such as an execution unit, compute unit, or streaming multiprocessor as described herein. In one embodiment, a set of instruction queues 3722A-3722N can store N sets of instructions that are pending execution by N threads of the execution logic 3720. Each thread has an associated pre-decode stage 3724A-3724N that stalls execution for the thread until dependencies for the current instruction are cleared. A thread is marked as ready when no dependencies for the current instruction remain. Once cleared, a thread may participate in the arbitration process that selects a thread for dispatch into the execution pipeline of one of ALUs.

[0421] The respective arbiters 3726A-3726M for each ALU can select one or more ready threads based on the availability of execution resources within an ALU. The instructions for the selected threads are then dispatched into the execution pipeline of the selected ALU. The execution pipeline stages for the illustrated ALUs include decodes stage 3728A-3728M, register read stages 3730A-3730M, execution stages 3732A-3732M, and write-back (WB) stages 3734A-3734M. In one embodiment a single dpXaV instruction can be expanded into multiple dot product instructions within a decode stage 3728A-3728M of the ALU that receives the instruction.

[0422] In one embodiment the number of ALUs within the execution logic 3720 is smaller than the number of threads that may be executed by the execution logic 3720. In one embodiment, multiple instructions may be issued to a single ALU. In one embodiment, only a subset of available ALUs may be configured for dpXaV instruction expansion and such instructions are executed by those specific ALUs. The total number of clocks the execution logic 3720 consumes to perform a dpXaV instruction depends on the number of clocks per dot product instruction, the repeat count, and the number of skipped instructions.

[0423] FIG. 37D shows a method 3750 to facilitate execution of a sparse dot product instruction. The method 3750 can be performed by logic of a compute or graphics processor pipeline of a graphics processor or parallel processor described herein. For example, the method 3750 can be performed by execution logic 3720 of FIG. 37C using instruction decode logic 3710 of FIG. 37B. The method 3750 can be used to facilitate execution of a hardware macro instruction, such as the dpXaV instruction 3700 of FIG. 37A, and/or as described herein.

[0424] In one embodiment, the method 3750 includes performing, via hardware logic of the graphics or parallel processor, an operation to fetch a hardware macro instruction (3751). The hardware macro instruction has a predicate mask, a repeat count, and a set of initial operands (<dst>, <src0>, <src1>, <src2>). The hardware macro instruction is configured to perform one or more dot product operations on input data associated with a set of matrices. The method 3750 additionally includes performing operations to decode the hardware macro instruction (3752). In one embodiment, the fetched hardware macro instruction can be stored in an instruction queue until logic of a pre-decode stage of the instruction execution pipeline (e.g., pre-decode 3724A-3724N) determines that dependencies, such as associated load or compare instructions, are satisfied. Once the dependencies are satisfied, the instruction can be automatically submitted for decode by decode logic of a processing resource described herein.

[0425] The decode operations can include or be performed at least in part to determine a set of hardware instructions to generate for execution. For example, a dot product instruction can be generated for each activated bit in the predicate mask. The predicate mask can be generated based on input stored in subregisters that correspond with the matrix A input operand, which may be, for example, the <src1> input. The specific inputs that may correspond with specific matrices may vary based in implementation. For example, a compare instruction can be used to set bits within a flag register by comparing the data to be used as the <src1> input with zero. The flag register can then be used as the predicate mask. The predicate mask and the repeat count can then be used to generate the set of hardware instructions for execution (3753). Instructions can be generated by an instruction generator 3713 as in FIG. 37B. The maximum number of instructions to generate can be determined based on the repeat count. Generation of specific hardware instructions is bypassed based when disabled by an associated bit within the predicate mask.

[0426] The specific operands to be used by the generated instructions in the set of instructions can be determined based on the predicate mask and the set of initial operands (3754). Operands for the instructions can be generated by an operand generator 3714 as in FIG. 37B. Successive contiguous registers are used for <dst> and <src0> operands and successive subregisters being used for <src1> operands. Operands are not generated for bypassed

instructions. The instruction execution pipeline of the graphics or parallel processor can then atomically execute the set of hardware instructions (3755). Atomic execution indicates that one or more operations performed during execution of the set of instructions is performed atomically, such that the instructions in the set of instructions are considered to be a single instruction. The specific number of clock cycles consumed by execution of the hardware macro instruction can vary based on the cycle count to execute a single multiply-add operation, the repeat count associated with the instruction, and the number of instructions or operations that are bypassed due to sparse input data. The instruction execution pipeline of the graphics or parallel processor can then retire the hardware macro instruction upon completion of each instruction in the set of hardware instructions (3756).

[0427] It will be noted that, depending on the sparseness of the input to a dot product instruction, the number of execution cycles may vary. Thus, the latency for the instruction may not be able to be determined at compile time to enable the scheduling of the next dependent instruction. Accordingly, the multiple instructions or operations that are performed by a dot product instruction are considered to be a single atomic instruction by the compiler and scheduling hardware. Thus, for a set of consecutive dependent dot product instructions, the instructions are executed in-order, such that a first dot product instruction is completed before a subsequent dependent instruction enters the execution pipeline.

[0428] Using the techniques above, one skilled in the art may implement, for example, a general-purpose graphics processing unit comprising a graphics processor comprising an instruction fetcher to fetch a hardware macro instruction having a predicate mask, a repeat count, and a set of initial operands, wherein the hardware macro instruction is to perform one or more dot product operations on input associated with a set of matrices. The dot product operations may be two element dot product operations of four element dot product operations. The general-purpose graphics processing unit additionally includes an instruction decoder to decode the hardware macro instruction and generate a set of hardware instructions to perform one or more multiply and add operations, the set of hardware instructions generated based on the predicate mask and the repeat count. The general-purpose graphics processing unit additionally includes a processing resource to execute the set of hardware instructions. The hardware macro instruction may be retired upon completion of the set of hardware instructions.

[0429] Using the techniques above, one skilled in the art may implement, for example, a general-purpose graphics processing unit configured to perform a method comprising fetching a hardware macro instruction having a predicate mask, a repeat count, and a set of initial operands, where the initial operands include a destination operand and multiple source operands. The repeat count is to indicate a maximum number of hardware instructions to generate for the set of

hardware instructions. The predicate mask indicate which instructions in the set of instructions are to be generated or bypassed. The predicate mask can be generated via a comparison between zero and each element associated with a source input operand. The hardware macro instruction is configured to perform one or more dot product operations on input data associated with a set of matrices.

[0430] Using the techniques above, one skilled in the art may implement, for example, a graphics processing system including processing logic to perform the above indicated method. A graphics processing system may also be implemented that include a memory device coupled with a general-purpose graphics processing unit as described above.

Depthwise Adapter for Systolic Array

[0431] The systolic arrays described herein suffer reduced efficiency when depthwise convolution. One embodiment provides an adapter circuit to perform image to column transformation on activation data before providing the activation data to the systolic array.

[0432] **FIG. 38A-38B** contrasts 3D convolution with depthwise convolution. Fig. 38A illustrates 3D convolution. FIG. 38B illustrates depthwise convolution. 3D depthwise convolution can be performed in a similar manner as 2D convolution, which is shown in FIG. 28, excepting that the input activations and weights/convolution kernel are 3D matrices. 3D convolution may be performed, for example for RGB images having red, green, and blue color channels.

[0433] As shown in FIG. 38A, given an input image 3802 of 12x12 pixel and three color channels, a 5x5x3 convolution operation 3803 can be performed on the input image 3802 using a 5x5 pixel, 3 channel convolution kernel 3804. The output 3805 of the 5x5x3 convolution operation 3803 on the 12x12x3 input image 3802 is an 8x8x1 image. If desired, additional convolution operations can be performed with different kernels to generate an output image with a greater number of channels. For example, to expand the number of output channels to 256, 256 separate 5x5x3 kernels can be used to generate 256 8x8x1 output images, which can be combined into 8x8x256 output image. For each step of the 5x5x3 convolution operation 3803, a minimum of 5x5x4=75 multiplication operations are performed. To generate the 8x8x1 output 3805, 8x8=64 steps are performed, for a minimum 64x75=4,800 multiplications. To expand the output to a 256 channel output image, 256x4800=1,228,800 multiplications are performed.

[0434] As shown in FIG. 38B, a depthwise convolution can be performed to obtain the same output image with a reduced number of calculations, particularly when output channel expansion is to be performed. To perform depthwise convolution given the input image 3802 of 12x12 pixels and three color channels, the input image 3802 can be divided into three 12x12x1 input

images 3812a-3812c. The 5x5x5 kernel 3804 can be divided into three 5x5x1 kernels 3814a-3814c. A depthwise convolution operation 3813 can then be performed in which each 5x5x1 kernels iterates one channel of the input images to generate three 8x8x1 output images 3815a-3815c. The output images 3815a-3815c can then be used as input into a pointwise convolution operation 3823 using a 1x1x3 kernel 3824 to generate the same 8x8x1 output 3805 as standard convolution. Output channel expansion can be performed during the pointwise convolution stage, where an 8x8x256 output image can be generated using 256 1x1x3 kernels to generate an 8x8x256 output image. To generate the 8x8x1 output image 3805 using depthwise convolution uses a minimum of 4,800 multiplications in the depthwise convolution stage, plus an additional 192 multiplications in the pointwise convolution stage. However, to perform output expansion to 256 channels, a total of 53,952 multiplications are performed, as opposed to 1,228,800 multiplications for standard convolution. Thus, it would be advantageous for a matrix unit or tensor array that is used to accelerate CNN operations to provide efficient support for depthwise convolution.

[0435] **FIG. 39A-39C** illustrate a system 3900 including a systolic tensor array 2808 and an image to column adapter 3910 for use during depth-wise convolution. As shown in FIG. 39A, in one embodiment the system 3900 includes a systolic array 2808 as in FIG. 28 and a general register file 3924 as in, for example, general register file 1824 as in FIG. 18B, register file 1906 as in FIG. 19, or a portion of shared register file 3114 as in FIG. 31. The systolic tensor array 2808 may also include aspects of the systolic array 1912 as in FIG. 19, systolic array 3000 as in FIG. 30A-30B, and/or systolic array 3112A-3112B. The system 3900 also includes an image to column adapter 3910 that includes circuitry to convert the data for an input image for a convolution operation, which may be configured as a three channel RGB image, into column formatted data. In the column formatted data, the data for the multiple input channels is separated and re-formatted to into transformed input 3932. Using the transformed input 3932 enhances the computational efficiency of depthwise convolution operations when performed via the systolic array 2808. The image to column adapter 3910 may be implemented as a set of input ports configured to read vector and/or packed data from the general register file 3924 that are mapped to a set of output ports that couple with the systolic tensor array 2808. The mapping of the input ports to the output ports may be adjusted based on the specifics of the input activation matrix to be transformed.

[0436] When performing depth-wise convolution, weights 3925 are provided as SRC2 input 3926 and input activation data 3930 is passed though the image to column adapter 3910 before being provided as SRC1 input 3931 as transformed input 3932. An incremental dpas (dot-product, accumulate, systolic) instruction is used that loads a single input register of transformed

input 3932 into SRC1 3931 per instruction. Multiple iterations of the dpas instruction may be used to process an entire set of input, depending on the size of the input data. An initialization mode is also used in which an initial set of registers worth of transformed input 3932 is loaded as SRC1 input 3931. Successive iterations of the incremental dpas instruction can be performed until convolution is performed for the entire set of input activations.

[0437] As shown in FIG. 39B, no significant transformation is required for weight inputs. The weight input may be read from the general register file 3924 during the incremental dpas initialization operation. Successive reads of weight data may be bypassed and a depthwise weight matrix 3928 may be generated for each iteration within the systolic tensor array 2808. To construct the depthwise weight matrix 3928, the SRC2 weights are constructed with zeroes on the appropriate locations so that only the correct rows of weights are used per cycle. The correct rows of weights to use for the depthwise convolution operation are moved down in the depthwise weight matrix 3928 each cycle along a sliding window 3927. The correct rows may eventually wrap around to the top of the depthwise weight matrix 3928.

[0438] As shown in FIG. 39C, the values of the input activation 3930 are swizzled according to an image to column transform 3940 when the data is read from the general register file 3924. The image to column transform 3940 repositions matrix elements of the input activation 3930 to separate the data of the input activation 3930, which may be a multi-channel image, into transformed input 3932 in the form of a 2D matrix in which each of the channels are separated and arrange side-by-side. The specific nature of the image to column transform 3940 may vary based on the number of channels of input data, the size of the convolution kernel, and the target output depth.

[0439] **FIG. 40** illustrates a method 4000 of performing depthwise convolution on a systolic array. Method 4000 can be performed by hardware or firmware logic of a systolic array in conjunction with a general register file and image to column adapter, for example, as illustrated in FIG. 39A, using data transforms illustrated in FIG. 39B and FIG. 39C.

[0440] Method 4000 includes for logic associated with a systolic array to load an initial input including a first set of multiple rows of a transformed matrix of input activations and a corresponding set of rows of a transformed weight matrix (4002). The transformed matrix of input activations can be read from a general register file and automatically transformed by an image to column adapter during the read. The number of rows of initial input may correspond with the size of the convolution operation to be performed. The logic can then perform a depthwise convolution calculation via the systolic array using the initial input (4004). The logic can then load a next row of the transformed matrix of input activations into the systolic array (4006). The loaded next row can be automatically transformed by the image to column adapter

upon the read from the general register file. The logic can then adjust the weight values of the transformed weight matrix within the systolic array (4008). The logic can then perform a next depthwise convolution calculation via the systolic array using the next row of the transformed matrix and the adjusted weight values (4010). The operations can continue at a one row quantum until all of the transformed input activation data is convolved with the transformed weights. If necessary, a pointwise convolution may be performed on the output of the depthwise convolution to either reduce or expand the number of channels of the output.

[0441] In one embodiment, the series of dot product instructions used to perform the depthwise convolution operations described herein may be implemented using the block sparse macro instruction (dpXasV) described herein when performing operations on sparse matrices.

[0442] Using the techniques above, one skilled in the art may implement, for example, a general-purpose graphics processing unit comprising a register file to store matrix elements, a tensor accelerator to perform a depthwise convolution operation on matrix elements read from the register file, and an adapter circuit to transform a data configuration of the matrix elements of an input activation tensor, the adapter circuit to transform the data configuration during a read of the matrix elements by the tensor accelerator from the register file. The adapter circuit is to perform an image to column transformation on the data configuration of the matrix elements of an input activation tensor.

[0443] Using the techniques above, one skilled in the art may implement, for example, a general-purpose graphics processing unit configured to perform a method comprising loading an initial input including a first set of multiple rows of a transformed matrix of input activations and a corresponding set of rows of a transformed weight matrix and performing a depthwise convolution calculation via the systolic array using the initial input. The transformation performed on the matrix of input activations may be an image to column transformation that is automatically performed when the matrix of input activations are read from a general register file of the general-purpose graphics processing unit. The method additionally includes loading the next row of the transformed matrix of input activations into the systolic array, where the next row is automatically transformed by the image to column adapter. The method additionally includes adjusting weight values of the transformed weight matrix. The adjustment of the weight values can be performed within the systolic array. The method additionally includes to perform a next depthwise convolution calculation via the systolic array using the next row of the transformed matrix and the adjusted weight values. The depthwise convolution operations may be dot product operations. The dot product operations may be performed by dot product instructions generated by a block sparse dot product macro instruction.

Additional Exemplary Computing Device

[0444] **FIG. 41** is a block diagram of a computing device 4100 including a graphics processor 4104, according to an embodiment. The computing device 4100 can be a computing device that includes functionality of each of the embodiments described above. The computing device 4100 may be or be included within a communication device such as a set-top box (e.g., Internet-based cable television set-top boxes, etc.), global positioning system (GPS)-based devices, etc. The computing device 4100 may also be or be included within mobile computing devices such as cellular phones, smartphones, personal digital assistants (PDAs), tablet computers, laptop computers, e-readers, smart televisions, television platforms, wearable devices (e.g., glasses, watches, bracelets, smartcards, jewelry, clothing items, etc.), media players, etc. For example, in one embodiment, the computing device 4100 includes a mobile computing device employing an integrated circuit ("IC"), such as system on a chip ("SoC" or "SOC"), integrating various hardware and/or software components of computing device 4100 on a single chip.

[0445] The computing device 4100 includes a graphics processor 4104. The graphics processor 4104 represents any graphics processor described herein. The graphics processor includes one or more graphics engine(s), graphics processor cores, and other graphics execution resources as described herein. Such graphics execution resources can be presented in the forms including but not limited to execution units, shader engines, fragment processors, vertex processors, streaming multiprocessors, graphics processor clusters, or any collection of computing resources suitable for the processing of graphics resources or image resources, or performing general purpose computational operations in a heterogeneous processor.

[0446] In one embodiment, the graphics processor 4104 includes a cache 4114, which can be a single cache or divided into multiple segments of cache memory, including but not limited to any number of L1, L2, L3, or L4 caches, render caches, depth caches, sampler caches, and/or shader unit caches. The cache 4114 may have a near and far region as described herein. The cache 4114 may also include dynamic hash logic that supports dynamic reconfiguration of a memory bank hash algorithm. In some embodiments, the graphics processor 4104 includes a GPGPU engine 4144 that includes shared local memory (SLM 4134), as well as a register file 4124, including includes registers for use by the GPGPU engine 4144. The register file 4124 can include general-purpose registers, architectural registers, configuration registers, and other types of registers. A general-purpose register file (GRF) and/or architectural register file (ARF) can also reside within processing resources within one or more blocks of compute units (e.g., compute 4150, compute 4155) within the GPGPU engine 4144. A shared fabric 4142 may also

be present that enables rapid communication between the various components of the GPGPU engine 4144.

[0447] As illustrated, in one embodiment, and in addition to the graphics processor 4104, the computing device 4100 may further include any number and type of hardware components and/or software components, including, but not limited to an application processor 4106, memory 4108, and input/output (I/O) sources 4110. The application processor 4106 can interact with a hardware graphics pipeline to share graphics pipeline functionality. Processed data is stored in a buffer in the hardware graphics pipeline and state information is stored in memory 4108. The resulting data can be transferred to a display controller for output via a display device as described herein. The display device may be of various types, such as Cathode Ray Tube (CRT), Thin Film Transistor (TFT), Liquid Crystal Display (LCD), Organic Light Emitting Diode (OLED) array, etc., and may be configured to display information to a user via a graphical user interface.

[0448] The application processor 4106 can include one or processors and may be the central processing unit (CPU) that is used at least in part to execute an operating system (OS) 4102 for the computing device 4100. The OS 4102 can serve as an interface between hardware and/or physical resources of the computing device 4100 and one or more users. The OS 4102 can include driver logic for various hardware devices in the computing device 4100, including graphics driver logic 4122, such as the user mode graphics driver 2326 and/or kernel mode graphics driver 2329 of FIG. 23.

[0449] It is contemplated that in some embodiments the graphics processor 4104 may exist as part of the application processor 4106 (such as part of a physical CPU package) in which case, at least a portion of the memory 4108 may be shared by the application processor 4106 and graphics processor 4104, although at least a portion of the memory 4108 may be exclusive to the graphics processor 4104, or the graphics processor 4104 may have a separate store of memory. The memory 4108 may comprise a pre-allocated region of a buffer (e.g., framebuffer); however, it should be understood by one of ordinary skill in the art that the embodiments are not so limited, and that any memory accessible to the lower graphics pipeline may be used. The memory 4108 may include various forms of random-access memory (RAM) (e.g., SDRAM, SRAM, etc.) comprising an application that makes use of the graphics processor 4104 to render a desktop or 3D graphics scene. A memory controller hub may access data in the memory 4108 and forward it to graphics processor 4104 for graphics pipeline processing. The memory 4108 may be made available to other components within the computing device 4100. For example, any data (e.g., input graphics data) received from various I/O sources 4110 of the computing device 4100 can be temporarily queued into memory 4108 prior to their being operated upon by

one or more processor(s) (e.g., application processor 4106) in the implementation of a software program or application. Similarly, data that a software program determines should be sent from the computing device 4100 to an outside entity through one of the computing system interfaces, or stored into an internal storage element, is often temporarily queued in memory 4108 prior to its being transmitted or stored.

[0450] The I/O sources can include devices such as touchscreens, touch panels, touch pads, virtual or regular keyboards, virtual or regular mice, ports, connectors, network devices, or the like, and can attach via an I/O hub 107 as in FIG. 1, Input/output (I/O) circuitry 363 as in FIG. 3,, a platform controller hub 1430 as in FIG. 14, or the like. Additionally, the I/O sources 4110 may include one or more I/O devices that are implemented for transferring data to and/or from the computing device 4100 (e.g., a networking adapter); or, for a large-scale non-volatile storage within the computing device 4100 (e.g., hard disk drive). User input devices, including alphanumeric and other keys, may be used to communicate information and command selections to graphics processor 4104. Another type of user input device is cursor control, such as a mouse, a trackball, a touchscreen, a touchpad, or cursor direction keys to communicate direction information and command selections to GPU and to control cursor movement on the display device. Camera and microphone arrays of the computing device 4100 may be employed to observe gestures, record audio and video and to receive and transmit visual and audio commands.

[0451] I/O sources 4110 configured as network interfaces can provide access to a network, such as a LAN, a wide area network (WAN), a metropolitan area network (MAN), a personal area network (PAN), Bluetooth, a cloud network, a cellular or mobile network (e.g., 3rd Generation (3G), 4th Generation (4G), 5th Generation (5G), etc.), a satellite network, an intranet, the Internet, etc. Network interface(s) may include, for example, a wireless network interface having one or more antenna(e). Network interface(s) may also include, for example, a wired network interface to communicate with remote devices via network cable, which may be, for example, an Ethernet cable, a coaxial cable, a fiber optic cable, a serial cable, or a parallel cable.

[0452] Network interface(s) may provide access to a LAN, for example, by conforming to IEEE 802.11 standards, and/or the wireless network interface may provide access to a personal area network, for example, by conforming to Bluetooth standards. Other wireless network interfaces and/or protocols, including previous and subsequent versions of the standards, may also be supported. In addition to, or instead of, communication via the wireless LAN standards, network interface(s) may provide wireless communication using, for example, Time Division, Multiple Access (TDMA) protocols, Global Systems for Mobile Communications (GSM)

protocols, Code Division, Multiple Access (CDMA) protocols, and/or any other type of wireless communications protocols.

[0453] It is to be appreciated that a lesser or more equipped system than the example described above may be preferred for certain implementations. Therefore, the configuration of the computing device 4100 may vary from implementation to implementation depending upon numerous factors, such as price constraints, performance requirements, technological improvements, or other circumstances. Examples include (without limitation) a mobile device, a personal digital assistant, a mobile computing device, a smartphone, a cellular telephone, a handset, a one-way pager, a two-way pager, a messaging device, a computer, a personal computer (PC), a desktop computer, a laptop computer, a notebook computer, a handheld computer, a tablet computer, a server, a server array or server farm, a web server, a network server, an Internet server, a work station, a mini-computer, a main frame computer, a supercomputer, a network appliance, a web appliance, a distributed computing system, multiprocessor systems, processor-based systems, consumer electronics, programmable consumer electronics, television, digital television, set top box, wireless access point, base station, subscriber station, mobile subscriber center, radio network controller, router, hub, gateway, bridge, switch, machine, or combinations thereof.

[0454] Embodiments may be implemented as any one, or a combination of one or more microchips or integrated circuits interconnected using a parent-board, hardwired logic, software stored by a memory device and executed by a microprocessor, firmware, an application specific integrated circuit (ASIC), and/or a field programmable gate array (FPGA). The term "logic" may include, by way of example, software or hardware and/or combinations of software and hardware.

[0455] Embodiments may be provided, for example, as a computer program product which may include one or more machine-readable media having stored thereon machine-executable instructions that, when executed by one or more machines such as a computer, network of computers, or other electronic devices, may result in the one or more machines carrying out operations in accordance with embodiments described herein. A machine-readable medium may include, but is not limited to, floppy diskettes, optical disks, CD-ROMs (Compact Disc-Read Only Memories), and magneto-optical disks, ROMs, RAMs, EPROMs (Erasable Programmable Read Only Memories), EEPROMs (Electrically Erasable Programmable Read Only Memories), magnetic or optical cards, flash memory, or other type of non-transitory machine-readable media suitable for storing machine-executable instructions.

[0456] Moreover, embodiments may be downloaded as a computer program product, wherein the program may be transferred from a remote computer (e.g., a server) to a requesting

computer (e.g., a client) by way of one or more data signals embodied in and/or modulated by a carrier wave or other propagation medium via a communication link (e.g., a modem and/or network connection).

[0457] Reference herein to “one embodiment” or “an embodiment” means that a particular feature, structure, or characteristic described in conjunction with the embodiment can be included in at least one embodiment of the invention. The appearances of the phrase “in one embodiment” in various places in the specification do not necessarily all refer to the same embodiment. The processes depicted in the figures that follow can be performed by processing logic that comprises hardware (e.g. circuitry, dedicated logic, etc.), software (as instructions on a non-transitory machine-readable storage medium), or a combination of both hardware and software. Reference will be made in detail to various embodiments, examples of which are illustrated in the accompanying drawings. In the following detailed description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one of ordinary skill in the art that the present invention may be practiced without these specific details. In other instances, well-known methods, procedures, components, circuits, and networks have not been described in detail so as not to unnecessarily obscure aspects of the embodiments.

[0458] It will also be understood that, although the terms first, second, etc. may be used herein to describe various elements, these elements should not be limited by these terms. These terms are only used to distinguish one element from another. For example, a first contact could be termed a second contact, and, similarly, a second contact could be termed a first contact, without departing from the scope of the present invention. The first contact and the second contact are both contacts, but they are not the same contact.

[0459] The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting as to all embodiments. As used in the description of the invention and the appended claims, the singular forms "a", "an" and "the" are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will also be understood that the term "and/or" as used herein refers to and encompasses any and all possible combinations of one or more of the associated listed items. It will be further understood that the terms "comprises" and/or "comprising," when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components, and/or groups thereof.

[0460] As used herein, the term "if" may be construed to mean "when" or "upon" or "in response to determining" or "in response to detecting," depending on the context. Similarly, the

phrase "if it is determined" or "if [a stated condition or event] is detected" may be construed to mean "upon determining" or "in response to determining" or "upon detecting [the stated condition or event]" or "in response to detecting [the stated condition or event]," depending on the context.

[0461] Embodiments described herein include, software, firmware, and hardware logic that provides techniques to perform arithmetic on sparse data via a systolic processing unit. One embodiment provides for data aware sparsity via compressed bitstreams. One embodiment provides for block sparse dot product instructions (e.g., dp4a, dp2a). One embodiment provides for a depth-wise adapter for a systolic array.

[0462] One embodiment provides for a graphics processor comprising a fetch and decode unit including an instruction fetcher to fetch a hardware macro instruction and an instruction decoder to decode the hardware macro instruction, where the hardware macro instruction includes a set of initial operands and to decode the hardware macro instruction includes to generate a set of hardware instructions including multiple dot product instructions, the multiple dot product instructions to be executed on input associated with a first matrix and a second matrix. The input is identified at least in part based on the initial set of operands. The graphics processor additionally includes a matrix accelerator including a processing resource to execute the set of hardware instructions.

[0463] One embodiment provides for a method comprising, on a graphics processing unit including a matrix accelerator, fetching a hardware macro instruction having a predicate mask, a repeat count, and a set of initial operands. The hardware macro instruction can enable generation of a set of dot product operations on input associated with a set of matrices. In one embodiment, the method additionally includes decoding the hardware macro instruction into a decoded hardware macro instruction and generating a set of hardware instructions based on the decoded hardware macro instruction, the set of hardware instructions including a set of multiple dot product operations, the set of hardware instructions generated based on the predicate mask and the repeat count. The method further includes executing the set of hardware instructions via a processing resource of a graphics processor

[0464] One embodiment provides a graphics processing system comprising a memory device and a graphics processor coupled with the memory device, the graphics processor including a fetch and decode unit including an instruction fetcher to fetch a hardware macro instruction and an instruction decoder to decode the hardware macro instruction, where the hardware macro instruction includes a set of initial operands and to decode the hardware macro instruction includes to generate a set of hardware instructions including multiple dot product instructions. The multiple dot product instructions can be executed on input associated with a first matrix and

a second matrix. The input can be identified at least in part based on the initial set of operands. In one embodiment the matrix accelerator including a processing resource to execute the set of hardware instructions.

[0001] The foregoing description and drawings are to be regarded in an illustrative rather than a restrictive sense. Persons skilled in the art will understand that various modifications and changes may be made to the embodiments described herein without departing from the broader spirit and scope of the features set forth in the appended claims.

CLAIMS

What is claimed is:

1. A graphics processor comprising:

a fetch and decode unit including an instruction fetcher to fetch a hardware macro instruction and an instruction decoder to decode the hardware macro instruction, wherein the hardware macro instruction includes a set of initial operands and to decode the hardware macro instruction includes to generate a set of hardware instructions including multiple dot product instructions, the multiple dot product instructions to be executed on input associated with a first matrix and a second matrix, the input identified at least in part based on the initial set of operands; and

a matrix accelerator including a processing resource to execute the set of hardware instructions.

2. The graphics processor as in claim 1, wherein the hardware macro instruction is retired upon completion of the set of hardware instructions.

3. The graphics processor as in claim 1, wherein the set of initial operands include a destination operand and multiple source operands.

4. The graphics processor as in claim 3, wherein the hardware macro instruction additionally includes a predicate mask and a repeat count, the set of hardware instructions generated based on the predicate mask and the repeat count.

5. The graphics processor as in claim 4, wherein the repeat count is to indicate a maximum number of hardware instructions to generate for the set of hardware instructions.

6. The graphics processor as in claim 5, wherein each of the multiple dot product instructions is to cause the processing resource to perform a dot product operation on multiple data elements.

7. The graphics processor as in claim 6, wherein to generate the set of hardware instructions, the instruction decoder is to:

generate a first hardware instruction for an active bit within the predicate mask; and

bypass generation of a second hardware instruction for an inactive bit within the predicate mask.

8. The graphics processor as in claim 7, wherein to generate the set of hardware instructions, the instruction decoder is to additionally to:

generate a first set of operands for the first hardware instruction, the first set of operands to indicate a destination register and multiple source registers for use by the first hardware instruction, wherein the first set of operands are generated based on the set of initial operands and an operand offset.

9. The graphics processor as in claim 8, wherein the predicate mask is a first predicate mask, to generate the first hardware instruction for an active bit includes to generate a second predicate mask for the first hardware instruction, and the second predicate mask to indicate one or more active channels to compute for the first hardware instruction.

10. The graphics processor as in claim 1, wherein the processing resource is a processing resource within a systolic array of processing resources.

11. The graphics processor as in claim 1, wherein the multiple dot product instructions include a four element dot product instruction.

12. The graphics processor as in claim 1, wherein the multiple dot product instructions include a two element dot product instruction.

13. A method comprising:

on a graphics processing unit including a matrix accelerator:

fetching a hardware macro instruction having a predicate mask, a repeat count, and a set of initial operands, wherein the hardware macro instruction is to enable generation of a set of dot product operations on input associated with a set of matrices;

decoding the hardware macro instruction into a decoded hardware macro instruction;

generating a set of hardware instructions based on the decoded hardware macro instruction, the set of hardware instructions including a set of multiple dot product

operations, the set of hardware instructions generated based on the predicate mask and the repeat count; and

executing the set of hardware instructions via a processing resource of a graphics processor.

14. The method as in claim 13, wherein the processing resource of the graphics processor is a processing resource within a matrix accelerator within the graphics processor.

15. The method as in claim 14, additionally comprising retiring the hardware macro instruction upon completion of the set of hardware instructions.

16. The method as in claim 13, wherein the set of initial operands include a destination operand and multiple source operands.

17. The method as in claim 13, wherein generating the set of hardware instructions includes:

generating a first hardware instruction for an active bit within the predicate mask; and

bypassing generation of a second hardware instruction for an inactive bit within the predicate mask.

18. The method as in claim 17, additionally comprising:

generating a first set of operands for the first hardware instruction, the first set of operands to indicate a destination register and multiple source registers for use by the first hardware instruction, wherein the first set of operands are generated based on the set of initial operands and an operand offset.

19. The method as in claim 18, wherein the predicate mask is a first predicate mask and generating the first hardware instruction for an active bit includes generating a second predicate mask for the first hardware instruction, the second predicate mask to indicate one or more active channels to compute for the first hardware instruction.

20. A graphics processing system comprising:

a memory device; and

a graphics processor coupled with the memory device, the graphics processor including:

a fetch and decode unit including an instruction fetcher to fetch a hardware macro instruction and an instruction decoder to decode the hardware macro instruction, wherein the hardware macro instruction includes a set of initial operands and to decode the hardware macro instruction includes to generate a set of hardware instructions including multiple dot product instructions, the multiple dot product instructions to be executed on input associated with a first matrix and a second matrix, the input identified at least in part based on the initial set of operands; and

a matrix accelerator including a processing resource to execute the set of hardware instructions.

21. The graphics processing system as in claim 20, wherein the hardware macro instruction is retired upon completion of the set of hardware instructions.

22. The graphics processing system as in claim 20, wherein the set of initial operands include a destination operand and multiple source operands, the hardware macro instruction additionally includes a predicate mask and a repeat count, the set of hardware instructions is generated based on the predicate mask and the repeat count, and the repeat count is to indicate a maximum number of hardware instructions to generate for the set of hardware instructions.

23. The graphics processing system as in claim 22, wherein each of the multiple dot product instructions is to cause the processing resource to perform a dot product operation on multiple data elements.

24. The graphics processing system as in claim 23, wherein to generate the set of hardware instructions, the instruction decoder is to:

generate a first hardware instruction for an active bit within the predicate mask; and

bypass generation of a second hardware instruction for an inactive bit within the predicate mask.

25. The graphics processing system as in claim 24, wherein the predicate mask is a first predicate mask, to generate the first hardware instruction for an active bit includes to

generate a second predicate mask for the first hardware instruction, and the second predicate mask to indicate one or more active channels to compute for the first hardware instruction.

