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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
31 August 2017 (31.08.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/143502 A1

(51) International Patent Classification:
G06F 9/44 (2006.01)

(21) International Application Number:
PCT/CN2016/074305

(22) International Filing Date:
23 February 2016 (23.02.2016)

(25) Filing Language: English

(26) Publication Language: English

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

(72) Inventors; and

(71) Applicants *(for BZ only)*: **GONG, Zhigang** [CN/CN]; Room 1709, Bldg 7#, Zhi Chun Road, Tai Yue Yuan, Haidian District, Beijing 100088 (CN). **FU, Wenqing** [CN/CN]; No.880 Zixing Rd, Min Hang, Shanghai 200063 (CN). **LI, Peng** [CN/CN]; 20F, Tower D, Beijing Global Trade Center, No. 36 North Third Road East, Dongcheng District, Beijing 100013 (CN). **QUE, Can** [CN/CN]; 880# Zixing Road, Minhang, Shanghai 200241 (CN). **WU, Zhiwen** [CN/CN]; Zixin Road No. 880 Intel Corp., Shanghai 200241 (CN).

(74) Agent: **CHINA PATENT AGENT (H.K.) LTD.**; 22/F., Great Eagle Center, 23 Harbour Road, Wanchai, Hong Kong (CN).

(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, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, 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, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, 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))*

(54) **Title**: OPTIMIZING STRUCTURES TO FIT INTO A COMPLETE CACHE LINE

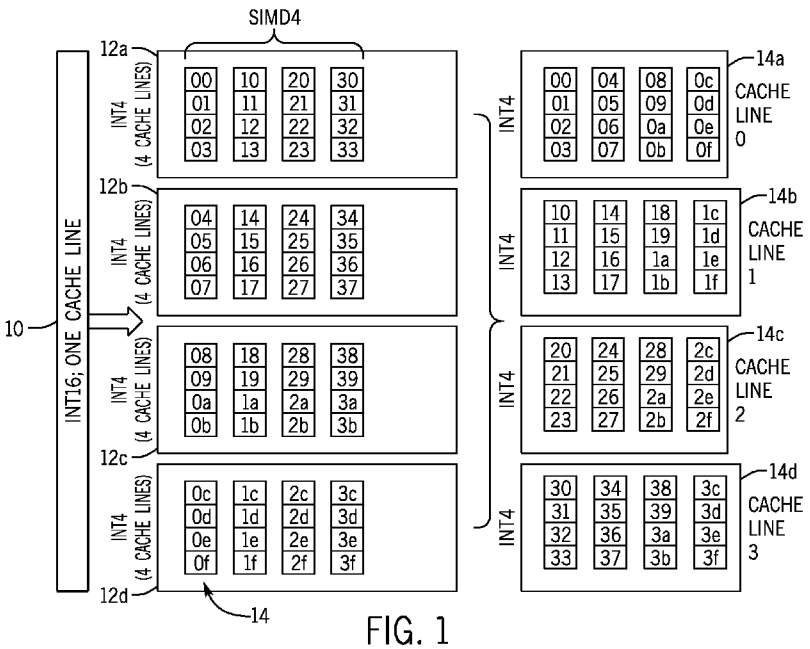


FIG. 1

(57) **Abstract**: An input data structure of a first size may be converted to a plurality of data structures of a second size smaller than the first size. The data structures of the second size are realigned such that each of the plurality of data structures fits in one cache line. The realigned data structures are compiled for use in a vector machine.

WO 2017/143502 A1

OPTIMIZING STRUCTURES TO FIT INTO A COMPLETE CACHE LINE

Background

[0001] This relates generally to graphics processing and particularly to optimizing structures used in graphics processing.

[0002] An example of a structure used in graphics processing is a vector load/store for an OpenCL (Open Computing Language) graphics processing unit backend. See OpenCL 2.1 revision 231 November 11, 2015 available in the Khronos Registry. Vector load and store structures are used in OpenCL to allow reading and writing vector types from a pointer to memory. In addition these load store operations are an important source of input/output intensive workloads.

[0003] For many graphics processing unit backend implementations, an implicitly vectorizing method is used to parallelize a kernel. So the vector type the user is using in the OpenCL kernel code will be translated into a vector in which the first level of the vector is the implicit single instruction multiple data (SIMD) lane, and each lane contains one vector to find in the OpenCL kernel code.

[0004] Typically the graphics processing unit backend cannot handle the full length of the user level vector. For example, even though a 64 byte or 16 double word (int16) cache line size is very common, some graphics processing units can only handle 4 double words (int4) at a time. Thus, to translate an int16 input would involve four int4 store instructions. If an input at SIMD 16 is used, then each int4 store instruction stores 16 int4 vectors at a time. Each int4 vector is a continuous address and the four vectors may be scattered to random addresses. This scattering of these vectors to random addresses results in inefficient retrieval.

[0005] Since the original destination is an int16 containing a complete cache line in some embodiments, the whole cache line is now split four times involving four different writes so it is extremely cache unfriendly.

Brief Description Of The Drawings

[0006] Some embodiments are described with respect to the following figures:

Figure 1 is a conceptual depiction of the flow according to one embodiment;

Figure 2 is a flow chart for one embodiment;

Figure 3 is a depiction of a compiler according to one embodiment;

Figure 4 is a block diagram of a processing system according to one embodiment;

Figure 5 is a block diagram of a processor according to one embodiment;

Figure 6 is a block diagram of a graphics processor according to one embodiment;

Figure 7 is a block diagram of a graphics processing engine according to one embodiment;

Figure 8 is a block diagram of another embodiment of a graphics processor;

Figure 9 is a depiction of thread execution logic according to one embodiment;

Figure 10 is a block diagram of a graphics processor instruction format according to some embodiments;

Figure 11 is a block diagram of another embodiment of a graphics processor;

Figure 12A is a block diagram of a graphics processor command format according to some embodiments;

Figure 12B is a block diagram illustrating a graphics processor command sequence according to some embodiments;

Figure 13 is a depiction of an exemplary graphics software architecture according to some embodiments;

Figure 14 is a block diagram illustrating an IP core development system according to some embodiments; and

Figure 15 is a block diagram showing an exemplary system on chip integrated circuit according to some embodiments.

Detailed Description

[0007] An input data structure of a first size may be converted to a plurality of data structures of a second size smaller than the first size. The data structures of the second size are realigned such that each of the plurality of data structures fits in one cache line. The realigned data structures are compiled for use in a vector machine.

[0008] In accordance with some embodiments, a structure such as a load/store may be rearranged so that the results end up on the same cache line. The OpenCL standard treats each work item as being invisible to the other so this is difficult to accomplish at the source code level. However, arranging results on the same cache line can be accomplished in the compiler backend. A compiler identifies those types of load/store structures easily and shuffles both data and address registers in a low cost fashion, keeping everything transparent to the application level.

[0009] A cache line is the smallest unit of memory that can be transferred between main memory and a cache memory.

[0010] While some embodiments are described with respect to a vector load/store, other structures may also be used so that the results fit into a complete cache line including:

```
struct{

float val[64];

}
```

[0011] In general, any data structure, including 16, 32 and 64 double word (4 byte) data structures, can be used here, particularly where the data structure is to be transformed into vector instructions. So 16, 32, 64 double word data structure all contain one or multiple cachelines in a 64 byte cacheline machine. Therefore, a data structure contains one or a multiple of one full cacheline. While OpenCL is described herein any computing language that can be used with vector machines can be used here.

[0012] By rearranging four int4 load/store operations into four full cache line load stores, a better input/output performance can be achieved for scatter/gather workloads in some embodiments.

[0013] **Figure 1** shows a flow in accordance with one exemplary embodiment using int16. In this case, the int16 input is on one cache line as indicated at 10. Then it is broken into four int4 structures, each int4 structure, such as the structures 12a, 12b and 12c and 12d, have four cache lines within them, in the case of an example using in SIMD4 with four lanes 14. The cache line number is indicated in the smaller boxes followed by the offset within that cache line. Since the first number in each small block is the cache line, there are components in each of four different cache lines in each structure 12a-12d.

[0014] The structures 12 are converted into the structures 14a, 14b, 14c, and 14d in which everything is in the same cache line. Thus, in structure 14a, all the int4 structures are all in cache line zero.

[0015] Since structures 12a-12d, each store instruction handles four different partial cache lines, these four different cache lines are transformed into the data layout 14a-14d. Then each instruction handles one full cache line.

[0016] To implement this solution, the proper size vector load store may be identified. At the compiler intermediate representation (IR) layer, it is then easy to identify an int16 or an equivalent load and store. All the candidate values are collected into a set for the consequent optimization. Next after all the other instruction register optimizations have completed, this optimization on the vector/store sets are collected from the first step. Firstly, the original address register is identified:

origin_addr:addr0_0 addr1_0 addr2_0 addr3_0 (SIMD4)

[0017] Then four new addresses are generated:

new_addr0: addr0_0 addr0_4 addr0_8 addr0_c

[0018] Thereafter, the corresponding result value is determined:

origin_value0 (one int4):

(val00, val10, val20, val30)

(val01, val11, val21, val31)

(val02, val12, val22, val32)

(val03, val13, val23, val33)

[0019] Next the register region feature is used to shuffle the above values to their new values:

new_value0

(val00, val04, val08, val0c)

(val01, val05, val09, val0d)

(val02, val06, val0a, val0e)

(val03, val07, val0b, val0f)

[0020] Finally, the old style four store instructions are replaced with the new four instructions:

store.int4 new_addr0 new_value0

store.int4 new_addr1 new_value1

store.int4 new_addr2 new_value2

store.int4 new_addr3 new_value3

[0021] The address generation is easier than the value shuffle. By identifying that the computation between load store is element-position independent, then the load address can be changed, avoiding shuffling the values. Thus, a four full cache lines can be mapped to one single load store int4 instruction under SIMD16 mode which boosts performance in some embodiments. Particularly in back-to-back load/store operations, there is no need to convert from the shuffled values after the load and before the store, so re-shuffling can be avoided.

[0022] Referring to **Figure 2**, a sequence 20 may be implemented in software, firmware and/or hardware. In software and firmware embodiments it may be implemented by computer executed instructions stored in one or more non-transitory

computer readable media such as magnetic, optical or semiconductor storages. For example, the storage may be part of a graphics processing unit, a processor, a controller, an accelerator, a system on a chip, to mention a few examples.

[0023] The sequence begins at block 22 by identifying, in the example of a vector load store, a vector load store that is of a size that is incompatible (e.g. too large) for a given processor. Then any load stores greater than a certain size is identified in block 24. A set of load stores is greater than that size, are gathered as indicated in block 26.

[0024] Next the original address registers for the data structure are found in block 28. New addresses are generated in block 30. The corresponding result values for multiple cache line data structure are determined in block 32.

[0025] These values are then shuffled to the new values wherein each data structure is in one cache line as indicated in block 34. Finally the old style store is replaced with new style instructions as indicated in block 36.

[0026] In some embodiments, the sequence shown in Figure 2 may be implemented in the backend of a compiler. A compiler is a computer program with a set of programs that transforms source code written in a programming language called the source language into another computer language called the target language with the latter often having a binary form known as object code. As shown in **Figure 3**, the compiler may receive source code in the compiler front end 40, and pass that source code to an intermediate code optimizer 42 that is then processed in the backend/code generator 44.

[0027] The front end verifies a syntax and semantics and generates an intermediate representation or IR the source code for processing by the middle end. It performs type checking by collecting type information, generates errors and warnings if any. The middle end performs optimizations including removal of useless or unreachable code, discovery and propagation of constant values, relocations of computations to less frequently executed places or specialization of computation based on context. It also generates another intermediate representation for the backend.

[0028] The backend generates assembly code, performing register allocation in process. It optimizes the target code utilization of the hardware by figuring out how to keep parallel execution units busy and filling the unused slots. In this case, the backend/code generator 44 may implement the sequence shown in Figure 2. It also does a code generation. The transformed intermediate language is translated into the output language, usually the native machine language of the system.

[0029] **Figure 4** is a block diagram of a processing system 100, according to an embodiment. In various embodiments the system 100 includes one or more processors 102 and one or more graphics processors 108, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 102 or processor cores 107. In on embodiment, the system 100 is a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices. The processing system may implement the compiler shown in Figure 3.

[0030] An embodiment of system 100 can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In some embodiments system 100 is a mobile phone, smart phone, tablet computing device or mobile Internet device. Data processing system 100 can also include, couple with, or be integrated within a wearable device, such as a smart watch wearable device, smart eyewear device, augmented reality device, or virtual reality device. In some embodiments, data processing system 100 is a television or set top box device having one or more processors 102 and a graphical interface generated by one or more graphics processors 108.

[0031] In some embodiments, the one or more processors 102 each include one or more processor cores 107 to process instructions which, when executed, perform operations for system and user software. In some embodiments, each of the one or more processor cores 107 is configured to process a specific instruction set 109. In some embodiments, instruction set 109 may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a Very Long Instruction Word (VLIW). Multiple processor cores 107 may each process

a different instruction set 109, which may include instructions to facilitate the emulation of other instruction sets. Processor core 107 may also include other processing devices, such a Digital Signal Processor (DSP).

[0032] In some embodiments, the processor 102 includes cache memory 104. Depending on the architecture, the processor 102 can have a single internal cache or multiple levels of internal cache. In some embodiments, the cache memory is shared among various components of the processor 102. In some embodiments, the processor 102 also uses an external cache (e.g., a Level-3 (L3) cache or Last Level Cache (LLC)) (not shown), which may be shared among processor cores 107 using known cache coherency techniques. A register file 106 is additionally included in processor 102 which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor 102.

