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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
8 June 2017 (08.06.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/095478 A1

(51) International Patent Classification:
G06T 1/20 (2006.01) *G06F 9/54* (2006.01)
G06T 1/60 (2006.01)

(21) International Application Number:
PCT/US2016/043190

(22) International Filing Date:
20 July 2016 (20.07.2016)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
14/954,899 30 November 2015 (30.11.2015) US

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

(72) Inventors: **TOTH, Robert M.**; Parternas Gränd 65, 22647 Lund, M (SE). **HASSELGREN, Jon N.**; Sparvvägen 4, 218 32 Bunkeflostrand, M (SE).

(74) Agents: **MALLIE, Michael J.** et al.; BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP, 1279 Oakmead Parkway, Sunnyvale, California 94085 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, 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.

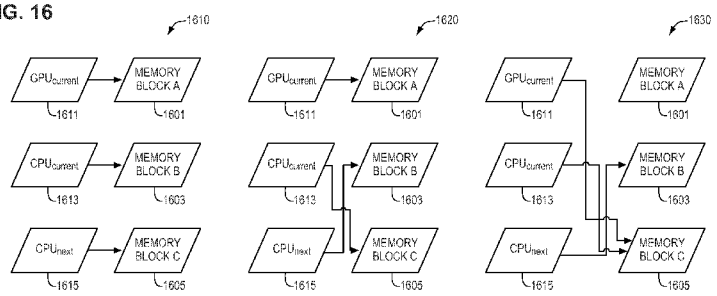
(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, 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: TRIPLE BUFFERED CONSTANT BUFFERS FOR EFFICIENT PROCESSING OF GRAPHICS DATA AT COMPUTING DEVICES

FIG. 16



(57) Abstract: A mechanism is described for facilitating efficient processing of graphics data using triple buffered constant buffers at computing devices. A method of embodiments, as described herein, includes detecting generation of a multi-block buffer by an application to perform data processing at a graphics processor of a computing device, and mapping a first memory block of the multi-block buffer to the graphics processor, where mapping further includes mapping a second memory block and a third memory block of the multi-block buffer to an application processor. The method further includes executing a swap operation to facilitate the graphics processor to process a current data set associated with the application processor.

WO 2017/095478 A1

TRIPLE BUFFERED CONSTANT BUFFERS FOR EFFICIENT PROCESSING OF GRAPHICS DATA AT COMPUTING DEVICES

FIELD

Embodiments described herein generally relate to computers. More particularly, embodiments are described for facilitating efficient processing of graphics data using triple buffered constant buffers at computing devices.

BACKGROUND

Virtual reality (VR) devices, such as Oculus™ Rift™, etc., are known for featuring head and position tracking to allow for user to move and look around in the virtual world. Accordingly, it is essential that the head tracking data is continuously polled and used to update the virtual world, both at a very high rate (such as at least 90Hz) and with low latency, because failure to do so is perceived by the users and can cause discomfort or even sickness, like nausea.

Conventional techniques designed to address these problems (such as trying to reduce any delays between a head tracking input and the rendered image, etc.) are cumbersome and inefficient as they typically require synchronizing graphics processing unit (GPU) and central processing unit (CPU) which causes stalling and reduction in performance.

BRIEF DESCRIPTION OF THE DRAWINGS

Embodiments are illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings in which like reference numerals refer to similar elements.

Figure 1 is a block diagram of a processing system, according to an embodiment.

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

Figure 3 is a block diagram of a graphics processor, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores.

Figure 4 is a block diagram of a graphics processing engine of a graphics processor in accordance with some embodiments.

Figure 5 is a block diagram of another embodiment of a graphics processor.

Figure 6 illustrates thread execution logic including an array of processing elements employed in some embodiments of a graphics processing engine.

Figure 7 is a block diagram illustrating a graphics processor instruction formats according to some embodiments.

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

Figure 9A is a block diagram illustrating a graphics processor command format according to an embodiment and **Figure 9B** is a block diagram illustrating a graphics processor command sequence according to an embodiment.

Figure 10 illustrates exemplary graphics software architecture for a data processing system according to some embodiments.