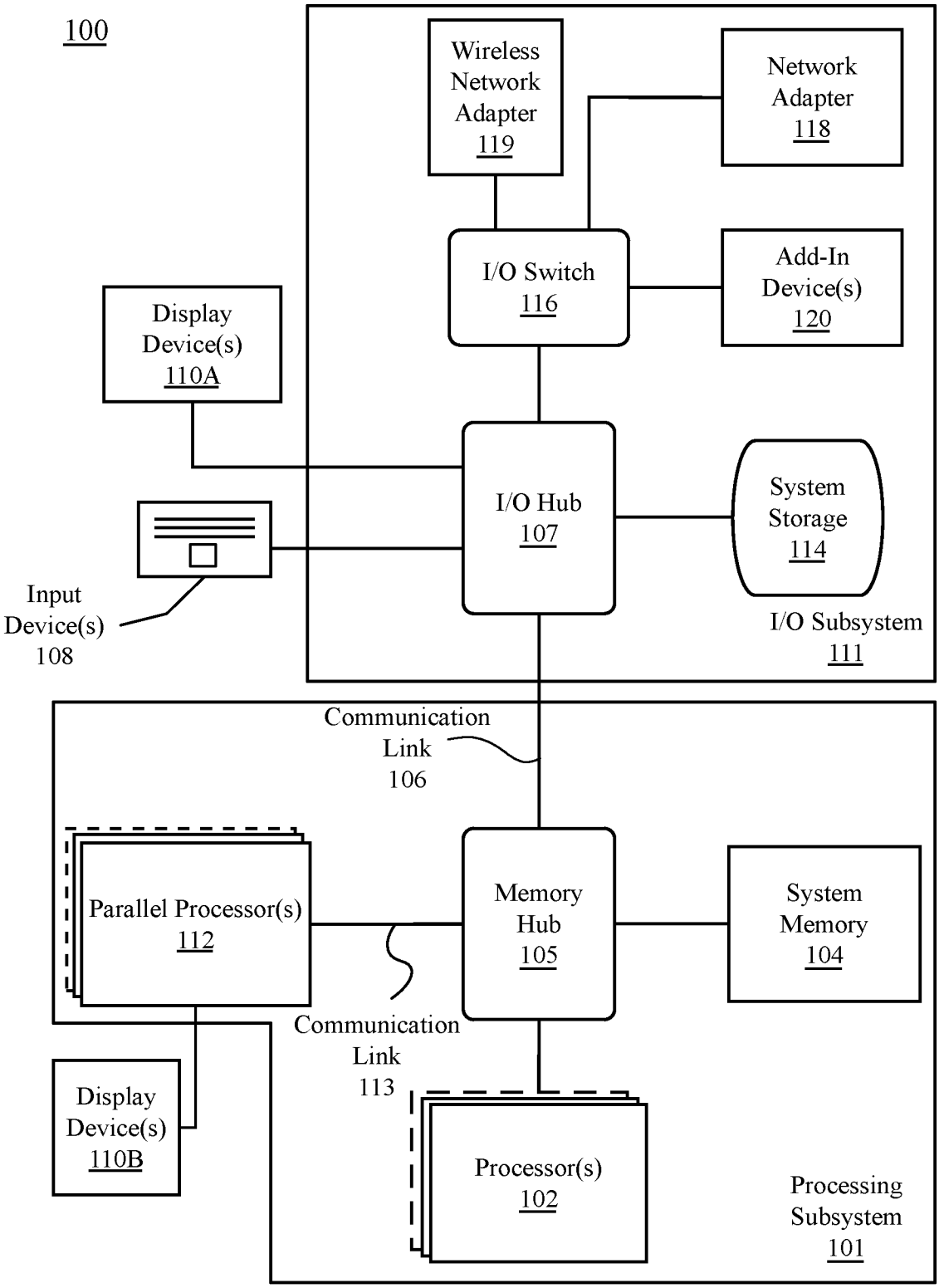

100
Wireless Network Adapter 119
Network Adapter 118
I/O Switch 116
Add-In Device(s) 120
Display Device(s) 110A
I/O Hub 107
System Storage 114
Input Device(s) 108
I/O Subsystem 111
Communication Link 106
Parallel Processor(s) 112
Memory Hub 105
System Memory 104
Communication Link 113
Display Device(s) 110B
Processor(s) 102
Processing Subsystem 101

FIG. 1

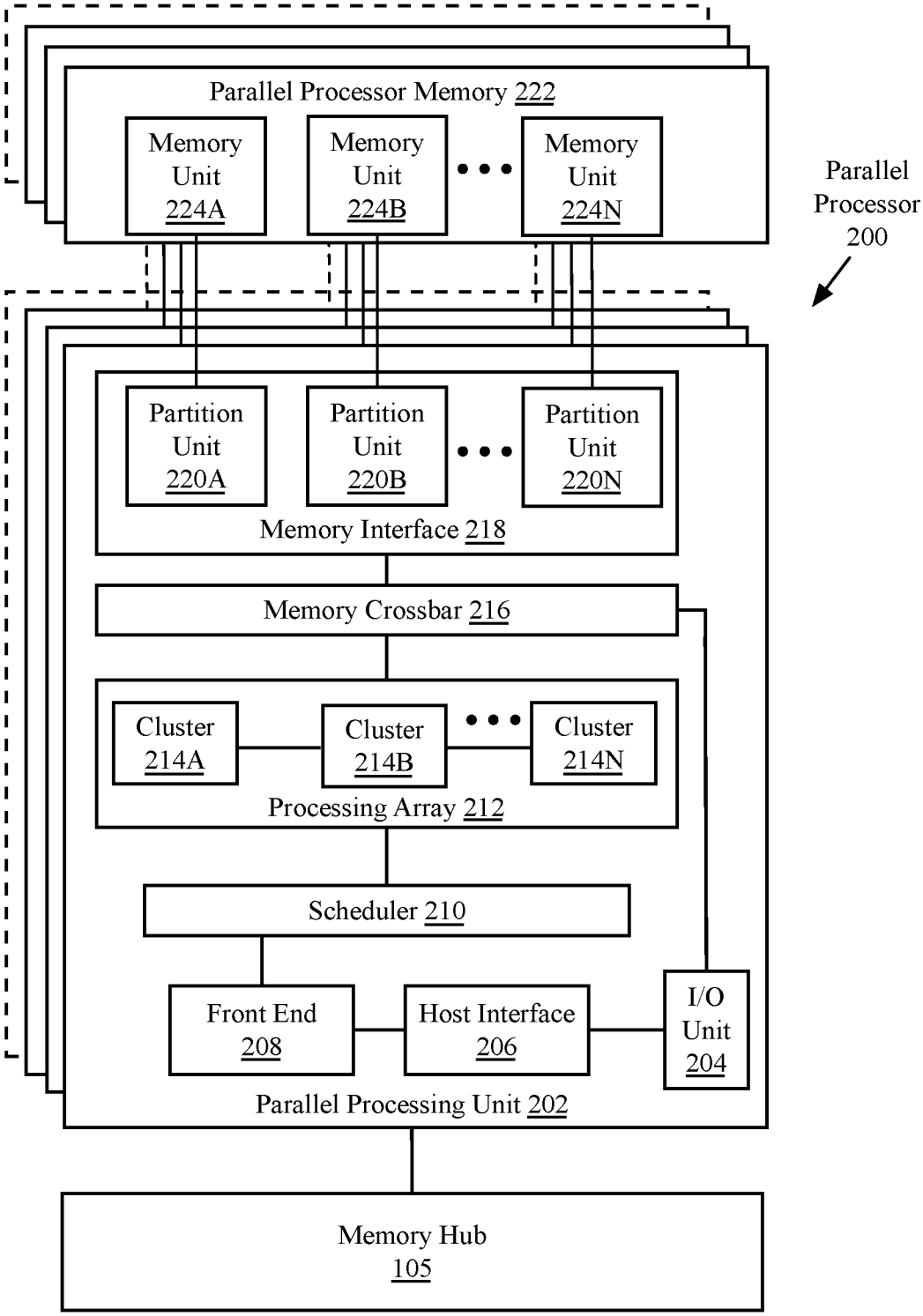

Parallel Processor Memory 222
Memory Unit 224A
Memory Unit 224B
Memory Unit 224N
Parallel Processor 200
Partition Unit 220A
Partition Unit 220B
Partition Unit 220N
Memory Interface 218
Memory Crossbar 216
Cluster 214A
Cluster 214B
Cluster 214N
Processing Array 212
Scheduler 210
Front End 208
Host Interface 206
I/O Unit 204
Parallel Processing Unit 202
Memory Hub 105

FIG. 2A

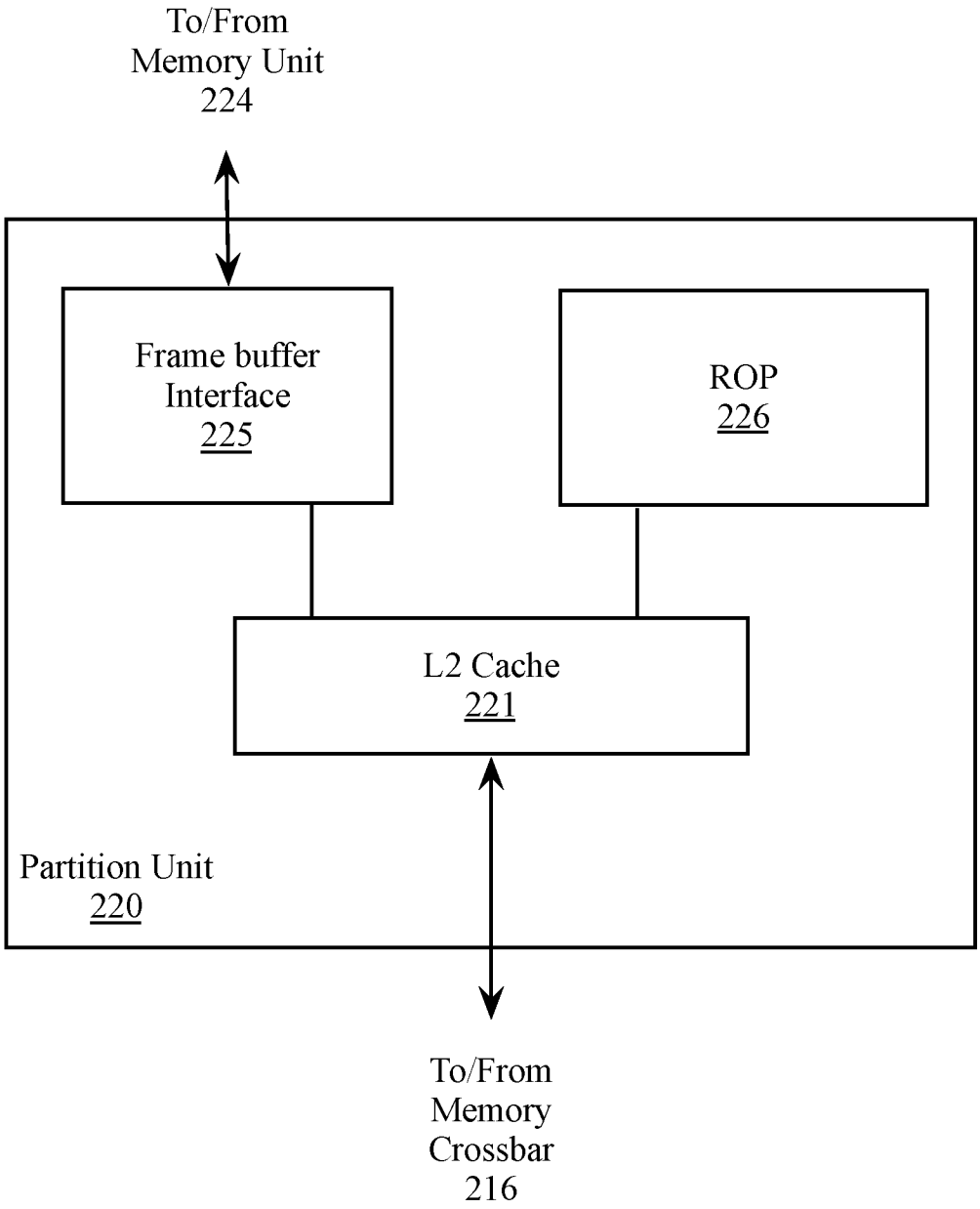

To/From
Memory Unit
224
Frame buffer
Interface
225
ROP
226
L2 Cache
221
Partition Unit
220
To/From
Memory
Crossbar
216

FIG. 2B

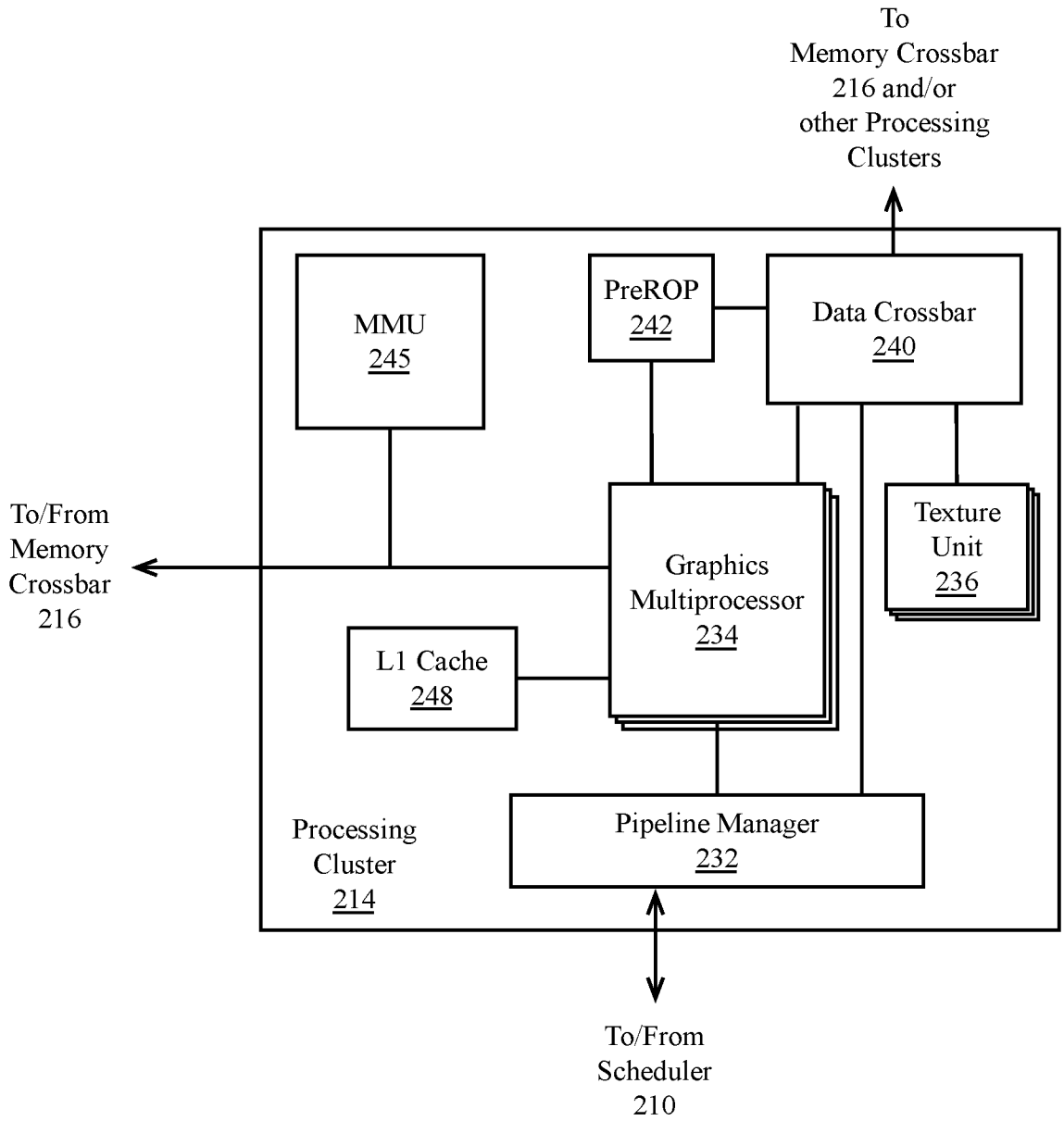

To
Memory Crossbar
216 and/or
other Processing
Clusters
MMU
245
PreROP
242
Data Crossbar
240
To/From
Memory
Crossbar
216
Graphics
Multiprocessor
234
Texture
Unit
236
L1 Cache
248
Processing
Cluster
214
Pipeline Manager
232
To/From
Scheduler
210

FIG. 2C

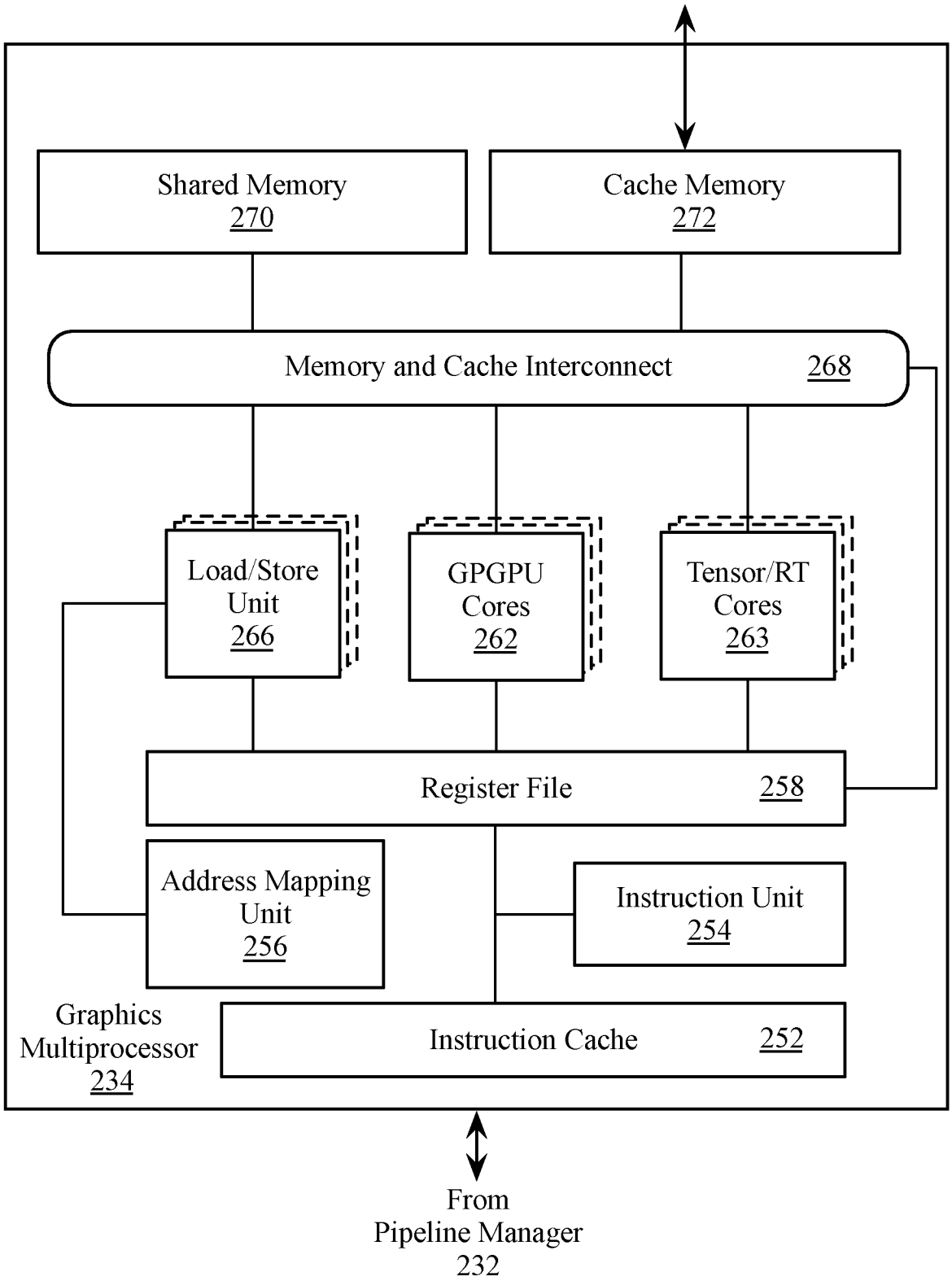

Shared Memory
270
Cache Memory
272
Memory and Cache Interconnect
268
Load/Store
Unit
266
GPGPU
Cores
262
Tensor/RT
Cores
263
Register File
258
Address Mapping
Unit
256
Instruction Unit
254
Graphics
Multiprocessor
234
Instruction Cache
252
From
Pipeline Manager
232

FIG. 2D

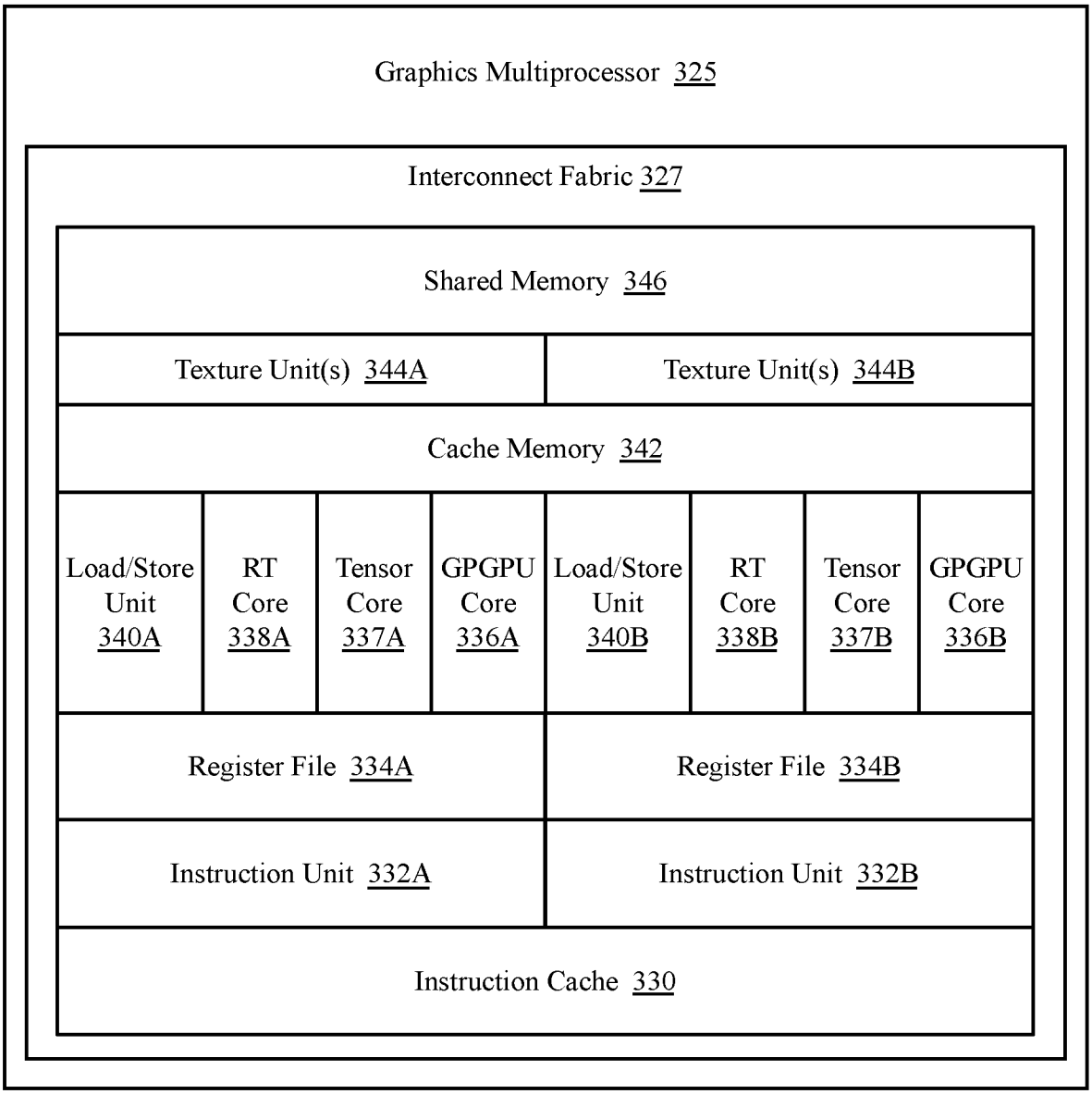

Graphics Multiprocessor 325
Interconnect Fabric 327
Shared Memory 346
Texture Unit(s) 344A
Texture Unit(s) 344B
Cache Memory 342
Load/Store Unit 340A
RT Core 338A
Tensor Core 337A
GPGPU Core 336A
Load/Store Unit 340B
RT Core 338B
Tensor Core 337B
GPGPU Core 336B
Register File 334A
Register File 334B
Instruction Unit 332A
Instruction Unit 332B
Instruction Cache 330

FIG. 3A

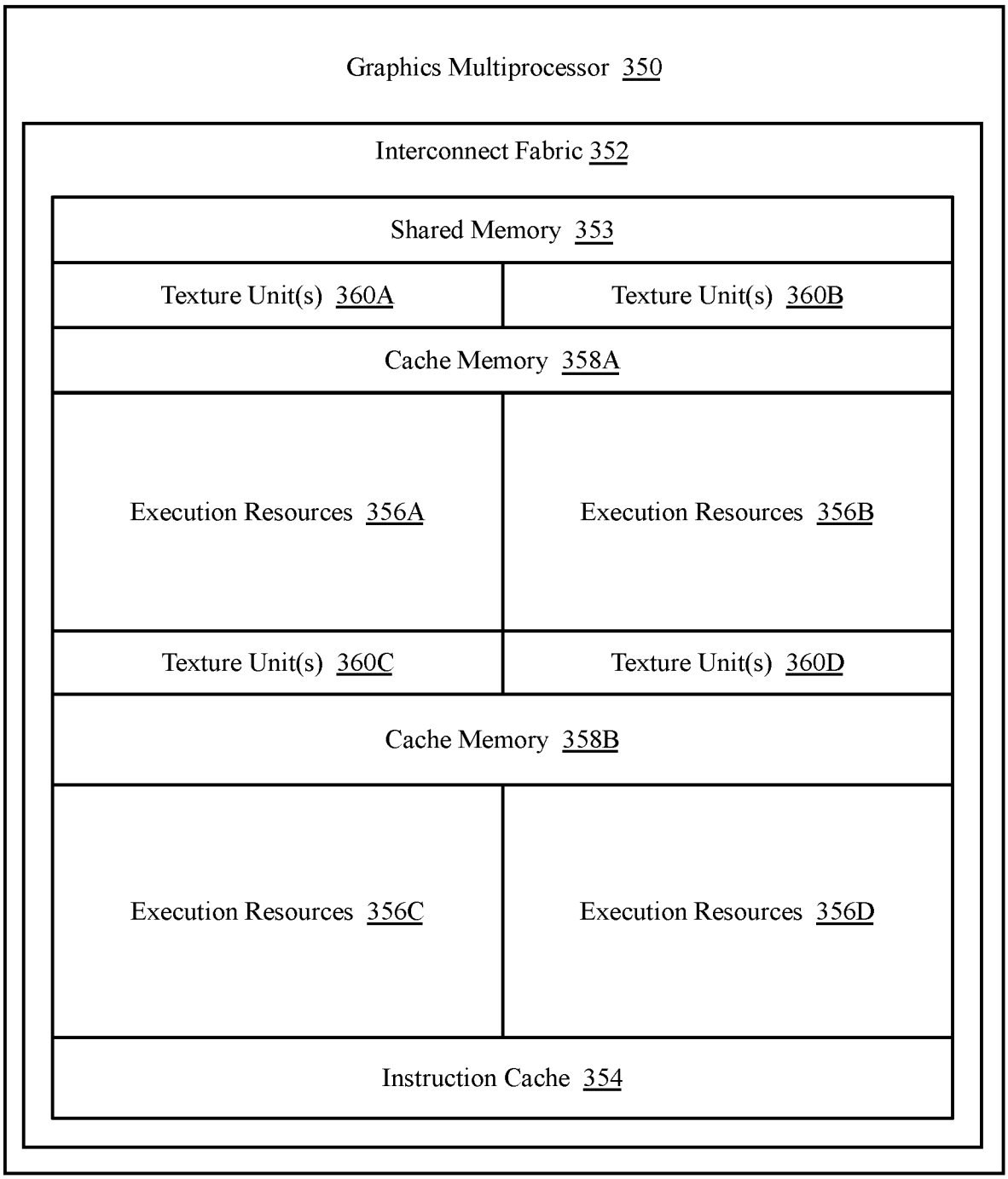

Graphics Multiprocessor 350
Interconnect Fabric 352
Shared Memory 353
Texture Unit(s) 360A
Texture Unit(s) 360B
Cache Memory 358A
Execution Resources 356A
Execution Resources 356B
Texture Unit(s) 360C
Texture Unit(s) 360D
Cache Memory 358B
Execution Resources 356C
Execution Resources 356D
Instruction Cache 354

FIG. 3B

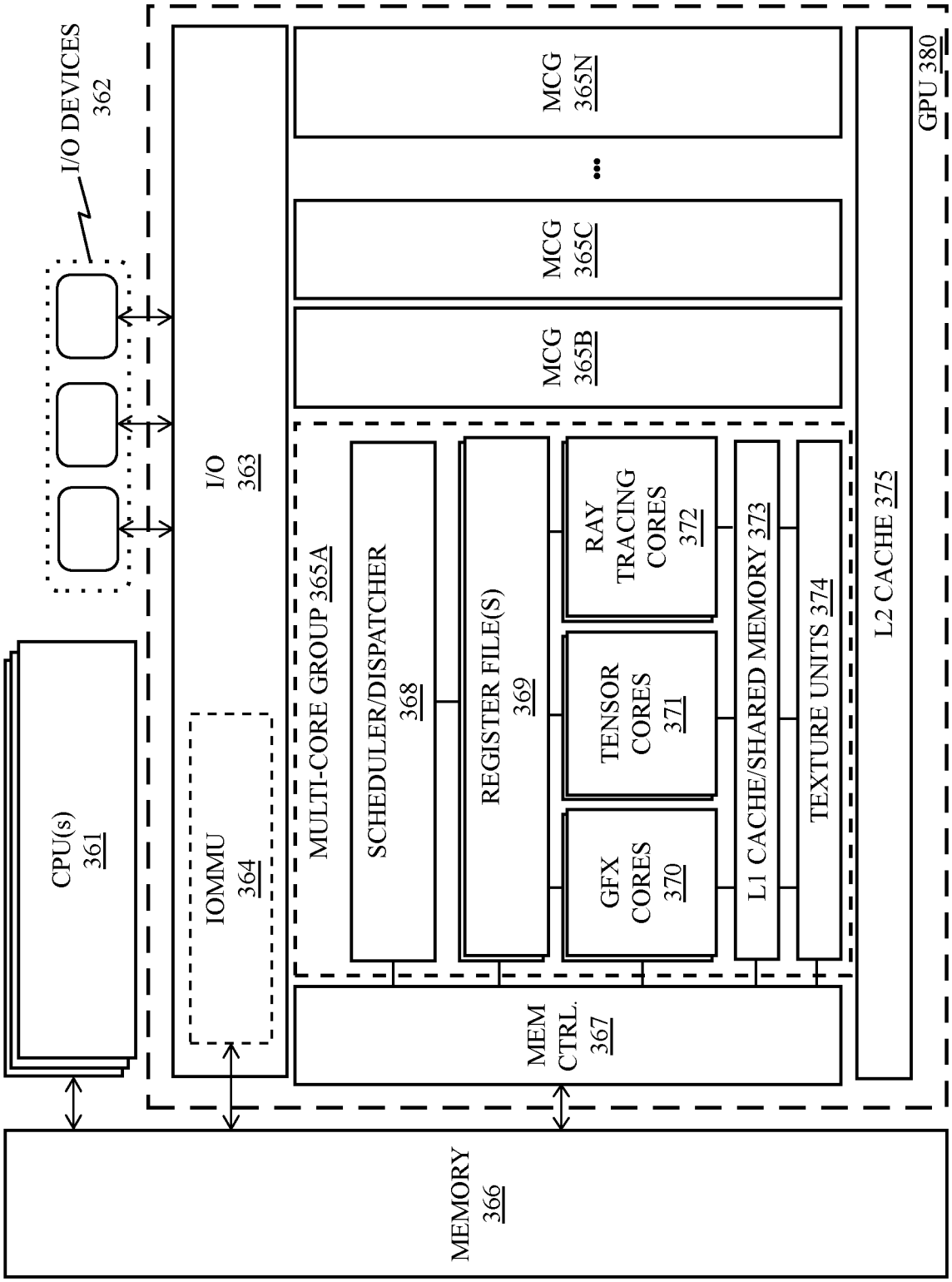

I/O DEVICES
362
CPU(s)
361
IOMMU
364
I/O
363
MULTI-CORE GROUP 365A
SCHEDULER/DISPATCHER
368
REGISTER FILE(S)
369
GFX
CORES
370
TENSOR
CORES
371
RAY
TRACING
CORES
372
L1 CACHE/SHARED MEMORY 373
TEXTURE UNITS 374
MCG
365B
MCG
365C
MCG
365N
L2 CACHE 375
GPU 380
MEM
CTRL.
367
MEMORY
366

FIG. 3C

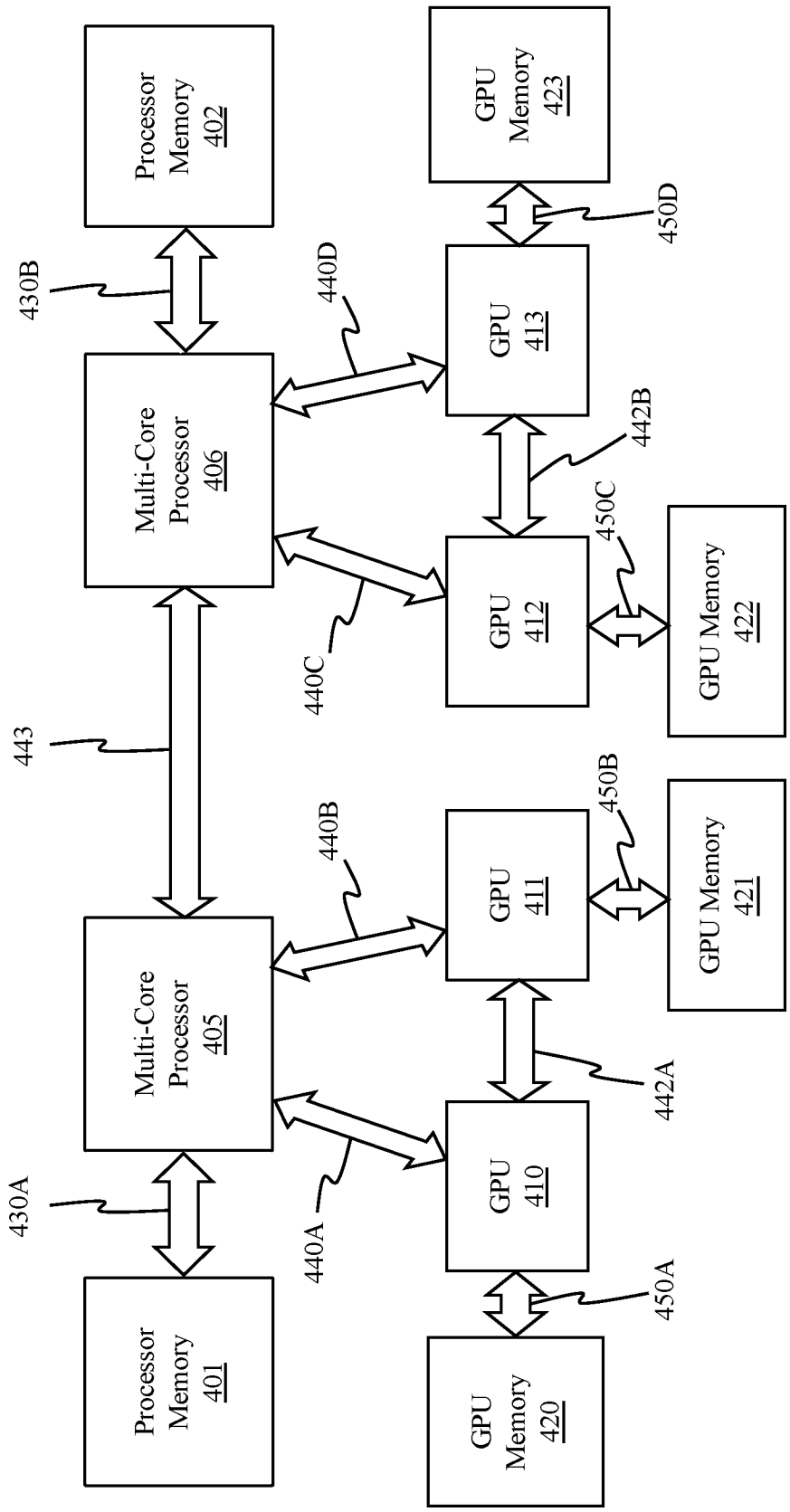

Processor Memory 401
Processor Memory 402
GPU Memory 420
GPU Memory 421
GPU Memory 422
GPU Memory 423
Multi-Core Processor 405
Multi-Core Processor 406
GPU 410
GPU 411
GPU 412
GPU 413
430A
430B
440A
440B
440C
440D
442A
442B
443
450A
450B
450C
450D

FIG. 4A

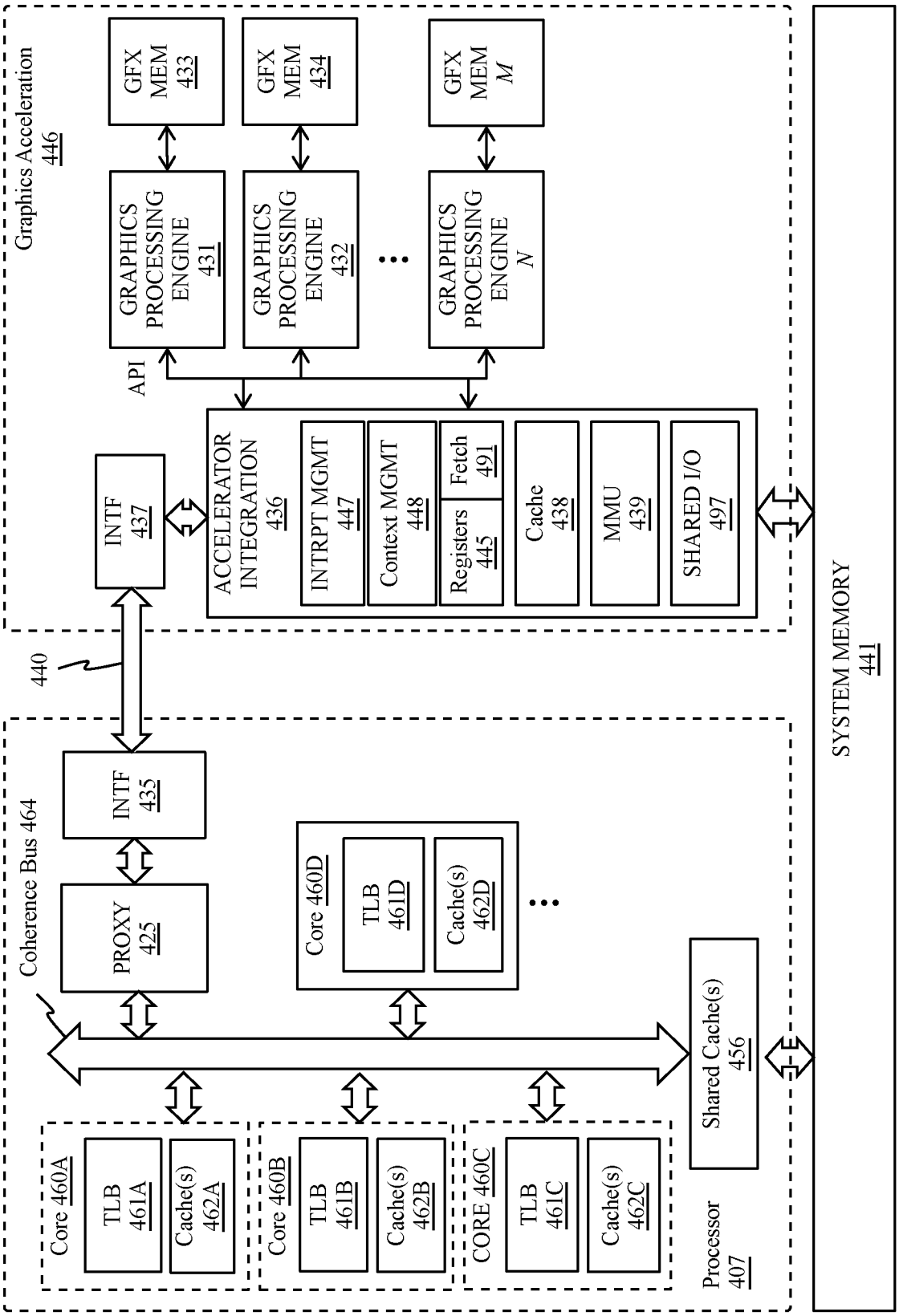

Graphics Acceleration 446
GFX MEM 433
GFX MEM 434
GFX MEM M
GRAPHICS PROCESSING ENGINE 431
GRAPHICS PROCESSING ENGINE 432
GRAPHICS PROCESSING ENGINE N
API
INTF 437
ACCELERATOR INTEGRATION 436
INTRPT MGMT 447
Context MGMT 448
Fetch 491
Registers 445
Cache 438
MMU 439
SHARED I/O 497
440
SYSTEM MEMORY 441
Coherence Bus 464
INTF 435
PROXY 425
Core 460D
TLB 461D
Cache(s) 462D
Shared Cache(s) 456
Core 460A
TLB 461A
Cache(s) 462A
Core 460B
TLB 461B
Cache(s) 462B
CORE 460C
TLB 461C
Cache(s) 462C
Processor 407