[0033] In some embodiments, processor 102 is coupled to a processor bus 110 to transmit communication signals such as address, data, or control signals between processor 102 and other components in system 100. In one embodiment the system 100 uses an exemplary 'hub' system architecture, including a memory controller hub 116 and an Input Output (I/O) controller hub 130. A memory controller hub 116 facilitates communication between a memory device and other components of system 100, while an I/O Controller Hub (ICH) 130 provides connections to I/O devices via a local I/O bus. In one embodiment, the logic of the memory controller hub 116 is integrated within the processor.

[0034] Memory device 120 can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. In one embodiment the memory device 120 can operate as system memory for the system 100, to store data 122 and instructions 121 for use when the one or more processors 102 executes an application or process. Memory controller hub 116 also couples with an optional external graphics processor 112, which may communicate with the one or more

graphics processors 108 in processors 102 to perform graphics and media operations. The memory device may store instructions to implement the sequence of Figure 2 in one embodiment.

[0035] In some embodiments, ICH 130 enables peripherals to connect to memory device 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller 146, a firmware interface 128, a wireless transceiver 126 (e.g., Wi-Fi, Bluetooth), a data storage device 124 (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller 140 for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 142 connect input devices, such as keyboard and mouse 144 combinations. A network controller 134 may also couple to ICH 130. In some embodiments, a high-performance network controller (not shown) couples to processor bus 110. It will be appreciated that the system 100 shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, the I/O controller hub 130 may be integrated within the one or more processor 102, or the memory controller hub 116 and I/O controller hub 130 may be integrated into a discreet external graphics processor, such as the external graphics processor 112.

[0036] **Figure 5** is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-202N, an integrated memory controller 214, and an integrated graphics processor 208. Those elements of Figure 5 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. Processor 200 can include additional cores up to and including additional core 202N represented by the dashed lined boxes. Each of processor cores 202A-202N includes one or more internal cache units 204A-204N. In some embodiments each processor core also has access to one or more shared cached units 206.

[0037] The internal cache units 204A-204N and shared cache units 206 represent a cache memory hierarchy within the processor 200. The cache memory hierarchy may include at least one level of instruction and data cache within each processor

core and one or more levels of shared mid-level cache, such as a Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the LLC. In some embodiments, cache coherency logic maintains coherency between the various cache units 206 and 204A-204N.

[0038] In some embodiments, processor 200 may also include a set of one or more bus controller units 216 and a system agent core 210. The one or more bus controller units 216 manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). System agent core 210 provides management functionality for the various processor components. In some embodiments, system agent core 210 includes one or more integrated memory controllers 214 to manage access to various external memory devices (not shown).

[0039] In some embodiments, one or more of the processor cores 202A-202N include support for simultaneous multi-threading. In such embodiment, the system agent core 210 includes components for coordinating and operating cores 202A-202N during multi-threaded processing. System agent core 210 may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of processor cores 202A-202N and graphics processor 208.

[0040] In some embodiments, processor 200 additionally includes graphics processor 208 to execute graphics processing operations. In some embodiments, the graphics processor 208 couples with the set of shared cache units 206, and the system agent core 210, including the one or more integrated memory controllers 214. In some embodiments, a display controller 211 is coupled with the graphics processor 208 to drive graphics processor output to one or more coupled displays. In some embodiments, display controller 211 may be a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor 208 or system agent core 210. The graphics processor may implement the compiler of Figure 3, for example.

[0041] In some embodiments, a ring based interconnect unit 212 is used to couple the internal components of the processor 200. However, an alternative interconnect

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

[0042] The exemplary I/O link 213 represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module 218, such as an eDRAM module. In some embodiments, each of the processor cores 202-202N and graphics processor 208 use embedded memory modules 218 as a shared Last Level Cache.

[0043] In some embodiments, processor cores 202A-202N are homogenous cores executing the same instruction set architecture. In another embodiment, processor cores 202A-202N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores 202A-N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set. In one embodiment processor cores 202A-202N are heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. Additionally, processor 200 can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.

[0044] **Figure 6** is a block diagram of a graphics processor 300, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores. In some embodiments, the graphics processor communicates via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor memory. In some embodiments, graphics processor 300 includes a memory interface 314 to access memory. Memory interface 314 can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

[0045] In some embodiments, graphics processor 300 also includes a display controller 302 to drive display output data to a display device 320. Display controller

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

[0046] In some embodiments, graphics processor 300 includes a block image transfer (BLIT) engine 304 to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations are performed using one or more components of graphics processing engine (GPE) 310. In some embodiments, graphics processing engine 310 is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

[0047] In some embodiments, GPE 310 includes a 3D pipeline 312 for performing 3D operations, such as rendering three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline 312 includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media sub-system 315. While 3D pipeline 312 can be used to perform media operations, an embodiment of GPE 310 also includes a media pipeline 316 that is specifically used to perform media operations, such as video post-processing and image enhancement.

[0048] In some embodiments, media pipeline 316 includes fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of video codec engine 306. In some embodiments, media pipeline 316 additionally includes a thread spawning unit to spawn threads for execution on 3D/Media sub-system 315. The spawned threads perform

computations for the media operations on one or more graphics execution units included in 3D/Media sub-system 315.

[0049] In some embodiments, 3D/Media subsystem 315 includes logic for executing threads spawned by 3D pipeline 312 and media pipeline 316. In one embodiment, the pipelines send thread execution requests to 3D/Media subsystem 315, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. In some embodiments, 3D/Media subsystem 315 includes one or more internal caches for thread instructions and data. In some embodiments, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

[0050] **Figure 7** is a block diagram of a graphics processing engine 410 of a graphics processor in accordance with some embodiments. In one embodiment, the GPE 410 is a version of the GPE 310 shown in Figure 6. Elements of Figure 7 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0051] In some embodiments, GPE 410 couples with a command streamer 403, which provides a command stream to the GPE 3D and media pipelines 412, 416. In some embodiments, command streamer 403 is coupled to memory, which can be system memory, or one or more of internal cache memory and shared cache memory. In some embodiments, command streamer 403 receives commands from the memory and sends the commands to 3D pipeline 412 and/or media pipeline 416. The commands are directives fetched from a ring buffer, which stores commands for the 3D and media pipelines 412, 416. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The 3D and media pipelines 412, 416 process the commands by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to an execution unit array 414. In some embodiments, execution

unit array 414 is scalable, such that the array includes a variable number of execution units based on the target power and performance level of GPE 410.

[0052] In some embodiments, a sampling engine 430 couples with memory (e.g., cache memory or system memory) and execution unit array 414. In some embodiments, sampling engine 430 provides a memory access mechanism for execution unit array 414 that allows execution array 414 to read graphics and media data from memory. In some embodiments, sampling engine 430 includes logic to perform specialized image sampling operations for media.

[0053] In some embodiments, the specialized media sampling logic in sampling engine 430 includes a de-noise/de-interlace module 432, a motion estimation module 434, and an image scaling and filtering module 436. In some embodiments, de-noise/de-interlace module 432 includes logic to perform one or more of a de-noise or a de-interlace algorithm on decoded video data. The de-interlace logic combines alternating fields of interlaced video content into a single fame of video. The de-noise logic reduces or removes data noise from video and image data. In some embodiments, the de-noise logic and de-interlace logic are motion adaptive and use spatial or temporal filtering based on the amount of motion detected in the video data. In some embodiments, the de-noise/de-interlace module 432 includes dedicated motion detection logic (e.g., within the motion estimation engine 434).

[0054] In some embodiments, motion estimation engine 434 provides hardware acceleration for video operations by performing video acceleration functions such as motion vector estimation and prediction on video data. The motion estimation engine determines motion vectors that describe the transformation of image data between successive video frames. In some embodiments, a graphics processor media codec uses video motion estimation engine 434 to perform operations on video at the macro-block level that may otherwise be too computationally intensive to perform with a general-purpose processor. In some embodiments, motion estimation engine 434 is generally available to graphics processor components to assist with video decode and processing functions that are sensitive or adaptive to the direction or magnitude of the motion within video data.

[0055] In some embodiments, image scaling and filtering module 436 performs image-processing operations to enhance the visual quality of generated images and video. In some embodiments, scaling and filtering module 436 processes image and video data during the sampling operation before providing the data to execution unit array 414.

[0056] In some embodiments, the GPE 410 includes a data port 444, which provides an additional mechanism for graphics subsystems to access memory. In some embodiments, data port 444 facilitates memory access for operations including render target writes, constant buffer reads, scratch memory space reads/writes, and media surface accesses. In some embodiments, data port 444 includes cache memory space to cache accesses to memory. The cache memory can be a single data cache or separated into multiple caches for the multiple subsystems that access memory via the data port (e.g., a render buffer cache, a constant buffer cache, etc.). In some embodiments, threads executing on an execution unit in execution unit array 414 communicate with the data port by exchanging messages via a data distribution interconnect that couples each of the sub-systems of GPE 410.

[0057] **Figure 8** is a block diagram of another embodiment of a graphics processor 500. Elements of Figure 8 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0058] In some embodiments, graphics processor 500 includes a ring interconnect 502, a pipeline front-end 504, a media engine 537, and graphics cores 580A-580N. In some embodiments, ring interconnect 502 couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In some embodiments, the graphics processor is one of many processors integrated within a multi-core processing system.

[0059] In some embodiments, graphics processor 500 receives batches of commands via ring interconnect 502. The incoming commands are interpreted by a command streamer 503 in the pipeline front-end 504. In some embodiments, graphics processor 500 includes scalable execution logic to perform 3D geometry

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

[0060] In some embodiments, graphics processor 500 includes scalable thread execution resources featuring modular cores 580A-580N (sometimes referred to as core slices), each having multiple sub-cores 550A-550N, 560A-560N (sometimes referred to as core sub-slices). In some embodiments, graphics processor 500 can have any number of graphics cores 580A through 580N. In some embodiments, graphics processor 500 includes a graphics core 580A having at least a first sub-core 550A and a second core sub-core 560A. In other embodiments, the graphics processor is a low power processor with a single sub-core (e.g., 550A). In some embodiments, graphics processor 500 includes multiple graphics cores 580A-580N, each including a set of first sub-cores 550A-550N and a set of second sub-cores 560A-560N. Each sub-core in the set of first sub-cores 550A-550N includes at least a first set of execution units 552A-552N and media/texture samplers 554A-554N. Each sub-core in the set of second sub-cores 560A-560N includes at least a second set of execution units 562A-562N and samplers 564A-564N. In some embodiments, each sub-core 550A-550N, 560A-560N shares a set of shared resources 570A-570N. In some embodiments, the shared resources include shared cache memory and pixel operation logic. Other shared resources may also be included in the various embodiments of the graphics processor.

[0061] **Figure 9** illustrates thread execution logic 600 including an array of processing elements employed in some embodiments of a GPE. Elements of Figure 9 having the same reference numbers (or names) as the elements of any other

figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0062] In some embodiments, thread execution logic 600 includes a pixel shader 602, a thread dispatcher 604, instruction cache 606, a scalable execution unit array including a plurality of execution units 608A-608N, a sampler 610, a data cache 612, and a data port 614. In one embodiment the included components are interconnected via an interconnect fabric that links to each of the components. In some embodiments, thread execution logic 600 includes one or more connections to memory, such as system memory or cache memory, through one or more of instruction cache 606, data port 614, sampler 610, and execution unit array 608A-608N. In some embodiments, each execution unit (e.g. 608A) is an individual vector processor capable of executing multiple simultaneous threads and processing multiple data elements in parallel for each thread. In some embodiments, execution unit array 608A-608N includes any number individual execution units.

[0063] In some embodiments, execution unit array 608A-608N is primarily used to execute "shader" programs. In some embodiments, the execution units in array 608A-608N execute 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).

[0064] Each execution unit in execution unit array 608A-608N operates on arrays of data elements. The number of data elements is the "execution size," or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical Arithmetic Logic Units (ALUs) or Floating Point Units (FPUs) for a particular graphics processor. In some embodiments, execution units 608A-608N support integer and floating-point data types.

[0065] The execution unit instruction set includes single instruction multiple data (SIMD) instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (Quad-Word (QW) size data elements), eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

[0066] One or more internal instruction caches (e.g., 606) are included in the thread execution logic 600 to cache thread instructions for the execution units. In some embodiments, one or more data caches (e.g., 612) are included to cache thread data during thread execution. In some embodiments, sampler 610 is included to provide texture sampling for 3D operations and media sampling for media operations. In some embodiments, sampler 610 includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

[0067] During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 600 via thread spawning and dispatch logic. In some embodiments, thread execution logic 600 includes a local thread dispatcher 604 that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units 608A-608N. For example, the geometry pipeline (e.g., 536 of Figure 8) dispatches vertex processing, tessellation, or geometry processing threads to thread execution logic 600 (Figure 9). In some embodiments, thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

[0068] Once a group of geometric objects has been processed and rasterized into pixel data, pixel shader 602 is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In some embodiments, pixel shader 602 calculates the values

of the various vertex attributes that are to be interpolated across the rasterized object. In some embodiments, pixel shader 602 then executes an application programming interface (API)-supplied pixel shader program. To execute the pixel shader program, pixel shader 602 dispatches threads to an execution unit (e.g., 608A) via thread dispatcher 604. In some embodiments, pixel shader 602 uses texture sampling logic in sampler 610 to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

[0069] In some embodiments, the data port 614 provides a memory access mechanism for the thread execution logic 600 output processed data to memory for processing on a graphics processor output pipeline. In some embodiments, the data port 614 includes or couples to one or more cache memories (e.g., data cache 612) to cache data for memory access via the data port.