Figure 11 is a block diagram illustrating an IP core development system that may be used to manufacture an integrated circuit to perform operations according to an embodiment.

Figure 12 is a block diagram illustrating an exemplary system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment.

Figure 13 illustrates a computing device employing a triple buffered constant buffer mechanism according to one embodiment.

Figure 14 illustrates a triple buffered constant buffer mechanism according to one embodiment.

Figures 15A, 15B, and **15C** illustrate conventional techniques requiring synchronization.

Figure 15D illustrates transaction sequence according to one embodiment.

Figure 16 illustrates a transaction sequence according to one embodiment.

Figure 17 illustrates architectural placements according to one embodiment.

Figures 18A, **18B,** and **18C** illustrate methods facilitating mapping, unmapping, and swapping operations, respectively, according to one embodiment.

Figure 19 illustrates method facilitating control and data flow for mapping, unmapping, and swapping operations according to one embodiment.

DETAILED DESCRIPTION

In the following description, numerous specific details are set forth. However, embodiments, as described herein, may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in details in order not to obscure the understanding of this description.

Embodiments provide for a novel technique offering triple buffered constant buffer ("TBCB") that are used to continuously update various contents (such as, but not limited to, head tracking positions for VR devices, etc.) without having to require any CPU-GPU synchronization or cause stalls or reduction in performance.

For example, GPUs are designed to perform as much work in parallel as possible and thus, it is common to buffer up several draw commands (sometimes spanning over multiple frames) that can introduce delay in a rendering pipeline. Consequently, if the head position is

queried at the beginning of issuing commands for a frame, the delay may grow to be considerably big by the time a final rendered content is presented on the user's screen.

Certain conventional techniques approach the delay problem by delaying the reading of a head tracking position for as long as possible before the GPU starts to process any render calls for a frame, and writing that position to a constant buffer.

For example, several applications start by doing view-insensitive rendering, such as environmental maps and shadow map creation, etc., and in such cases, it might be feasible to read the head track position after performing a view-insensitive (also referenced as "view-agnostic" or "view-independent") rendering, reducing the delay a bit; however, it is difficult for an application to know when the GPU has finished the view-independent rendering as doing so requires synchronization of CPU and GPU which, as aforementioned, causes stalling and severe reduction in performance.

In one embodiment, the novel TBCB includes two memory blocks visible to a CPU (including a display driver) and one memory block visible to a GPU. This enables lockless updates from the application. Each time the buffer is mapped by an application, such as map function, a pointer to one of the CPU-visible memory blocks is returned in a cyclic function, while, only one memory block currently updated maybe in an inconsistent state. For example, one the GPU side, a new swap command is introduced into the command stream, where this swap command facilitates swapping of the GPU-visible memory block with the most recently updated CPU-visible memory block that is not current mapped. This swap may be implemented either by swapping the pointers of the GPU-visible and CPU-visible memory blocks or copying the contents of the CPU-visible memory block into the GPU-visible memory block.

It is contemplated that terms like "request", "query", "job", "work", "work item", and "workload" may be referenced interchangeably throughout this document. Similarly, an "application" or "agent" may refer to or include a computer program, a software application, a game, a workstation application, etc., offered thorough an API, such as a free rendering API, such as Open Graphics Library (OpenGL®), DirectX® 11, DirectX® 12, etc., where "dispatch" may be interchangeably referred to as "work unit" or "draw" and similarly, "application" may be interchangeably referred to as "workflow" or simply "agent". For example, a workload, such as that of a 3D game, may include and issue any number and type of "frames" where each frame may represent an image (e.g., sailboat, human face). Further, each frame may include and offer any number and type of work units, where each work unit may represent a part (e.g., mast of sailboat, forehead of human face) of the image (e.g., sailboat, human face) represented by its

corresponding frame. However, for the sake of consistency, each item may be referenced by a single term (e.g., "dispatch", "agent", etc.) throughout this document.