FIG. 4B

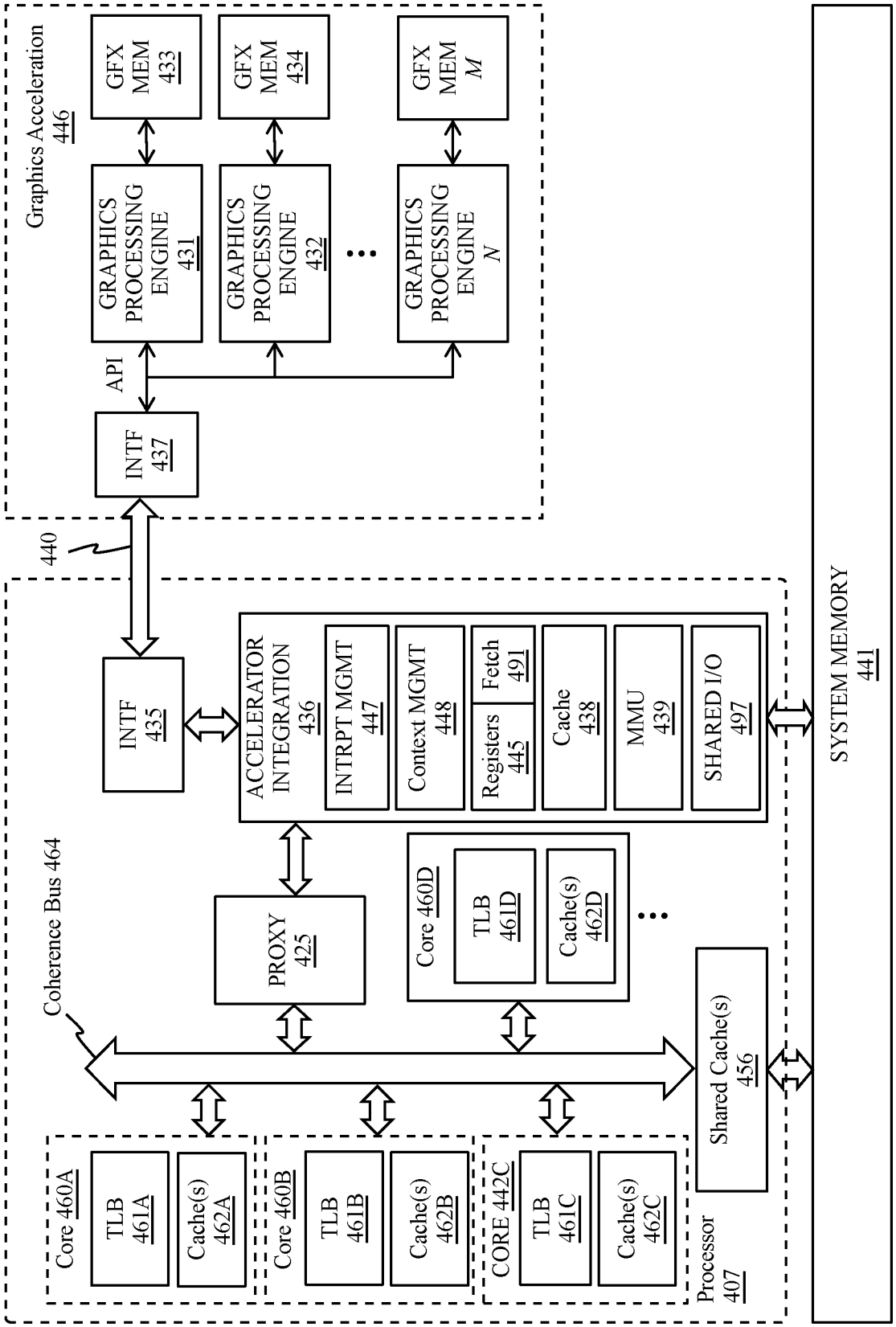

Graphics Acceleration 446
GFX MEM 433
GFX MEM 434
GFX MEM M
GRAPHICS PROCESSING ENGINE 431
GRAPHICS PROCESSING ENGINE 432
GRAPHICS PROCESSING ENGINE N
API
INTF 437
440
INTF 435
ACCELERATOR INTEGRATION 436
INTRPT MGMT 447
Context MGMT 448
Registers 445
Fetch 491
Cache 438
MMU 439
SHARED I/O 497
SYSTEM MEMORY 441
Coherence Bus 464
PROXY 425
Core 460D
TLB 461D
Cache(s) 462D
Shared Cache(s) 456
Core 460A
TLB 461A
Cache(s) 462A
Core 460B
TLB 461B
Cache(s) 462B
CORE 442C
TLB 461C
Cache(s) 462C
Processor 407

FIG. 4C

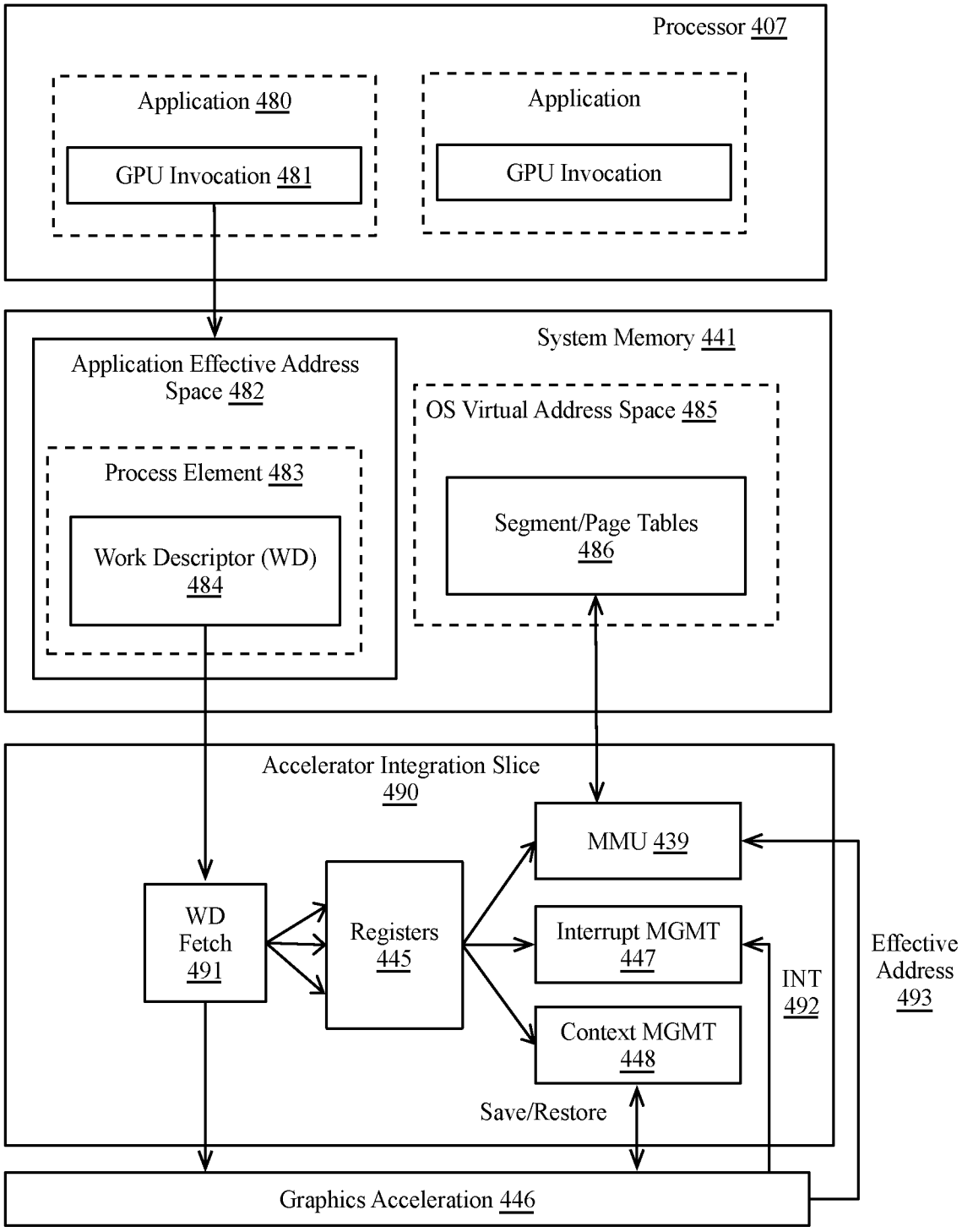

Processor 407
Application 480
GPU Invocation 481
Application
GPU Invocation
System Memory 441
Application Effective Address Space 482
Process Element 483
Work Descriptor (WD) 484
OS Virtual Address Space 485
Segment/Page Tables 486
Accelerator Integration Slice 490
MMU 439
WD Fetch 491
Registers 445
Interrupt MGMT 447
Context MGMT 448
INT 492
Effective Address 493
Save/Restore
Graphics Acceleration 446

FIG. 4D

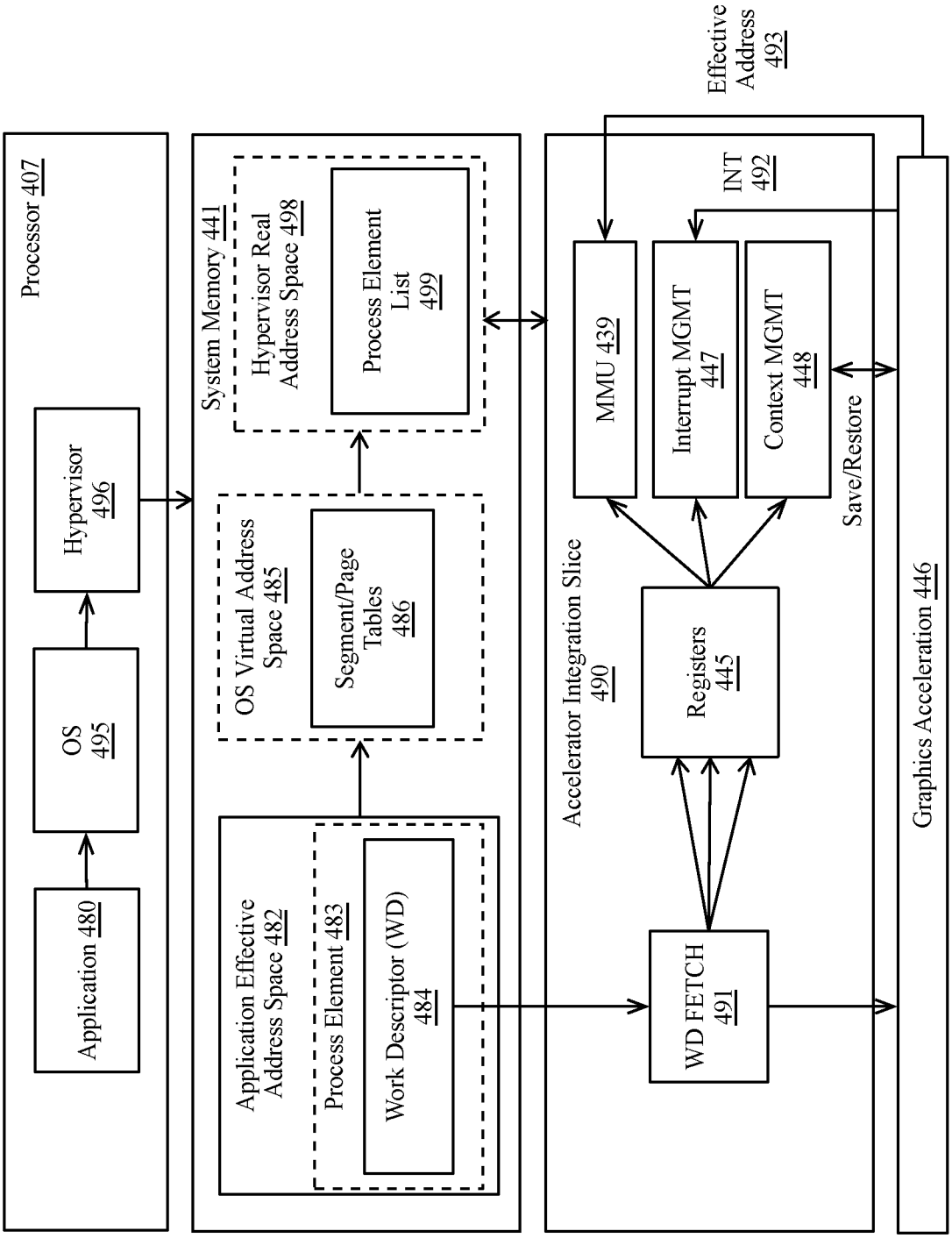

Processor 407
Application 480
OS 495
Hypervisor 496
System Memory 441
Application Effective Address Space 482
Process Element 483
Work Descriptor (WD) 484
OS Virtual Address Space 485
Segment/Page Tables 486
Hypervisor Real Address Space 498
Process Element List 499
Accelerator Integration Slice 490
WD FETCH 491
Registers 445
MMU 439
Interrupt MGMT 447
Context MGMT 448
Save/Restore
INT 492
Effective Address 493
Graphics Acceleration 446

FIG. 4E

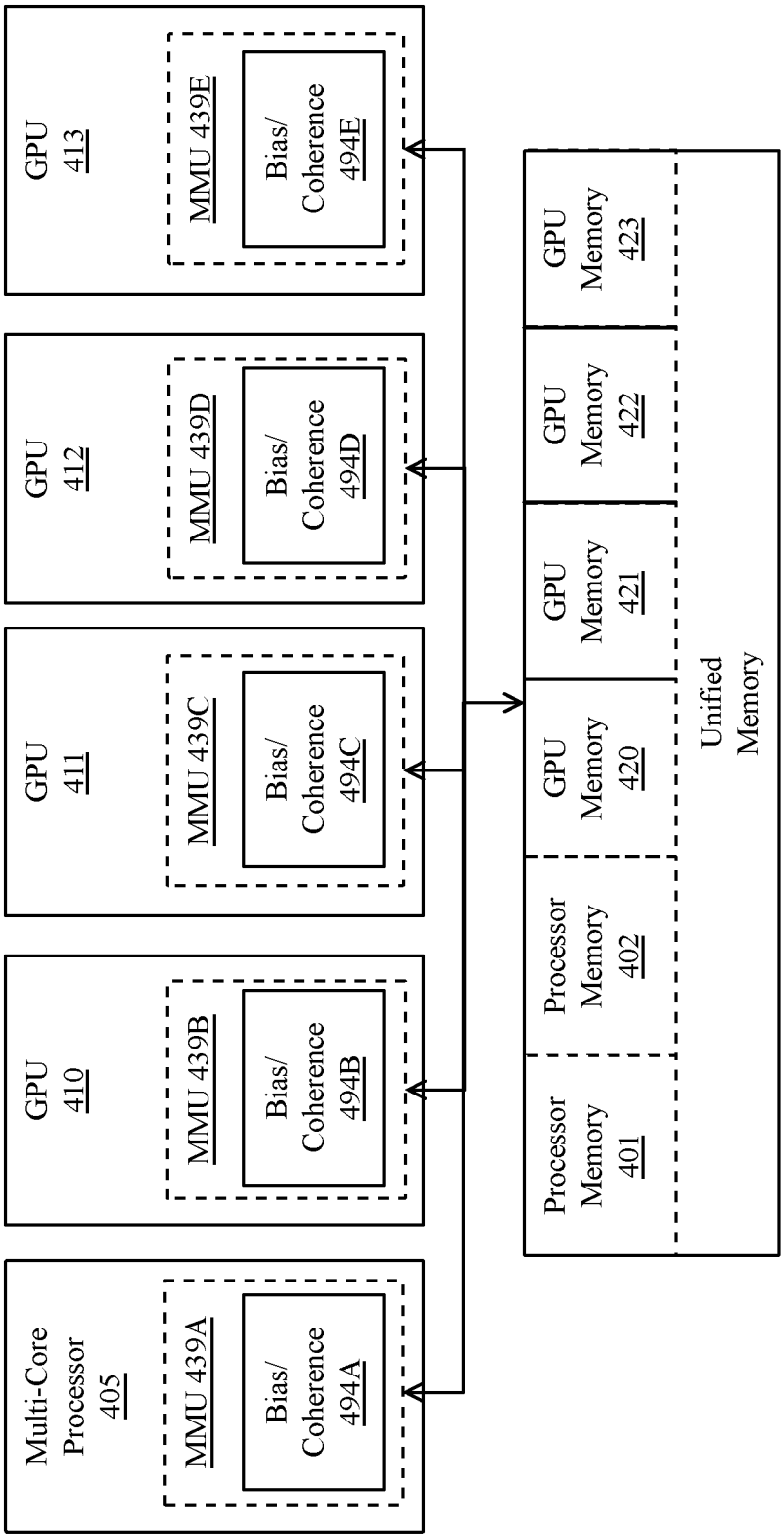

Multi-Core Processor 405
MMU 439A
Bias/ Coherence 494A
GPU 410
MMU 439B
Bias/ Coherence 494B
GPU 411
MMU 439C
Bias/ Coherence 494C
GPU 412
MMU 439D
Bias/ Coherence 494D
GPU 413
MMU 439E
Bias/ Coherence 494E
Processor Memory 401
Processor Memory 402
GPU Memory 420
GPU Memory 421
GPU Memory 422
GPU Memory 423
Unified Memory

FIG. 4F

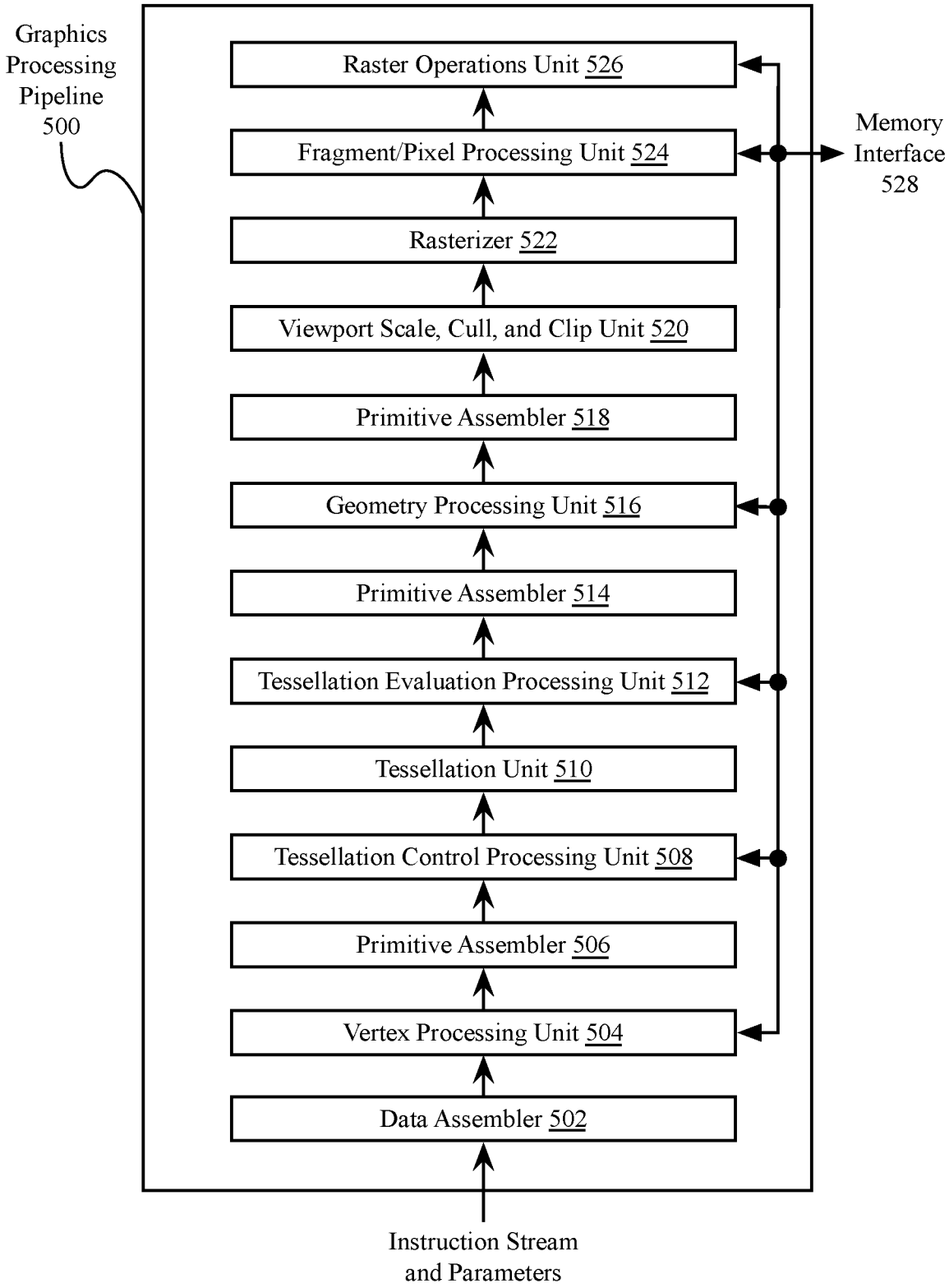

Graphics Processing Pipeline 500
Raster Operations Unit 526
Fragment/Pixel Processing Unit 524
Rasterizer 522
Viewport Scale, Cull, and Clip Unit 520
Primitive Assembler 518
Geometry Processing Unit 516
Primitive Assembler 514
Tessellation Evaluation Processing Unit 512
Tessellation Unit 510
Tessellation Control Processing Unit 508
Primitive Assembler 506
Vertex Processing Unit 504
Data Assembler 502
Memory Interface 528
Instruction Stream and Parameters

FIG. 5

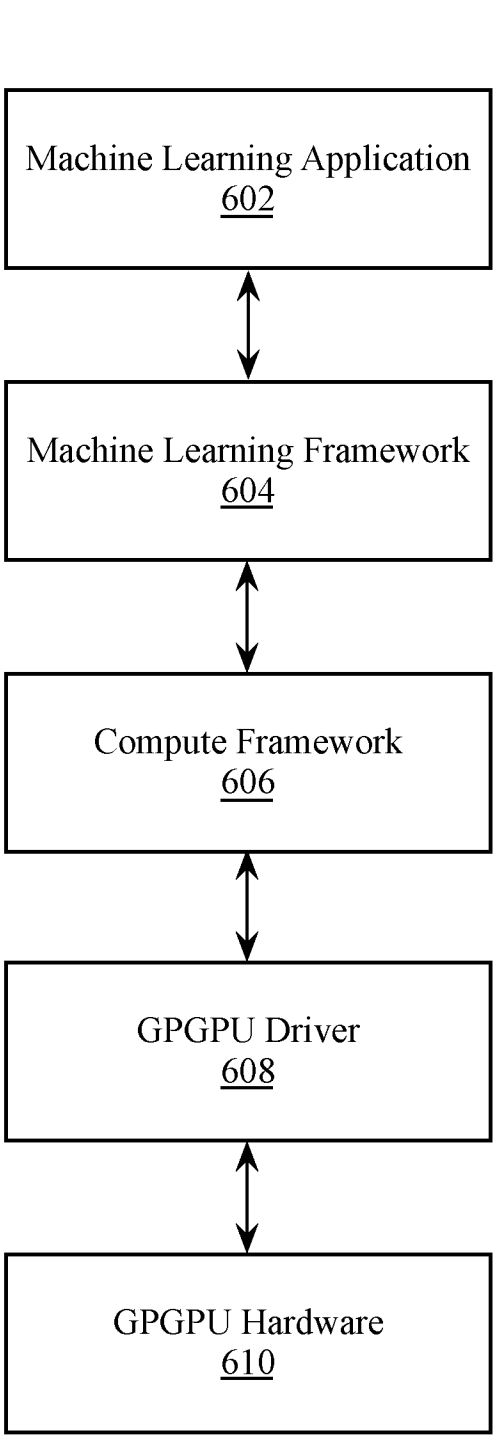

600
Machine Learning Application
602
Machine Learning Framework
604
Compute Framework
606
GPGPU Driver
608
GPGPU Hardware
610

FIG. 6

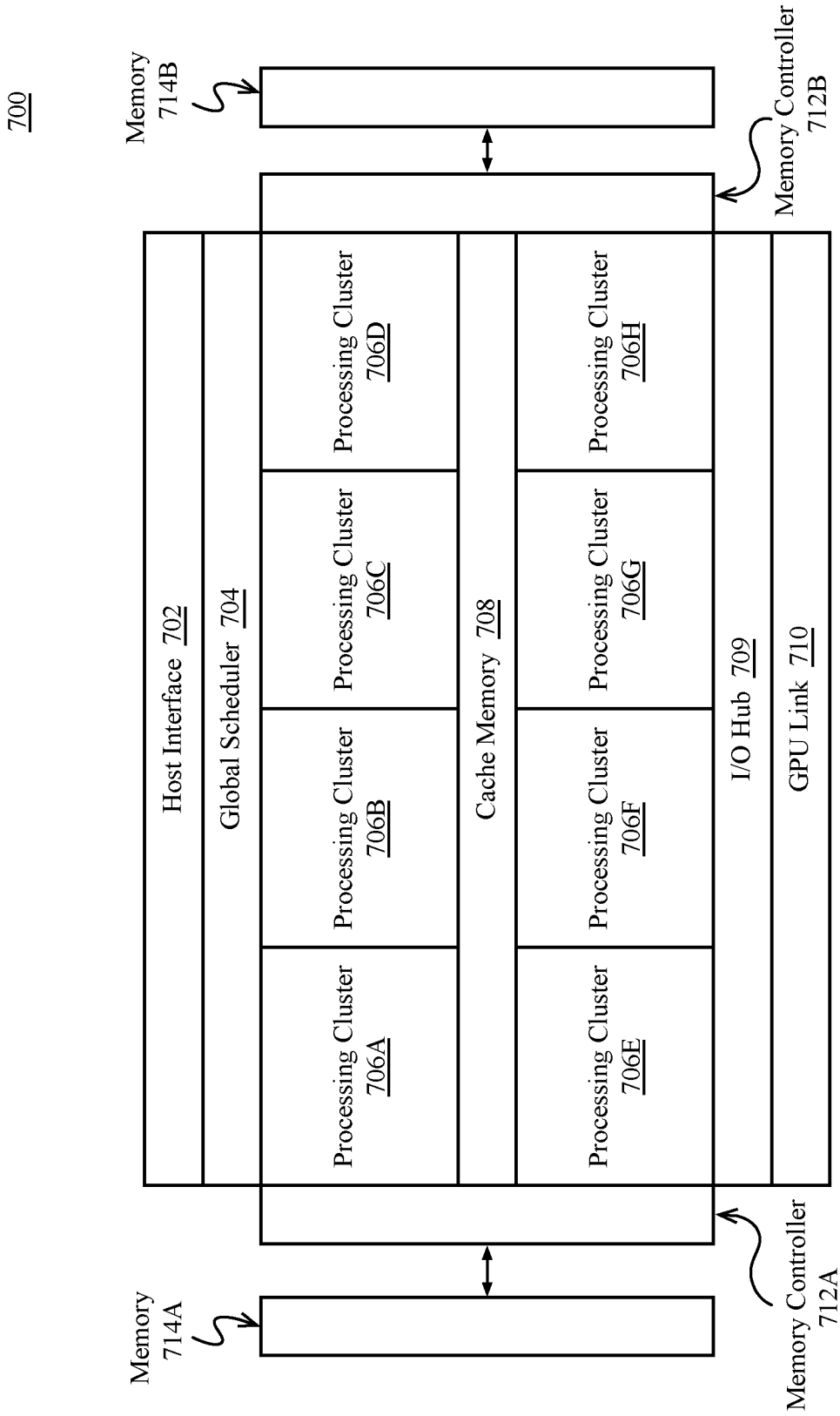

700
Memory 714B
Memory Controller 712B
Host Interface 702
Global Scheduler 704
Processing Cluster 706A
Processing Cluster 706B
Processing Cluster 706C
Processing Cluster 706D
Cache Memory 708
Processing Cluster 706E
Processing Cluster 706F
Processing Cluster 706G
Processing Cluster 706H
I/O Hub 709
GPU Link 710
Memory 714A
Memory Controller 712A

FIG. 7

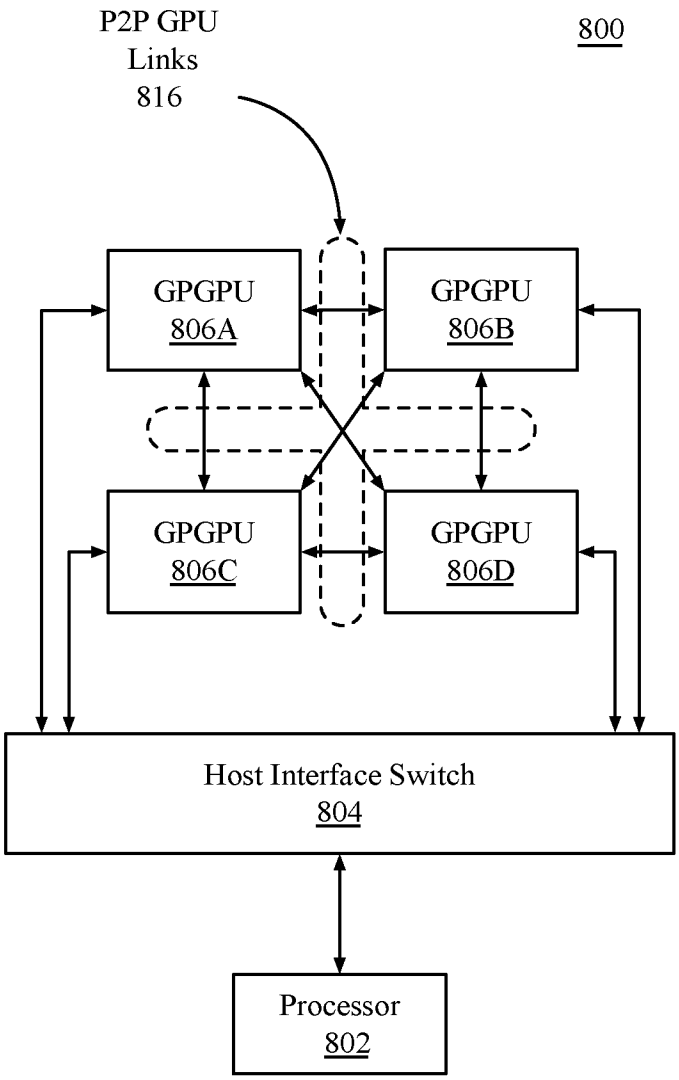

P2P GPU
Links
816
800
GPGPU
806A
GPGPU
806B
GPGPU
806C
GPGPU
806D
Host Interface Switch
804
Processor
802

FIG. 8

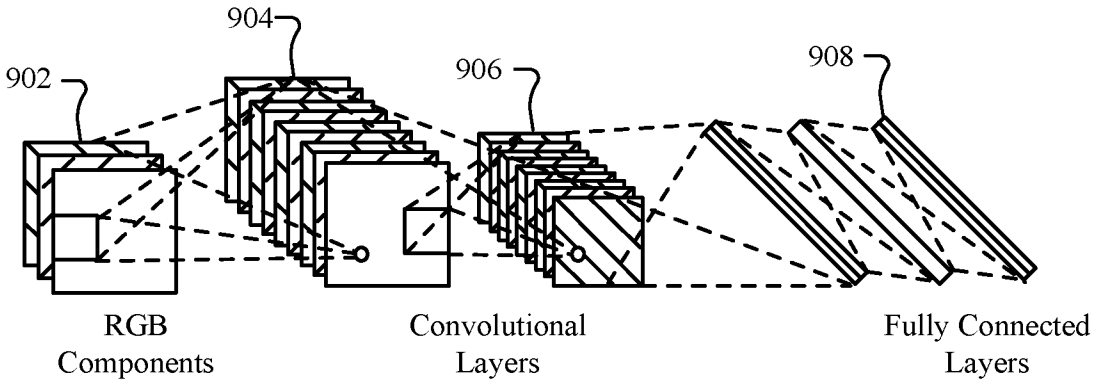

902
904
906
908
RGB Components
Convolutional Layers
Fully Connected Layers

FIG. 9A

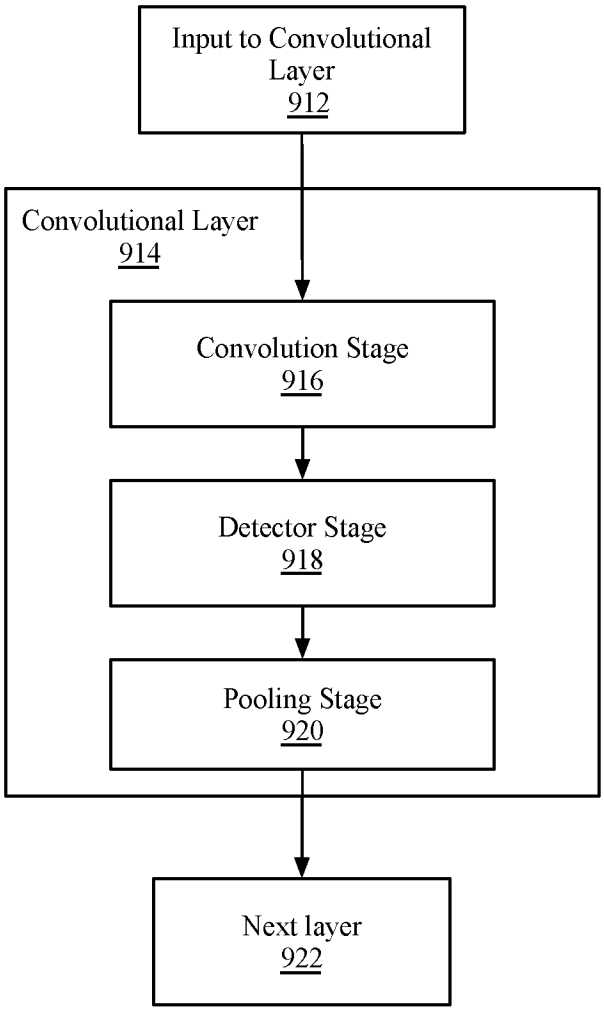

Input to Convolutional Layer
912
Convolutional Layer
914
Convolution Stage
916
Detector Stage
918
Pooling Stage
920
Next layer
922