[0070] **Figure 10** is a block diagram illustrating a graphics processor instruction formats 700 according to some embodiments. In one or more embodiment, the graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. In some embodiments, instruction format 700 described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

[0071] In some embodiments, the graphics processor execution units natively support instructions in a 128-bit format 710. A 64-bit compacted instruction format 730 is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit format 710 provides access to all instruction options, while some options and operations are restricted in the 64-bit format 730. The native instructions available in the 64-bit format 730 vary by embodiment. In some embodiments, the instruction is compacted in part using a set

of index values in an index field 713. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit format 710.

[0072] For each format, instruction opcode 712 defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. In some embodiments, instruction control field 714 enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For 128-bit instructions 710 an exec-size field 716 limits the number of data channels that will be executed in parallel. In some embodiments, exec-size field 716 is not available for use in the 64-bit compact instruction format 730.

[0073] Some execution unit instructions have up to three operands including two source operands, src0 722, src1 722, and one destination 718. In some embodiments, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 724), where the instruction opcode 712 determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

[0074] In some embodiments, the 128-bit instruction format 710 includes an access/address mode information 726 specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction 710.

[0075] In some embodiments, the 128-bit instruction format 710 includes an access/address mode field 726, which specifies an address mode and/or an access mode for the instruction. In one embodiment the access mode to define a data

access alignment for the instruction. Some embodiments support access modes including a 16-byte aligned access mode and a 1-byte aligned access mode, where the byte alignment of the access mode determines the access alignment of the instruction operands. For example, when in a first mode, the instruction 710 may use byte-aligned addressing for source and destination operands and when in a second mode, the instruction 710 may use 16-byte-aligned addressing for all source and destination operands.

[0076] In one embodiment, the address mode portion of the access/address mode field 726 determines whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction 710 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.

[0077] In some embodiments instructions are grouped based on opcode 712 bit-fields to simplify Opcode decode 740. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group 742 includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group 742 shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group 744 (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group 746 includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group 748 includes component-wise arithmetic instructions (e.g., add, multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group 748 performs the arithmetic operations in parallel across data channels. The vector math group 750 includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

[0078] **Figure 11** is a block diagram of another embodiment of a graphics processor 800. Elements of Figure 11 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0079] In some embodiments, graphics processor 800 includes a graphics pipeline 820, a media pipeline 830, a display engine 840, thread execution logic 850, and a render output pipeline 870. In some embodiments, graphics processor 800 is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor 800 via a ring interconnect 802. In some embodiments, ring interconnect 802 couples graphics processor 800 to other processing components, such as other graphics processors or general-purpose processors. Commands from ring interconnect 802 are interpreted by a command streamer 803, which supplies instructions to individual components of graphics pipeline 820 or media pipeline 830.

[0080] In some embodiments, command streamer 803 directs the operation of a vertex fetcher 805 that reads vertex data from memory and executes vertex-processing commands provided by command streamer 803. In some embodiments, vertex fetcher 805 provides vertex data to a vertex shader 807, which performs coordinate space transformation and lighting operations to each vertex. In some embodiments, vertex fetcher 805 and vertex shader 807 execute vertex-processing instructions by dispatching execution threads to execution units 852A, 852B via a thread dispatcher 831.

[0081] In some embodiments, execution units 852A, 852B are an array of vector processors having an instruction set for performing graphics and media operations. In some embodiments, execution units 852A, 852B have an attached L1 cache 851 that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

[0082] In some embodiments, graphics pipeline 820 includes tessellation components to perform hardware-accelerated tessellation of 3D objects. In some embodiments, a programmable hull shader 811 configures the tessellation operations. A programmable domain shader 817 provides back-end evaluation of tessellation output. A tessellator 813 operates at the direction of hull shader 811 and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to graphics pipeline 820. In some embodiments, if tessellation is not used, tessellation components 811, 813, 817 can be bypassed.

[0083] In some embodiments, complete geometric objects can be processed by a geometry shader 819 via one or more threads dispatched to execution units 852A, 852B, or can proceed directly to the clipper 829. In some embodiments, the geometry shader operates on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader 819 receives input from the vertex shader 807. In some embodiments, geometry shader 819 is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

[0084] Before rasterization, a clipper 829 processes vertex data. The clipper 829 may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In some embodiments, a rasterizer/depth 873 in the render output pipeline 870 dispatches pixel shaders to convert the geometric objects into their per pixel representations. In some embodiments, pixel shader logic is included in thread execution logic 850. In some embodiments, an application can bypass the rasterizer 873 and access un-rasterized vertex data via a stream out unit 823.

[0085] The graphics processor 800 has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units 852A, 852B and associated cache(s) 851, texture and media sampler 854, and texture/sampler cache 858 interconnect via a data port 856 to perform memory access and communicate with render output pipeline components of the processor.

In some embodiments, sampler 854, caches 851, 858 and execution units 852A, 852B each have separate memory access paths.

[0086] In some embodiments, render output pipeline 870 contains a rasterizer and depth test component 873 that converts vertex-based objects into an associated pixel-based representation. In some embodiments, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache 878 and depth cache 879 are also available in some embodiments. A pixel operations component 877 performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine 841, or substituted at display time by the display controller 843 using overlay display planes. In some embodiments, a shared L3 cache 875 is available to all graphics components, allowing the sharing of data without the use of main system memory.

[0087] In some embodiments, graphics processor media pipeline 830 includes a media engine 837 and a video front end 834. In some embodiments, video front end 834 receives pipeline commands from the command streamer 803. In some embodiments, media pipeline 830 includes a separate command streamer. In some embodiments, video front-end 834 processes media commands before sending the command to the media engine 837. In some embodiments, media engine 337 includes thread spawning functionality to spawn threads for dispatch to thread execution logic 850 via thread dispatcher 831.

[0088] In some embodiments, graphics processor 800 includes a display engine 840. In some embodiments, display engine 840 is external to processor 800 and couples with the graphics processor via the ring interconnect 802, or some other interconnect bus or fabric. In some embodiments, display engine 840 includes a 2D engine 841 and a display controller 843. In some embodiments, display engine 840 contains special purpose logic capable of operating independently of the 3D pipeline. In some embodiments, display controller 843 couples with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

[0089] In some embodiments, graphics pipeline 820 and media pipeline 830 are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL) and Open Computing Language (OpenCL) from the Khronos Group, the Direct3D library from the Microsoft Corporation, or support may be provided to both OpenGL and D3D. 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.

[0090] **Figure 12A** is a block diagram illustrating a graphics processor command format 900 according to some embodiments. **Figure 12B** is a block diagram illustrating a graphics processor command sequence 910 according to an embodiment. The solid lined boxes in Figure 12A illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format 900 of Figure 12A includes data fields to identify a target client 902 of the command, a command operation code (opcode) 904, and the relevant data 906 for the command. A sub-opcode 905 and a command size 908 are also included in some commands.

[0091] In some embodiments, client 902 specifies the client unit of the graphics device that processes the command data. In some embodiments, a graphics processor command parser examines the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. In some embodiments, the graphics processor client units include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit has a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads

the opcode 904 and, if present, sub-opcode 905 to determine the operation to perform. The client unit performs the command using information in data field 906. For some commands an explicit command size 908 is expected to specify the size of the command. In some embodiments, the command parser automatically determines the size of at least some of the commands based on the command opcode. In some embodiments commands are aligned via multiples of a double word.

[0092] The flow diagram in Figure 12B shows an exemplary graphics processor command sequence 910. In some embodiments, software or firmware of a data processing system that features an embodiment of a graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only as embodiments are not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.

[0093] In some embodiments, the graphics processor command sequence 910 may begin with a pipeline flush command 912 to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In some embodiments, the 3D pipeline 922 and the media pipeline 924 do not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. In some embodiments, pipeline flush command 912 can be used for pipeline synchronization or before placing the graphics processor into a low power state.

[0094] In some embodiments, a pipeline select command 913 is used when a command sequence requires the graphics processor to explicitly switch between pipelines. In some embodiments, a pipeline select command 913 is required only

once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. In some embodiments, a pipeline flush command is 912 is required immediately before a pipeline switch via the pipeline select command 913.

[0095] In some embodiments, a pipeline control command 914 configures a graphics pipeline for operation and is used to program the 3D pipeline 922 and the media pipeline 924. In some embodiments, pipeline control command 914 configures the pipeline state for the active pipeline. In one embodiment, the pipeline control command 914 is used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.

[0096] In some embodiments, return buffer state commands 916 are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write intermediate data during processing. In some embodiments, the graphics processor also uses one or more return buffers to store output data and to perform cross thread communication. In some embodiments, the return buffer state 916 includes selecting the size and number of return buffers to use for a set of pipeline operations.

[0097] The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination 920, the command sequence is tailored to the 3D pipeline 922 beginning with the 3D pipeline state 930, or the media pipeline 924 beginning at the media pipeline state 940.

[0098] The commands for the 3D pipeline state 930 include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based the particular 3D API in use. In some embodiments, 3D pipeline state 930 commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

[0099] In some embodiments, 3D primitive 932 command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive 932 command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive 932 command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. In some embodiments, 3D primitive 932 command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, 3D pipeline 922 dispatches shader execution threads to graphics processor execution units.

[0100] In some embodiments, 3D pipeline 922 is triggered via an execute 934 command or event. In some embodiments, a register write triggers command execution. In some embodiments execution is triggered via a 'go' or 'kick' command in the command sequence. In one embodiment command execution is triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

[0101] In some embodiments, the graphics processor command sequence 910 follows the media pipeline 924 path when performing media operations. In general, the specific use and manner of programming for the media pipeline 924 depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. In some embodiments, the media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general purpose processing cores. In one embodiment, the media pipeline also includes elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using

computational shader programs that are not explicitly related to the rendering of graphics primitives.

[0102] In some embodiments, media pipeline 924 is configured in a similar manner as the 3D pipeline 922. A set of media pipeline state commands 940 are dispatched or placed into in a command queue before the media object commands 942. In some embodiments, media pipeline state commands 940 include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. In some embodiments, media pipeline state commands 940 also support the use one or more pointers to "indirect" state elements that contain a batch of state settings.

[0103] In some embodiments, media object commands 942 supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. In some embodiments, all media pipeline states must be valid before issuing a media object command 942. Once the pipeline state is configured and media object commands 942 are queued, the media pipeline 924 is triggered via an execute command 944 or an equivalent execute event (e.g., register write). Output from media pipeline 924 may then be post processed by operations provided by the 3D pipeline 922 or the media pipeline 924. In some embodiments, GPGPU operations are configured and executed in a similar manner as media operations.

[0104] **Figure 13** illustrates exemplary graphics software architecture for a data processing system 1000 according to some embodiments. In some embodiments, software architecture includes a 3D graphics application 1010, an operating system 1020, and at least one processor 1030. In some embodiments, processor 1030 includes a graphics processor 1032 and one or more general-purpose processor core(s) 1034. The graphics application 1010 and operating system 1020 each execute in the system memory 1050 of the data processing system.

[0105] In some embodiments, 3D graphics application 1010 contains one or more shader programs including shader instructions 1012. The shader language

instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL Shader Language (GLSL). The application also includes executable instructions 1014 in a machine language suitable for execution by the general-purpose processor core 1034. The application also includes graphics objects 1016 defined by vertex data.

[0106] In some embodiments, operating system 1020 is a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. When the Direct3D API is in use, the operating system 1020 uses a front-end shader compiler 1024 to compile any shader instructions 1012 in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. In some embodiments, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application 1010.

[0107] In some embodiments, user mode graphics driver 1026 contains a back-end shader compiler 1027 to convert the shader instructions 1012 into a hardware specific representation. When the OpenGL API is in use, shader instructions 1012 in the GLSL high-level language are passed to a user mode graphics driver 1026 for compilation. In some embodiments, user mode graphics driver 1026 uses operating system kernel mode functions 1028 to communicate with a kernel mode graphics driver 1029. In some embodiments, kernel mode graphics driver 1029 communicates with graphics processor 1032 to dispatch commands and instructions.

[0108] One or more aspects of at least one embodiment may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as “IP cores,” are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model

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

[0109] **Figure 14** is a block diagram illustrating an IP core development system 1100 that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system 1100 may be used to generate modular, re-usable designs that can be incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility 1130 can generate a software simulation 1110 of an IP core design in a high level programming language (e.g., C/C++). The software simulation 1110 can be used to design, test, and verify the behavior of the IP core using a simulation model 1112. The simulation model 1112 may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design can then be created or synthesized from the simulation model 1112. The RTL design 1115 is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design 1115, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

[0110] The RTL design 1115 or equivalent may be further synthesized by the design facility into a hardware model 1120, which may be in a hardware description language (HDL), or some other representation of physical design data. The HDL may be further simulated or tested to verify the IP core design. The IP core design can be stored for delivery to a 3rd party fabrication facility 1165 using non-volatile memory 1140 (e.g., hard disk, flash memory, or any non-volatile storage medium). Alternatively, the IP core design may be transmitted (e.g., via the Internet) over a wired connection 1150 or wireless connection 1160. The fabrication facility 1165 may then fabricate an integrated circuit that is based at least in part on the IP core design. The fabricated integrated circuit can be configured to perform operations in accordance with at least one embodiment described herein.