In some embodiments, terms like "display screen" and "display surface" may be used interchangeably referring to the visible portion of a display device while the rest of the display device may be embedded into a computing device, such as a smartphone, a wearable device, etc. It is contemplated and to be noted that embodiments are not limited to any particular computing device, software application, hardware component, display device, display screen or surface, protocol, standard, etc. For example, embodiments may be applied to and used with any number and type of real-time applications on any number and type of computers, such as desktops, laptops, tablet computers, smartphones, head-mounted displays and other wearable devices, and/or the like. Further, for example, rendering scenarios for efficient performance using this novel technique may range from simple scenarios, such as desktop compositing, to complex scenarios, such as 3D games, augmented reality applications, etc.

System Overview

Figure 1 is a block diagram of a processing system 100, according to an embodiment. In various embodiments the system 100 includes one or more processors 102 and one or more graphics processors 108, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 102 or processor cores 107. In 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.

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

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

(RISC), or computing via a Very Long Instruction Word (VLIW). Multiple processor cores 107 may each process a different instruction set 109, which may include instructions to facilitate the emulation of other instruction sets. Processor core 107 may also include other processing devices, such a Digital Signal Processor (DSP).

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

In some embodiments, processor 102 is coupled 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.

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

In some embodiments, ICH 130 enables peripherals to connect to memory device 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller 146, a firmware interface 128, a wireless transceiver 126 (e.g., Wi-Fi, Bluetooth), a data storage device 124 (e.g., hard disk drive, flash memory, etc.), and a legacy I/O

controller 140 for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 142 connect input devices, such as keyboard and mouse 144 combinations. A network controller 134 may also couple 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.

Figure 2 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 **Fig. 2** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. Processor 200 can include additional cores up to and including additional core 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.

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.

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

In some embodiments, one or more of the processor cores 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.

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

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

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

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.

Figure 3 is a block diagram of a graphics processor 300, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores. In some embodiments, the graphics processor communicates via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor

memory. In some embodiments, graphics processor 300 includes a memory interface 314 to access memory. Memory interface 314 can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

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

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

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

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

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

3D/Media Processing

Figure 4 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 **Fig. 3**. Elements of **Fig. 4** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

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.

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.

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).

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.

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.

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.

Execution Units

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

In some embodiments, graphics processor 500 includes a ring interconnect 502, a pipeline front-end 504, a media engine 537, and graphics cores 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.

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.

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.

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

In some embodiments, thread execution logic 600 includes a 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.

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).

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.

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.

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.

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 **Fig. 5**) dispatches vertex processing, tessellation, or geometry processing threads to thread execution logic 600 (**Fig. 6**). In some embodiments, thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

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.

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.

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

In some embodiments, the graphics processor execution units natively support instructions in a 128-bit 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.

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.

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.

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.

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.

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.

In some embodiments instructions are grouped based on opcode 712 bit-fields to simplify Opcode decode 740. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group 742 includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group 742 shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group 744 (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group 746 includes a mix of instructions, including synchronization

instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group 748 includes component-wise arithmetic instructions (e.g., add, multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group 748 performs the arithmetic operations in parallel across data channels. The vector math group 750 includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

Graphics Pipeline

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

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

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

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.

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.

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.

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

The graphics processor 800 has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units 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.

In some embodiments, render output pipeline 870 contains a rasterizer and depth test component 873 that converts vertex-based objects into an associated pixel-based representation. In some embodiments, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache 878 and depth cache 879 are also available in some embodiments. A pixel operations component 877 performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations

(e.g. bit block image transfers with blending) are performed by the 2D engine 841, or substituted at display time by the display controller 843 using overlay display planes. In some embodiments, a shared L3 cache 875 is available to all graphics components, allowing the sharing of data without the use of main system memory.

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

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

In some embodiments, graphics pipeline 820 and media pipeline 830 are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL) 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.

Graphics Pipeline Programming

Figure 9A is a block diagram illustrating a graphics processor command format 900 according to some embodiments. **Figure 9B** is a block diagram illustrating a graphics processor command sequence 910 according to an embodiment. The solid lined boxes in **Fig. 9A** illustrate

the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format 900 of **Fig. 9A** includes data fields to identify a target client 902 of the command, a command operation code (opcode) 904, and the relevant data 906 for the command. A sub-opcode 905 and a command size 908 are also included in some commands.