FIG. 9B

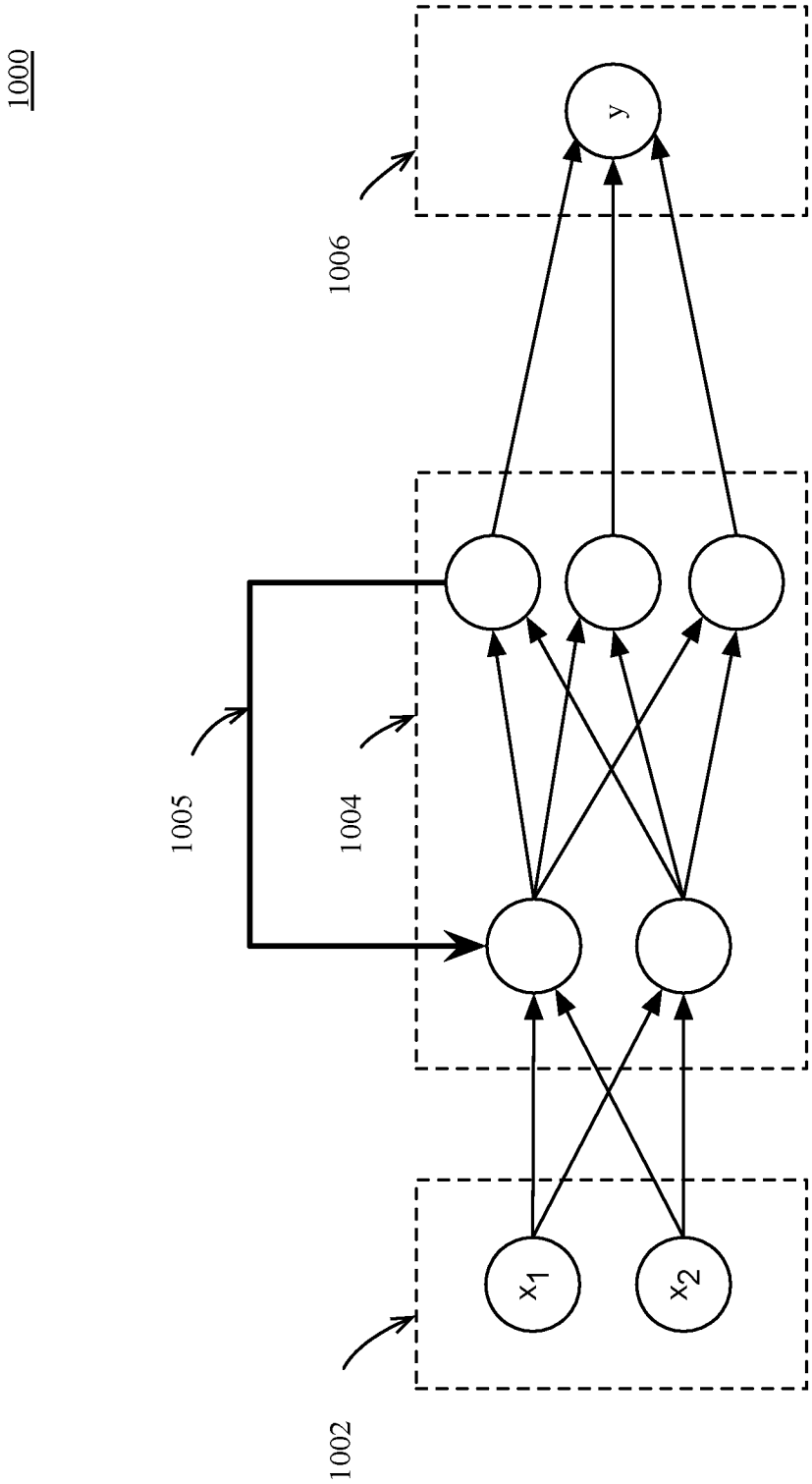


FIG. 10

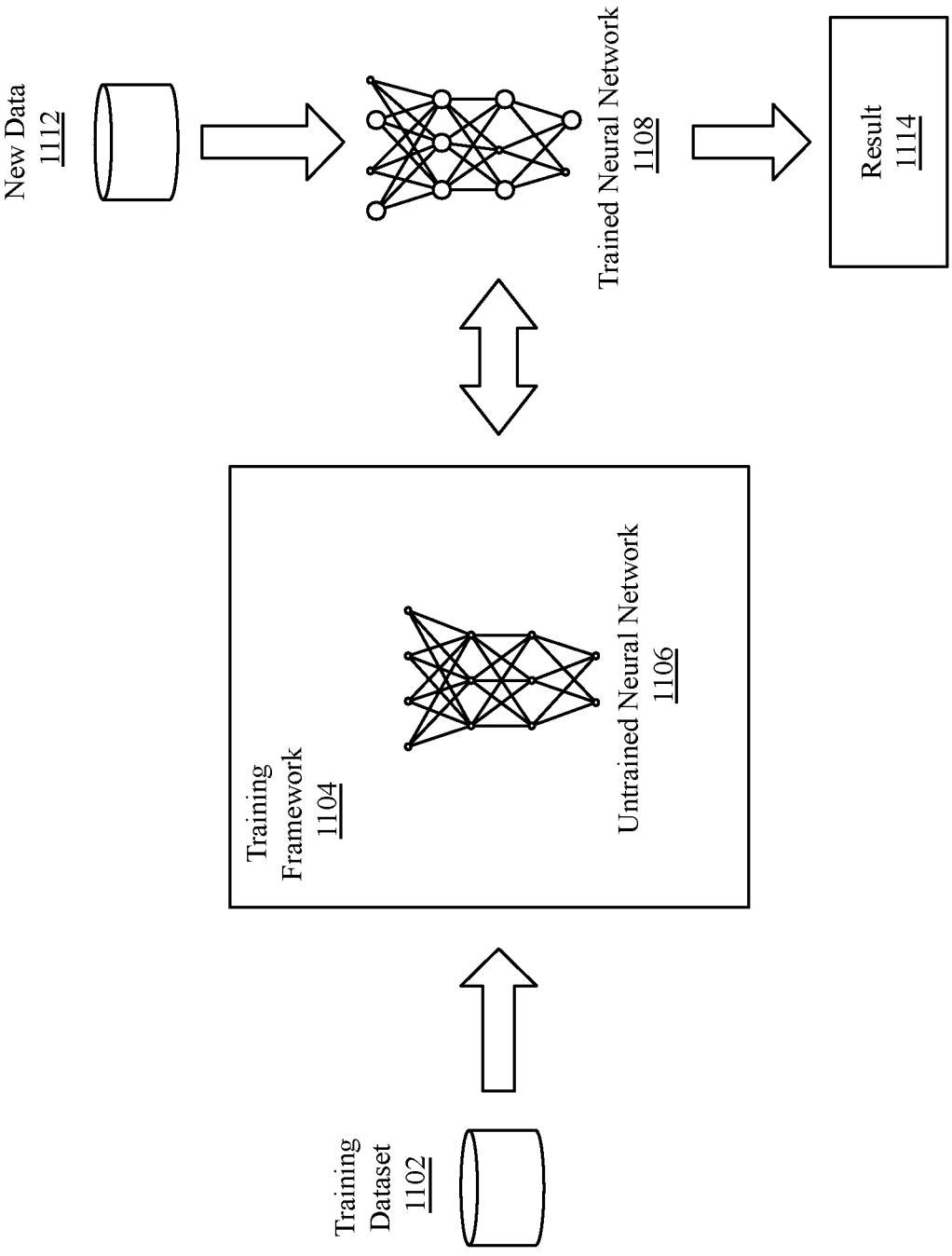

New Data
1112
Trained Neural Network
1108
Result
1114
Training
Framework
1104
Untrained Neural Network
1106
Training
Dataset
1102

FIG. 11

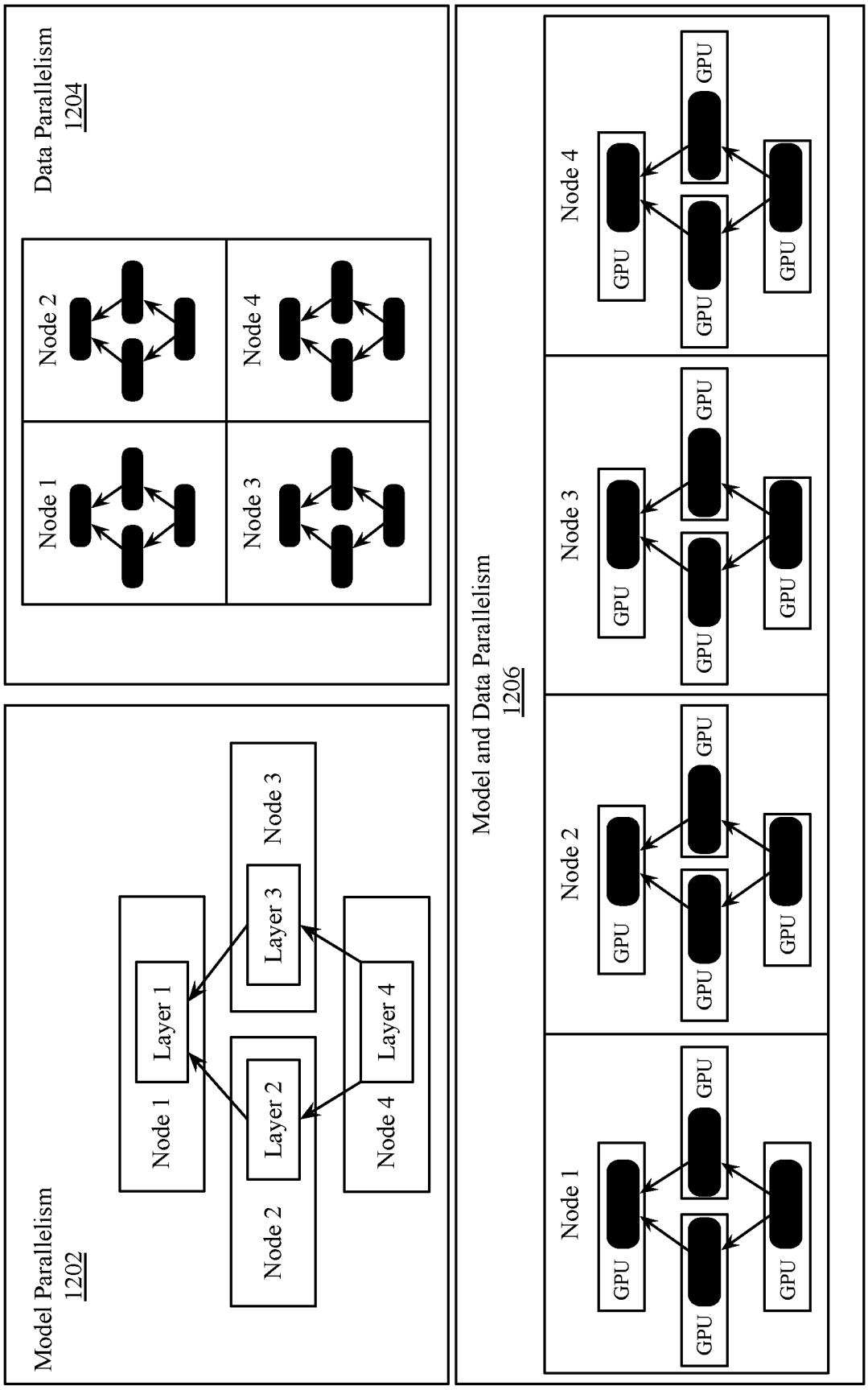

Model Parallelism
1202
Node 1
Layer 1
Node 2
Layer 2
Node 3
Layer 3
Node 4
Layer 4
Data Parallelism
1204
Node 1
Node 2
Node 3
Node 4
Model and Data Parallelism
1206
Node 1
Node 2
Node 3
Node 4
GPU

FIG. 12

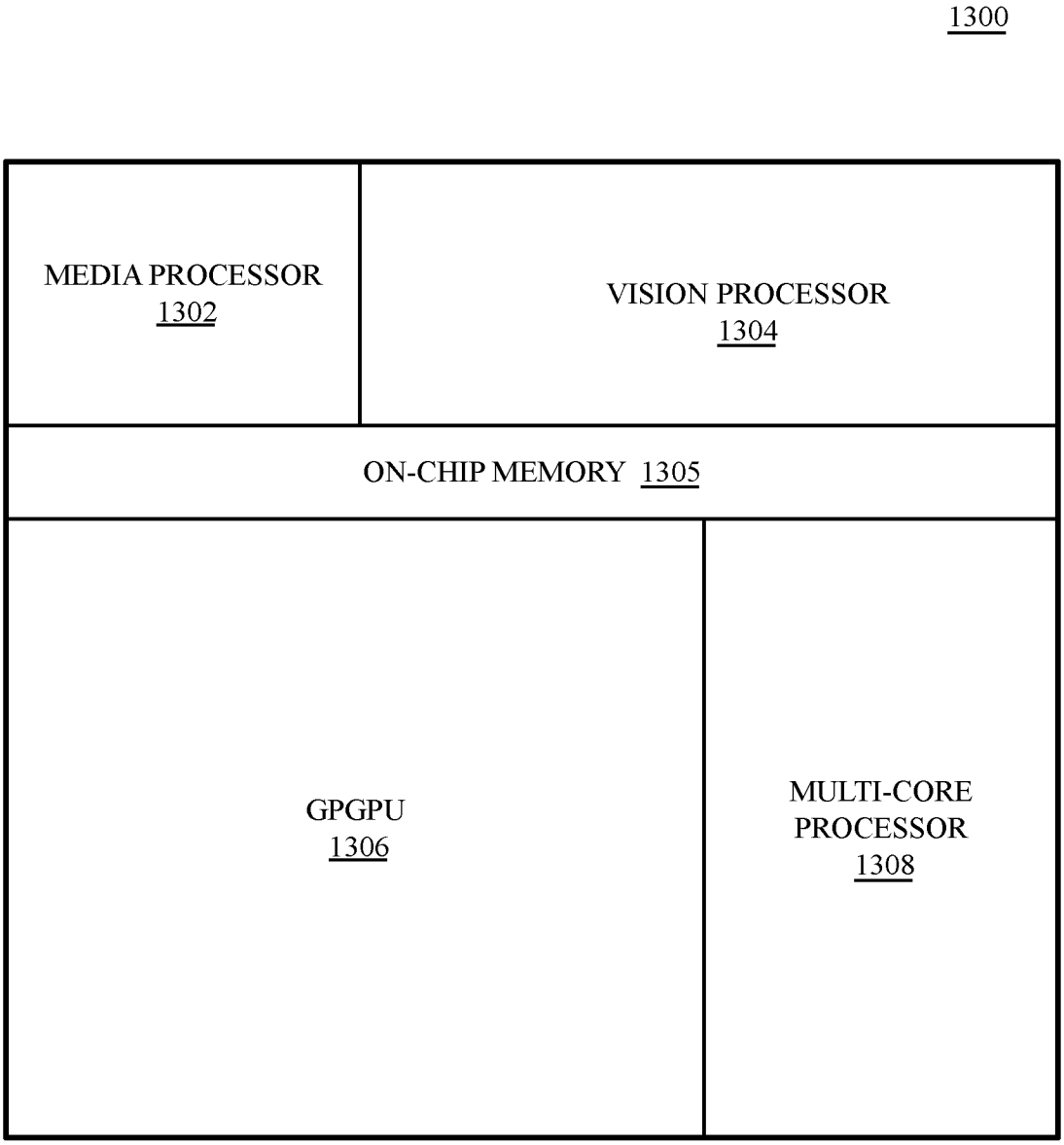

1300
MEDIA PROCESSOR
1302
VISION PROCESSOR
1304
ON-CHIP MEMORY 1305
GPGPU
1306
MULTI-CORE
PROCESSOR
1308

FIG. 13

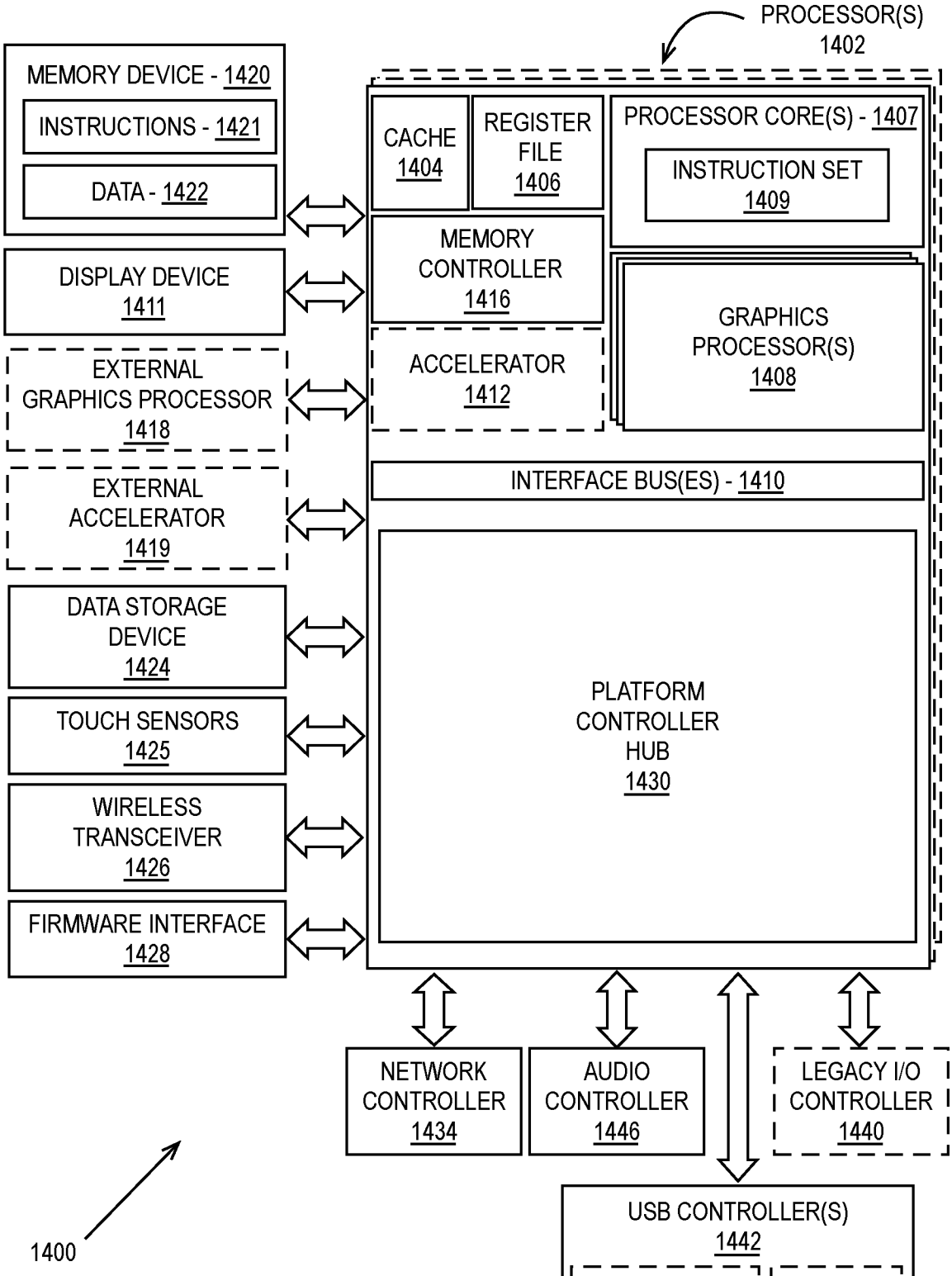

PROCESSOR(S)
1402
MEMORY DEVICE - 1420
INSTRUCTIONS - 1421
DATA - 1422
CACHE
1404
REGISTER
FILE
1406
PROCESSOR CORE(S) - 1407
INSTRUCTION SET
1409
MEMORY
CONTROLLER
1416
DISPLAY DEVICE
1411
GRAPHICS
PROCESSOR(S)
1408
ACCELERATOR
1412
EXTERNAL
GRAPHICS PROCESSOR
1418
INTERFACE BUS(ES) - 1410
EXTERNAL
ACCELERATOR
1419
DATA STORAGE
DEVICE
1424
PLATFORM
CONTROLLER
HUB
1430
TOUCH SENSORS
1425
WIRELESS
TRANSCEIVER
1426
FIRMWARE INTERFACE
1428
NETWORK
CONTROLLER
1434
AUDIO
CONTROLLER
1446
LEGACY I/O
CONTROLLER
1440
USB CONTROLLER(S)
1442
KEYBOARD/
MOUSE - 1443
CAMERA
1444
1400

FIG. 14

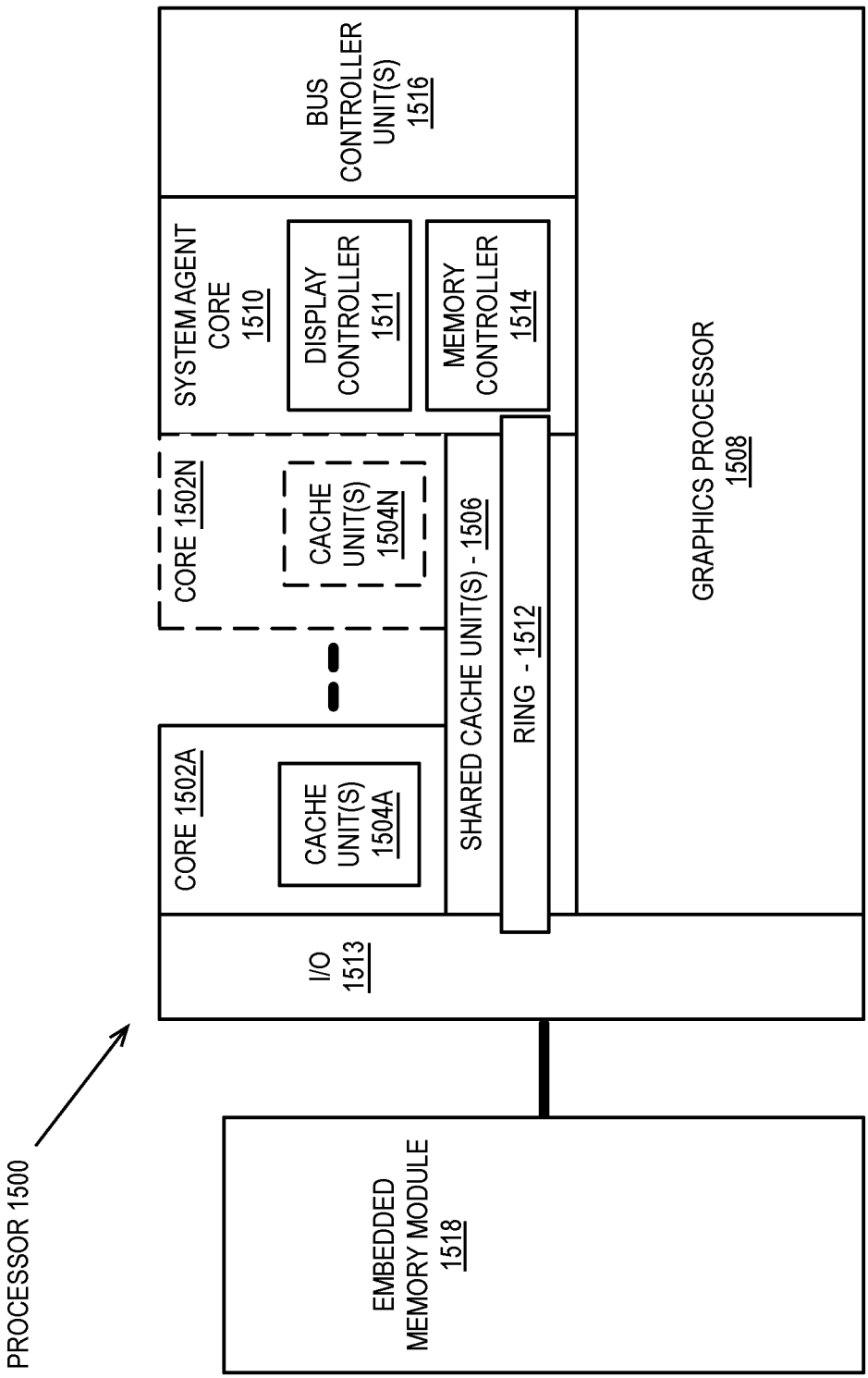

PROCESSOR 1500
EMBEDDED MEMORY MODULE 1518
I/O 1513
CORE 1502A
CACHE UNIT(S) 1504A
CORE 1502N
CACHE UNIT(S) 1504N
SHARED CACHE UNIT(S) - 1506
RING - 1512
GRAPHICS PROCESSOR 1508
SYSTEM AGENT CORE 1510
DISPLAY CONTROLLER 1511
MEMORY CONTROLLER 1514
BUS CONTROLLER UNIT(S) 1516

FIG. 15A

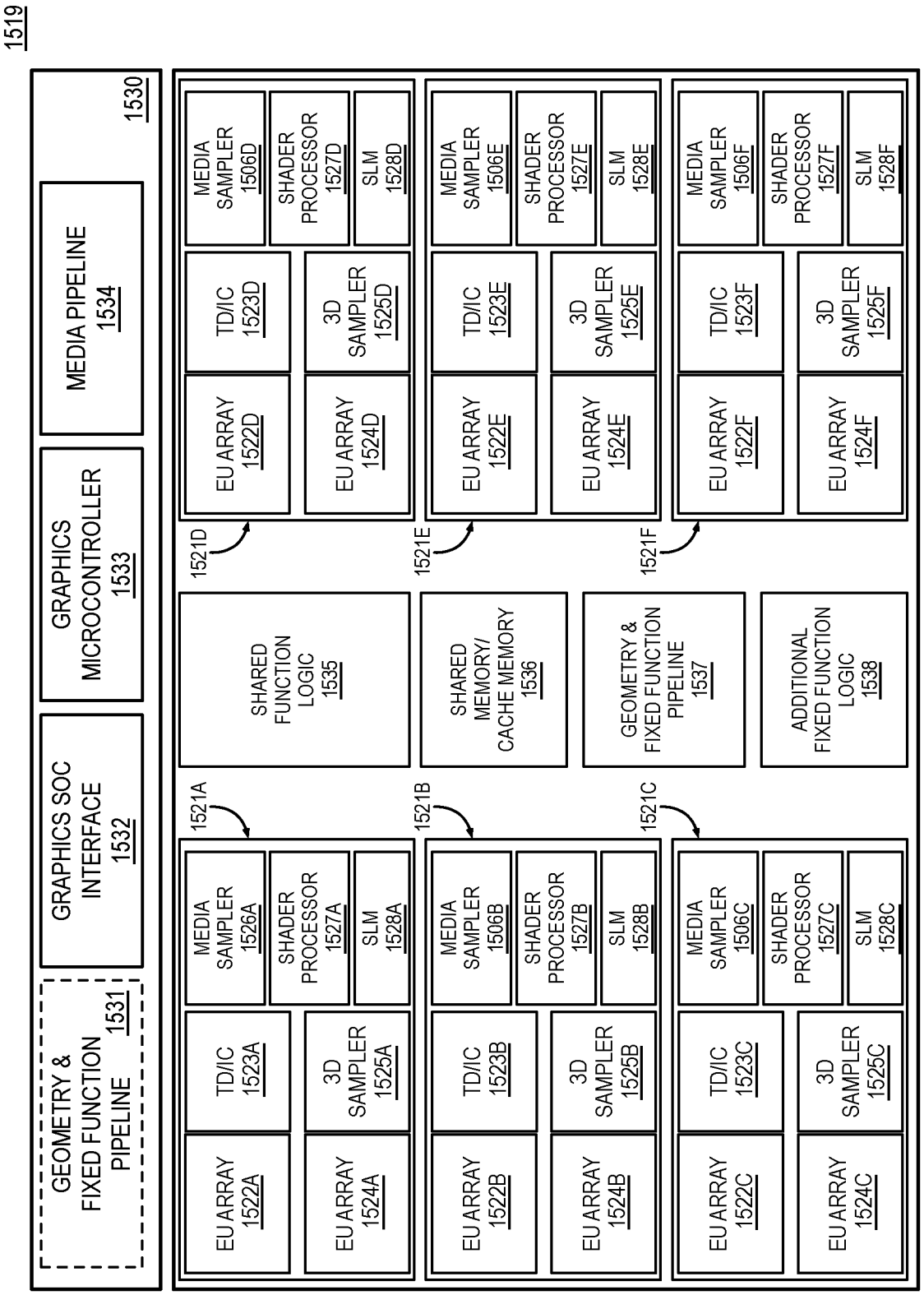

1519
GEOMETRY & FIXED FUNCTION PIPELINE 1531
GRAPHICS SOC INTERFACE 1532
GRAPHICS MICROCONTROLLER 1533
MEDIA PIPELINE 1534
1530
SHARED FUNCTION LOGIC 1535
SHARED MEMORY/ CACHE MEMORY 1536
GEOMETRY & FIXED FUNCTION PIPELINE 1537
ADDITIONAL FIXED FUNCTION LOGIC 1538
1521A
EU ARRAY 1522A
TD/IC 1523A
3D SAMPLER 1525A
EU ARRAY 1524A
MEDIA SAMPLER 1526A
SHADER PROCESSOR 1527A
SLM 1528A
1521B
EU ARRAY 1522B
TD/IC 1523B
3D SAMPLER 1525B
EU ARRAY 1524B
MEDIA SAMPLER 1506B
SHADER PROCESSOR 1527B
SLM 1528B
1521C
EU ARRAY 1522C
TD/IC 1523C
3D SAMPLER 1525C
EU ARRAY 1524C
MEDIA SAMPLER 1506C
SHADER PROCESSOR 1527C
SLM 1528C
1521D
EU ARRAY 1522D
TD/IC 1523D
3D SAMPLER 1525D
EU ARRAY 1524D
MEDIA SAMPLER 1506D
SHADER PROCESSOR 1527D
SLM 1528D
1521E
EU ARRAY 1522E
TD/IC 1523E
3D SAMPLER 1525E
EU ARRAY 1524E
MEDIA SAMPLER 1506E
SHADER PROCESSOR 1527E
SLM 1528E
1521F
EU ARRAY 1522F
TD/IC 1523F
3D SAMPLER 1525F
EU ARRAY 1524F
MEDIA SAMPLER 1506F
SHADER PROCESSOR 1527F
SLM 1528F

FIG. 15B

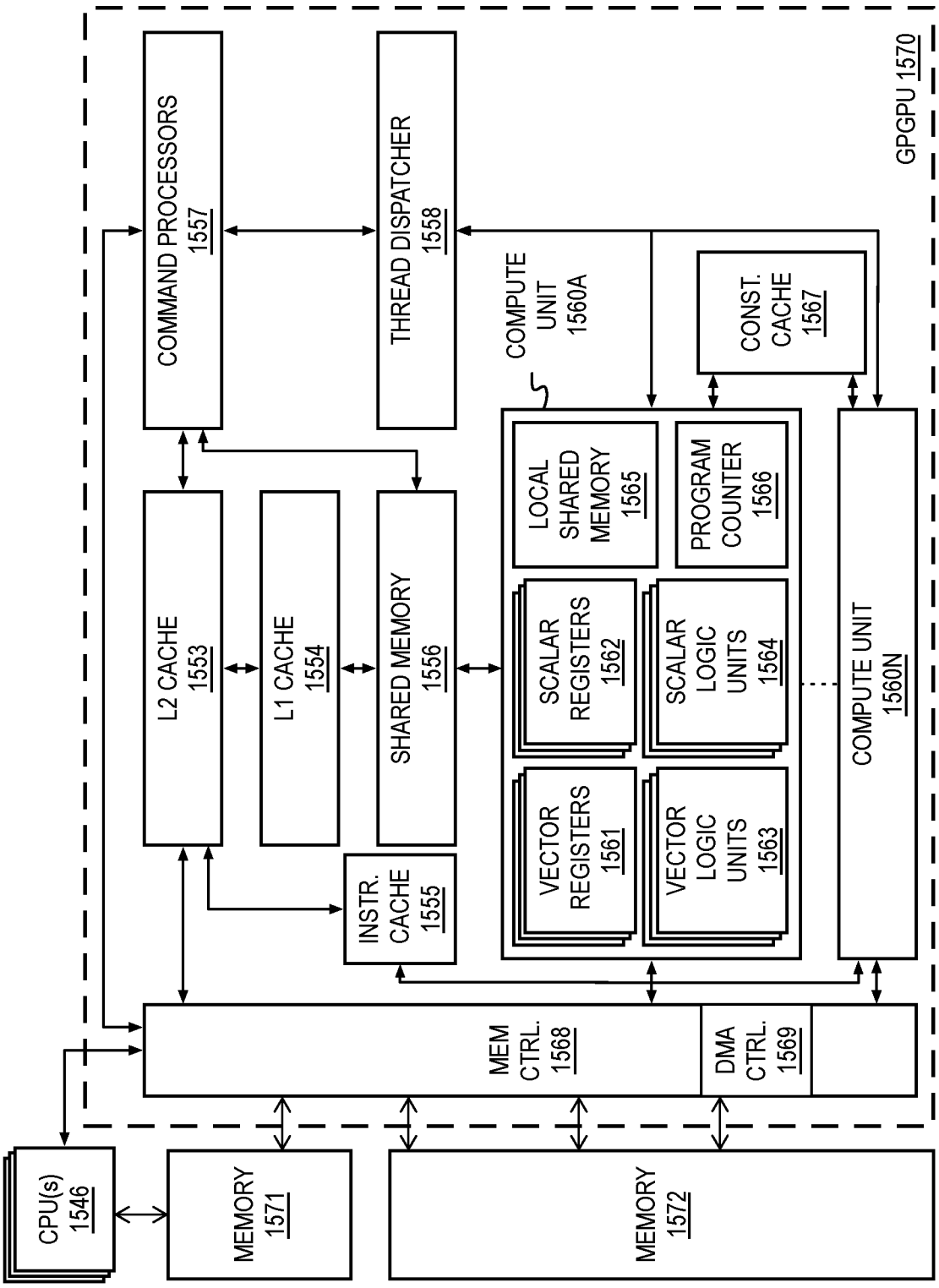

GPGPU 1570
COMMAND PROCESSORS 1557
THREAD DISPATCHER 1558
COMPUTE UNIT 1560A
CONST. CACHE 1567
LOCAL SHARED MEMORY 1565
PROGRAM COUNTER 1566
L2 CACHE 1553
L1 CACHE 1554
SHARED MEMORY 1556
SCALAR REGISTERS 1562
SCALAR LOGIC UNITS 1564
VECTOR REGISTERS 1561
VECTOR LOGIC UNITS 1563
COMPUTE UNIT 1560N
INSTR. CACHE 1555
MEM CTRL. 1568
DMA CTRL. 1569
CPU(s) 1546
MEMORY 1571
MEMORY 1572

FIG. 15C

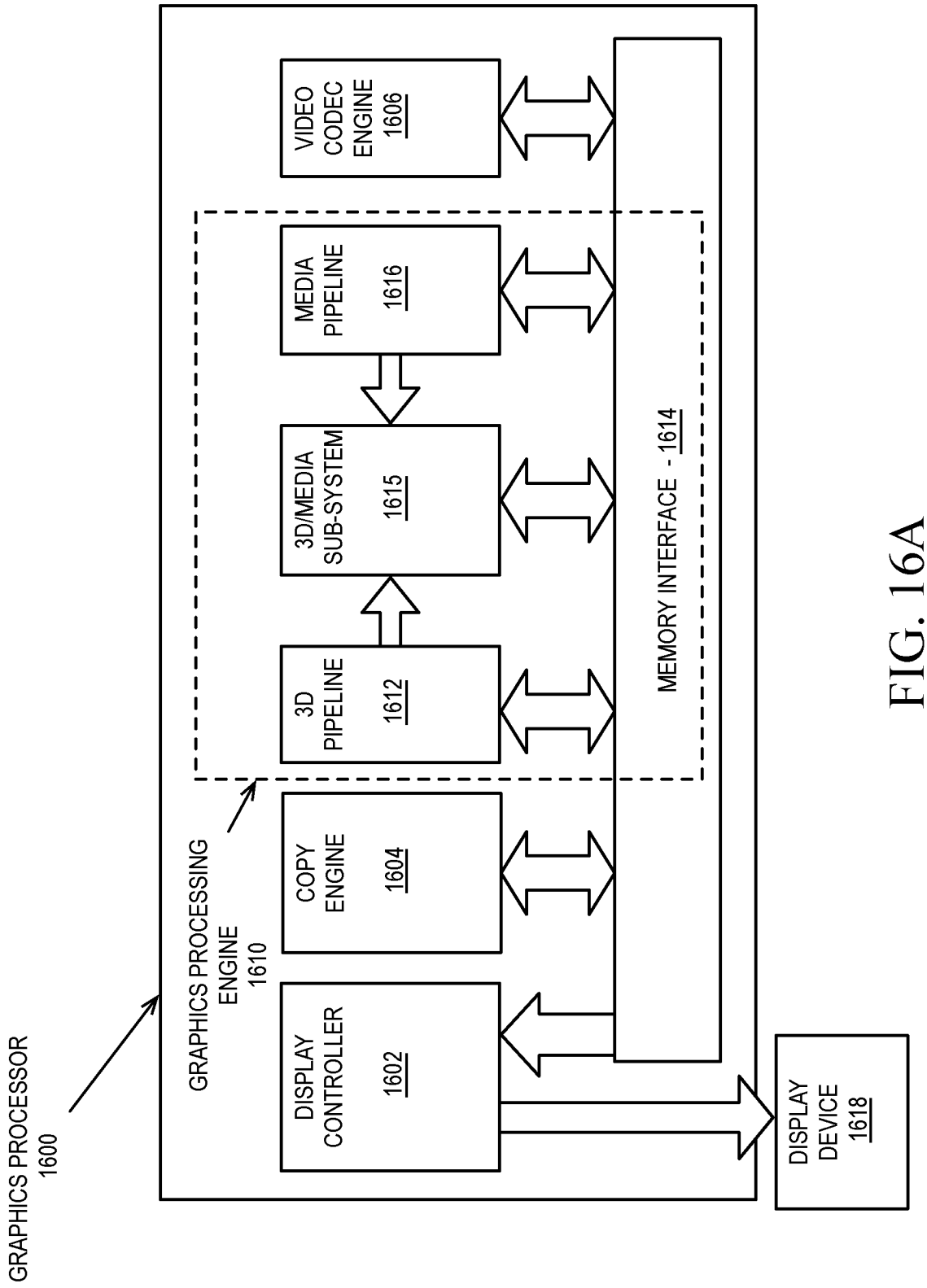

GRAPHICS PROCESSOR
1600
GRAPHICS PROCESSING
ENGINE
1610
DISPLAY
CONTROLLER
1602
COPY
ENGINE
1604
3D
PIPELINE
1612
3D/MEDIA
SUB-SYSTEM
1615
MEDIA
PIPELINE
1616
VIDEO
CODEC
ENGINE
1606
MEMORY INTERFACE - 1614
DISPLAY
DEVICE
1618