[0111] **Figure 15** is a block diagram illustrating an exemplary system on a chip integrated circuit 1200 that may be fabricated using one or more IP cores, according to an embodiment. The exemplary integrated circuit includes one or more application processors 1205 (e.g., CPUs), at least one graphics processor 1210, and may additionally include an image processor 1215 and/or a video processor 1220, any of which may be a modular IP core from the same or multiple different design facilities. The integrated circuit includes peripheral or bus logic including a USB controller 1225, UART controller 1230, an SPI/SDIO controller 1235, and an I^2S/I^2C controller 1240. Additionally, the integrated circuit can include a display device 1245 coupled to one or more of a high-definition multimedia interface (HDMI) controller 1250 and a mobile industry processor interface (MIPI) display interface 1255. Storage may be provided by a flash memory subsystem 1260 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller 1265 for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine 1270.

[0112] Additionally, other logic and circuits may be included in the processor of integrated circuit 1200, including additional graphics processors/cores, peripheral interface controllers, or general purpose processor cores.

[0113] The following clauses and/or examples pertain to further embodiments:

One example embodiment may be a method comprising converting an input data structure of a first size to a plurality of data structures of a second size smaller than said first size, realigning said data structures of said second size such that each of said plurality of data structures fits in one cache line, and compiling with said realized data structures for use in a vector machine. The method may also include converting a vector load store data structure. The method may also include in a identifying, in compiler, data structures exceeding a predetermined size. The method may also include locating address registers for said data structures exceeding said predetermined size. The method may also include generating new addresses for said data structures exceeding said predetermined size. The method may also include shuffling values for said data structures exceeding said predetermined size. The method may also include shuffling said values such that

each data structure fits in only one cache line. The method may also include performing said method in an intermediate code optimizer of a compiler.

[0114] Another example embodiment may be at least one or more non-transitory computer readable media storing instructions to perform a sequence comprising converting an input data structure of a first size to a plurality of data structures of a second size smaller than said first size, realigning said data structures of said second size such that each of said plurality of data structures fits in one cache line, and compiling with said realized data structures for use in a vector machine. The media may include said sequence including converting a vector load store data structure The media may include said sequence including further storing instructions to perform a sequence including in a identifying, in compiler, data structures exceeding a predetermined size. The sequence may include said sequence including further storing instructions to perform a sequence including locating address registers for said data structures exceeding said predetermined size. The sequence may include said sequence including further storing instructions to perform a sequence including generating new addresses for said data structures exceeding said predetermined size. The sequence may include said sequence including further storing instructions to perform a sequence including shuffling values for said data structures exceeding said predetermined size. The sequence may include said sequence including further storing instructions to perform a sequence including shuffling said values such that each data structure fits in only one cache line. The sequence may include said sequence including further storing instructions to perform a sequence including performing said method in an intermediate code optimizer of a compiler.

[0115] In another example embodiment may be an apparatus comprising a processor to convert an input data structure of a first size to a plurality of data structures of a second size smaller than said first size, realign said data structures of said second size such that each of said plurality of data structures fits in one cache line, and compile with said realized data structures for use in a vector machine, and a memory coupled to said processor. The apparatus may include said processor to convert a vector load store data structure. The apparatus may include a in compiler

to identify data structures exceeding a predetermined size. The apparatus may include said processor to locate address registers for said data structures exceeding said predetermined size. The apparatus may include said processor to generate new addresses for said data structures exceeding said predetermined size. The apparatus may include said processor to shuffle values for said data structures exceeding said predetermined size. The apparatus may include said processor to shuffle said values such that each data structure fits in only one cache line. The apparatus may include said processor to perform said method in an intermediate code optimizer of a compiler. The apparatus may include wherein said processor is a graphics processing unit. The apparatus may include wherein said apparatus is a system-on-a chip.

[0116] The graphics processing techniques described herein may be implemented in various hardware architectures. For example, graphics functionality may be integrated within a chipset. Alternatively, a discrete graphics processor may be used. As still another embodiment, the graphics functions may be implemented by a general purpose processor, including a multicore processor.

[0117] References throughout this specification to “one embodiment” or “an embodiment” mean that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one implementation encompassed within the present disclosure. Thus, appearances of the phrase “one embodiment” or “in an embodiment” are not necessarily referring to the same embodiment. Furthermore, the particular features, structures, or characteristics may be instituted in other suitable forms other than the particular embodiment illustrated and all such forms may be encompassed within the claims of the present application.

[0118] While a limited number of embodiments have been described, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this disclosure.

What is claimed is:

1. A method comprising:
converting an input data structure of a first size to a plurality of data structures of a second size smaller than said first size;
realigning said data structures of said second size such that each of said plurality of data structures fits in one cache line; and
compiling with said realized data structures for use in a vector machine.

2. The method of claim 1 including converting a vector load store data structure.

3. The method of claim 1 including in a identifying, in compiler, data structures exceeding a predetermined size.

4. The method of claim 3 including locating address registers for said data structures exceeding said predetermined size.

5. The method of claim 4 including generating new addresses for said data structures exceeding said predetermined size.

6. The method of claim 5 including shuffling values for said data structures exceeding said predetermined size.

7. The method of claim 6 including shuffling said values such that each data structure fits in only one cache line.

8. The method of claim 1 including performing said method in an intermediate code optimizer of a compiler.

9. One or more non-transitory computer readable media storing instructions to perform a sequence comprising:

converting an input data structure of a first size to a plurality of data structures of a second size smaller than said first size;

realigning said data structures of said second size such that each of said plurality of data structures fits in one cache line; and

compiling with said realized data structures for use in a vector machine.

10. The media of claim 9, further storing instructions to perform a sequence including converting a vector load store data structure.

11. The method of claim 9, further storing instructions to perform a sequence including in a identifying, in compiler, data structures exceeding a predetermined size.

12. The media of claim 11, further storing instructions to perform a sequence including locating address registers for said data structures exceeding said predetermined size.

13. The media of claim 12, further storing instructions to perform a sequence including generating new addresses for said data structures exceeding said predetermined size.

14. The media of claim 13, further storing instructions to perform a sequence including shuffling values for said data structures exceeding said predetermined size.

15. The media of claim 14, further storing instructions to perform a sequence including shuffling said values such that each data structure fits in only one cache line.

16. The media of claim 9, further storing instructions to perform a sequence including performing said method in an intermediate code optimizer of a compiler.

17. An apparatus comprising:

a processor to convert an input data structure of a first size to a plurality of data structures of a second size smaller than said first size, realign said data structures of said second size such that each of said plurality of data structures fits in one cache line, and compile with said realized data structures for use in a vector machine; and

a memory coupled to said processor.

18. The apparatus of claim 17, said processor to convert a vector load store data structure.

19. The apparatus of claim 18, including a in compiler to identify data structures exceeding a predetermined size.

20. The apparatus of claim 19, said processor to locate address registers for said data structures exceeding said predetermined size.

21. The apparatus of claim 20, said processor to generate new addresses for said data structures exceeding said predetermined size.

22. The apparatus of claim 21, said processor to shuffle values for said data structures exceeding said predetermined size.

23. The apparatus of claim 22, said processor to shuffle said values such that each data structure fits in only one cache line.

24. The apparatus of claim 17, said processor to perform said method in an intermediate code optimizer of a compiler.

25. The apparatus of claim 17 wherein said processor is a graphics processing unit.

26. The apparatus of claim 17 wherein said apparatus is a system-on-a chip.

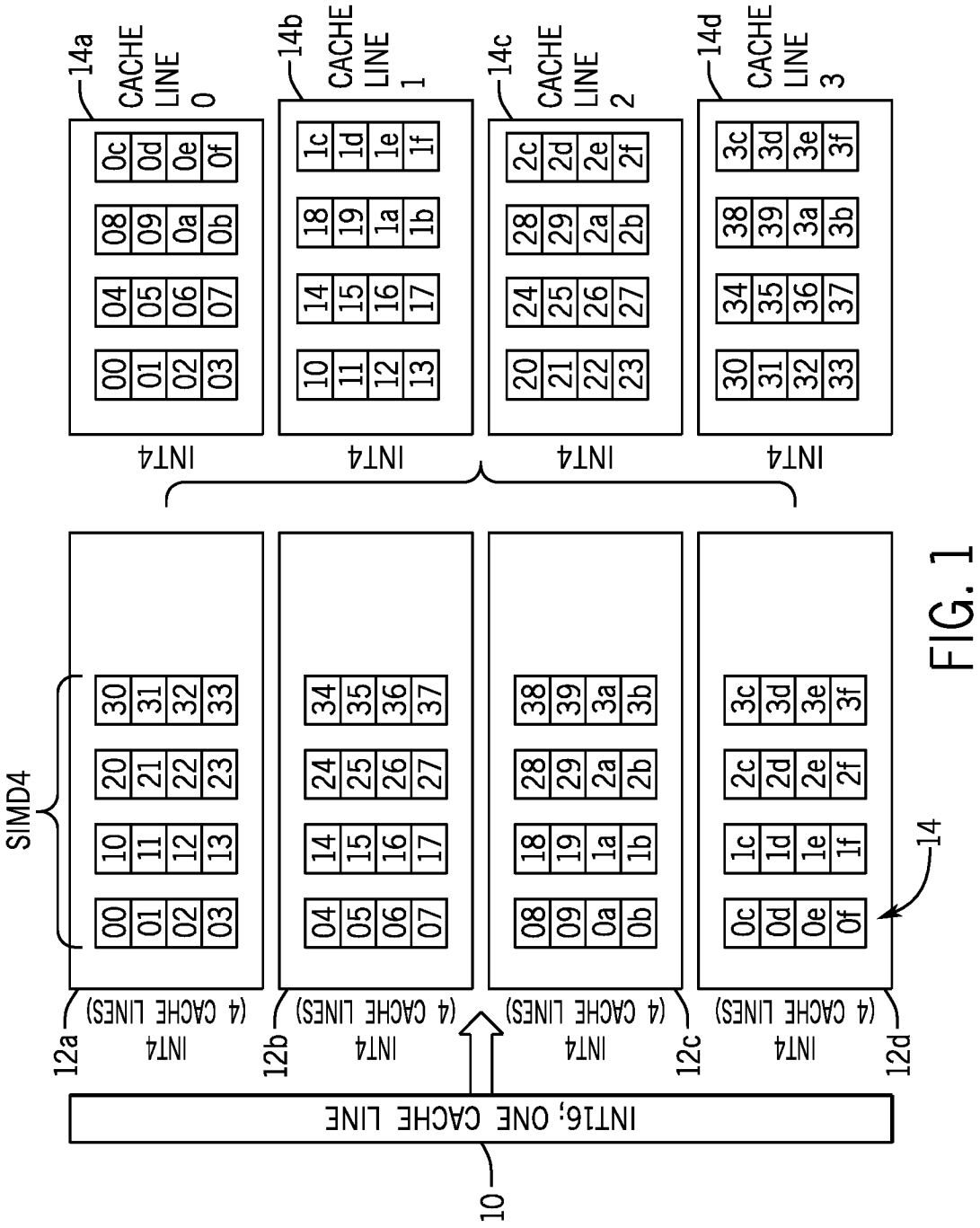

14a
CACHE LINE 0
14b
CACHE LINE 1
14c
CACHE LINE 2
14d
CACHE LINE 3
0c 0d 0e 0f
08 09 0a 0b
04 05 06 07
00 01 02 03
1c 1d 1e 1f
18 19 1a 1b
14 15 16 17
10 11 12 13
2c 2d 2e 2f
28 29 2a 2b
24 25 26 27
20 21 22 23
3c 3d 3e 3f
38 39 3a 3b
34 35 36 37
30 31 32 33
INT4
SIMD4
(4 CACHE LINES)
12a
12b
12c
12d
14
INT16; ONE CACHE LINE
10