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

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

In some embodiments, the graphics processor command sequence 910 may begin with a pipeline flush command 912 to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In some embodiments, the 3D pipeline 922 and the media pipeline 924 do not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated.

Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. In some embodiments, pipeline flush command 912 can be used for pipeline synchronization or before placing the graphics processor into a low power state.

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

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

In some embodiments, 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.

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

The commands 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.

In some embodiments, 3D primitive 932 command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the

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

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

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

In some embodiments, media pipeline 924 is configured in a similar manner as the 3D pipeline 922. A set of 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.

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

Graphics Software Architecture

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

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

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

In some embodiments, user mode graphics driver 1026 contains a back-end shader compiler 1027 to convert the shader instructions 1012 into a hardware specific representation. When the OpenGL API is in use, shader instructions 1012 in the GLSL high-level language are

passed to a user mode graphics driver 1026 for compilation. In some embodiments, user mode graphics driver 1026 uses operating system kernel mode functions 1028 to communicate with a kernel mode graphics driver 1029. In some embodiments, kernel mode graphics driver 1029 communicates with graphics processor 1032 to dispatch commands and instructions.

IP Core Implementations

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

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

The RTL design 1115 or equivalent may be further synthesized by the design facility into a hardware model 1120, which may be in a hardware description language (HDL), or some other representation of physical design data. The HDL may be further simulated or tested to verify the

IP core design. The IP core design can be stored for delivery to a 3rd party fabrication facility 1165 using non-volatile memory 1140 (e.g., hard disk, flash memory, or any non-volatile storage medium). Alternatively, the IP core design may be transmitted (e.g., via the Internet) over a wired connection 1150 or wireless connection 1160. The fabrication facility 1165 may then fabricate an integrated circuit that is based at least in part on the IP core design. The fabricated integrated circuit can be configured to perform operations in accordance with at least one embodiment described herein.

Figure 12 is a block diagram illustrating an exemplary system on a chip integrated circuit 1200 that may be fabricated using one or more IP cores, according to an embodiment. 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.

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.

Figure 13 illustrates a computing device 1300 employing a triple buffered constant buffer mechanism 1310 according to one embodiment. Computing device 1300 (e.g., smart wearable devices, VR device, head-mounted display (HMD), mobile computer, Internet of Things (IoT) devices, laptop computer, desktop computer, server computer, etc.) may be the same as data processing system 100 of **Figure 1** and accordingly, for brevity, clarity, and ease of understanding, many of the details stated above with reference to **Figures 1-12** are not further discussed or repeated hereafter. As illustrated, in one embodiment, computing device 1300 is shown as hosting triple buffered constant buffer mechanism ("buffer mechanism") 1310.

In the illustrated embodiment, buffer mechanism 1310 is shown as being hosted by graphics driver 1316; however, it is contemplated that embodiments are not limited as such. For

example, in one embodiment, buffer mechanism 1310 may be part of firmware of GPU 1314 or, in another embodiment, hosted by operating system 1306. In yet another embodiment, buffer mechanism 1310 may be a hardware component hosted by GPU 1314. In yet another embodiment, buffer mechanism 1310 may be partially and simultaneously hosted by multiple components of computing device 1300, such as one or more of driver 1316, GPU 1314, GPU firmware, operating system, and/or the like.

For example, as will be further illustrated with respect to **Figure 17**, although buffer mechanism 110 may host swapping logic to facilitate adding and implementation of adding of a swap command, GPU 1314 may be capable of hosting swapping execution logic 1320 capable of performing or executing the corresponding swap functions, while, for example, an application programming interface (API) for mapping, unmapping, and adding swap commands, etc., may be provided or exposed through graphics API/runtime 1703, in one embodiment, or graphics driver 1316, in another embodiment.

Throughout the document, the term "user" may be interchangeably referred to as "viewer", "observer", "person", "individual", "end-user", and/or the like. It is to be noted that throughout this document, terms like "graphics domain" may be referenced interchangeably with "graphics processing unit" or simply "GPU" and similarly, "CPU domain" or "host domain" may be referenced interchangeably with "computer processing unit" or simply "CPU".