FIG. 16A

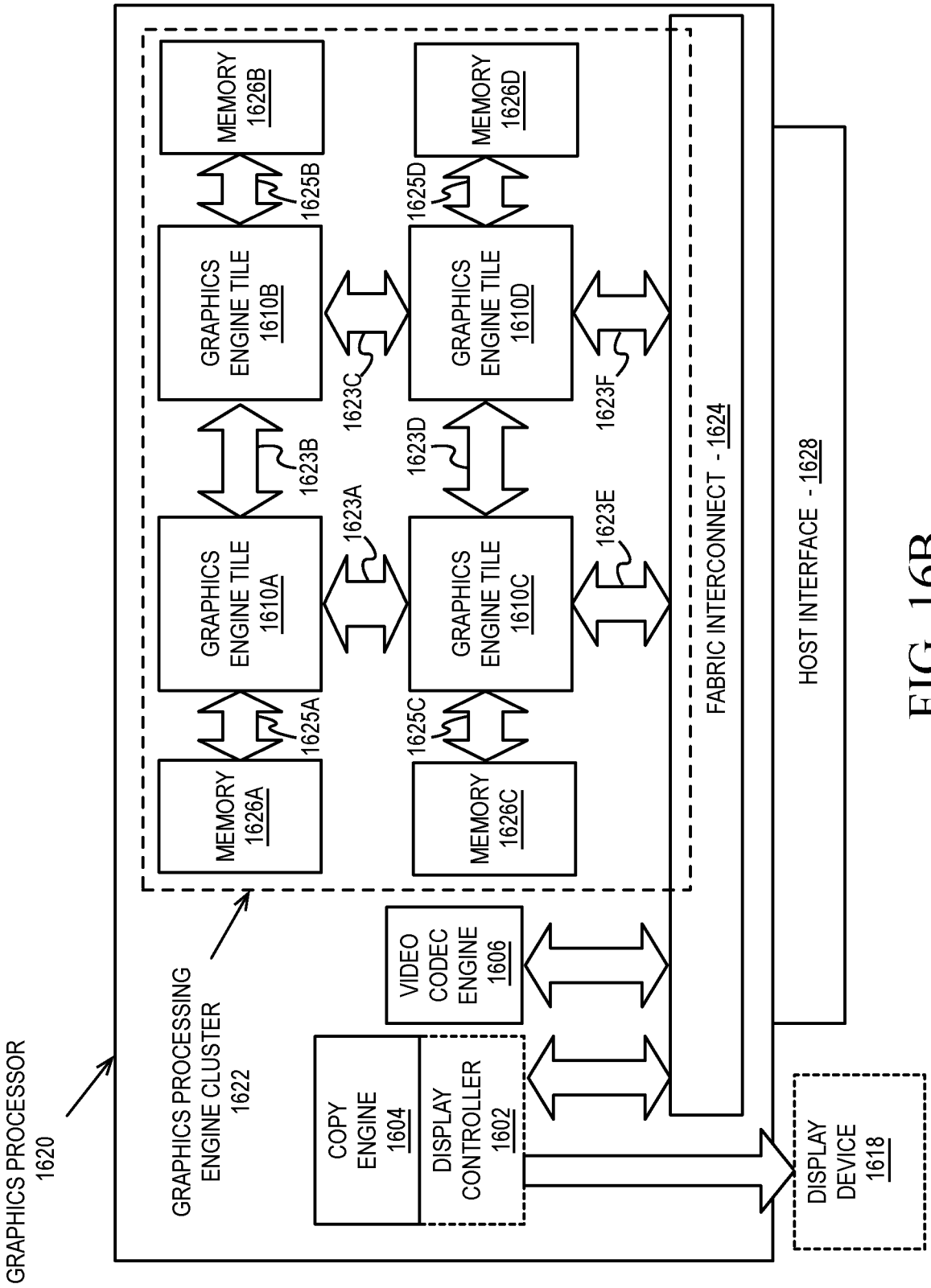

GRAPHICS PROCESSOR 1620
GRAPHICS PROCESSING ENGINE CLUSTER 1622
GRAPHICS ENGINE TILE 1610A
GRAPHICS ENGINE TILE 1610B
GRAPHICS ENGINE TILE 1610C
GRAPHICS ENGINE TILE 1610D
MEMORY 1626A
MEMORY 1626B
MEMORY 1626C
MEMORY 1626D
1625A
1625B
1625C
1625D
1623A
1623B
1623C
1623D
1623E
1623F
FABRIC INTERCONNECT - 1624
HOST INTERFACE - 1628
VIDEO CODEC ENGINE 1606
COPY ENGINE 1604
DISPLAY CONTROLLER 1602
DISPLAY DEVICE 1618

FIG. 16B

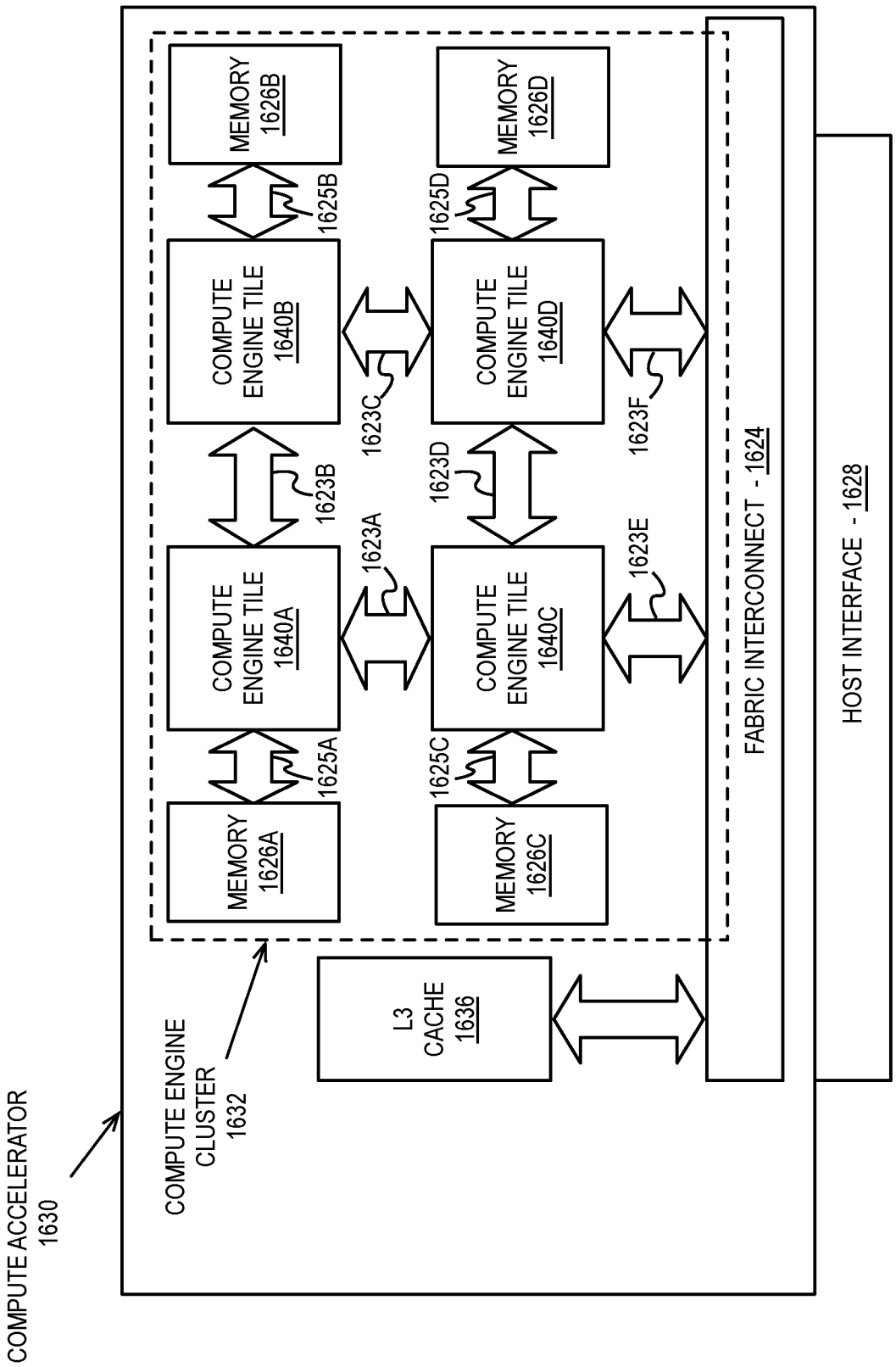

COMPUTE ACCELERATOR
1630
COMPUTE ENGINE
CLUSTER
1632
MEMORY
1626A
MEMORY
1626B
MEMORY
1626C
MEMORY
1626D
COMPUTE
ENGINE TILE
1640A
COMPUTE
ENGINE TILE
1640B
COMPUTE
ENGINE TILE
1640C
COMPUTE
ENGINE TILE
1640D
1625A
1625B
1625C
1625D
1623A
1623B
1623C
1623D
1623E
1623F
L3
CACHE
1636
FABRIC INTERCONNECT - 1624
HOST INTERFACE - 1628

FIG. 16C

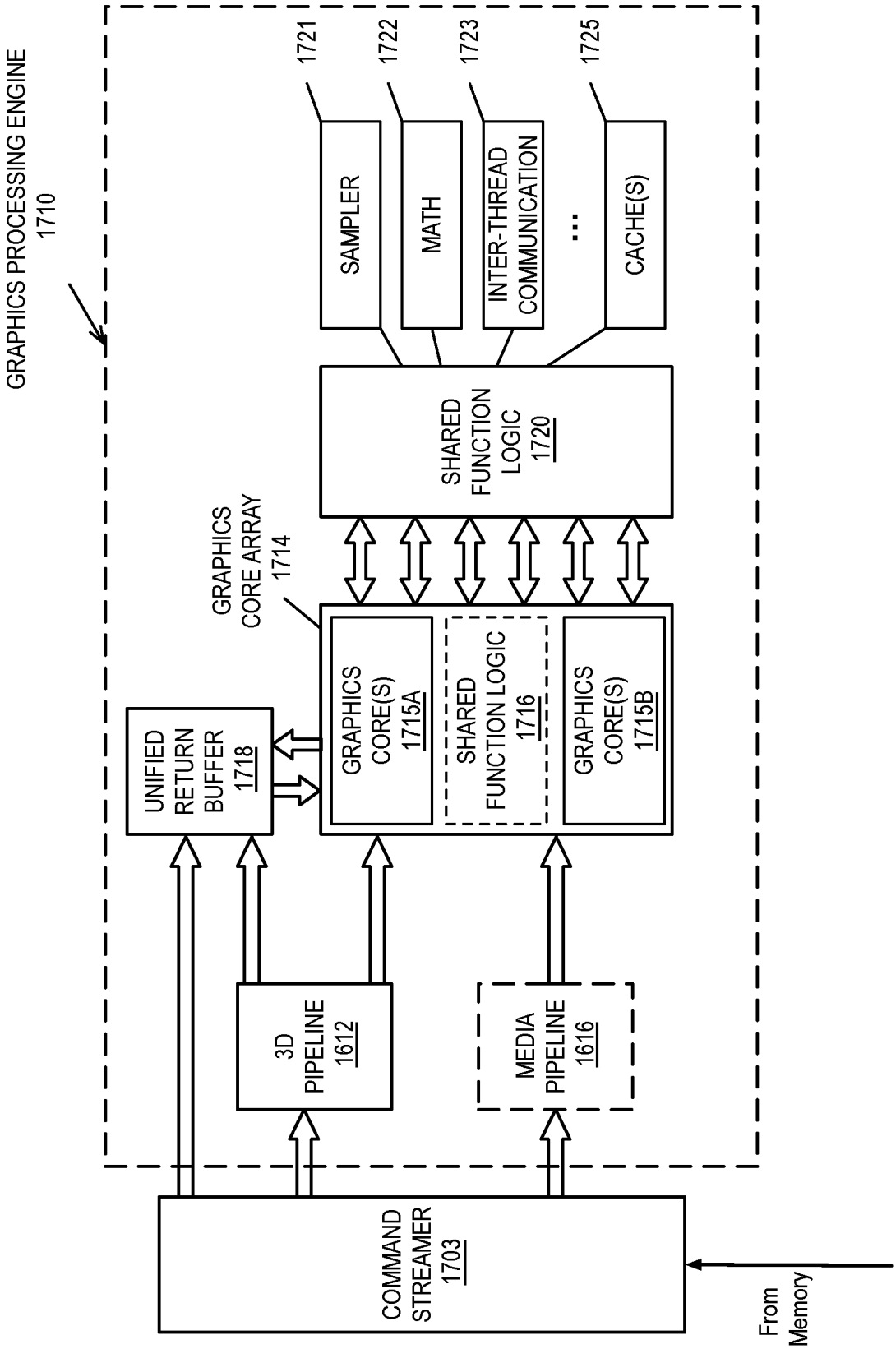

GRAPHICS PROCESSING ENGINE
1710
1721
1722
1723
1725
SAMPLER
MATH
INTER-THREAD COMMUNICATION
...
CACHE(S)
SHARED FUNCTION LOGIC 1720
GRAPHICS CORE ARRAY 1714
GRAPHICS CORE(S) 1715A
SHARED FUNCTION LOGIC 1716
GRAPHICS CORE(S) 1715B
UNIFIED RETURN BUFFER 1718
3D PIPELINE 1612
MEDIA PIPELINE 1616
COMMAND STREAMER 1703
From Memory

FIG. 17

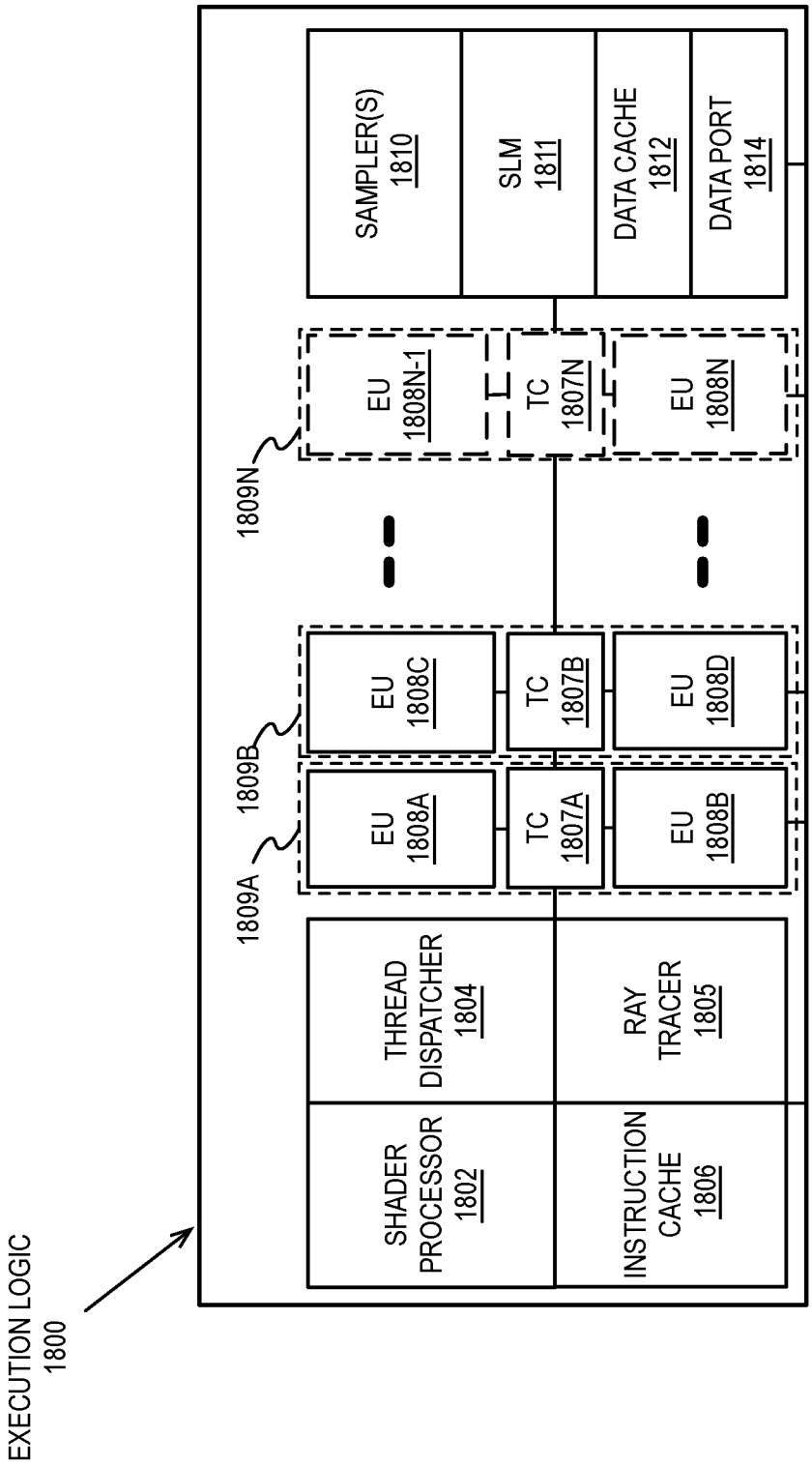

EXECUTION LOGIC
1800
SHADER PROCESSOR 1802
THREAD DISPATCHER 1804
INSTRUCTION CACHE 1806
RAY TRACER 1805
1809A
1809B
1809N
EU 1808A
TC 1807A
EU 1808B
EU 1808C
TC 1807B
EU 1808D
EU 1808N-1
TC 1807N
EU 1808N
SAMPLER(S) 1810
SLM 1811
DATA CACHE 1812
DATA PORT 1814

FIG. 18A

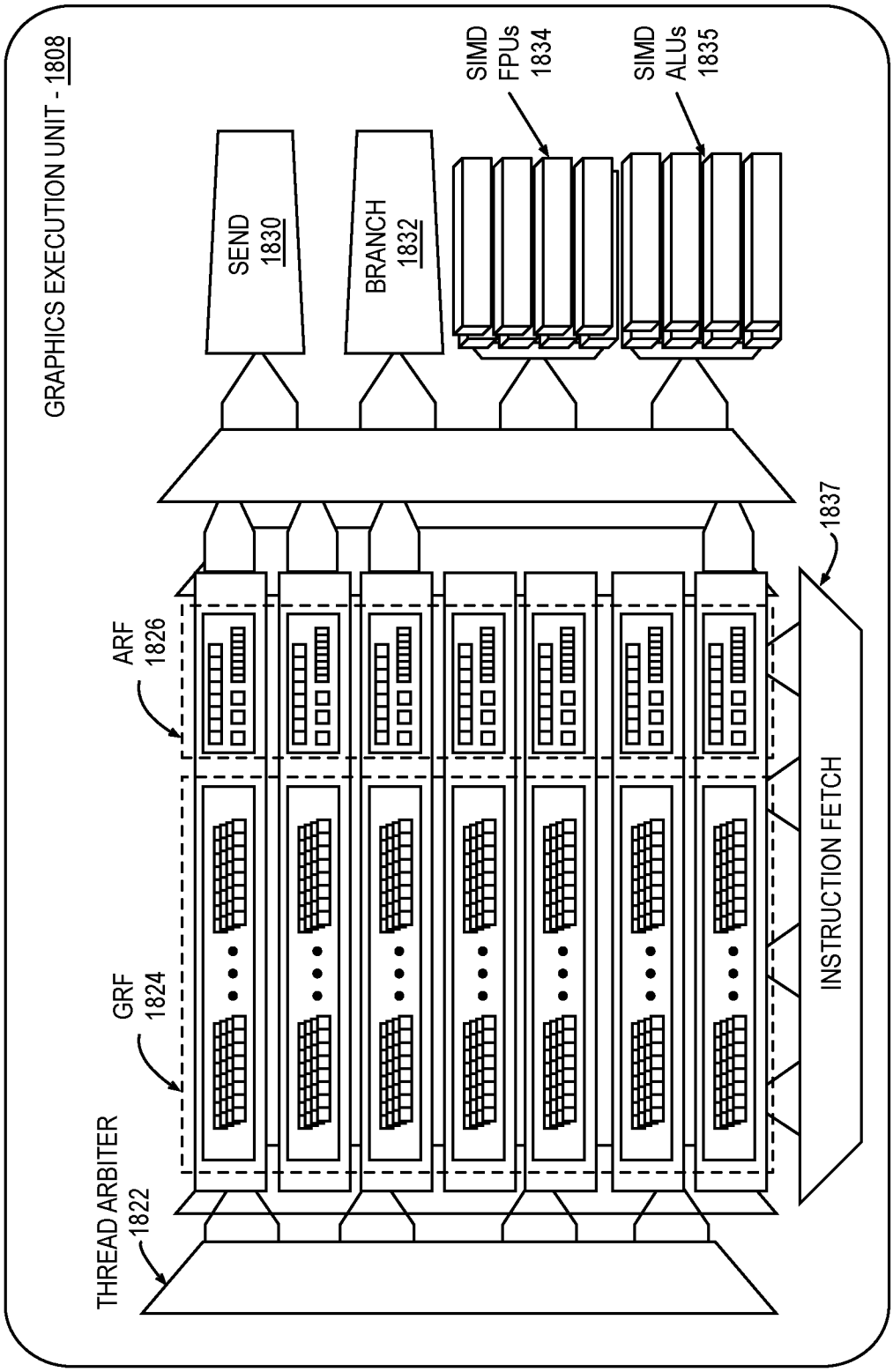

GRAPHICS EXECUTION UNIT - 1808
SEND 1830
BRANCH 1832
SIMD FPUs 1834
SIMD ALUs 1835
1837
INSTRUCTION FETCH
ARF 1826
GRF 1824
THREAD ARBITER 1822

FIG. 18B

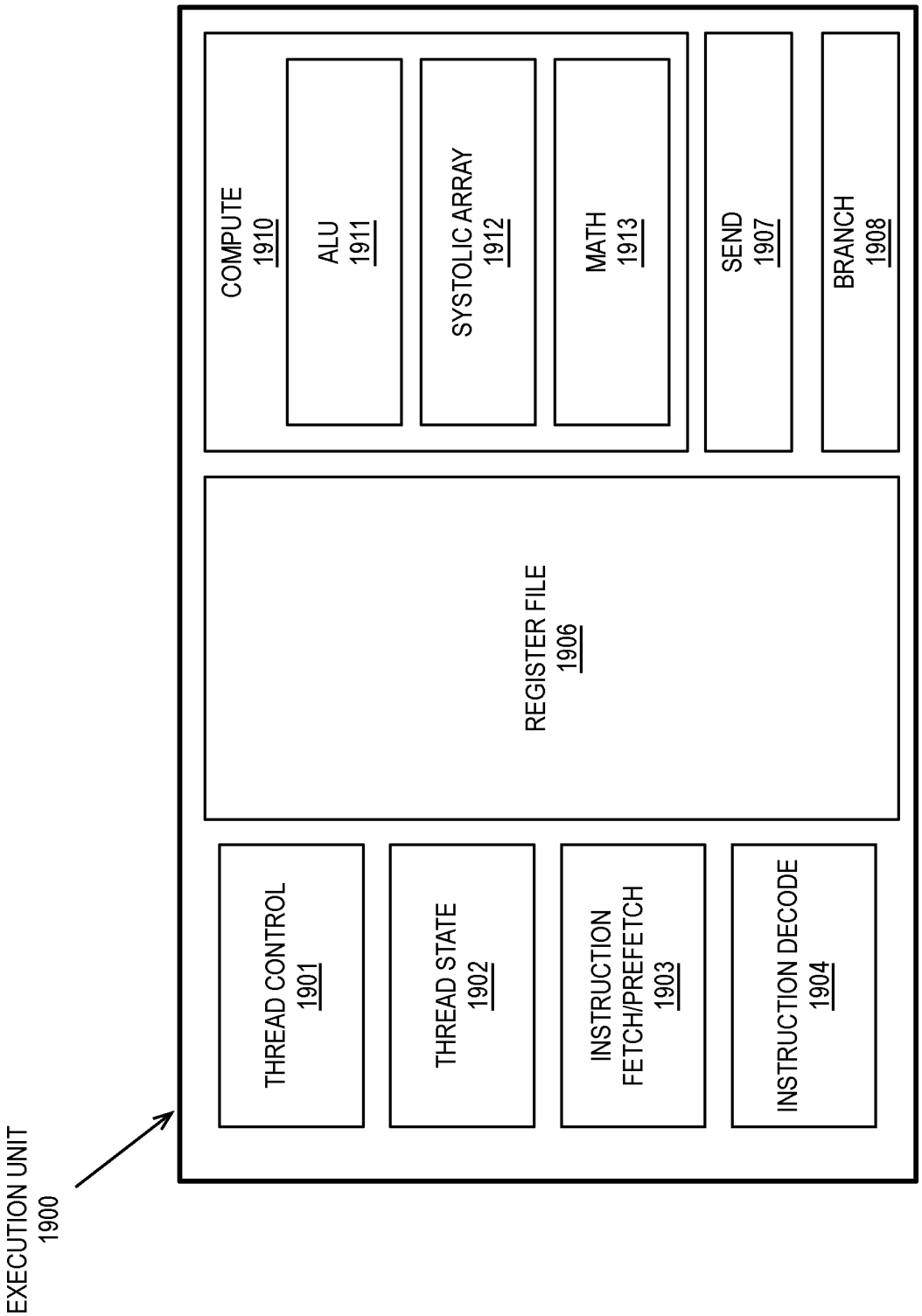

EXECUTION UNIT
1900
THREAD CONTROL
1901
THREAD STATE
1902
INSTRUCTION
FETCH/PREFETCH
1903
INSTRUCTION DECODE
1904
REGISTER FILE
1906
COMPUTE
1910
ALU
1911
SYSTOLIC ARRAY
1912
MATH
1913
SEND
1907
BRANCH
1908

FIG. 19

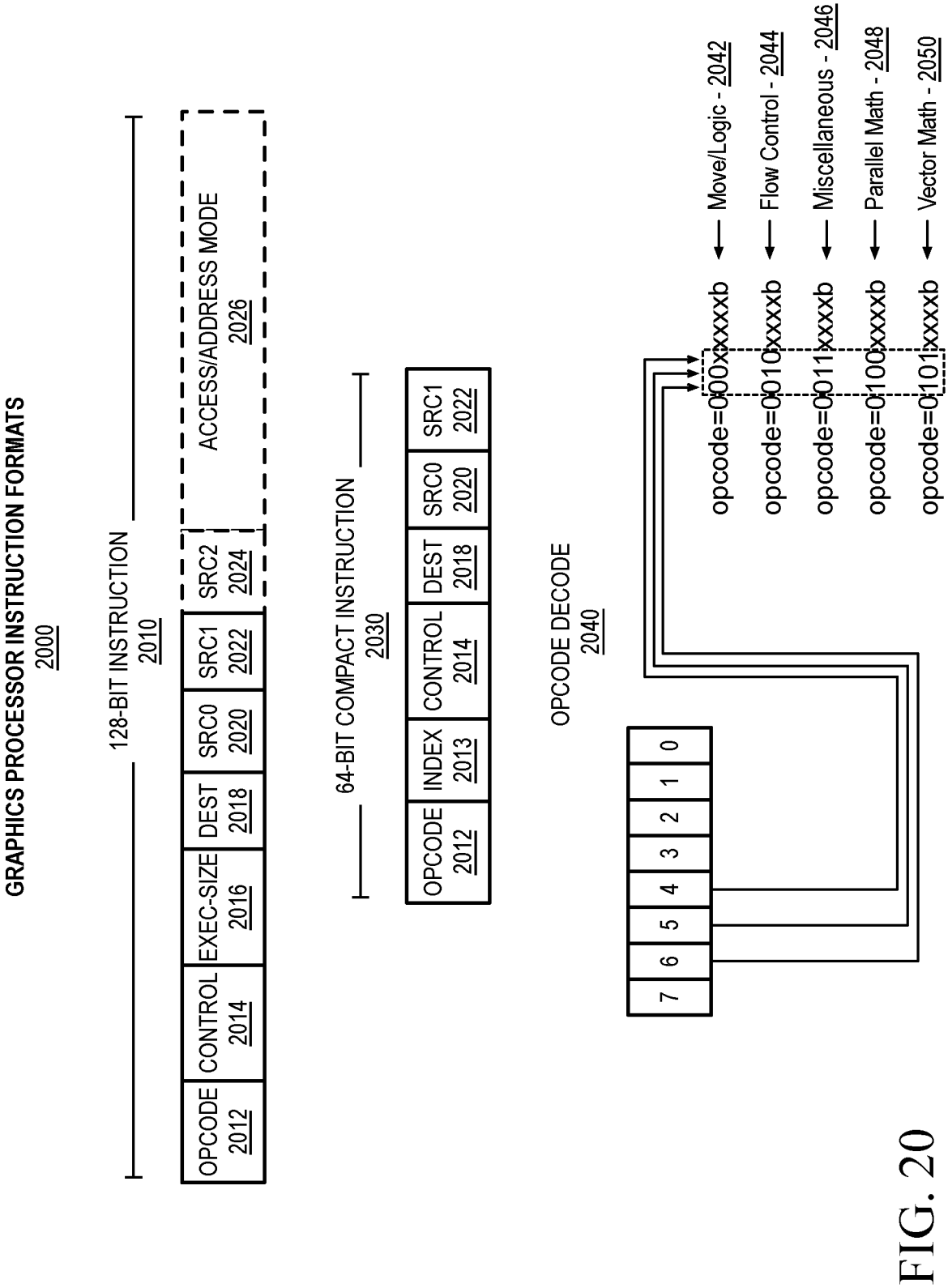

GRAPHICS PROCESSOR INSTRUCTION FORMATS
2000
128-BIT INSTRUCTION
2010
OPCODE 2012
CONTROL 2014
EXEC-SIZE 2016
DEST 2018
SRC0 2020
SRC1 2022
SRC2 2024
ACCESS/ADDRESS MODE 2026
64-BIT COMPACT INSTRUCTION
2030
OPCODE 2012
INDEX 2013
CONTROL 2014
DEST 2018
SRC0 2020
SRC1 2022
OPCODE DECODE
2040
7 6 5 4 3 2 1 0
opcode=000xxxxb
opcode=001xxxxb
opcode=011xxxxb
opcode=0100xxxxb
opcode=0101xxxxb
Move/Logic - 2042
Flow Control - 2044
Miscellaneous - 2046
Parallel Math - 2048
Vector Math - 2050

FIG. 20

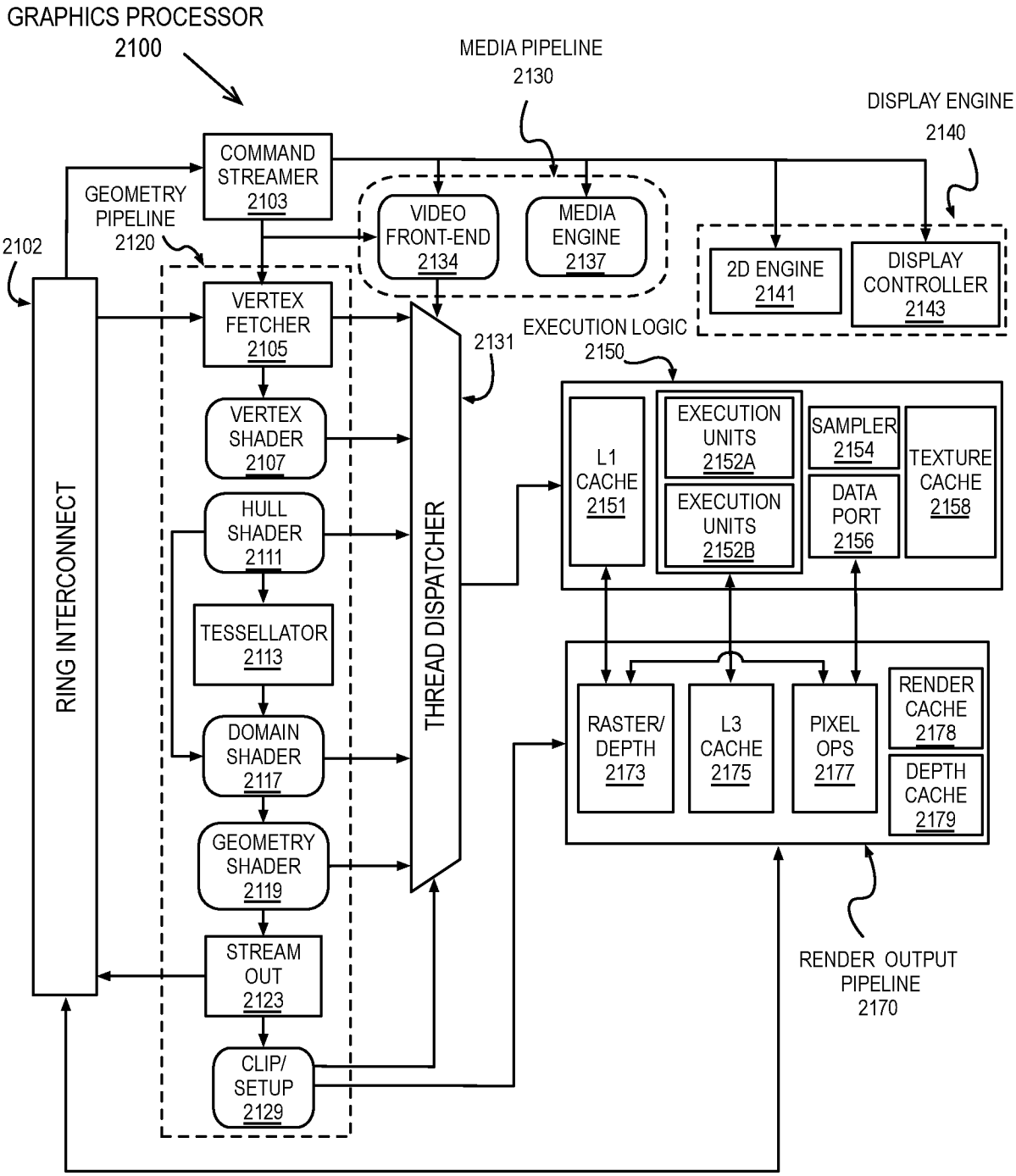

GRAPHICS PROCESSOR
2100
MEDIA PIPELINE
2130
DISPLAY ENGINE
2140
COMMAND STREAMER
2103
GEOMETRY PIPELINE
2120
2102
VIDEO FRONT-END
2134
MEDIA ENGINE
2137
2D ENGINE
2141
DISPLAY CONTROLLER
2143
VERTEX FETCHER
2105
2131
EXECUTION LOGIC
2150
VERTEX SHADER
2107
HULL SHADER
2111
TESSELLATOR
2113
DOMAIN SHADER
2117
GEOMETRY SHADER
2119
STREAM OUT
2123
CLIP/ SETUP
2129
RING INTERCONNECT
THREAD DISPATCHER
L1 CACHE
2151
EXECUTION UNITS
2152A
EXECUTION UNITS
2152B
SAMPLER
2154
DATA PORT
2156
TEXTURE CACHE
2158
RASTER/ DEPTH
2173
L3 CACHE
2175
PIXEL OPS
2177
RENDER CACHE
2178
DEPTH CACHE
2179
RENDER OUTPUT PIPELINE
2170