FIG. 1

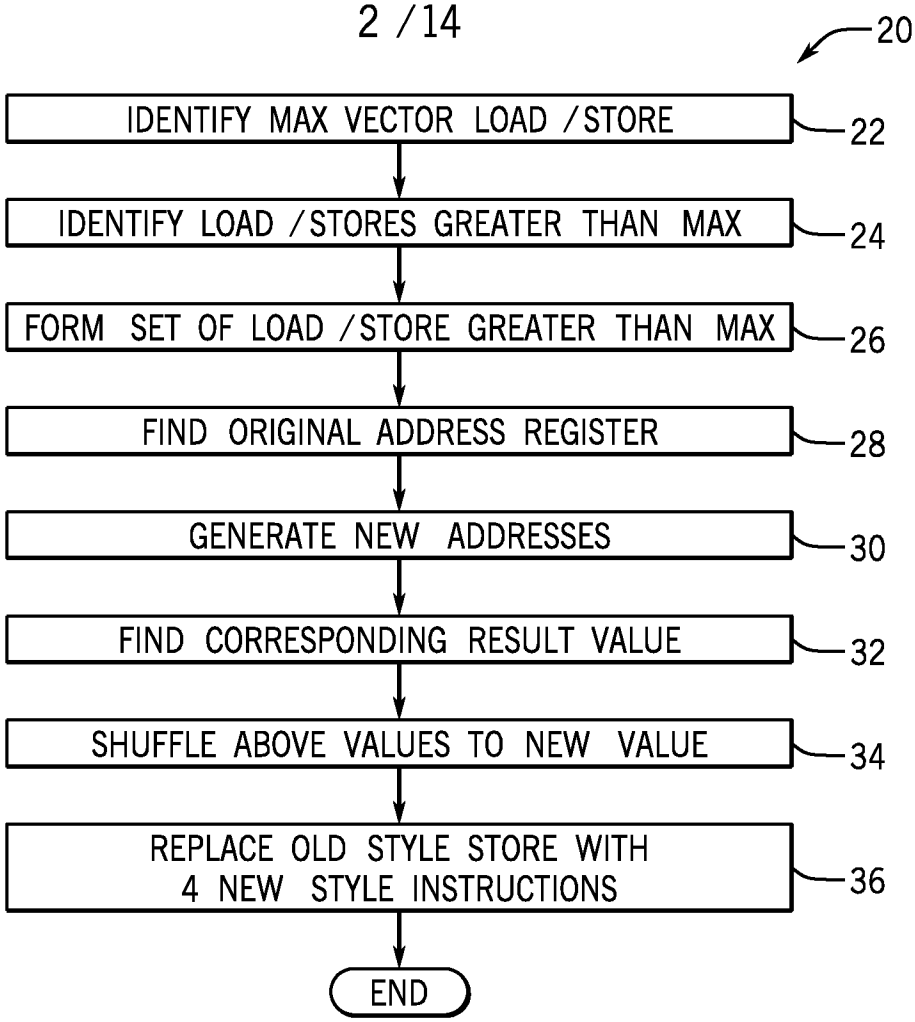

20
IDENTIFY MAX VECTOR LOAD /STORE
22
IDENTIFY LOAD /STORES GREATER THAN MAX
24
FORM SET OF LOAD /STORE GREATER THAN MAX
26
FIND ORIGINAL ADDRESS REGISTER
28
GENERATE NEW ADDRESSES
30
FIND CORRESPONDING RESULT VALUE
32
SHUFFLE ABOVE VALUES TO NEW VALUE
34
REPLACE OLD STYLE STORE WITH
4 NEW STYLE INSTRUCTIONS
36
END

FIG. 2

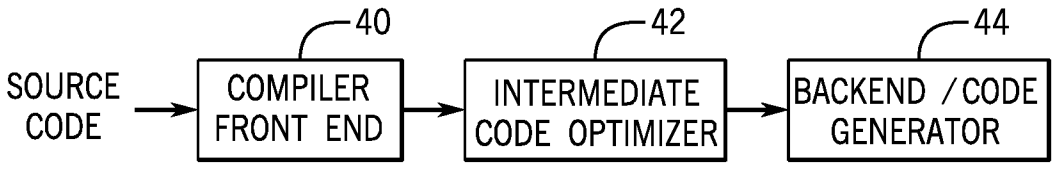

SOURCE CODE
40
COMPILER FRONT END
42
INTERMEDIATE CODE OPTIMIZER
44
BACKEND /CODE GENERATOR

FIG. 3

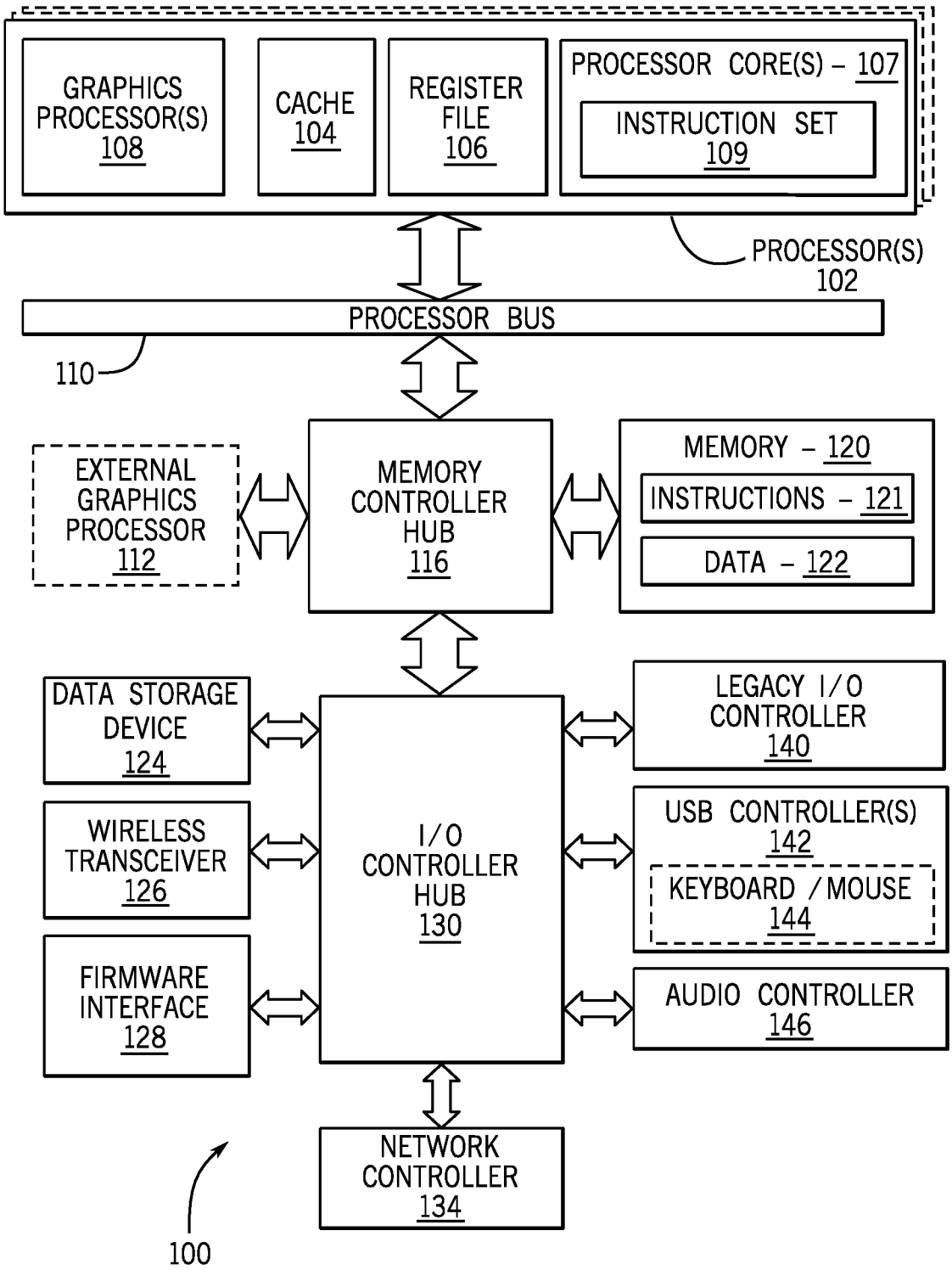

GRAPHICS PROCESSOR(S) 108
CACHE 104
REGISTER FILE 106
PROCESSOR CORE(S) – 107
INSTRUCTION SET 109
PROCESSOR(S) 102
PROCESSOR BUS
110
EXTERNAL GRAPHICS PROCESSOR 112
MEMORY CONTROLLER HUB 116
MEMORY – 120
INSTRUCTIONS – 121
DATA – 122
DATA STORAGE DEVICE 124
WIRELESS TRANSCEIVER 126
FIRMWARE INTERFACE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

FIG. 4

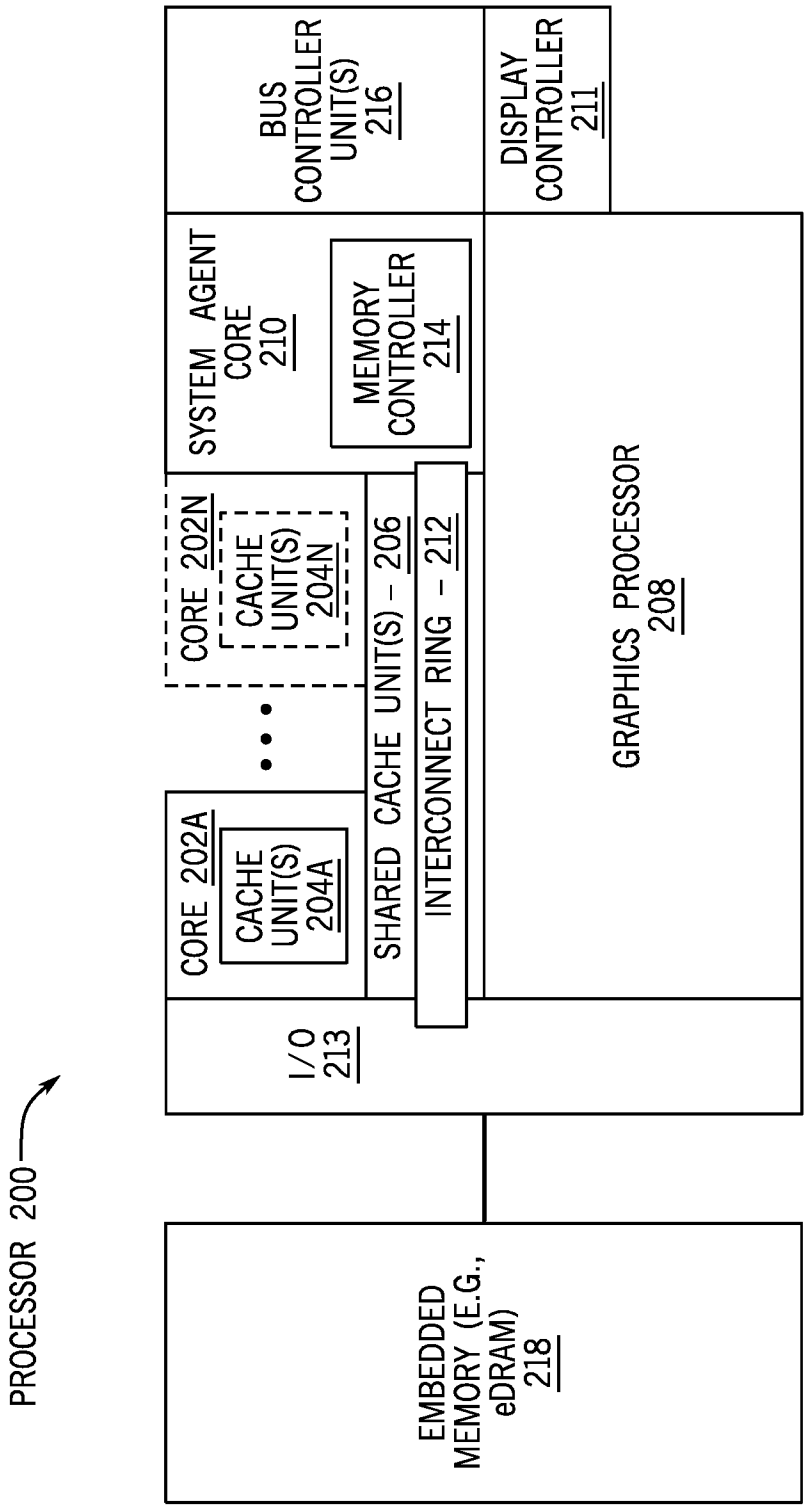

PROCESSOR 200
EMBEDDED MEMORY (E.G., eDRAM) 218
I/O 213
CORE 202A
CACHE UNIT(S) 204A
CORE 202N
CACHE UNIT(S) 204N
SHARED CACHE UNIT(S) – 206
INTERCONNECT RING – 212
GRAPHICS PROCESSOR 208
SYSTEM AGENT CORE 210
MEMORY CONTROLLER 214
BUS CONTROLLER UNIT(S) 216
DISPLAY CONTROLLER 211