Computing device 1300 may include any number and type of communication devices, such as large computing systems, such as server computers, desktop computers, etc., and may further include set-top boxes (e.g., Internet-based cable television set-top boxes, etc.), global positioning system (GPS)-based devices, etc. Computing device 1300 may include mobile computing devices serving as communication devices, such as cellular phones including smartphones, personal digital assistants (PDAs), tablet computers, laptop computers, e-readers, smart televisions, television platforms, wearable devices (e.g., glasses, watches, bracelets, smartcards, jewelry, clothing items, etc.), media players, etc. For example, in one embodiment, computing device 1300 may include a mobile computing device employing an integrated circuit ("IC"), such as system on a chip ("SoC" or "SOC"), integrating various hardware and/or software components of computing device 1300 on a single chip.

As illustrated, in one embodiment, computing device 1300 may include any number and type of hardware and/or software components, such as (without limitation) graphics processing unit 1314, graphics driver (also referred to as "GPU driver", "graphics driver logic", "driver logic", user-mode driver (UMD), UMD, user-mode driver framework (UMDF), UMDF, or simply "driver") 1316, central processing unit 1312, memory 1308, network devices, drivers, or

the like, as well as input/output (I/O) sources 1304, such as touchscreens, touch panels, touch pads, virtual or regular keyboards, virtual or regular mice, ports, connectors, etc. Computing device 1300 may include operating system (OS) 1306 serving as an interface between hardware and/or physical resources of the computer device 1300 and a user. It is contemplated that CPU 1312 may include one or processors, such as processor(s) 102 of **Figure 1**, while GPU 1314 may include one or more graphics processors, such as graphics processor(s) 108 of **Figure 1**.

It is to be noted that terms like "node", "computing node", "server", "server device", "cloud computer", "cloud server", "cloud server computer", "machine", "host machine", "device", "computing device", "computer", "computing system", and the like, may be used interchangeably throughout this document. It is to be further noted that terms like "application", "software application", "program", "software program", "package", "software package", and the like, may be used interchangeably throughout this document. Also, terms like "job", "input", "request", "message", and the like, may be used interchangeably throughout this document.

It is contemplated and as further described with reference to **Figures 1-12**, some processes of the graphics pipeline as described above are implemented in software, while the rest are implemented in hardware. A graphics pipeline may be implemented in a graphics coprocessor design, where CPU 1312 is designed to work with GPU 1314 which may be included in or co-located with CPU 1312. In one embodiment, GPU 1314 may employ any number and type of conventional software and hardware logic to perform the conventional functions relating to graphics rendering as well as novel software and hardware logic to execute any number and type of instructions, such as instructions 121 of **Figure 1**, to perform the various novel functions of buffer mechanism 1310 as disclosed throughout this document.

As aforementioned, memory 1308 may include a random access memory (RAM) comprising application database having object information. A memory controller hub, such as memory controller hub 116 of **Figure 1**, may access data in the RAM and forward it to GPU 1314 for graphics pipeline processing. RAM may include double data rate RAM (DDR RAM), extended data output RAM (EDO RAM), etc. CPU 1312 interacts with a hardware graphics pipeline, as illustrated with reference to **Figure 3**, to share graphics pipelining functionality. Processed data is stored in a buffer in the hardware graphics pipeline, and state information is stored in memory 1308. The resulting image is then transferred to I/O sources 1304, such as a display component, such as display device 320 of **Figure 3**, for displaying of the image. It is contemplated that the display device may be of various types, such as Cathode Ray Tube (CRT), Thin Film Transistor (TFT), Liquid Crystal Display (LCD), Organic Light Emitting Diode (OLED) array, etc., to display information to a user.

Memory 1308 may comprise a pre-allocated region of a buffer (e.g., frame buffer); however, it should be understood by one of ordinary skill in the art that the embodiments are not so limited, and that any memory accessible to the lower graphics pipeline may be used. Computing device 1300 may further include input/output (I/O) control hub (ICH) 130 as referenced in **Figure 1**, one or more I/O sources 1304, etc.