FIG. 21

FIG. 22A GRAPHICS PROCESSOR COMMAND FORMAT
2200

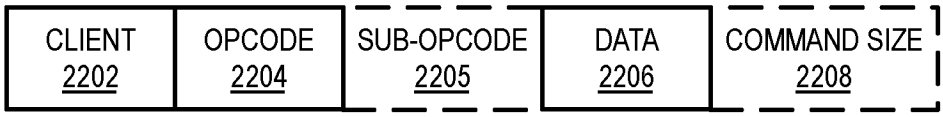
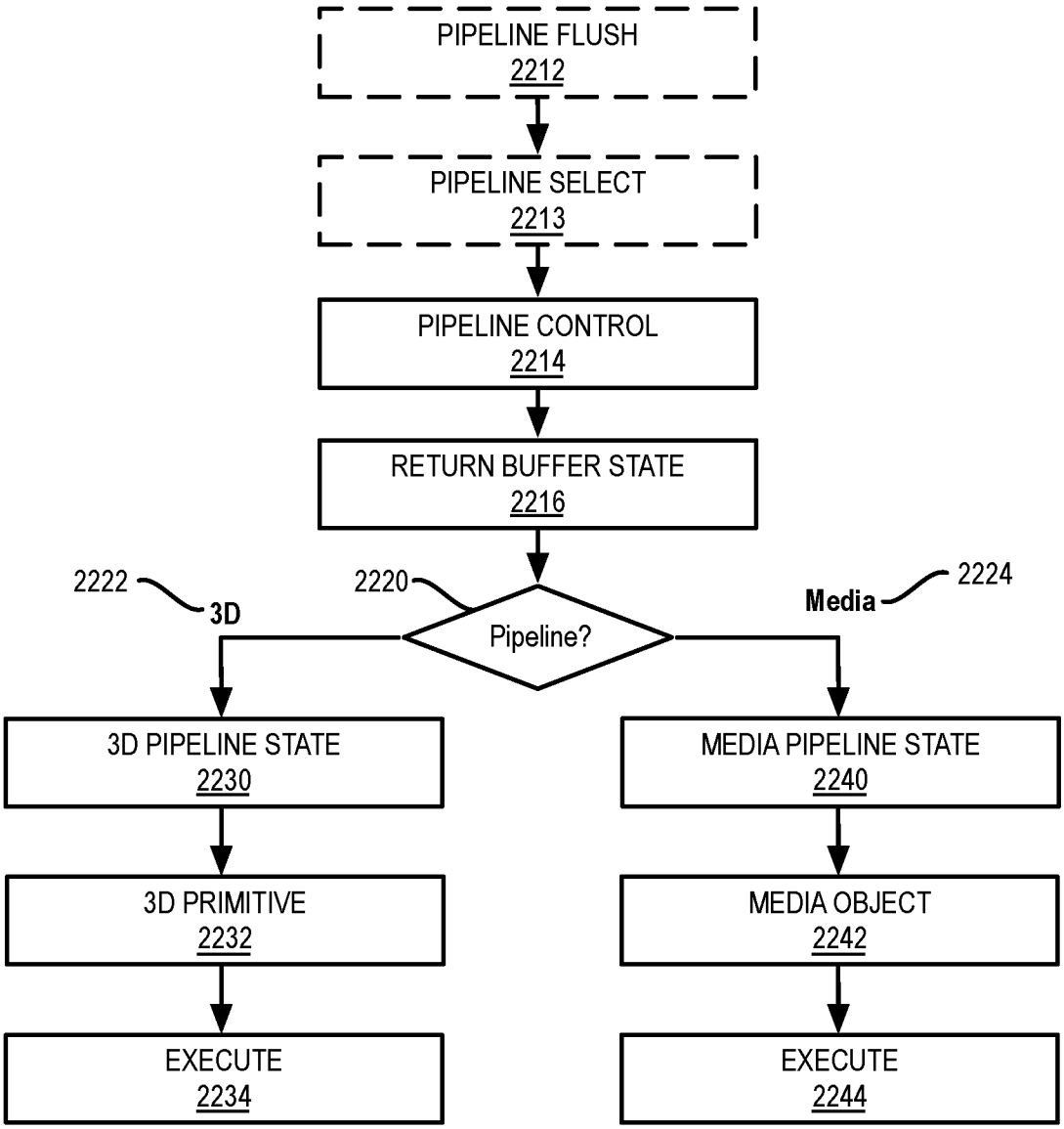


FIG. 22B GRAPHICS PROCESSOR COMMAND SEQUENCE
2210



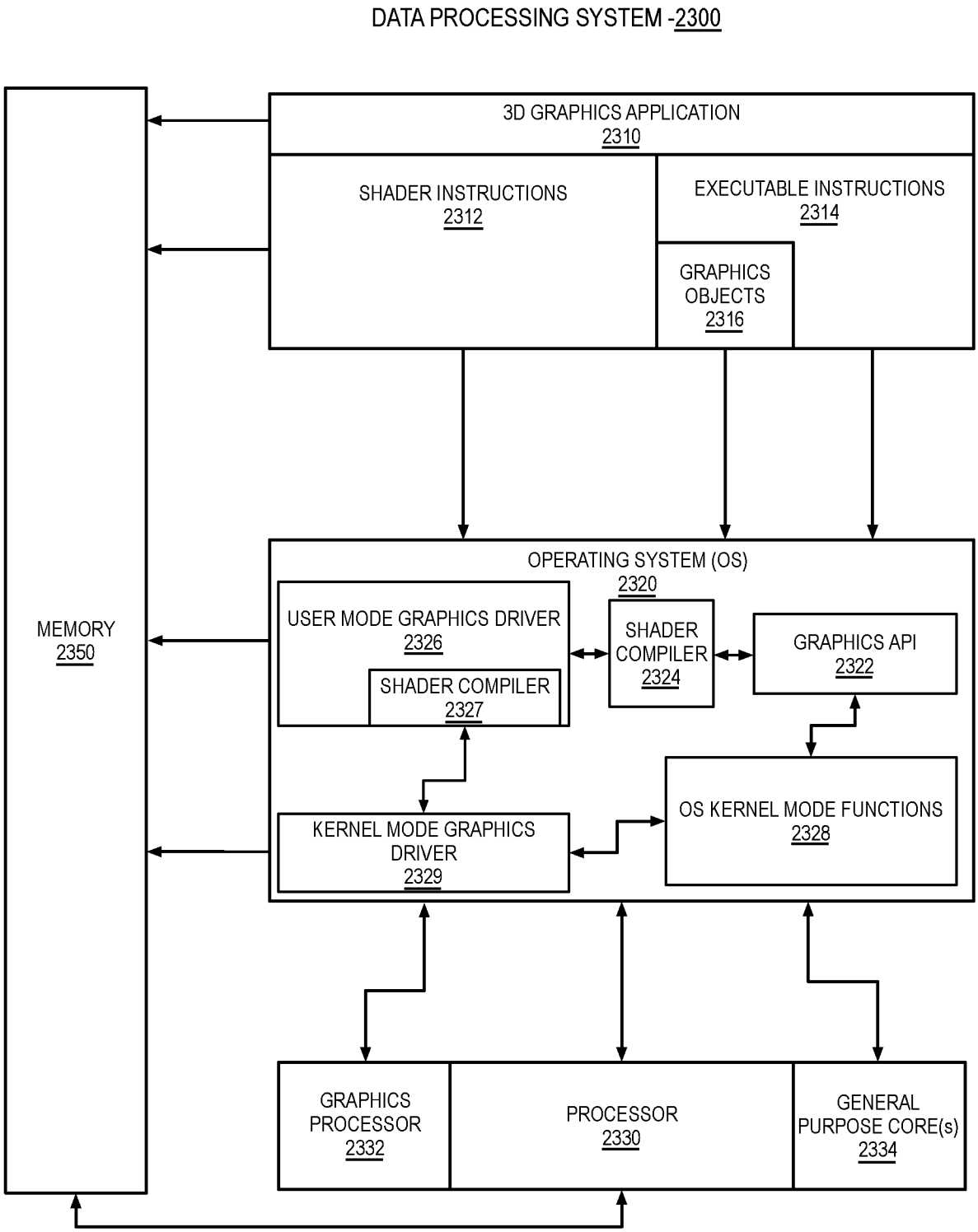

DATA PROCESSING SYSTEM -2300
3D GRAPHICS APPLICATION
2310
SHADER INSTRUCTIONS
2312
EXECUTABLE INSTRUCTIONS
2314
GRAPHICS OBJECTS
2316
MEMORY
2350
OPERATING SYSTEM (OS)
2320
USER MODE GRAPHICS DRIVER
2326
SHADER COMPILER
2327
SHADER COMPILER
2324
GRAPHICS API
2322
KERNEL MODE GRAPHICS DRIVER
2329
OS KERNEL MODE FUNCTIONS
2328
GRAPHICS PROCESSOR
2332
PROCESSOR
2330
GENERAL PURPOSE CORE(s)
2334

FIG. 23

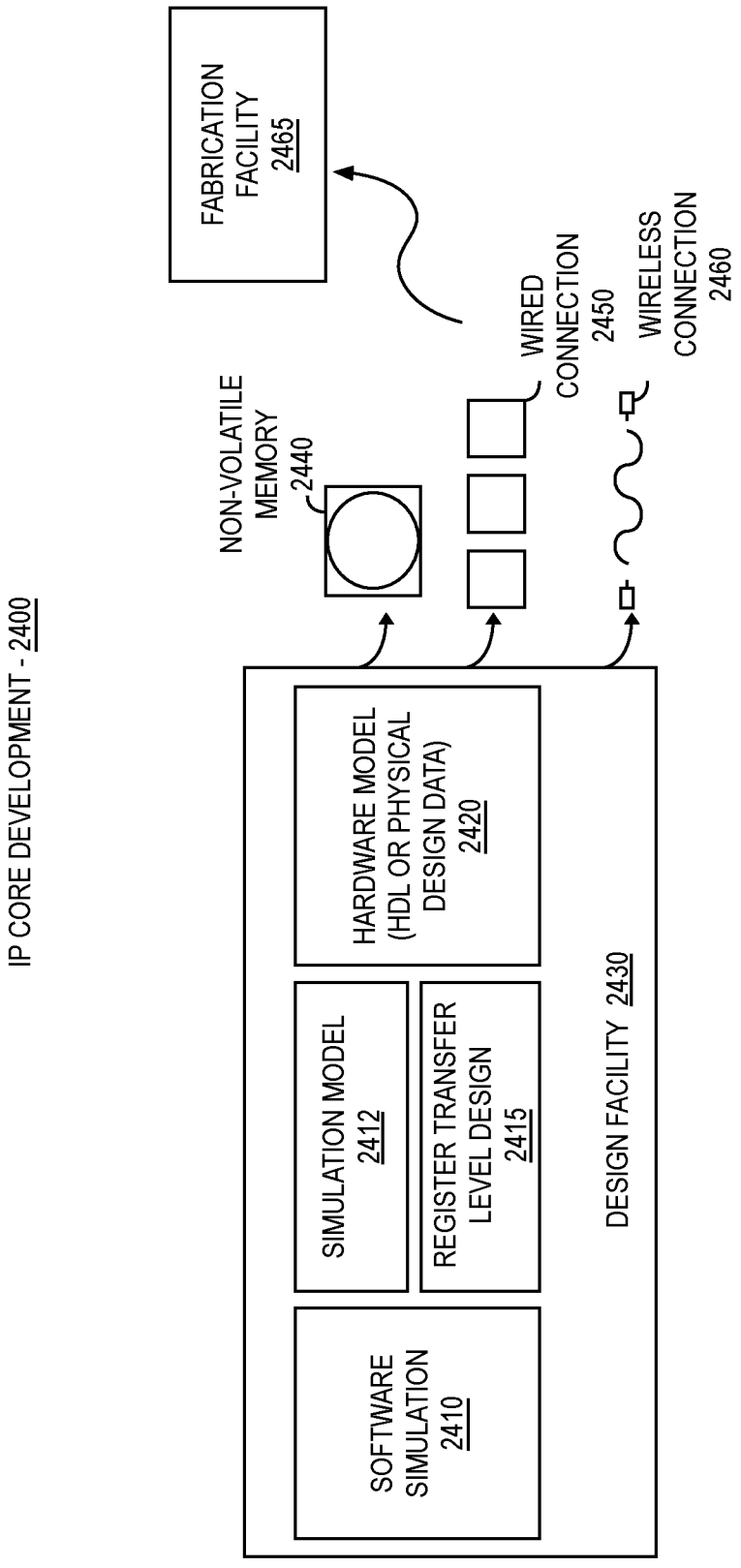

IP CORE DEVELOPMENT - 2400
SOFTWARE SIMULATION 2410
SIMULATION MODEL 2412
REGISTER TRANSFER LEVEL DESIGN 2415
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA) 2420
DESIGN FACILITY 2430
NON-VOLATILE MEMORY 2440
WIRED CONNECTION 2450
WIRELESS CONNECTION 2460
FABRICATION FACILITY 2465

FIG. 24A

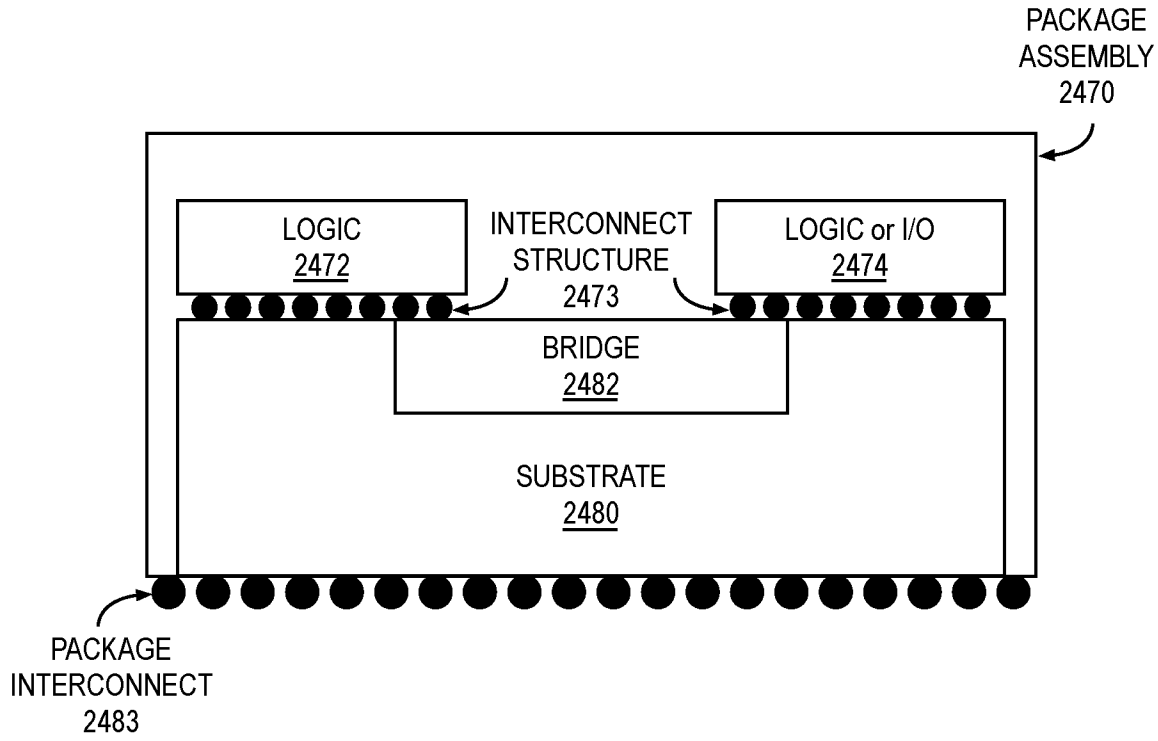

PACKAGE
ASSEMBLY
2470
LOGIC
2472
INTERCONNECT
STRUCTURE
2473
LOGIC or I/O
2474
BRIDGE
2482
SUBSTRATE
2480
PACKAGE
INTERCONNECT
2483

FIG. 24B

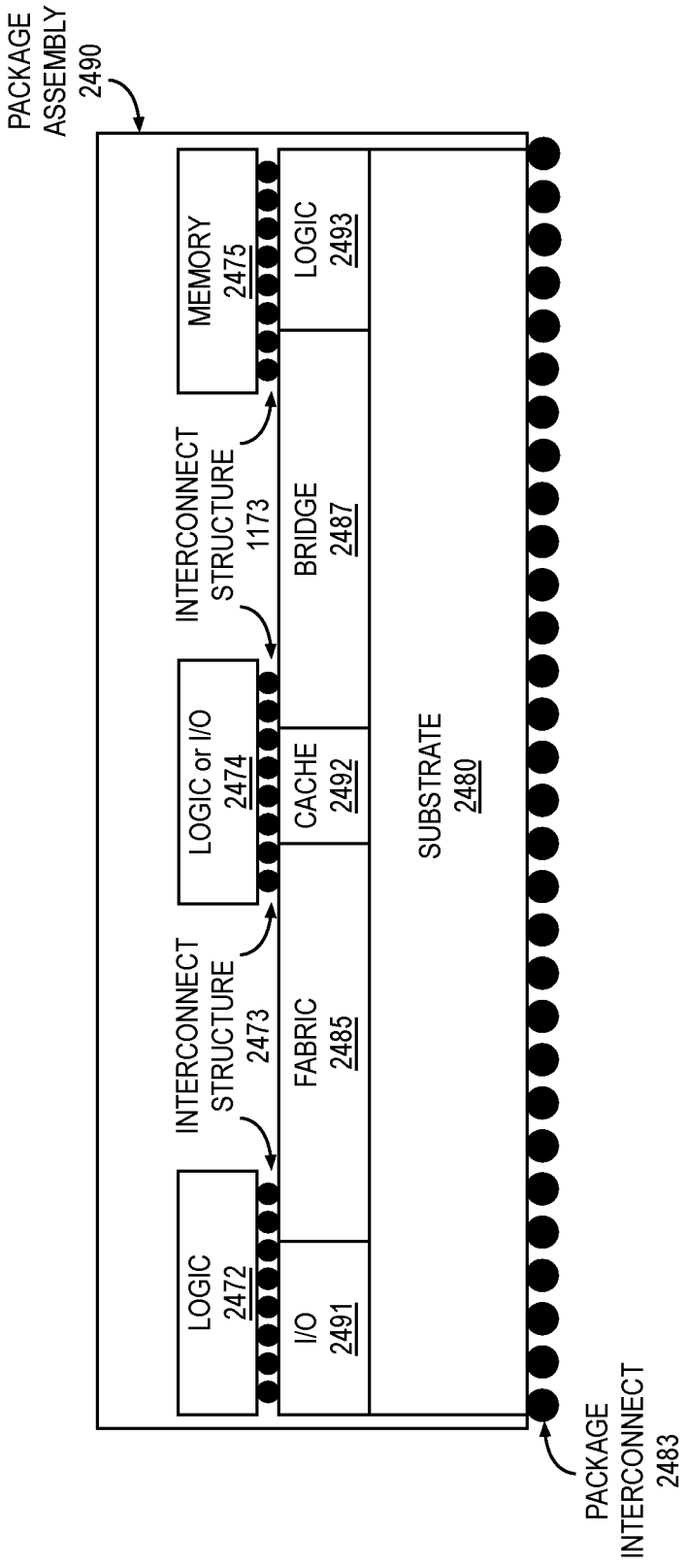

PACKAGE ASSEMBLY 2490
LOGIC 2472
I/O 2491
INTERCONNECT STRUCTURE 2473
FABRIC 2485
LOGIC or I/O 2474
CACHE 2492
INTERCONNECT STRUCTURE 1173
BRIDGE 2487
MEMORY 2475
LOGIC 2493
SUBSTRATE 2480
PACKAGE INTERCONNECT 2483

FIG. 24C

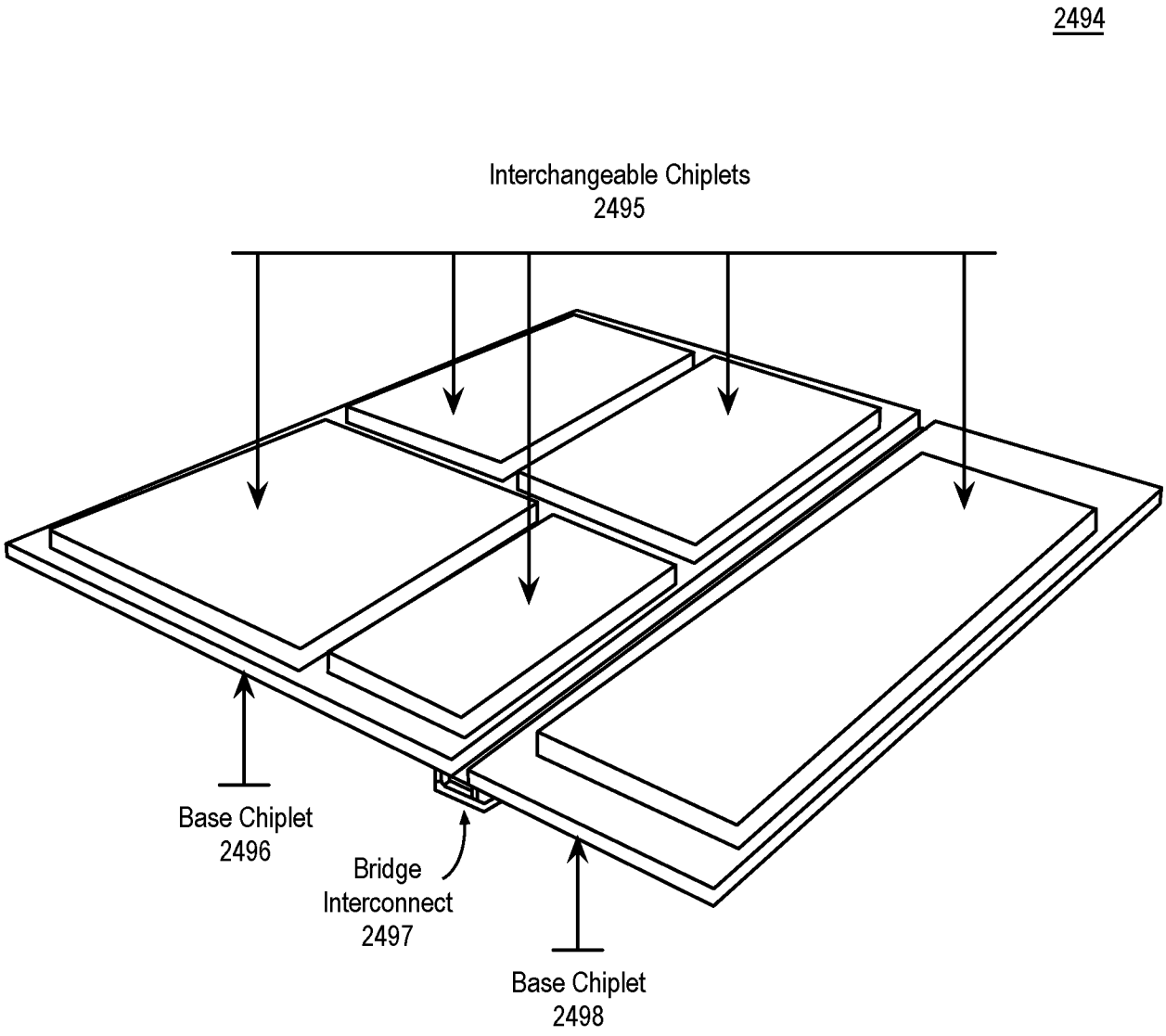

2494
Interchangeable Chiplets
2495
Base Chiplet
2496
Bridge
Interconnect
2497
Base Chiplet
2498

FIG. 24D

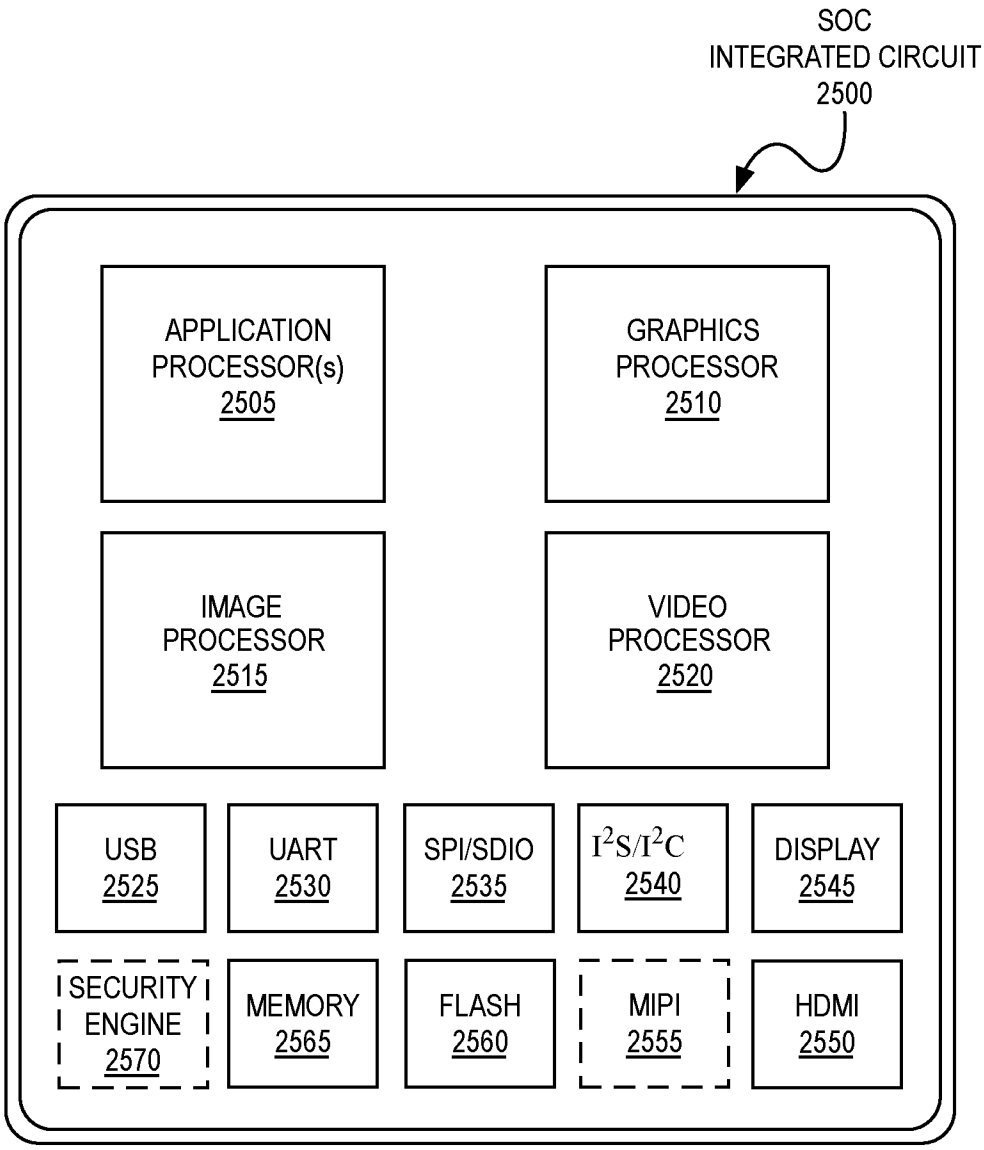

SOC
INTEGRATED CIRCUIT
2500
APPLICATION PROCESSOR(s) 2505
GRAPHICS PROCESSOR 2510
IMAGE PROCESSOR 2515
VIDEO PROCESSOR 2520
USB 2525
UART 2530
SPI/SDIO 2535
I2S/I2C 2540
DISPLAY 2545
SECURITY ENGINE 2570
MEMORY 2565
FLASH 2560
MIPI 2555
HDMI 2550

FIG. 25

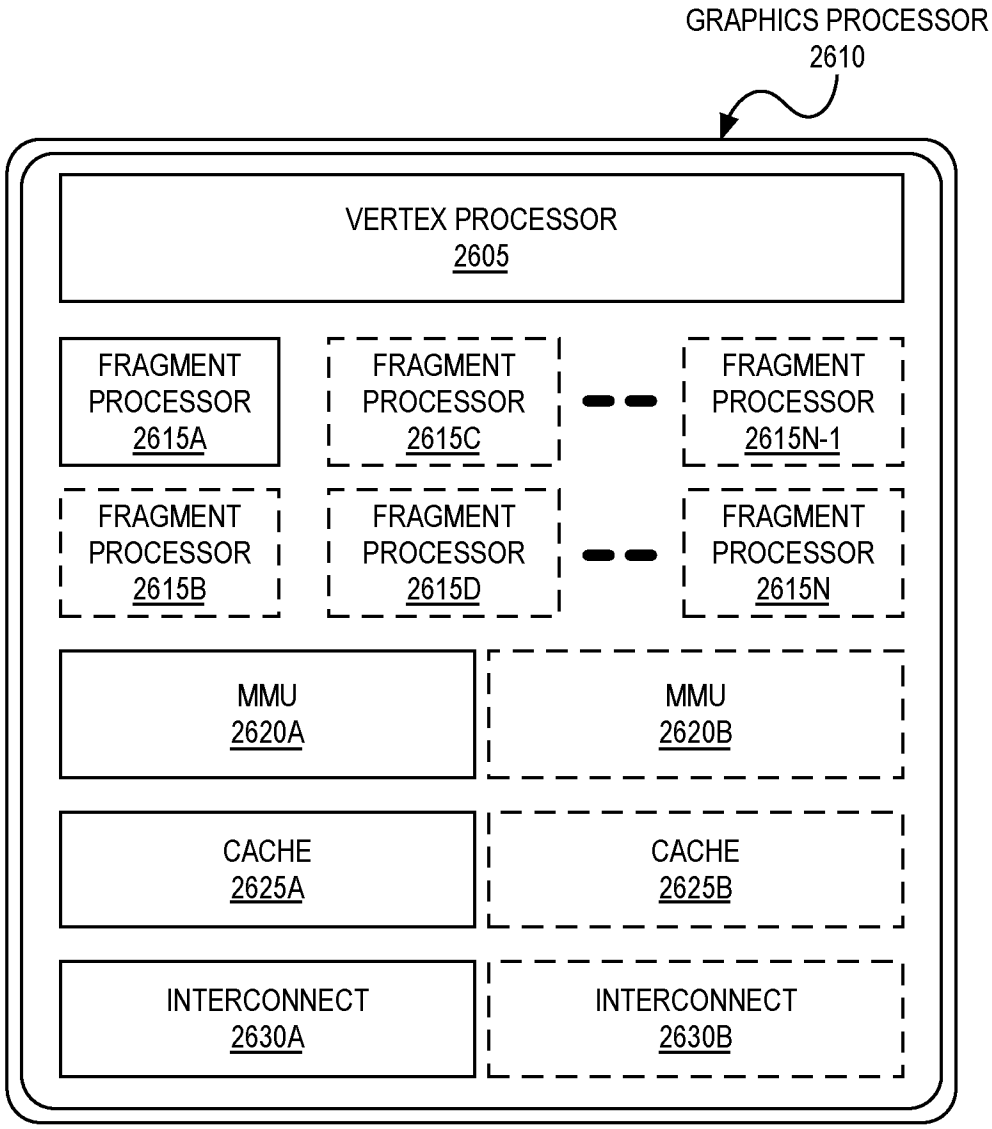

GRAPHICS PROCESSOR
2610
VERTEX PROCESSOR
2605
FRAGMENT PROCESSOR
2615A
FRAGMENT PROCESSOR
2615C
FRAGMENT PROCESSOR
2615N-1
FRAGMENT PROCESSOR
2615B
FRAGMENT PROCESSOR
2615D
FRAGMENT PROCESSOR
2615N
MMU
2620A
MMU
2620B
CACHE
2625A
CACHE
2625B
INTERCONNECT
2630A
INTERCONNECT
2630B

FIG. 26A

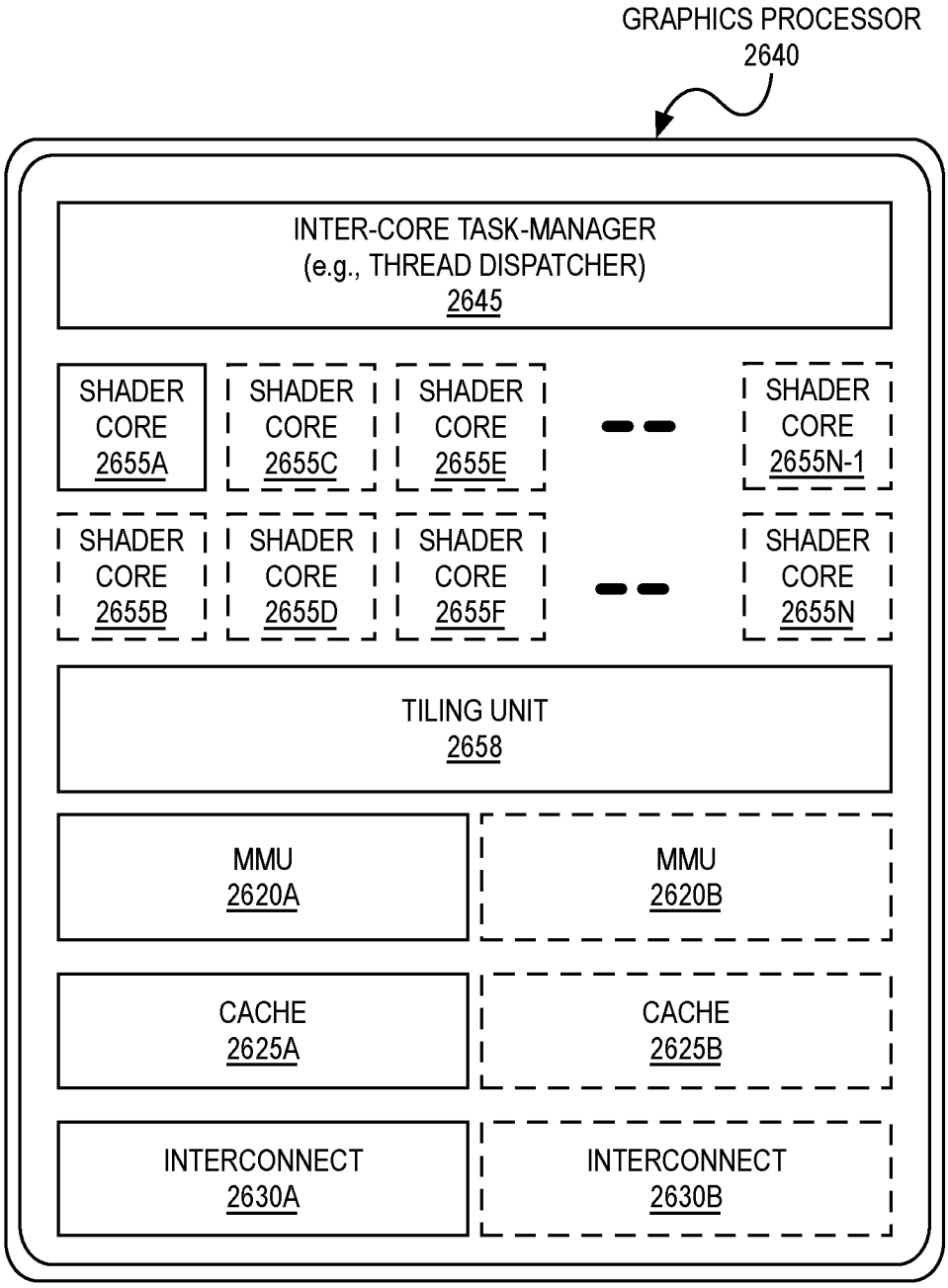

GRAPHICS PROCESSOR
2640
INTER-CORE TASK-MANAGER
(e.g., THREAD DISPATCHER)
2645
SHADER CORE 2655A
SHADER CORE 2655C
SHADER CORE 2655E
SHADER CORE 2655N-1
SHADER CORE 2655B
SHADER CORE 2655D
SHADER CORE 2655F
SHADER CORE 2655N
TILING UNIT
2658
MMU
2620A
MMU
2620B
CACHE
2625A
CACHE
2625B
INTERCONNECT
2630A
INTERCONNECT
2630B

FIG. 26B

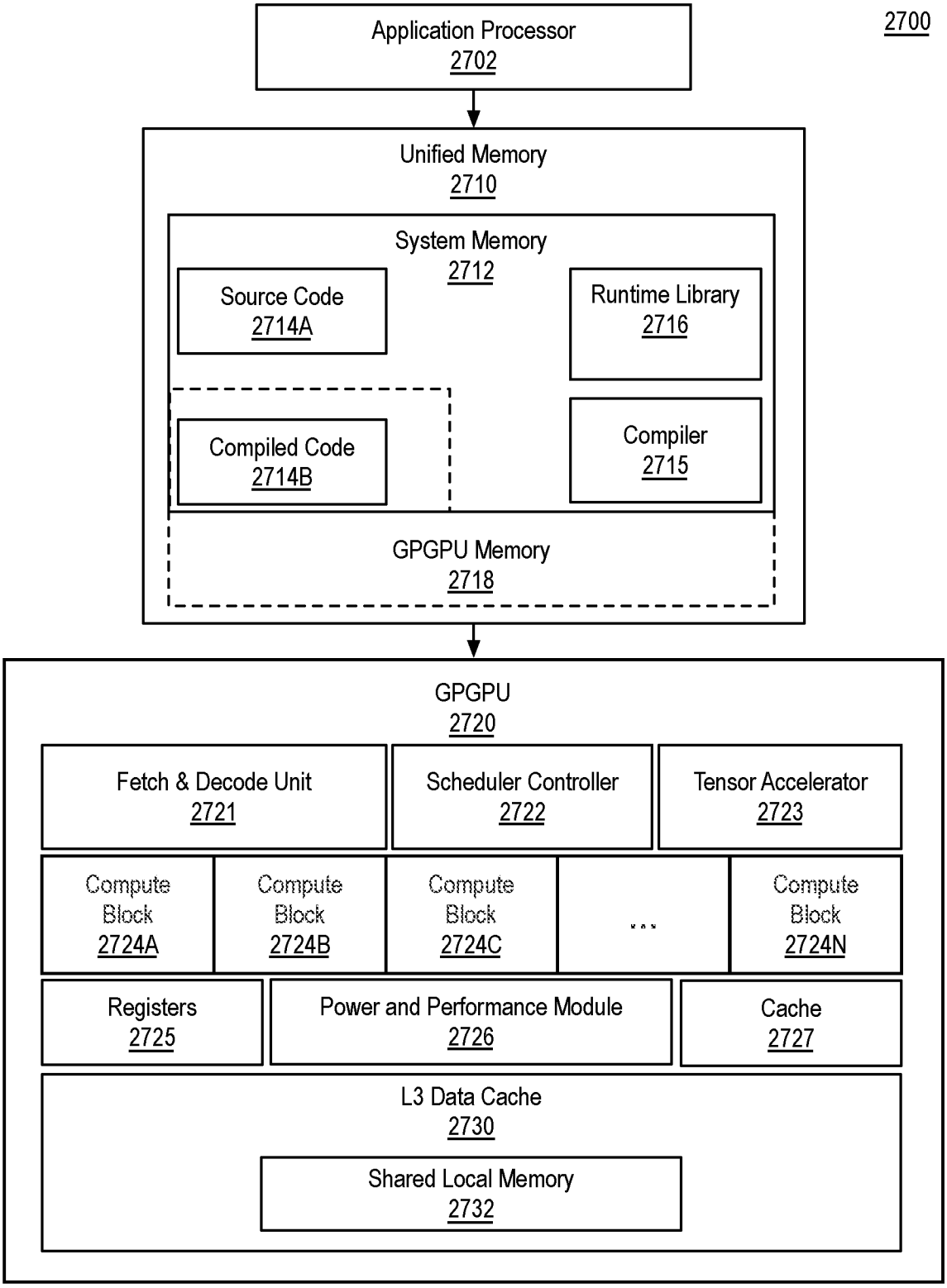

2700
Application Processor
2702
Unified Memory
2710
System Memory
2712
Source Code
2714A
Runtime Library
2716
Compiled Code
2714B
Compiler
2715
GPGPU Memory
2718
GPGPU
2720
Fetch & Decode Unit
2721
Scheduler Controller
2722
Tensor Accelerator
2723
Compute Block
2724A
Compute Block
2724B
Compute Block
2724C
...
Compute Block
2724N
Registers
2725
Power and Performance Module
2726
Cache
2727
L3 Data Cache
2730
Shared Local Memory
2732