FIG. 5

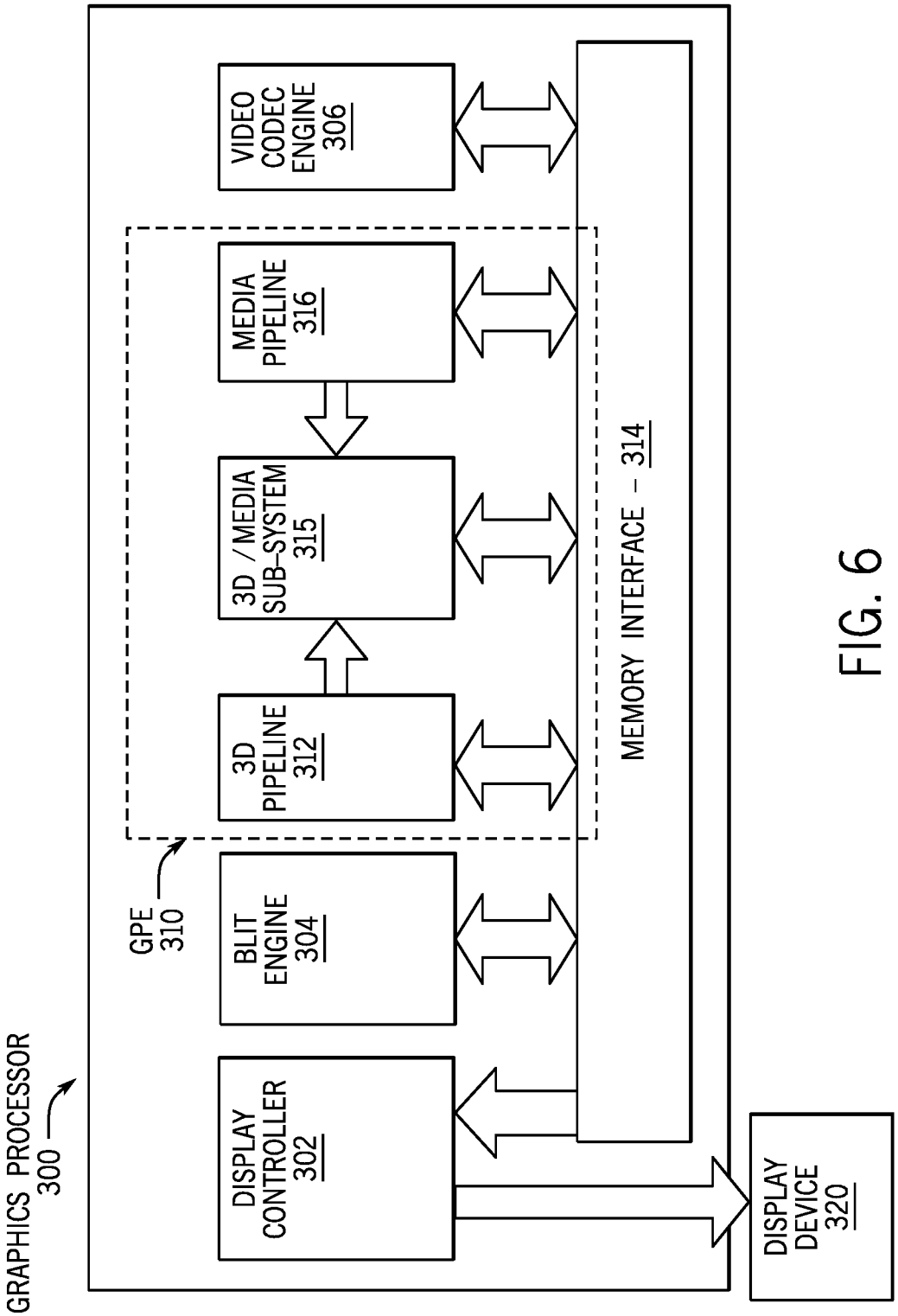

GRAPHICS PROCESSOR 300
DISPLAY CONTROLLER 302
BLIT ENGINE 304
GPE 310
3D PIPELINE 312
3D /MEDIA SUB-SYSTEM 315
MEDIA PIPELINE 316
VIDEO CODEC ENGINE 306
MEMORY INTERFACE – 314
DISPLAY DEVICE 320

FIG. 6

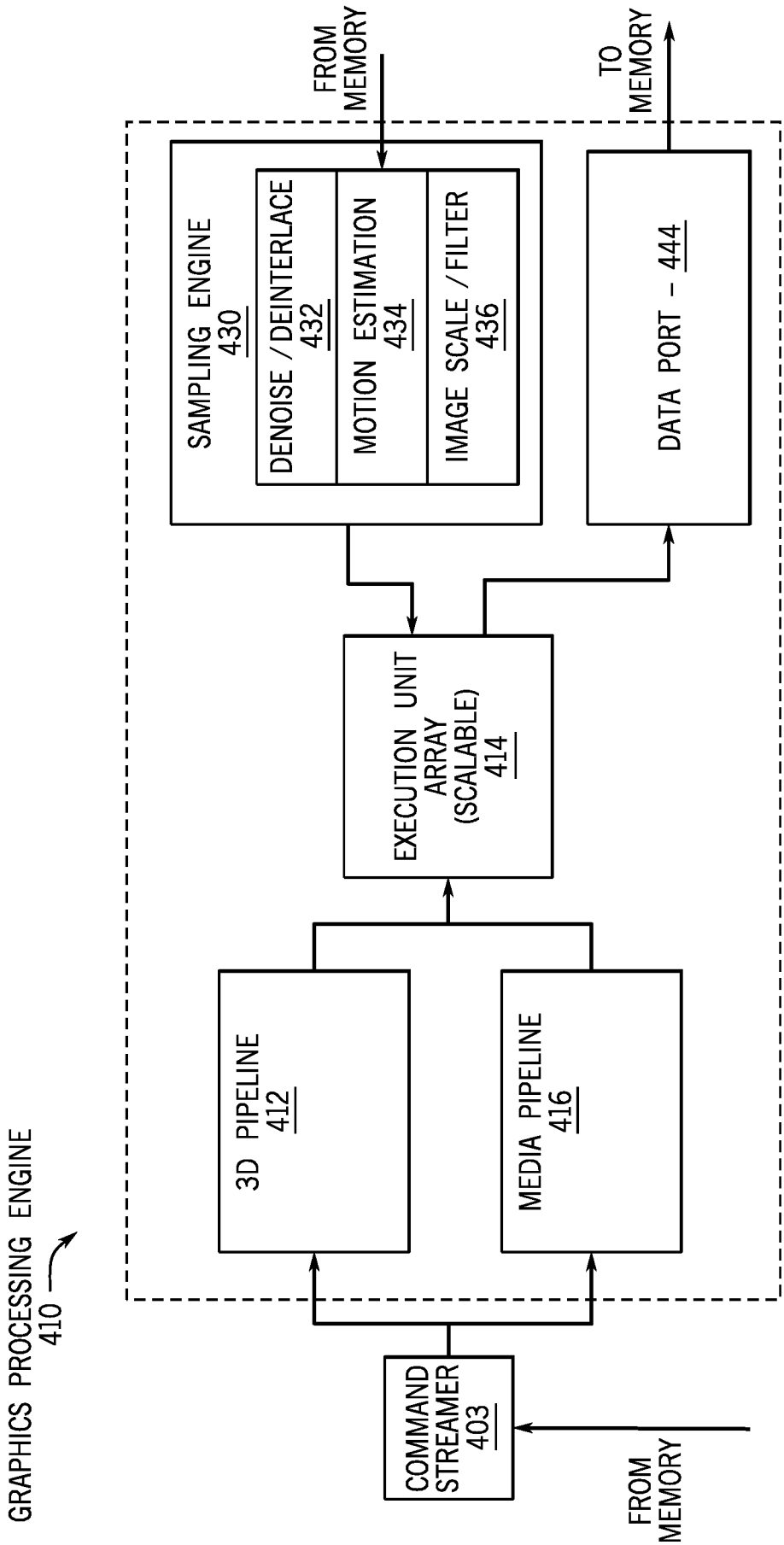

GRAPHICS PROCESSING ENGINE
410
FROM MEMORY
TO MEMORY
SAMPLING ENGINE
430
DENOISE / DEINTERLACE
432
MOTION ESTIMATION
434
IMAGE SCALE / FILTER
436
DATA PORT – 444
EXECUTION UNIT ARRAY (SCALABLE)
414
3D PIPELINE
412
MEDIA PIPELINE
416
COMMAND STREAMER
403
FROM MEMORY

FIG. 7

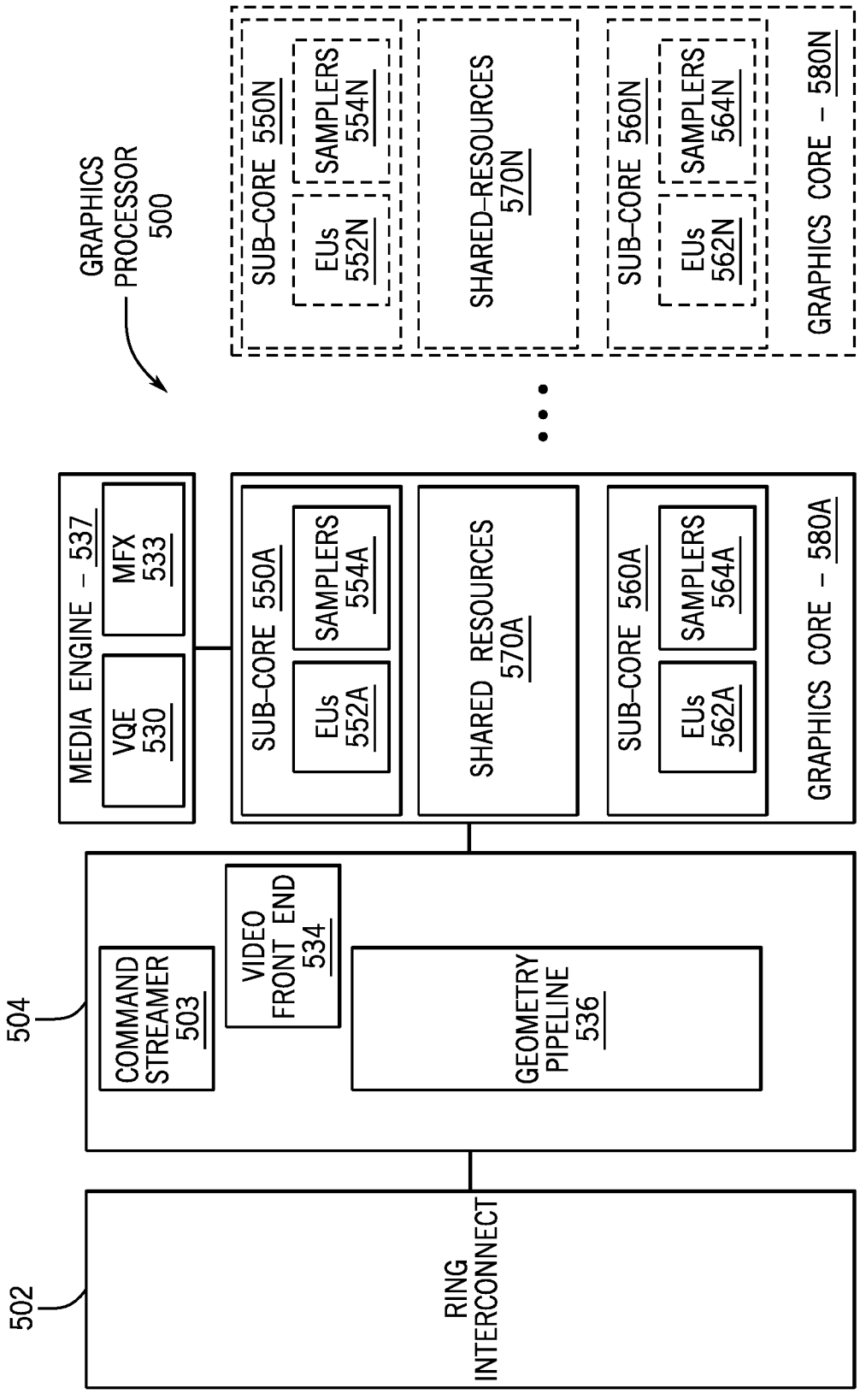

GRAPHICS PROCESSOR 500
SUB-CORE 550N
EUs 552N
SAMPLERS 554N
SHARED-RESOURCES 570N
SUB-CORE 560N
EUs 562N
SAMPLERS 564N
GRAPHICS CORE – 580N
MEDIA ENGINE – 537
VQE 530
MFX 533
SUB-CORE 550A
EUs 552A
SAMPLERS 554A
SHARED RESOURCES 570A
SUB-CORE 560A
EUs 562A
SAMPLERS 564A
GRAPHICS CORE – 580A
504
COMMAND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

FIG. 8

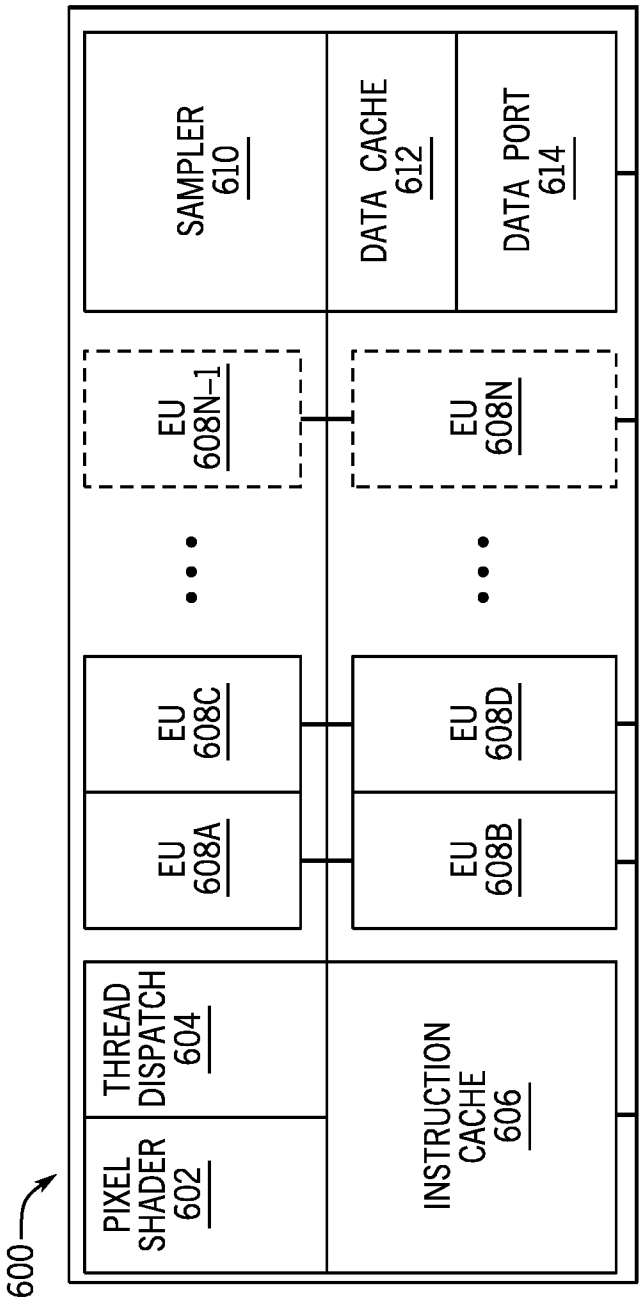

600
PIXEL SHADER 602
THREAD DISPATCH 604
INSTRUCTION CACHE 606
EU 608A
EU 608C
EU 608N–1
EU 608B
EU 608D
EU 608N
SAMPLER 610
DATA CACHE 612
DATA PORT 614