CPU 1312 may include one or more processors to execute instructions in order to perform whatever software routines the computing system implements. The instructions frequently involve some sort of operation performed upon data. Both data and instructions may be stored in system memory 1308 and any associated cache. Cache is typically designed to have shorter latency times than system memory 1308; for example, cache might be integrated onto the same silicon chip(s) as the processor(s) and/or constructed with faster static RAM (SRAM) cells whilst the system memory 1308 might be constructed with slower dynamic RAM (DRAM) cells. By tending to store more frequently used instructions and data in the cache as opposed to the system memory 1308, the overall performance efficiency of computing device 1300 improves. It is contemplated that in some embodiments, GPU 1314 may exist as part of CPU 1312 (such as part of a physical CPU package) in which case, memory 1308 may be shared by CPU 1312 and GPU 1314 or kept separated.

System memory 1308 may be made available to other components within the computing device 1300. For example, any data (e.g., input graphics data) received from various interfaces to the computing device 1300 (e.g., keyboard and mouse, printer port, Local Area Network (LAN) port, modem port, etc.) or retrieved from an internal storage element of the computer device 1300 (e.g., hard disk drive) are often temporarily queued into system memory 1308 prior to their being operated upon by the one or more processor(s) in the implementation of a software program. Similarly, data that a software program determines should be sent from the computing device 1300 to an outside entity through one of the computing system interfaces, or stored into an internal storage element, is often temporarily queued in system memory 1308 prior to its being transmitted or stored.

Further, for example, an ICH, such as ICH 130 of **Figure 1**, may be used for ensuring that such data is properly passed between the system memory 1308 and its appropriate corresponding computing system interface (and internal storage device if the computing system is so designed) and may have bi-directional point-to-point links between itself and the observed I/O sources/devices 1304. Similarly, an MCH, such as MCH 116 of **Figure 1**, may be used for managing the various contending requests for system memory 1308 accesses amongst CPU 1312 and GPU 1314, interfaces and internal storage elements that may proximately arise in time with

respect to one another.

I/O sources 1304 may include one or more I/O devices that are implemented for transferring data to and/or from computing device 1300 (e.g., a networking adapter); or, for a large scale non-volatile storage within computing device 1300 (e.g., hard disk drive). User input device, including alphanumeric and other keys, may be used to communicate information and command selections to GPU 1314. Another type of user input device is cursor control, such as a mouse, a trackball, a touchscreen, a touchpad, or cursor direction keys to communicate direction information and command selections to GPU 1314 and to control cursor movement on the display device. Camera and microphone arrays of computer device 1300 may be employed to observe gestures, record audio and video and to receive and transmit visual and audio commands.

Computing device 1300 may further include network interface(s) to provide access to a network, such as a LAN, a wide area network (WAN), a metropolitan area network (MAN), a personal area network (PAN), Bluetooth, a cloud network, a mobile network (e.g., 3rd Generation (3G), 4th Generation (4G), etc.), an intranet, the Internet, etc. Network interface(s) may include, for example, a wireless network interface having antenna, which may represent one or more antenna(e). Network interface(s) may also include, for example, a wired network interface to communicate with remote devices via network cable, which may be, for example, an Ethernet cable, a coaxial cable, a fiber optic cable, a serial cable, or a parallel cable.

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

Network interface(s) may include one or more communication interfaces, such as a modem, a network interface card, or other well-known interface devices, such as those used for coupling to the Ethernet, token ring, or other types of physical wired or wireless attachments for purposes of providing a communication link to support a LAN or a WAN, for example. In this manner, the computer system may also be coupled to a number of peripheral devices, clients, control surfaces, consoles, or servers via a conventional network infrastructure, including an

Intranet or the Internet, for example.