FIG. 27

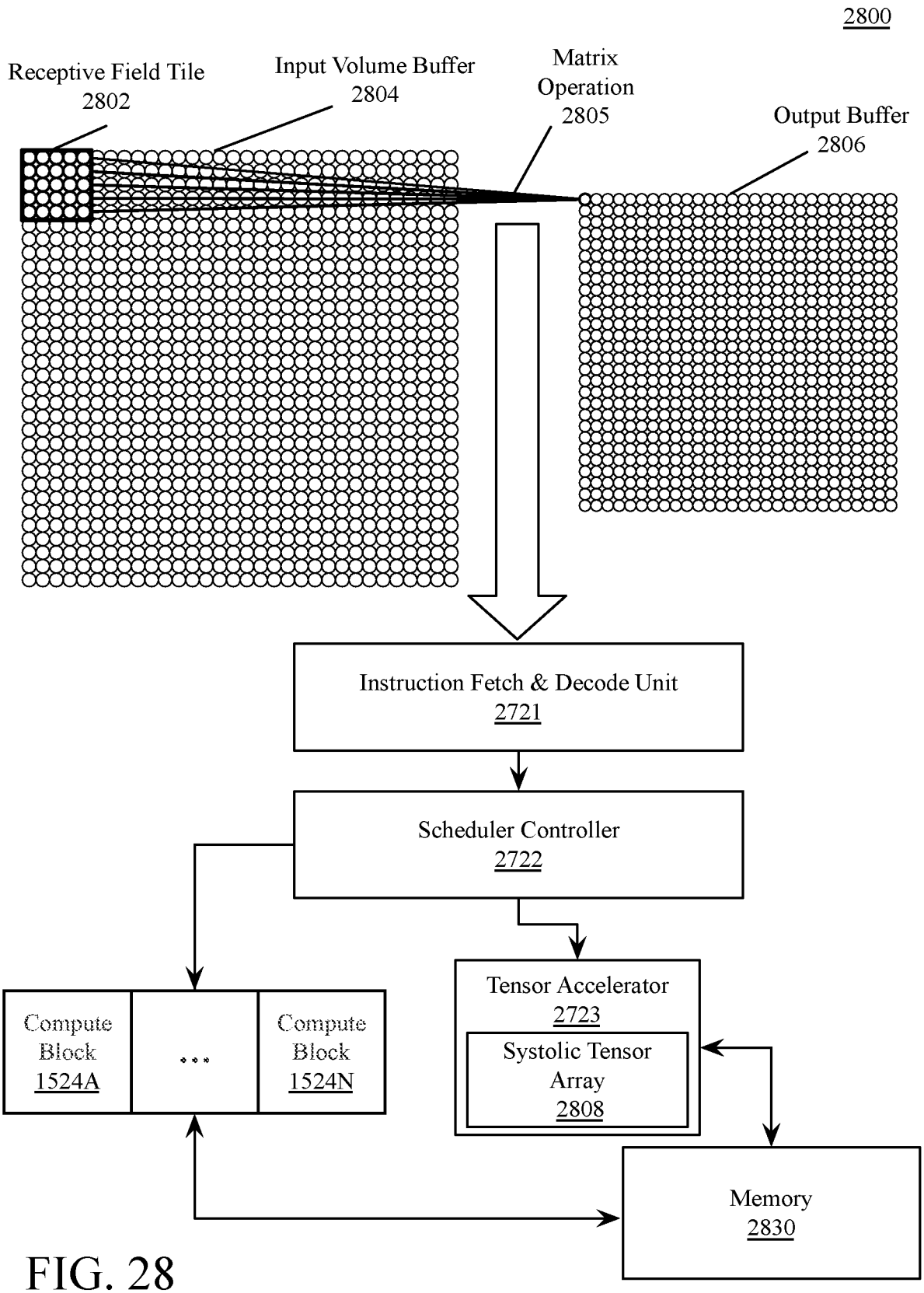

2800
Receptive Field Tile
2802
Input Volume Buffer
2804
Matrix
Operation
2805
Output Buffer
2806
Instruction Fetch & Decode Unit
2721
Scheduler Controller
2722
Compute
Block
1524A
...
Compute
Block
1524N
Tensor Accelerator
2723
Systolic Tensor
Array
2808
Memory
2830

FIG. 28

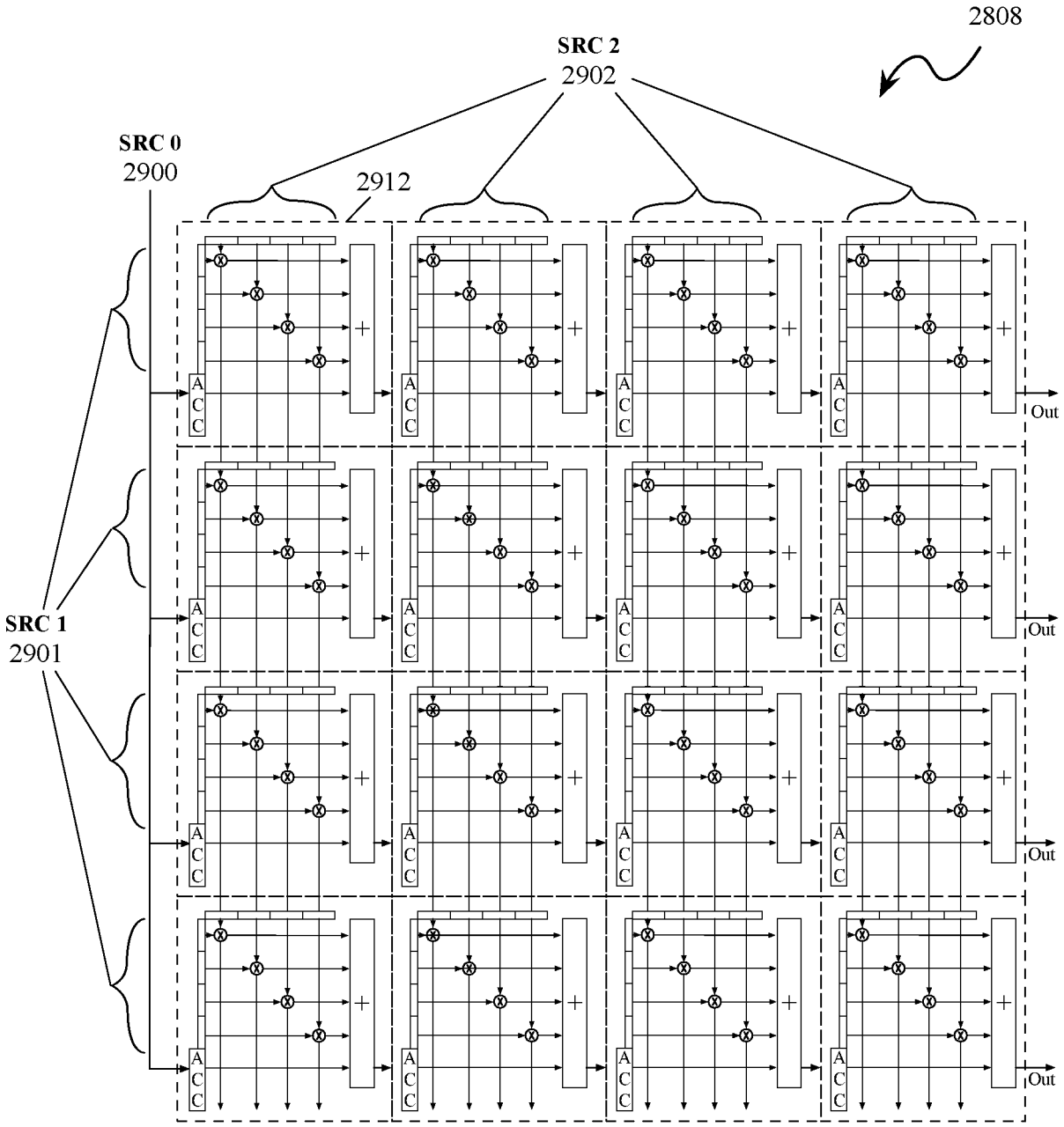


FIG. 29A

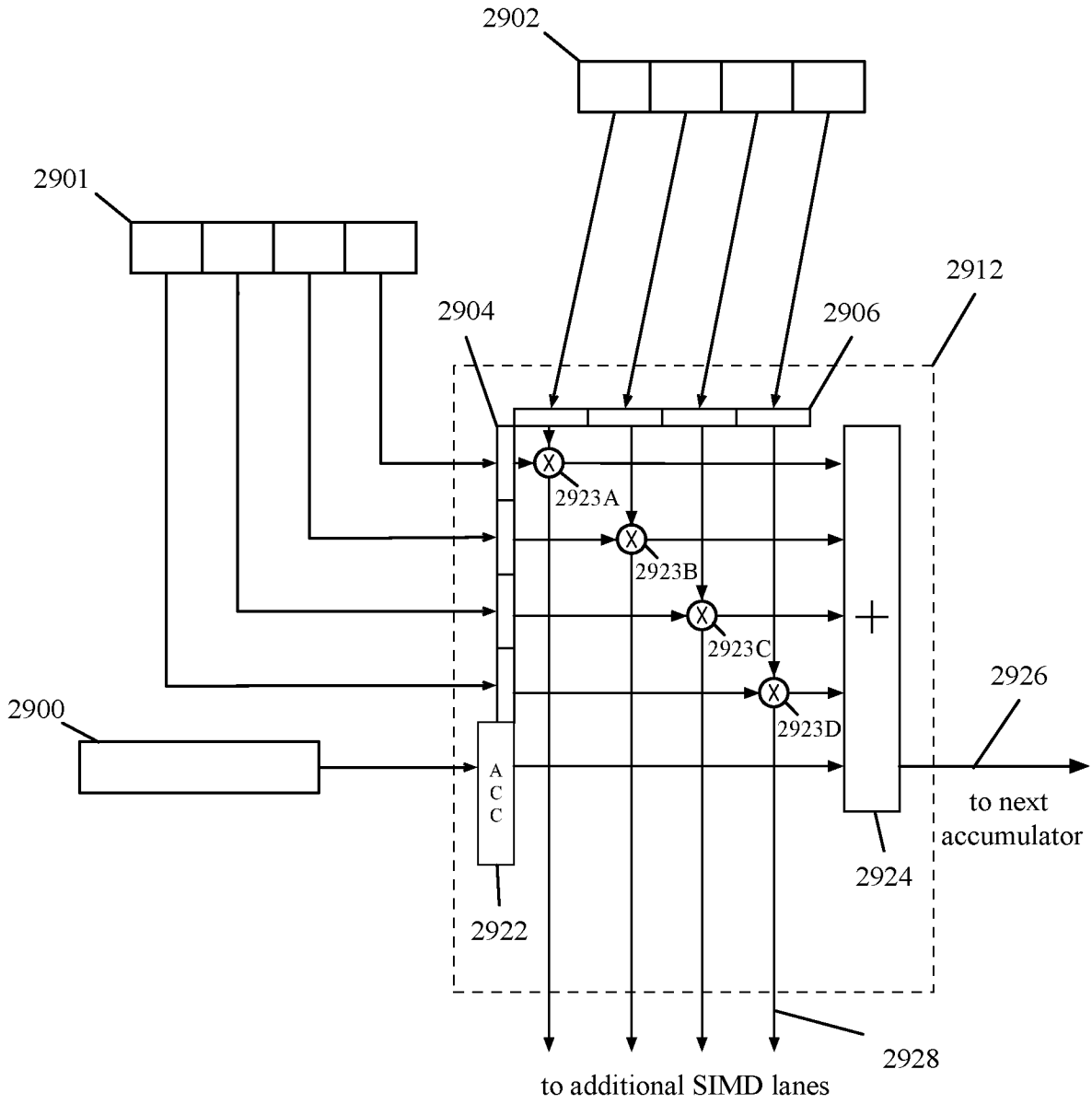

2902
2901
2912
2904
2906
2923A
2923B
2923C
2923D
2900
2926
ACC
to next accumulator
2924
2922
2928
to additional SIMD lanes

FIG. 29B

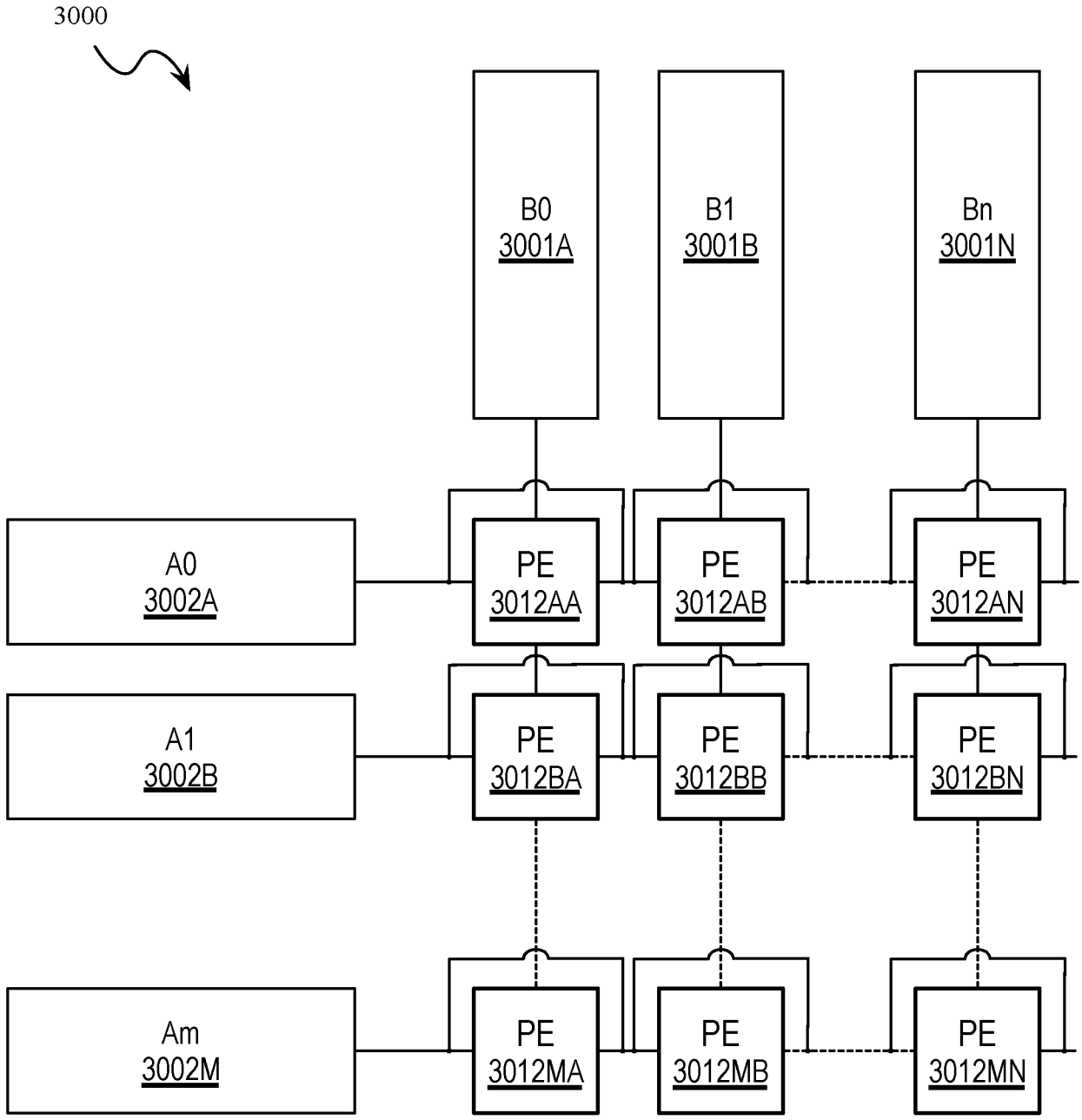

3000
B0
3001A
B1
3001B
Bn
3001N
A0
3002A
A1
3002B
Am
3002M
PE
3012AA
PE
3012AB
PE
3012AN
PE
3012BA
PE
3012BB
PE
3012BN
PE
3012MA
PE
3012MB
PE
3012MN

FIG. 30

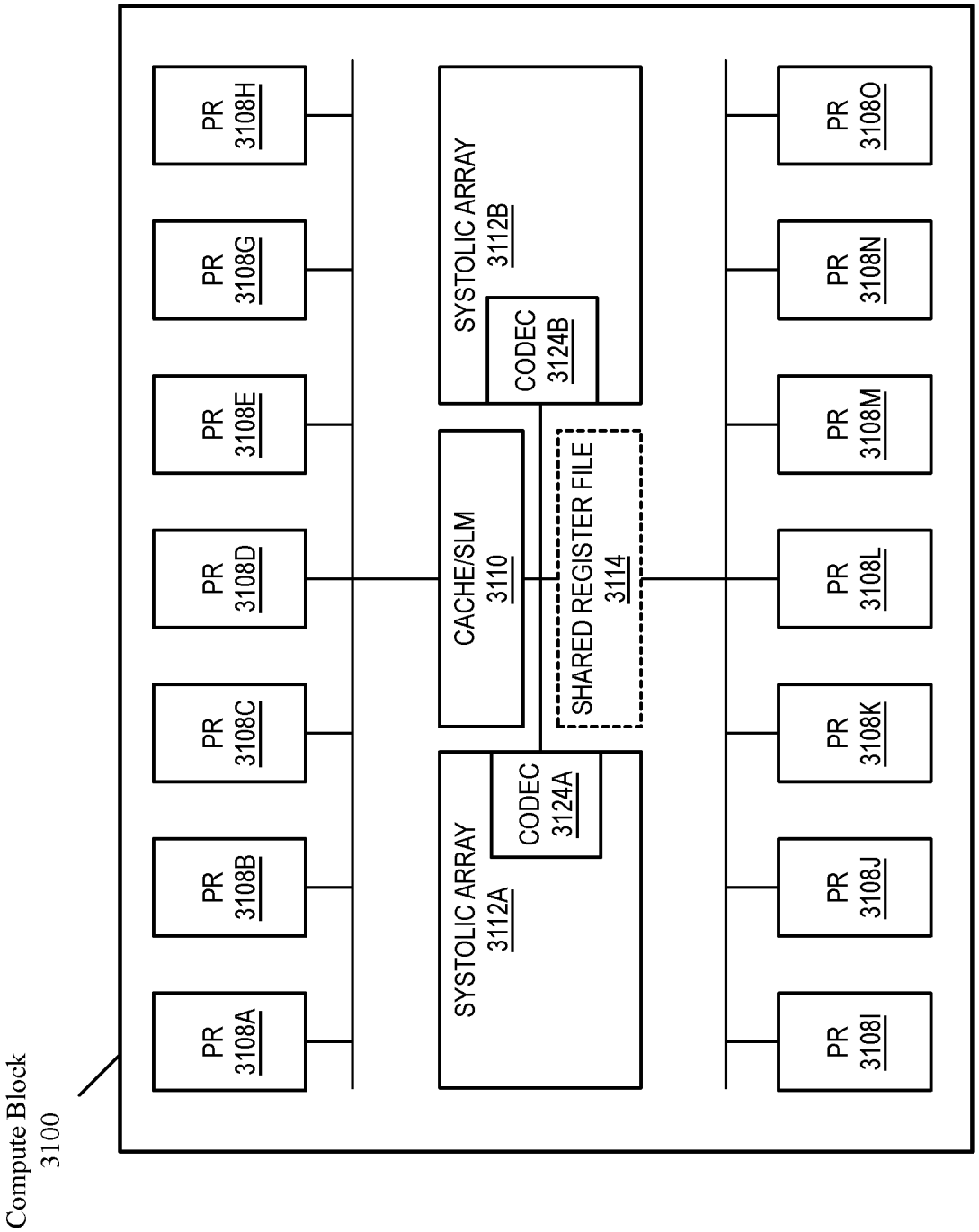

Compute Block
3100
PR
3108A
PR
3108B
PR
3108C
PR
3108D
PR
3108E
PR
3108G
PR
3108H
SYSTOLIC ARRAY
3112A
CODEC
3124A
CACHE/SLM
3110
SHARED REGISTER FILE
3114
SYSTOLIC ARRAY
3112B
CODEC
3124B
PR
3108I
PR
3108J
PR
3108K
PR
3108L
PR
3108M
PR
3108N
PR
3108O

FIG. 31A

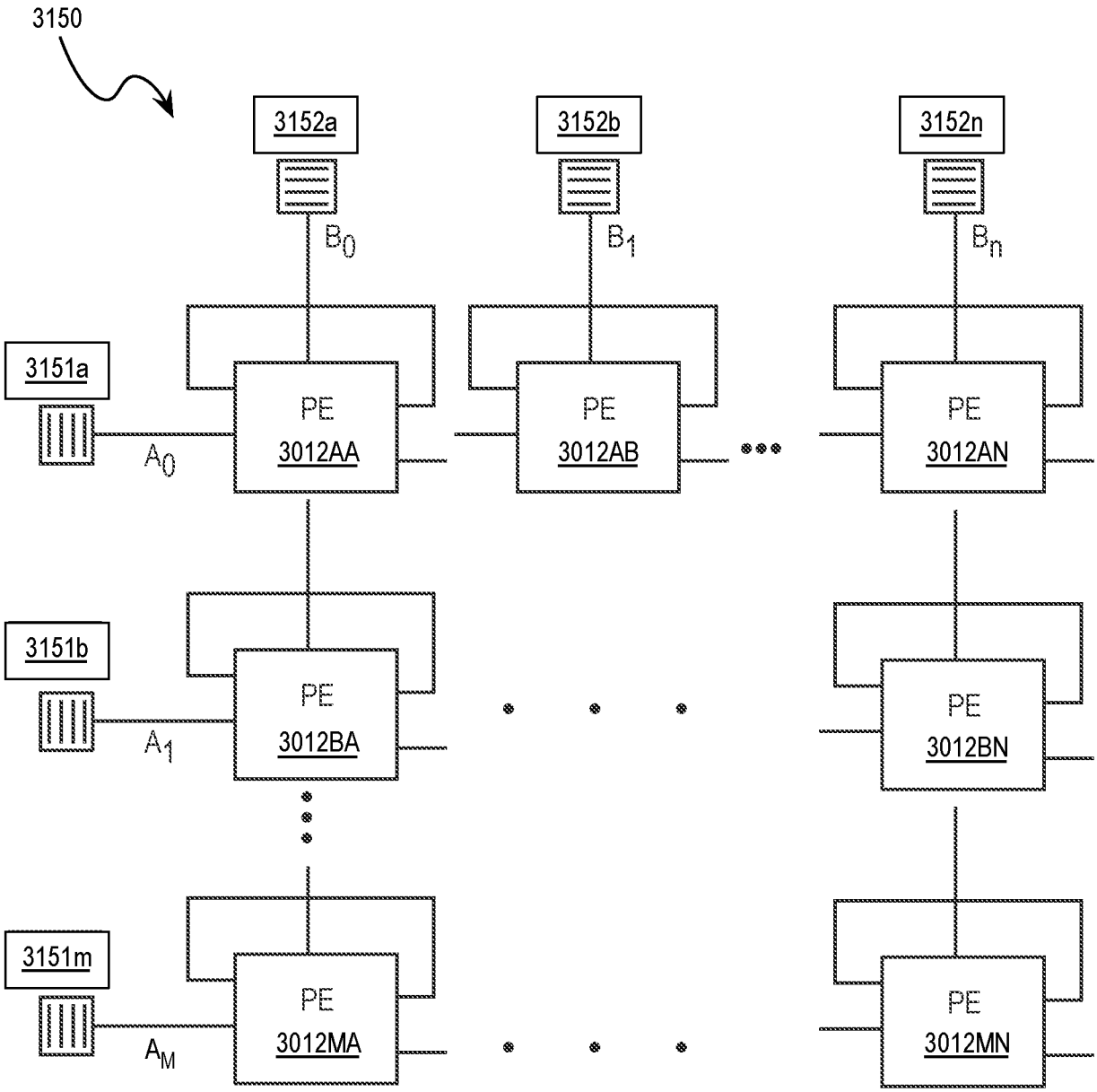

3150
3152a
3152b
3152n
B0
B1
Bn
3151a
3151b
3151m
A0
A1
AM
PE
3012AA
PE
3012AB
PE
3012AN
PE
3012BA
PE
3012BN
PE
3012MA
PE
3012MN