FIG. 9

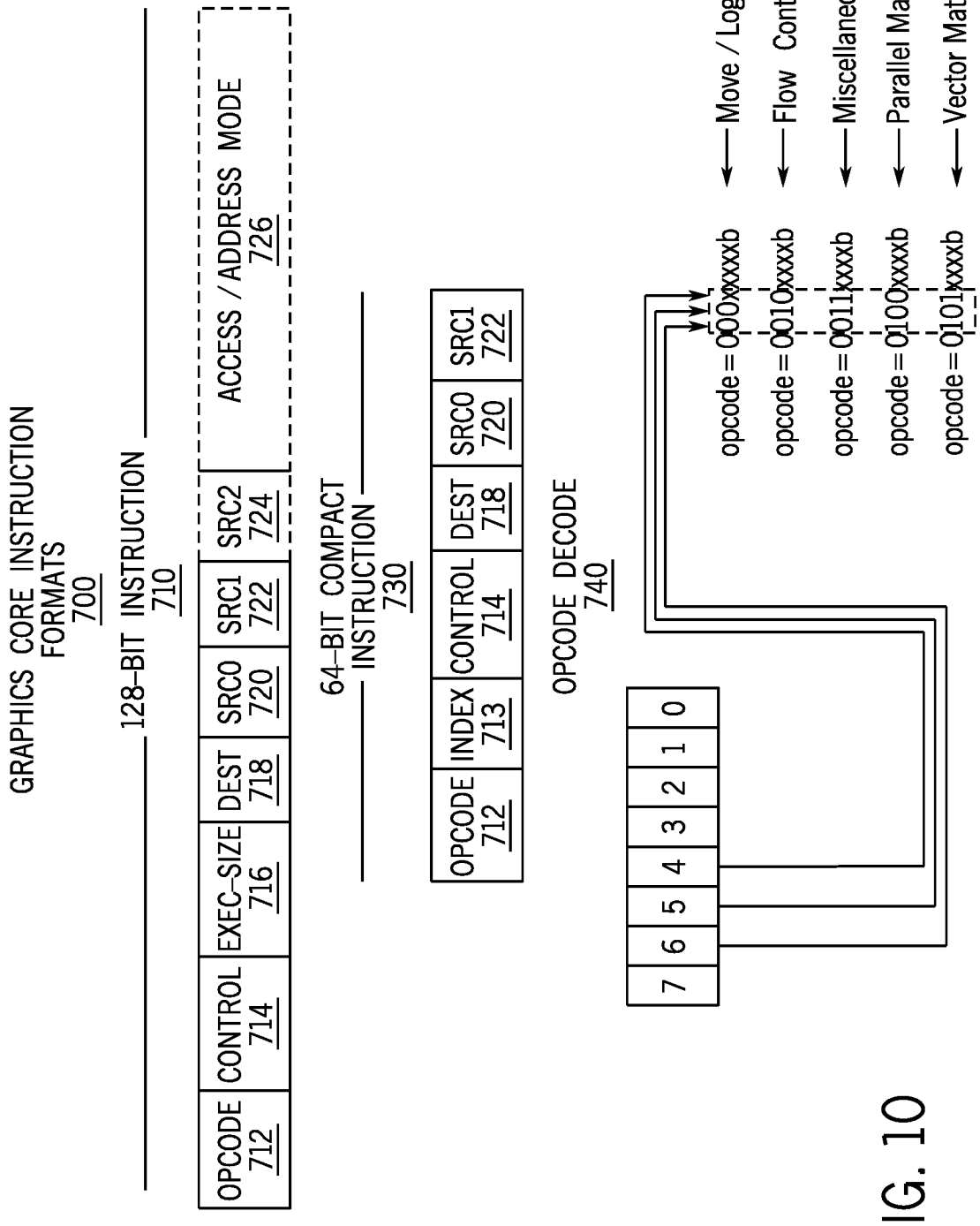

GRAPHICS CORE INSTRUCTION FORMATS 700
128-BIT INSTRUCTION 710
OPCODE 712
CONTROL 714
EXEC-SIZE 716
DEST 718
SRC0 720
SRC1 722
SRC2 724
ACCESS / ADDRESS MODE 726
64-BIT COMPACT INSTRUCTION 730
OPCODE 712
INDEX 713
CONTROL 714
DEST 718
SRC0 720
SRC1 722
OPCODE DECODE 740
7 6 5 4 3 2 1 0
opcode = 000xxxxb
opcode = 001xxxxb
opcode = 010xxxxb
opcode = 011xxxxb
opcode = 0100xxxxb
opcode = 0101xxxxb
Move / Logic – 742
Flow Control – 744
Miscellaneous – 746
Parallel Math – 748
Vector Math – 750

FIG. 10

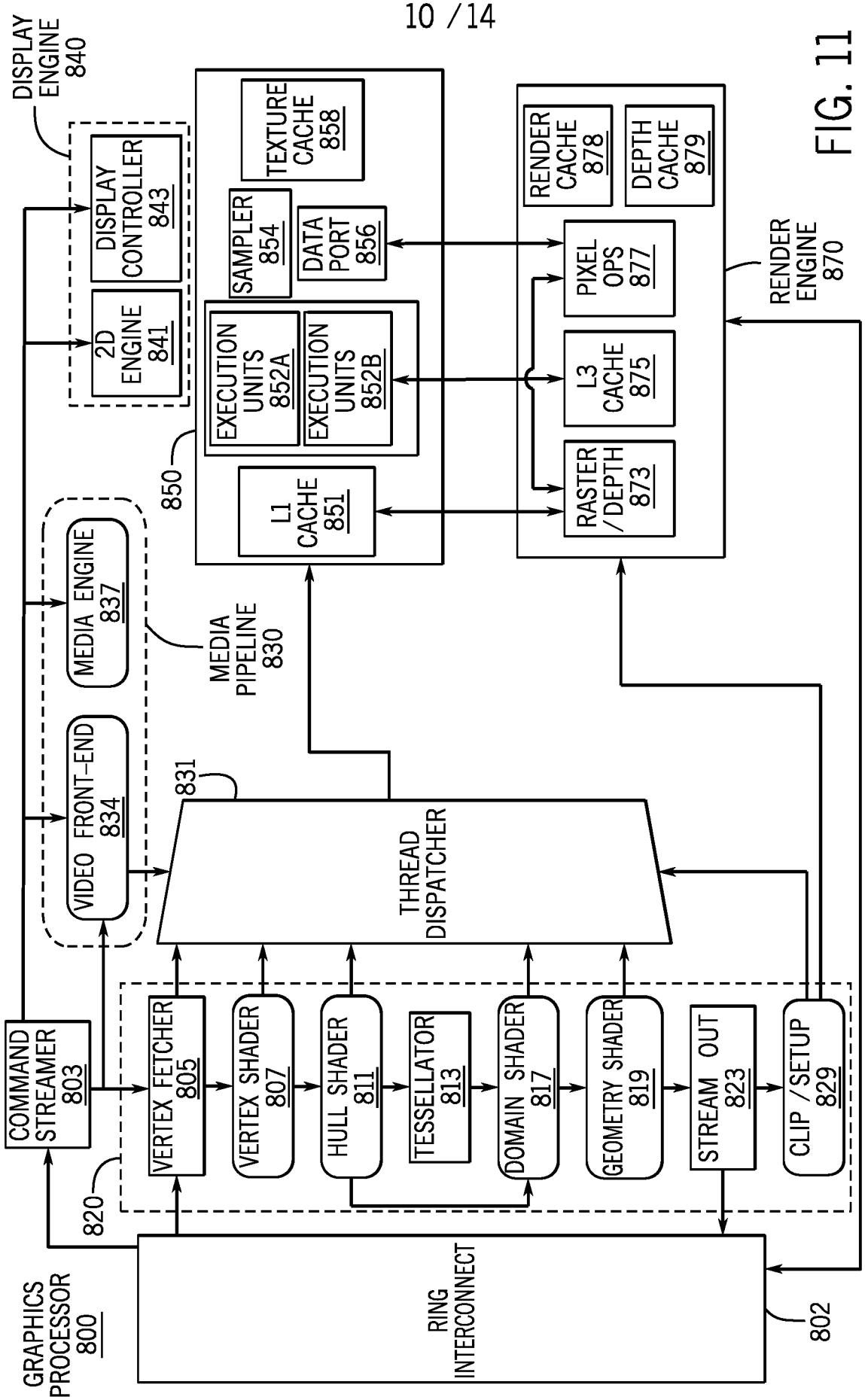

GRAPHICS PROCESSOR 800
RING INTERCONNECT
802
COMMAND STREAMER 803
820
VERTEX FETCHER 805
VERTEX SHADER 807
HULL SHADER 811
TESSELLATOR 813
DOMAIN SHADER 817
GEOMETRY SHADER 819
STREAM OUT 823
CLIP /SETUP 829
THREAD DISPATCHER
831
MEDIA PIPELINE 830
VIDEO FRONT-END 834
MEDIA ENGINE 837
DISPLAY ENGINE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