It is to be appreciated that a lesser or more equipped system than the example described above may be preferred for certain implementations. Therefore, the configuration of computing device 1300 may vary from implementation to implementation depending upon numerous factors, such as price constraints, performance requirements, technological improvements, or other circumstances. Examples of the electronic device or computer system 1300 may include (without limitation) a mobile device, a personal digital assistant, a mobile computing device, a smartphone, a cellular telephone, a handset, a one-way pager, a two-way pager, a messaging device, a computer, a personal computer (PC), a desktop computer, a laptop computer, a notebook computer, a handheld computer, a tablet computer, a server, a server array or server farm, a web server, a network server, an Internet server, a work station, a mini-computer, a main frame computer, a supercomputer, a network appliance, a web appliance, a distributed computing system, multiprocessor systems, processor-based systems, consumer electronics, programmable consumer electronics, television, digital television, set top box, wireless access point, base station, subscriber station, mobile subscriber center, radio network controller, router, hub, gateway, bridge, switch, machine, or combinations thereof.

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

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

Moreover, embodiments may be downloaded as a computer program product, wherein the program may be transferred from a remote computer (e.g., a server) to a requesting computer (e.g., a client) by way of one or more data signals embodied in and/or modulated by a carrier

wave or other propagation medium via a communication link (e.g., a modem and/or network connection).

Figure 14 illustrates a triple buffered constant buffer mechanism 1310 according to one embodiment. For brevity, many of the details already discussed with reference to **Figures 1-13** are not repeated or discussed hereafter. In one embodiment, buffer mechanism 1310 may include any number and type of components, such as (without limitation): detection/reception logic 1401; buffer logic 1403; mapping logic 1405; unmapping logic 1407; swapping logic 1409; accessing/interfacing logic 1411; and communication/compatibility logic 1413. In one embodiment, GPU 1314 is shown to host swapping execution logic 1320.

Computing device 1300 is further shown to be in communication with one or more repositories, datasets, and/or databases, such as database(s)1430 (e.g., cloud storage, non-cloud storage, etc.), where the communication may be direct and/or over communication medium, such as one or more networks (e.g., a cloud network, a proximity network, a mobile network, an intranet, the Internet, etc.). Computing device 1300 may further in communication with one or more computing devices (e.g., gaming devices, mobile devices, HMD, VR devices, etc.) over one or more networks.

As previously described, buffer mechanism 1310 offers a technique that does not require any explicit CPU-GPU synchronization and accordingly, allows for updating of a TBCB periodically and opportunistically, as desired or necessitated. Further, for example, an application, such as a software application running at or being hosted or facilitated by computing device 1300 may run a thread polling head tracking positions, such as every millisecond (ms), updating the buffer. Accordingly, in this case, the position may be at most 1ms old when GPU 1314 performs the buffer swap, as facilitated by swap logic 1320, just before rendering view-dependent (also referred to as "view-sensitive") draw calls. The period may be adjusted to find the best balance between low latency and low CPU overhead, as will be further illustrated with reference to **Figure 15B**.

It is contemplated and to be noted that although, for the sake of brevity and clarity, a single TBCB is discussed throughout this document, embodiments are not limited as such and that more than one TBCB may be created by an application, such as multiple TBCBs may be generated and operated independently of each other. Further, in one embodiment, each TBCB may have its own set of three underlying memory blocks as well as a set of pointers, such as set of $CPU_{current}$, $GPU_{current}$ and CPU_{next} pointers, where the three memory blocks of a TBCB may or may not be parts of a common single continuous memory block. Further, $CPU_{current}$, $GPU_{current}$ and CPU_{next} pointers may be stored as memory pointers or as indices into an (implicit or explicit)

list of memory pointers.

In one embodiment, detection/reception logic 1401 may be used to detect or received various contents and components, such as CPU 1312 of **Figure 13**, GPU 1314, memory, etc., to facilitate any number and type of operations as facilitated by buffer mechanism 1310. For example, detection/reception logic 1401 may be used to detected with a TBCB is generated, such as by an application or as facilitated by buffer logic 1403, along with detecting TBCB's allocation of memory to store equally sized memory blocks, where for the sake of brevity and clarity, the three memory blocks may be arbitrarily referenced, throughout this document, as memory blocks "A", "B", and "C". Further, for example, as two of the three memory blocks may be visible from the CPU, such as $CPU_{current}$ and CPU_{next}, while the remaining one of the three memory blocks, such as $GPU_{current}$, may be visible from GPU 1314. Now, referring back to the reference A, B, and C, once the TBCB is created, the referenced and the memory blocks may be matched as follows: $CPU_{current}$ = A, CPU_{next} = B, and $GPU_{current}$ =C, as facilitated by block logic 1403.