FIG. 31B

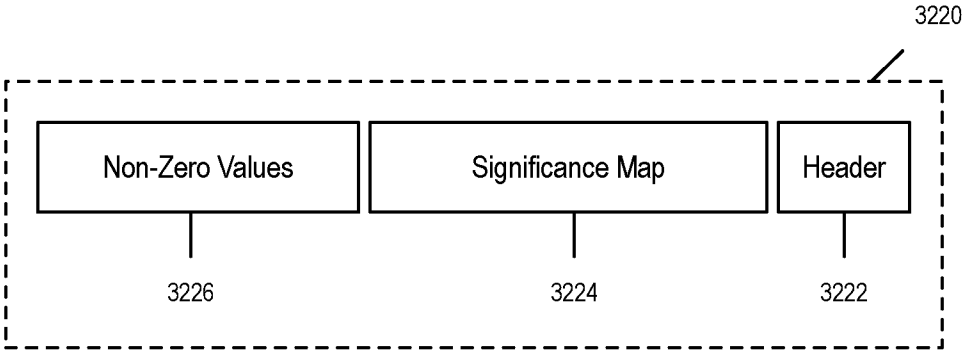


FIG. 32A

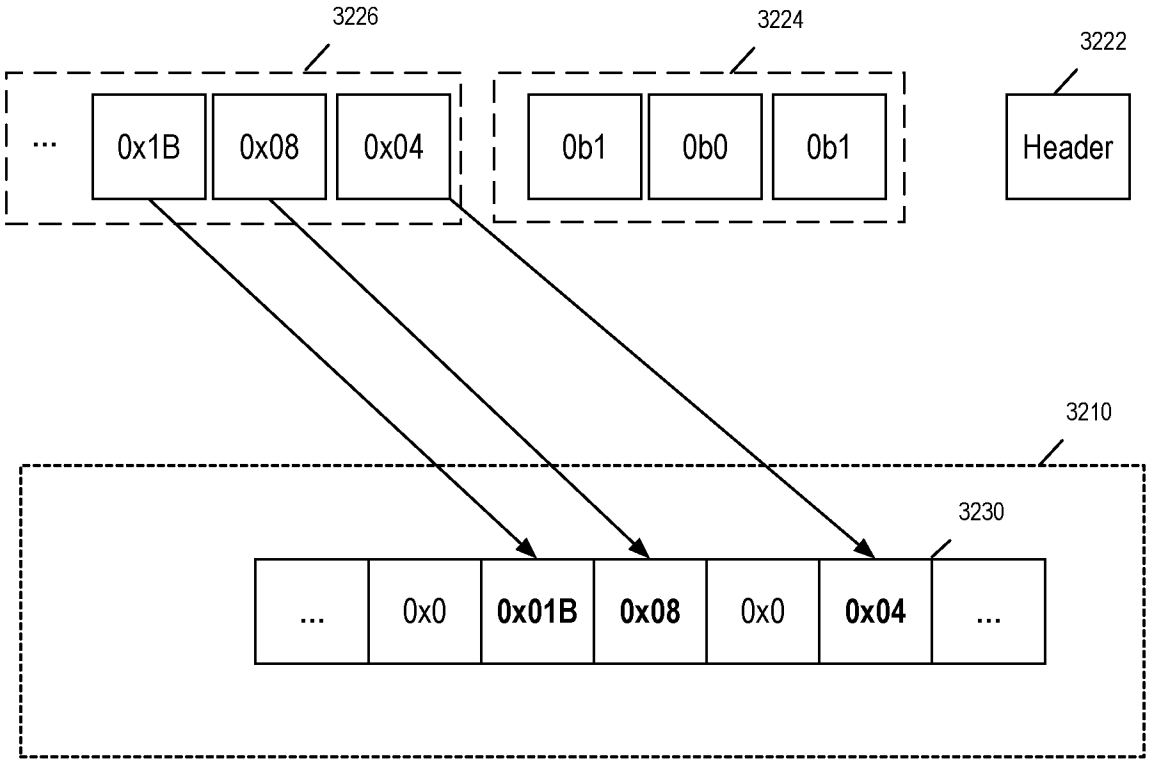


FIG. 32B

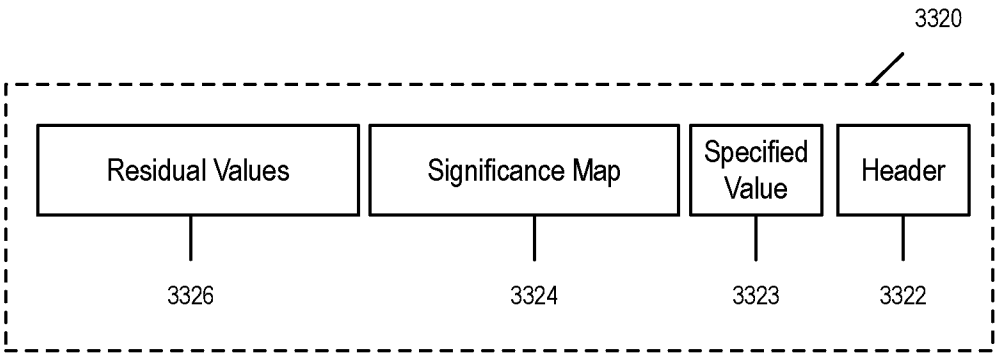


FIG. 33A

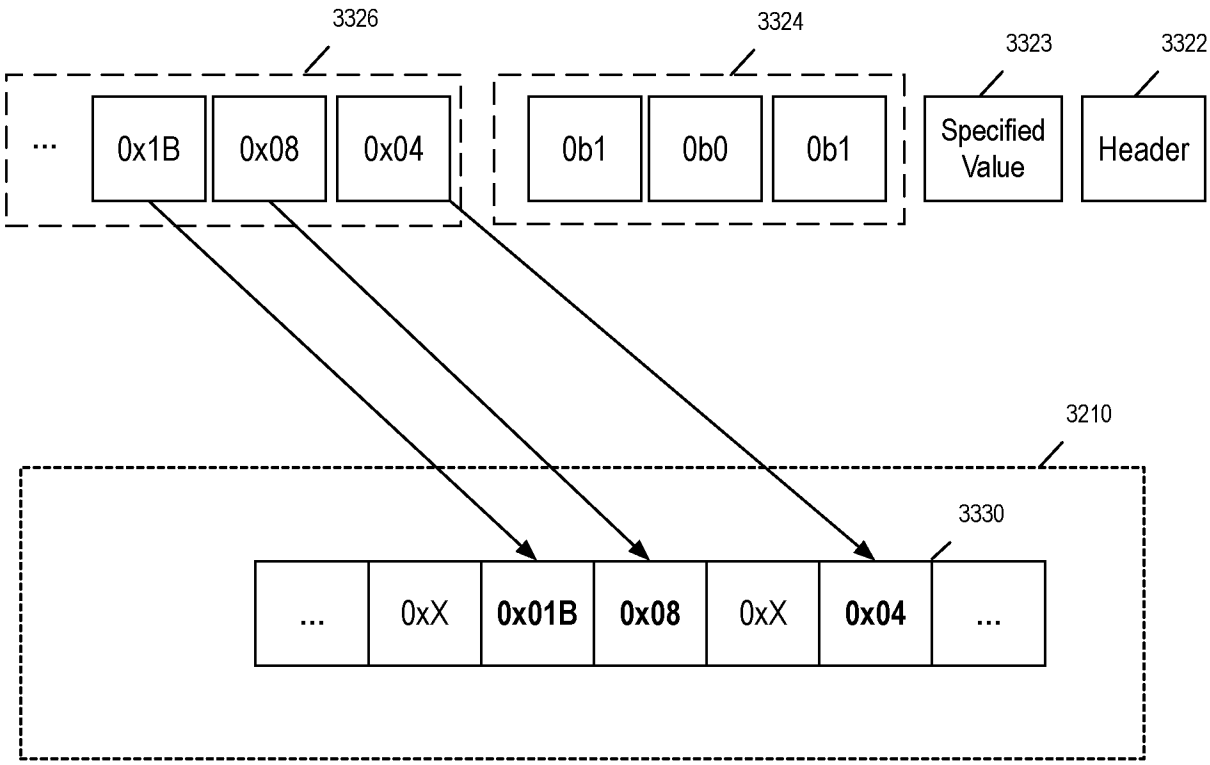


FIG. 33B

3402

0	-0.2	-0.5	0
0	0	0.1	0.2
0.2	0	-0.5	0
-0.3	0	-0.3	-0.5

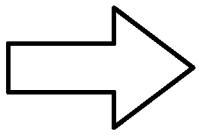
3404

0	0	-0.3	0
0	0	0.5	0.8
0	0.1	0	0
0.3	0.7	0	0

FIG. 34A

3412

0	-0.2	-0.2	0.3
0	0.3	0.2	0.3
0.2	0.1	0.1	-0.2
0.1	-0.2	-0.1	-0.1



3414

0	0	-0.2	0.3
0	0	0.2	0.3
0	0	0	0
0	0	0	0

FIG. 34B

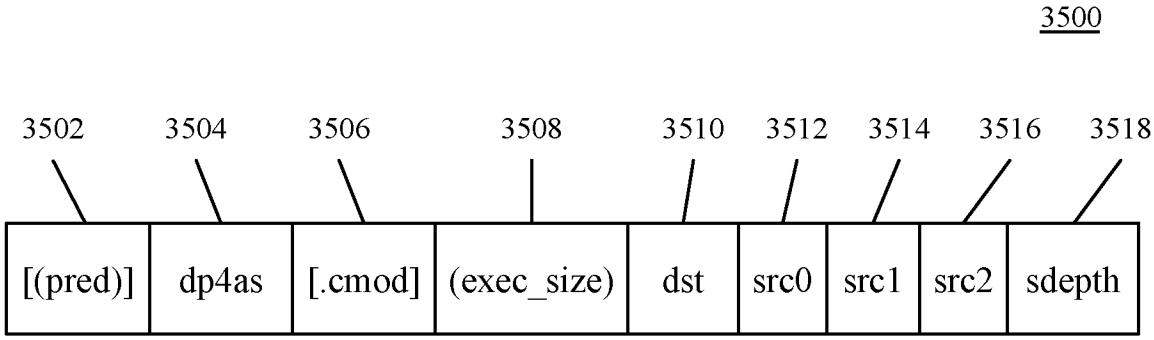


FIG. 35A

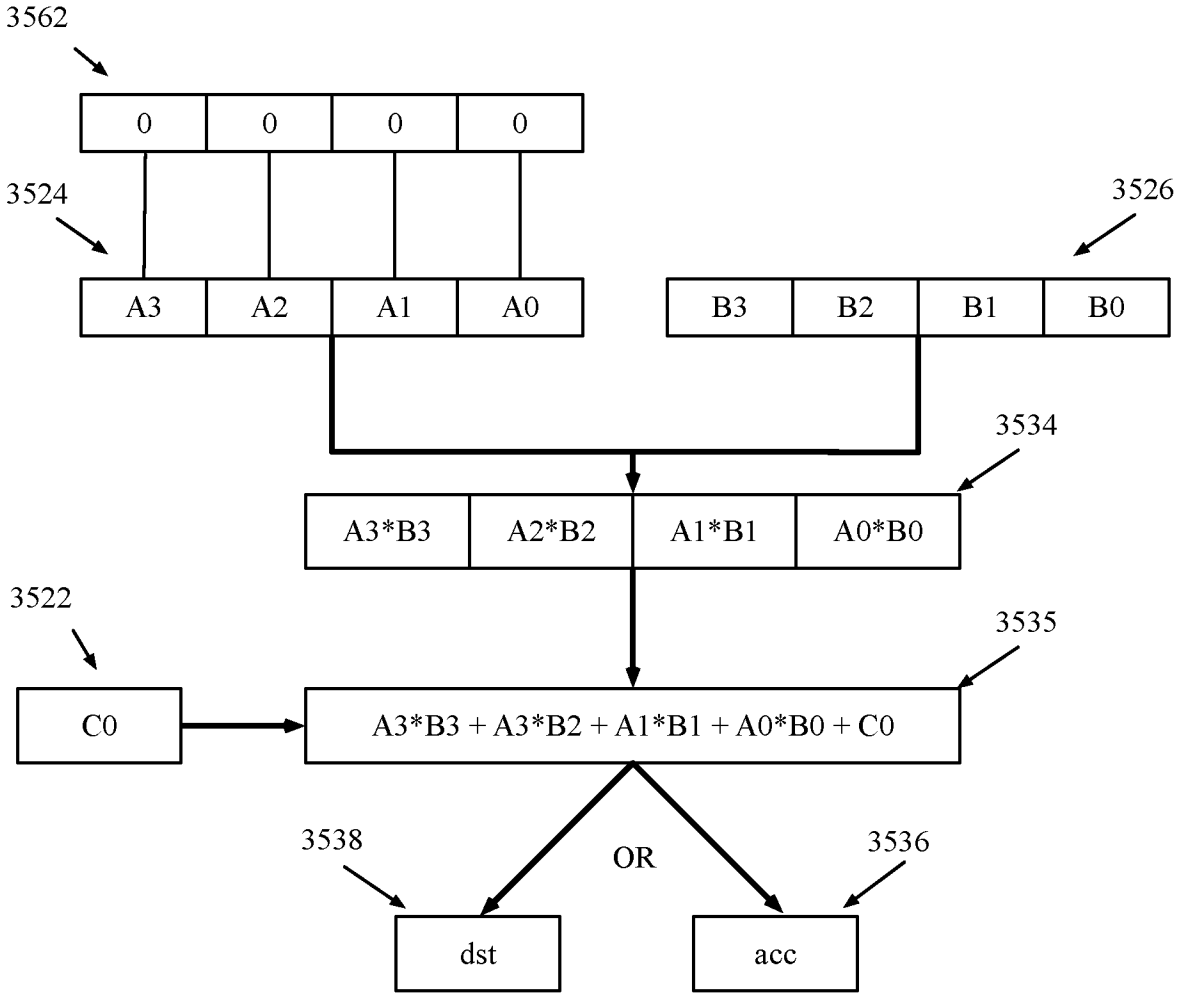


FIG. 35B

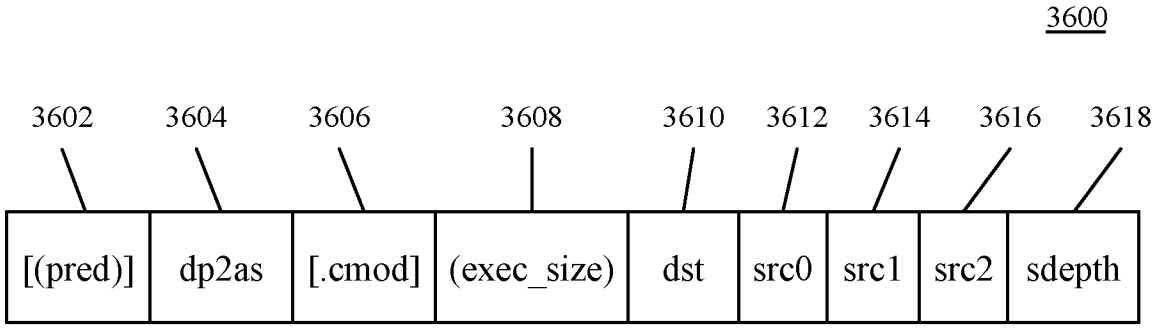


FIG. 36A

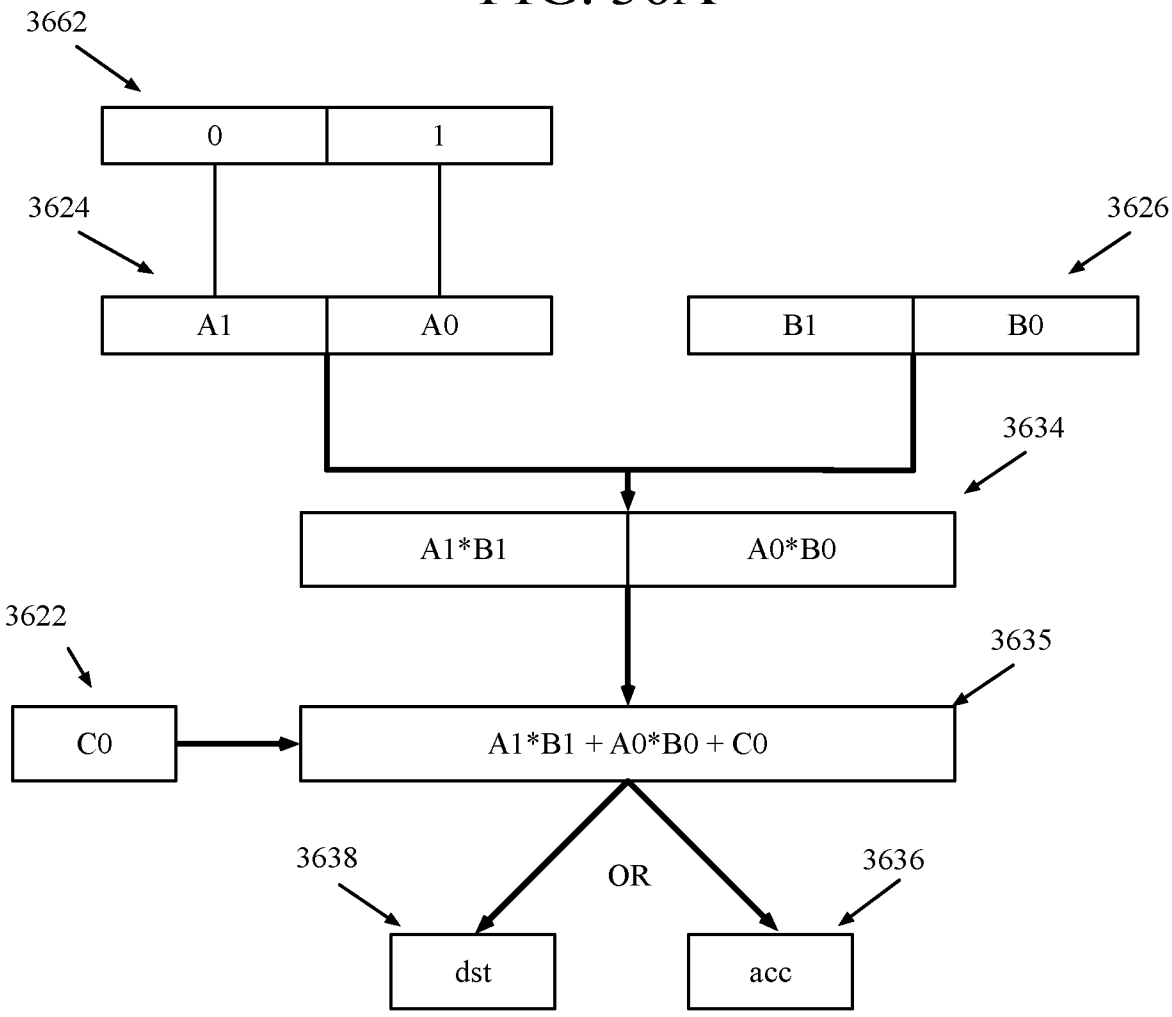


FIG. 36B

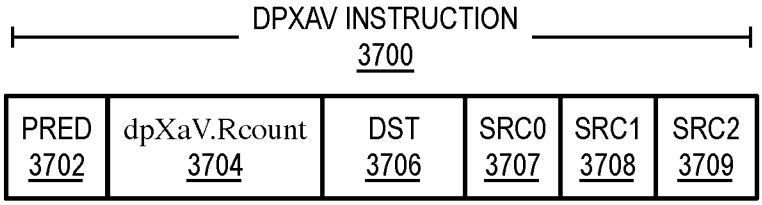


FIG. 37A

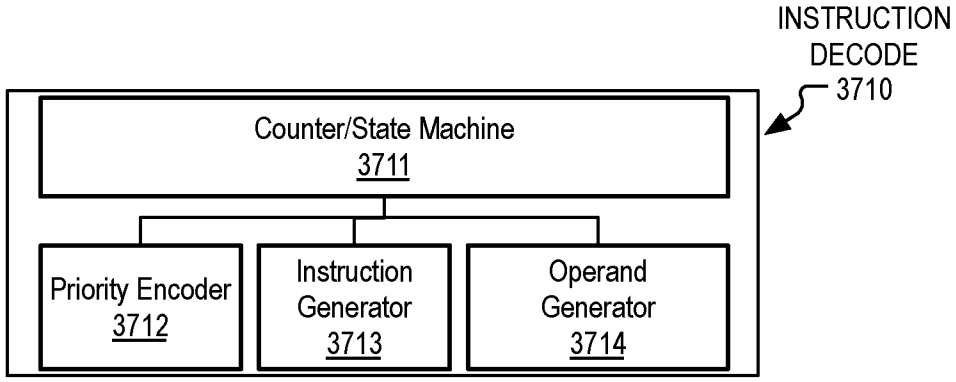


FIG. 37B

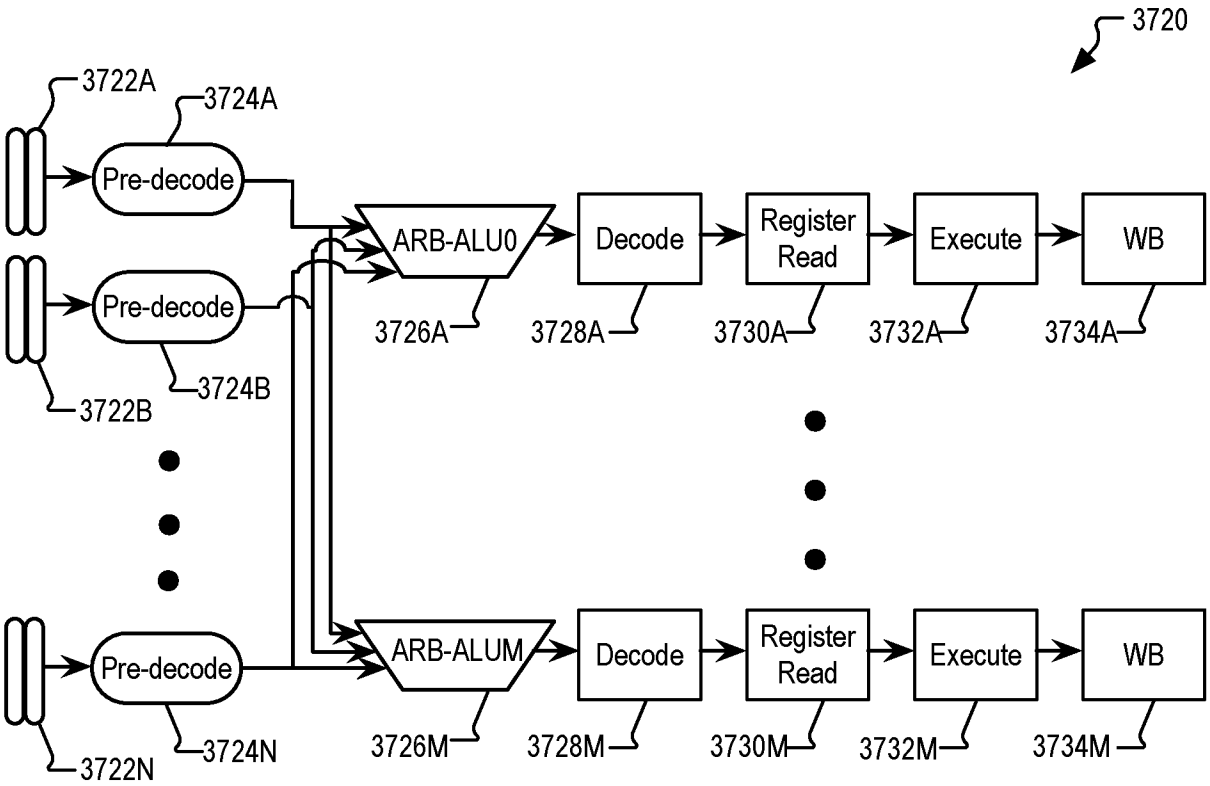


FIG. 37C

3750

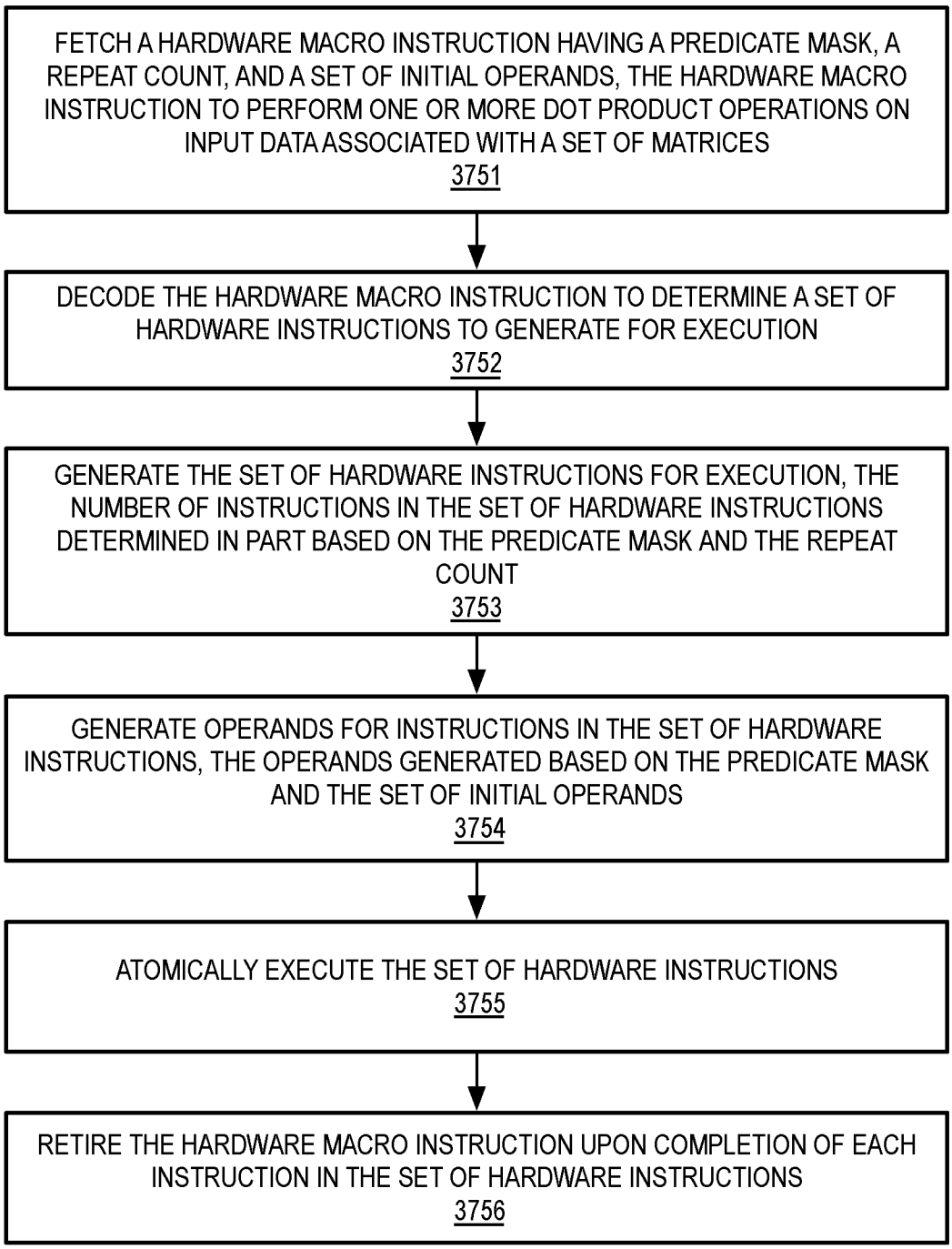

FETCH A HARDWARE MACRO INSTRUCTION HAVING A PREDICATE MASK, A REPEAT COUNT, AND A SET OF INITIAL OPERANDS, THE HARDWARE MACRO INSTRUCTION TO PERFORM ONE OR MORE DOT PRODUCT OPERATIONS ON INPUT DATA ASSOCIATED WITH A SET OF MATRICES
3751
DECODE THE HARDWARE MACRO INSTRUCTION TO DETERMINE A SET OF HARDWARE INSTRUCTIONS TO GENERATE FOR EXECUTION
3752
GENERATE THE SET OF HARDWARE INSTRUCTIONS FOR EXECUTION, THE NUMBER OF INSTRUCTIONS IN THE SET OF HARDWARE INSTRUCTIONS DETERMINED IN PART BASED ON THE PREDICATE MASK AND THE REPEAT COUNT
3753
GENERATE OPERANDS FOR INSTRUCTIONS IN THE SET OF HARDWARE INSTRUCTIONS, THE OPERANDS GENERATED BASED ON THE PREDICATE MASK AND THE SET OF INITIAL OPERANDS
3754
ATOMICALLY EXECUTE THE SET OF HARDWARE INSTRUCTIONS
3755
RETIRE THE HARDWARE MACRO INSTRUCTION UPON COMPLETION OF EACH INSTRUCTION IN THE SET OF HARDWARE INSTRUCTIONS
3756

FIG. 37D

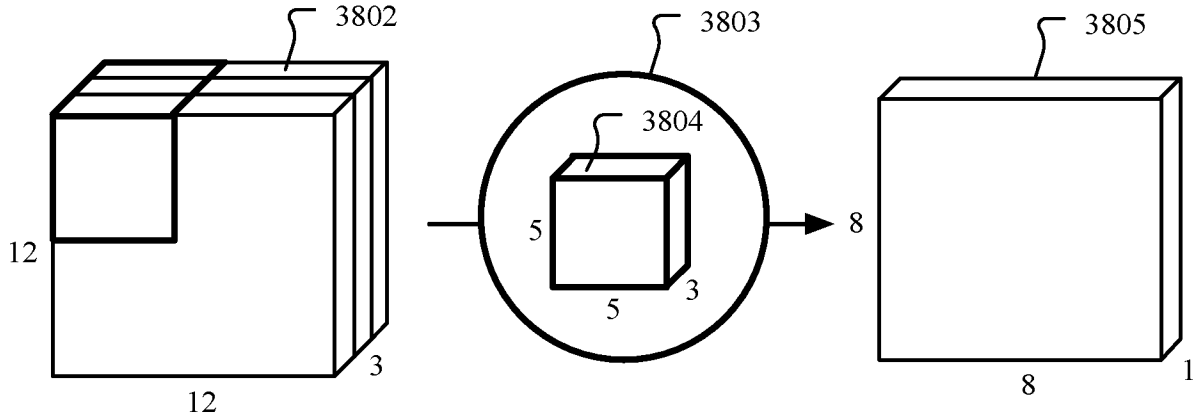


FIG. 38A

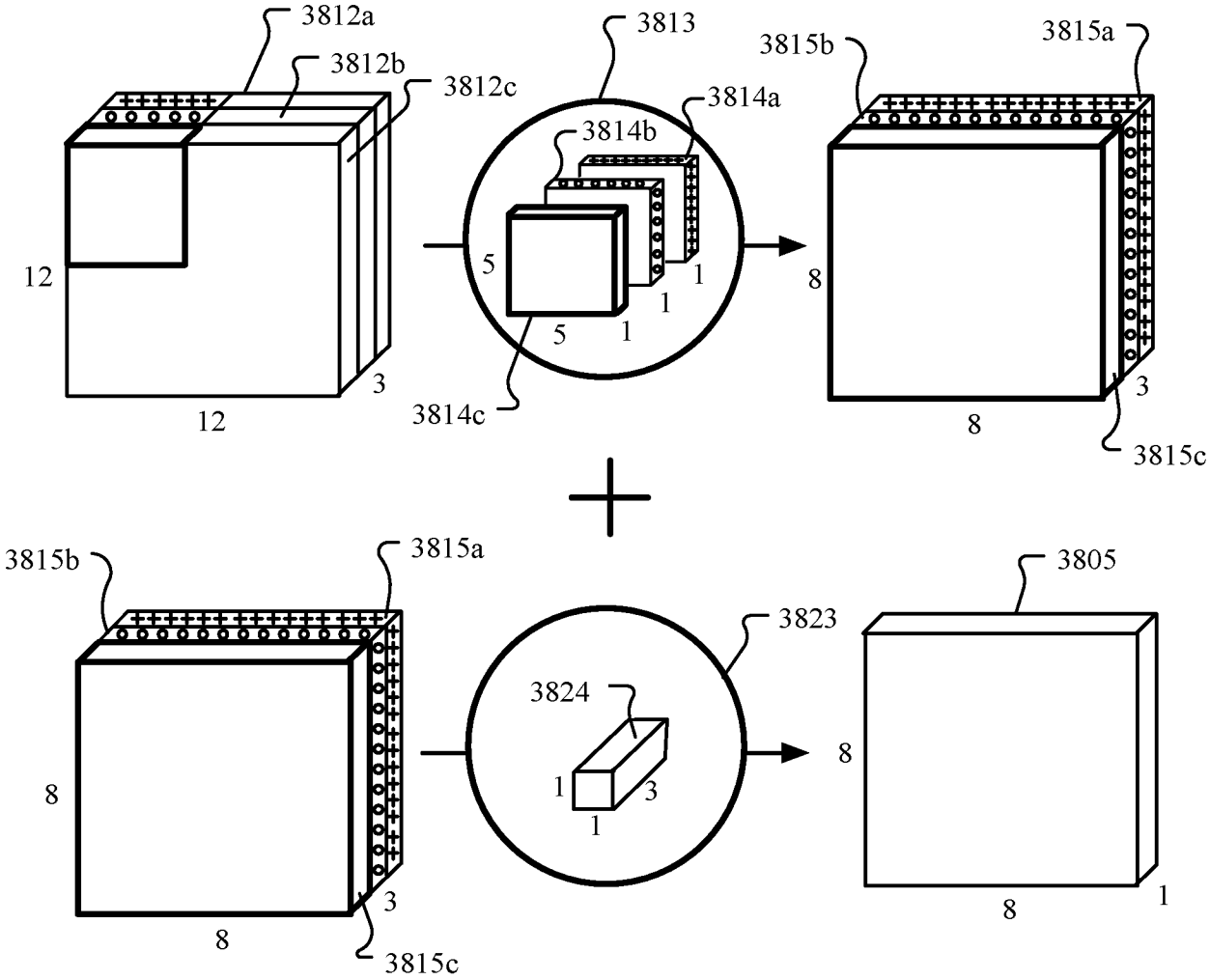


FIG. 38B

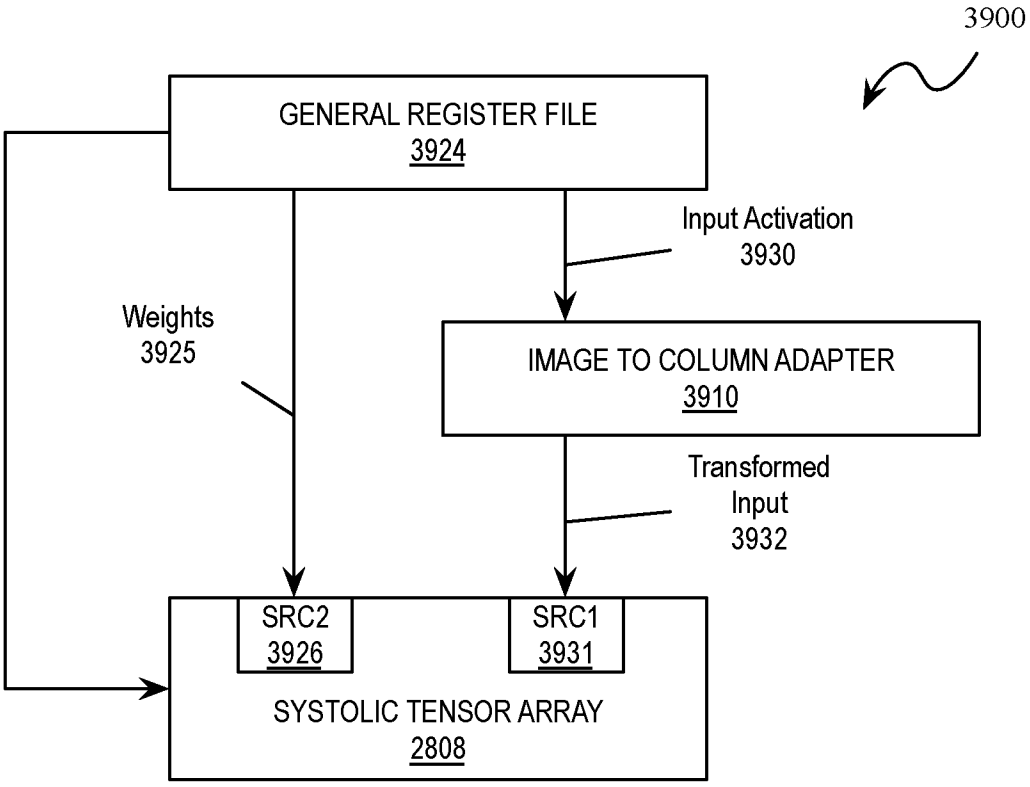


FIG. 39A

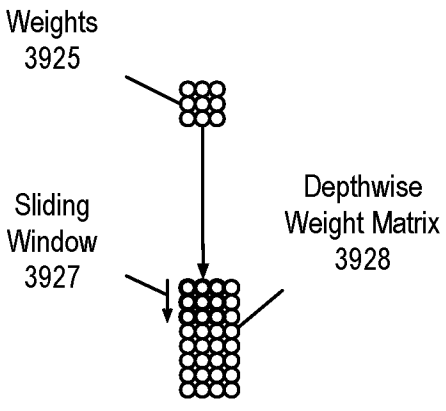


FIG. 39B

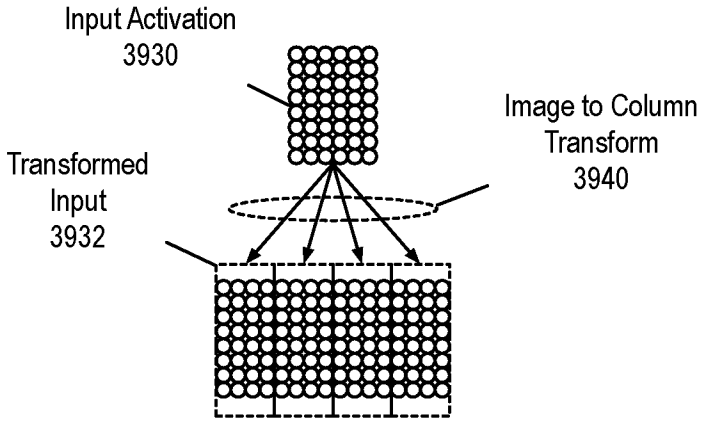


FIG. 39C

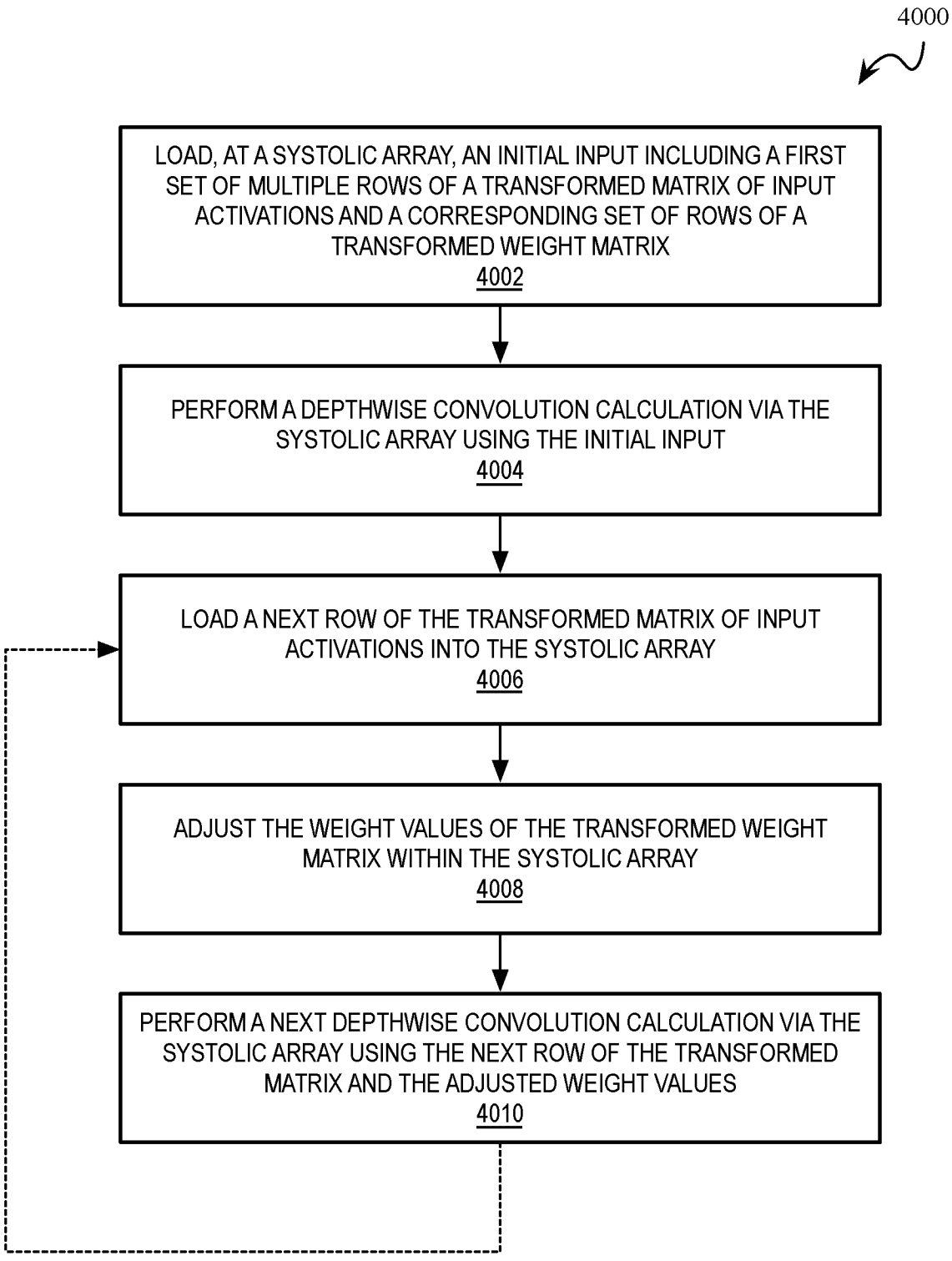

4000
LOAD, AT A SYSTOLIC ARRAY, AN INITIAL INPUT INCLUDING A FIRST SET OF MULTIPLE ROWS OF A TRANSFORMED MATRIX OF INPUT ACTIVATIONS AND A CORRESPONDING SET OF ROWS OF A TRANSFORMED WEIGHT MATRIX
4002
PERFORM A DEPTHWISE CONVOLUTION CALCULATION VIA THE SYSTOLIC ARRAY USING THE INITIAL INPUT
4004
LOAD A NEXT ROW OF THE TRANSFORMED MATRIX OF INPUT ACTIVATIONS INTO THE SYSTOLIC ARRAY
4006
ADJUST THE WEIGHT VALUES OF THE TRANSFORMED WEIGHT MATRIX WITHIN THE SYSTOLIC ARRAY
4008
PERFORM A NEXT DEPTHWISE CONVOLUTION CALCULATION VIA THE SYSTOLIC ARRAY USING THE NEXT ROW OF THE TRANSFORMED MATRIX AND THE ADJUSTED WEIGHT VALUES
4010

FIG. 40

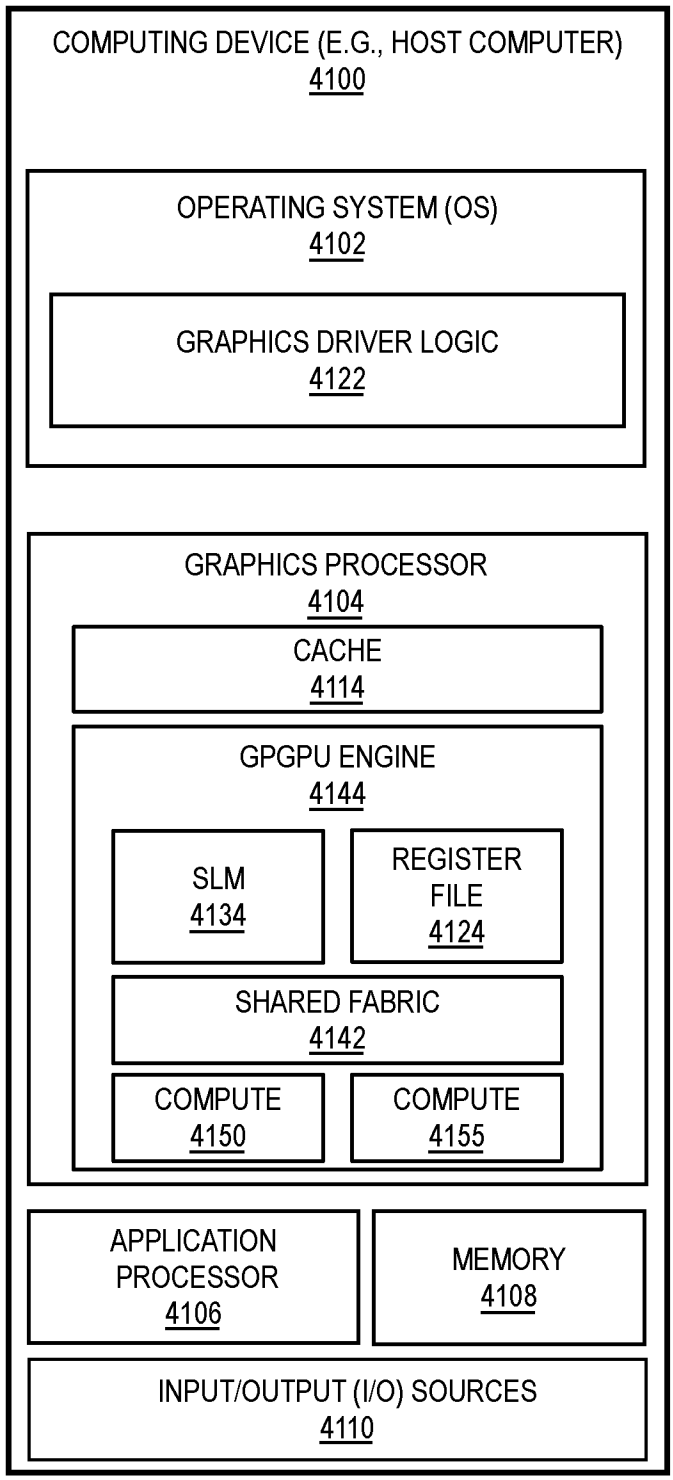

COMPUTING DEVICE (E.G., HOST COMPUTER)
4100
OPERATING SYSTEM (OS)
4102
GRAPHICS DRIVER LOGIC
4122
GRAPHICS PROCESSOR
4104
CACHE
4114
GPGPU ENGINE
4144
SLM
4134
REGISTER FILE
4124
SHARED FABRIC
4142
COMPUTE
4150
COMPUTE
4155
APPLICATION PROCESSOR
4106
MEMORY
4108
INPUT/OUTPUT (I/O) SOURCES
4110

FIG. 41

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2020/022847

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/30 G06F9/38
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	WO 2013/101018 A1 (INTEL CORP [US]; KRISHNAN KARTHIKEYAN [US] ET AL.) 4 July 2013 (2013-07-04) the whole document	1-25
Y	US 2008/071851 A1 (ZOHAR RONEN [US] ET AL) 20 March 2008 (2008-03-20) paragraph [0069] - paragraph [0094]	1-25
A	EP 3 407 183 A2 (INTEL CORP [US]) 28 November 2018 (2018-11-28) the whole document	1-23

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
26 June 2020	06/07/2020

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Moraiti, Marina

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2020/022847

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 2013101018 A1	04-07-2013	CN 104137055 A	05-11-2014
		EP 2798457 A1	05-11-2014
		TW 201349105 A	01-12-2013
		US 2014195783 A1	10-07-2014
		WO 2013101018 A1	04-07-2013
US 2008071851 A1	20-03-2008	CN 101187861 A	28-05-2008
		CN 102004628 A	06-04-2011
		CN 102622203 A	01-08-2012
		CN 105022605 A	04-11-2015
		CN 107741842 A	27-02-2018
		DE 112007002101 T5	09-07-2009
		JP 4697639 B2	08-06-2011
		JP 2008077663 A	03-04-2008
		KR 20090042329 A	29-04-2009
		KR 20110112453 A	12-10-2011
		RU 2009114818 A	27-10-2010
		US 2008071851 A1	20-03-2008
		US 2013290392 A1	31-10-2013
		US 2014032624 A1	30-01-2014
		US 2014032881 A1	30-01-2014
		US 2017364476 A1	21-12-2017
		WO 2008036859 A1	27-03-2008
EP 3407183 A2	28-11-2018	NONE	