FIG. 11

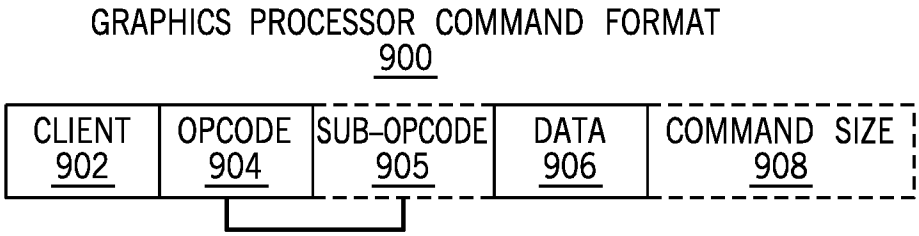


FIG. 12A

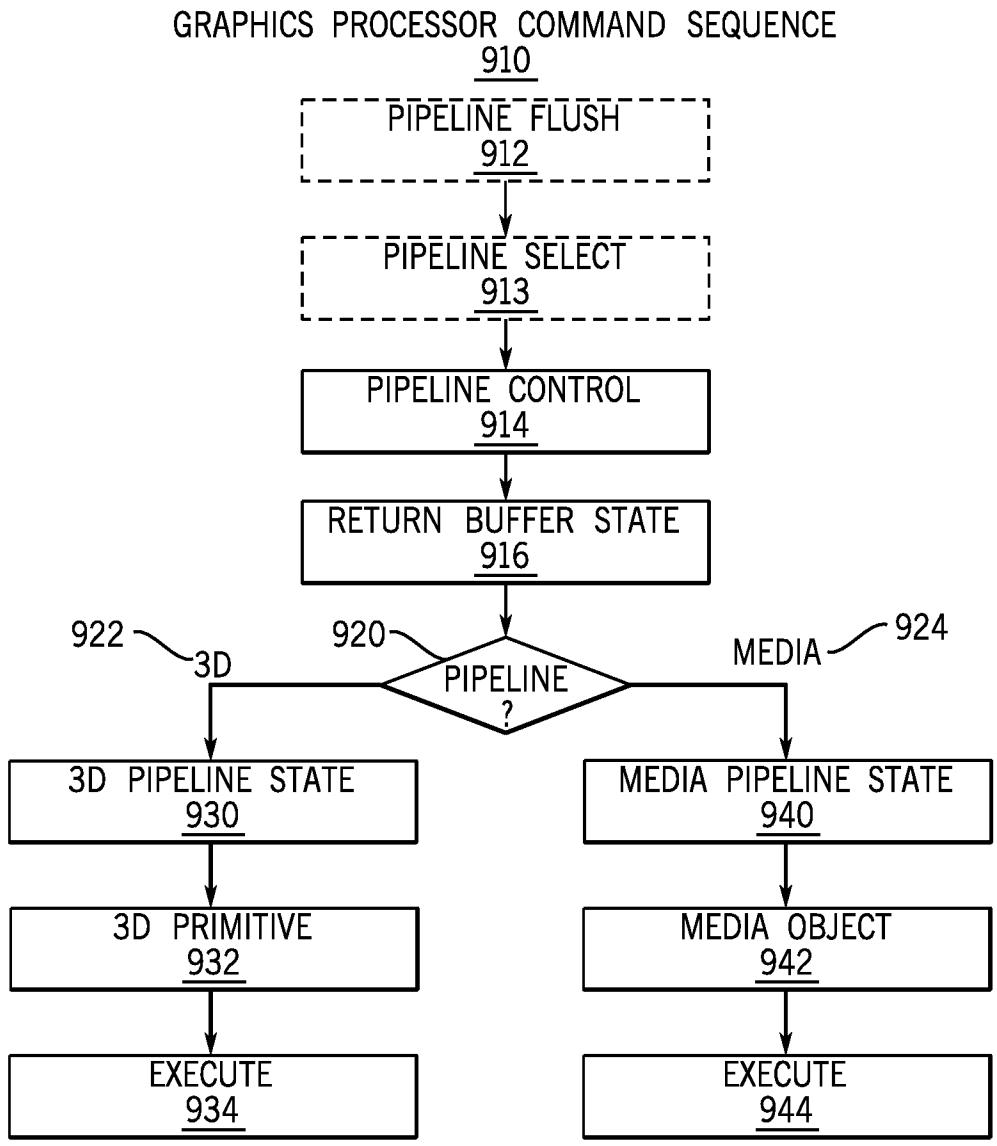


FIG. 12B

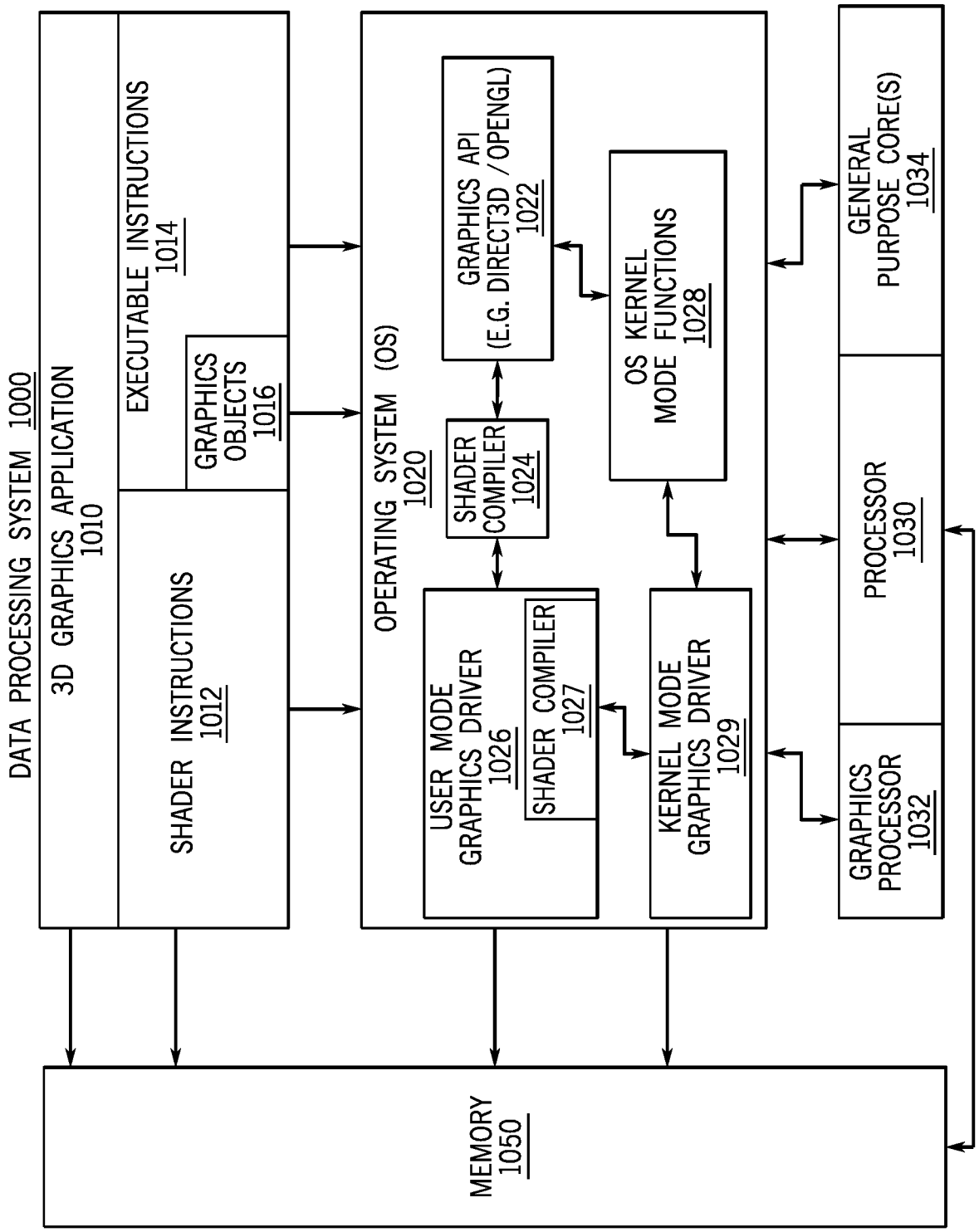

DATA PROCESSING SYSTEM 1000
3D GRAPHICS APPLICATION 1010
EXECUTABLE INSTRUCTIONS 1014
GRAPHICS OBJECTS 1016
SHADER INSTRUCTIONS 1012
OPERATING SYSTEM (OS) 1020
GRAPHICS API (E.G. DIRECT3D /OPENGL) 1022
SHADER COMPILER 1024
OS KERNEL MODE FUNCTIONS 1028
USER MODE GRAPHICS DRIVER 1026
SHADER COMPILER 1027
KERNEL MODE GRAPHICS DRIVER 1029
GENERAL PURPOSE CORE(S) 1034
PROCESSOR 1030
GRAPHICS PROCESSOR 1032
MEMORY 1050

FIG. 13

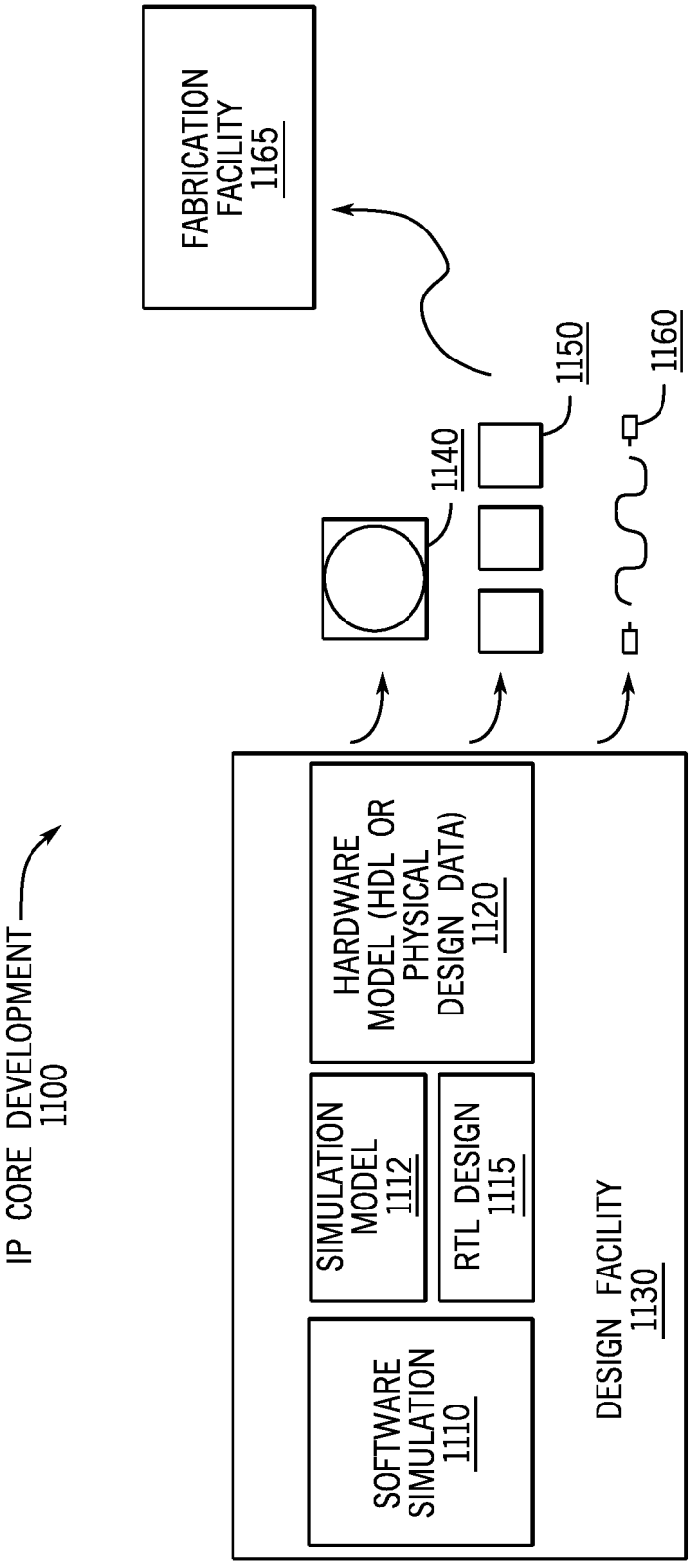

IP CORE DEVELOPMENT
1100
DESIGN FACILITY
1130
SOFTWARE SIMULATION
1110
SIMULATION MODEL
1112
RTL DESIGN
1115
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA)
1120
1140
1150
1160
FABRICATION FACILITY
1165

FIG. 14

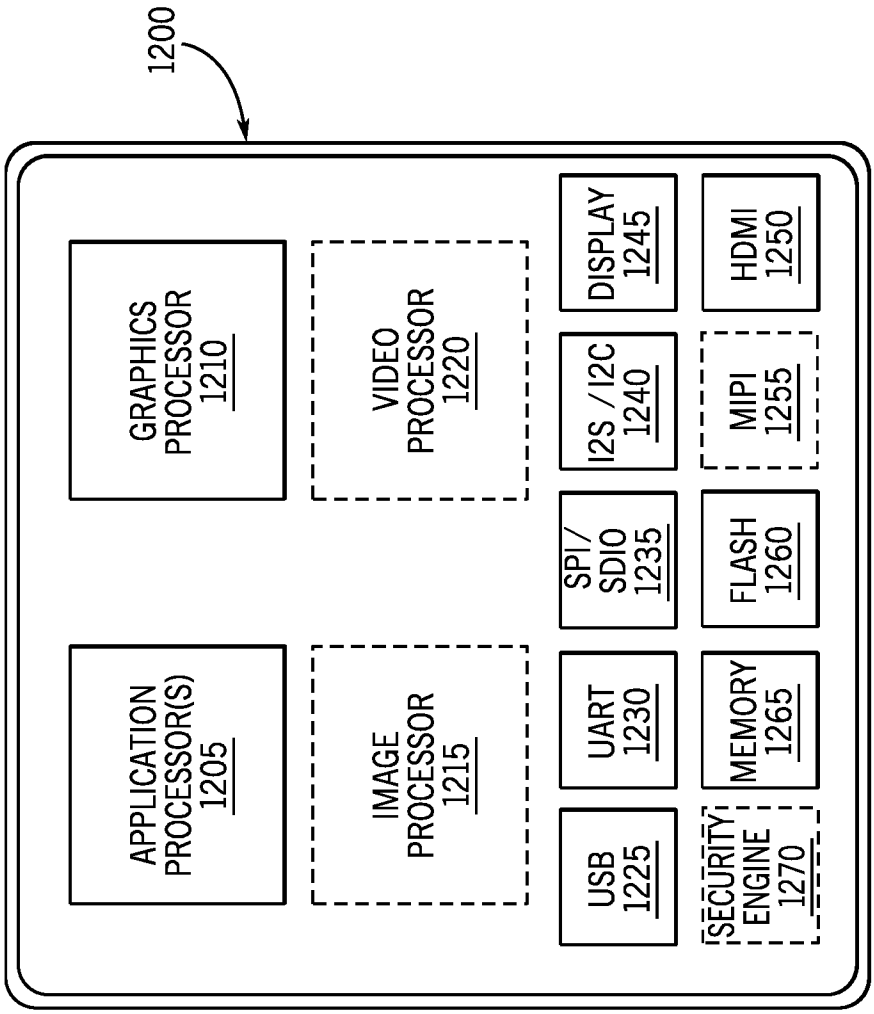

1200
APPLICATION PROCESSOR(S) 1205
GRAPHICS PROCESSOR 1210
IMAGE PROCESSOR 1215
VIDEO PROCESSOR 1220
USB 1225
UART 1230
SPI/ SDIO 1235
I2S / I2C 1240
DISPLAY 1245
SECURITY ENGINE 1270
MEMORY 1265
FLASH 1260
MIPI 1255
HDMI 1250

FIG. 15

INTERNATIONAL SEARCH REPORT

International application No.
PCT/CN2016/074305

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/44(2006.01)i

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)

G06F9/-, G06F17/-, G06F 12/-

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)

CNABS, CNKI, SIPOABS, DWPI: cache, vector, vector machine, seperat+, cache line, opencl, data structure, divid+, data, realign+, convert+

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2009228657 A1 (NEC CORPORATION) 10 September 2009 (2009-09-10) the entire document	1-26
A	US 2012290815 A1 (CANON KABUSHIKI KAISHA) 15 November 2012 (2012-11-15) the entire document	1-26
A	CN 105068971 A (UNIV SOUTHEAST) 18 November 2015 (2015-11-18) the entire document	1-26
A	CN 105022631 A (UNIV XI AN TECHNOLOGY) 04 November 2015 (2015-11-04) the entire document	1-26

☐ 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
01 November 2016	**09 November 2016**

Name and mailing address of the ISA/CN	Authorized officer
STATE INTELLECTUAL PROPERTY OFFICE OF THE P.R.CHINA **6, Xitucheng Rd., Jimen Bridge, Haidian District, Beijing 100088** **China**	**SUN,Yan**
Facsimile No. **(86-10)62019451**	Telephone No. **(86-10)62412061**

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/CN2016/074305

Patent document cited in search report	Publication date (day/month/year)	Patent family member(s)	Publication date (day/month/year)
US 2009228657 A1	10 September 2009	US 8095732 B2	10 January 2012
		JP 4687729 B2	25 May 2011
		JP 2009211368 A	17 September 2009
US 2012290815 A1	15 November 2012	JP 2012242855 A	10 December 2012
		JP 5706754 B2	22 April 2015
		US 9256439 B2	09 February 2016
CN 105068971 A	18 November 2015	None	
CN 105022631 A	04 November 2015	None	