As aforementioned, three operations may be performed on the TBCB, such as mapping, unmapping, and swapping. In one embodiment, In one embodiment, mapping logic 1405 may be triggered to map set CPU_{next} to a memory block, such as A, that is neither referenced by $CPU_{current}$ nor by $GPU_{current}$, to ensure that contents of the memory block referenced by CPU_{next} are not being, and may not be, read by GPU 1314 until it is unmapped by unmapping logic 1407. If $CPU_{current}$ and $GPU_{current}$ reference the same memory block, then any of the two remaining memory blocks may be assigned to CPU_{next}. Further, in some embodiments, the determination of CPU_{next} may be performed at the end of the unmapping operation as opposed to during the mapping operation such that mapping merely returns the predetermined CPU_{next}. In one embodiment, the mapping function to map TBCB, as facilitated by mapping logic 1405, may simply return the pointer to CPU_{next} to the CPU application.

In one embodiment, unmapping logic 1407 is triggered to set $CPU_{current}$ to CPU_{next}, marking the newly unmapped memory block as being the most recent data available for consumption. As mentioned above, in some embodiments, determination of CPU_{next} may be performed at the end of the unmapping operation.

In one embodiment, mapping and unmapping may be executed synchronously (such as immediately when called) on the CPU by mapping logic 1405 and unmapping logic 1407, respectively, while swapping is issued asynchronously on the CPU as facilitated by swapping logic 1407 and later executed on GPU 1314 as facilitated by swapping execution logic 1320. Graphics commands that reference a TBCB are executed on the GPU read data from the memory

block referenced by $GPU_{current}$.

In one embodiment, the unmapping function, as facilitated by unmapping logic 1407, may swap the $CPU_{current}$ and CPU_{next} pointers, changing the TBCB pointers such that the recently updated memory block becomes "current", while the other memory block becomes the target for the next update.

Further, in one embodiment, swapping logic 1409, in contrast to the mapping and unmapping operations that are executed on the CPU by driver 1316, may insert a swap command into a command stream that is executed by GPU 1314. For example, when executing this function, GPU 1314 makes $GPU_{current}$ target the $CPU_{current}$ data, causing the "current" CPU memory block to become visible to GPU 1314. This may, preferably, be implemented as a simply pointer swap, or by copying the memory, depending on one or more factors, such as hardware limitations of computing device 1300.

Once a swap command is inserted in the command stream or queue, and thus initiated, by swapping logic 1409, swap may then be executed by GPU 1314, as facilitated by swap execution logic 1320, when the swap command is identified and processed from the command stream. For example, in one embodiment, swap gets $GPU_{current}$ to $CPU_{current}$, making any subsequent render commands read from the memory block that, at the time of executing swap, contained the most recent available data.

In one embodiment, accessing/interfacing logic 1411 may be used to facilitate interfacing and performing transactions between the memory blocks, such as A, B, C, of TBCB, such as accessing data in one or more memory blocks.

It is contemplated that head tracking and VR devices and applications are throughout this document, it is contemplated that embodiments are not limited as such. For example, changes to the application, such as main application at computing device 1300, may be relatively minor and upon creation, the constant buffer containing head tracking data may be flagged as a TBCB and further, before submitting viewpoint-dependent draw calls, a swap command may be submitted fro the constant buffer. With regard to head tracking threads, the application may create a thread that polls head tracking data periodically (e.g., every millisecond), where the constant buffer is mapped (e.g., calling ID3D11DeviceContext::Map), the data is updated, and the buffer is unmapped (e.g., calling ID3DIIDeviceContext::Unmap).

Communication/compatibility logic 1413 may be used to facilitate dynamic communication and compatibility between computing device 1300 and any number and type of other computing devices (such as mobile computing device, desktop computer, server computing device, etc.), processing devices (such as CPUs, GPUs, etc.), capturing/sensing/detecting devices

(such as capturing/sensing components including cameras, depth sensing cameras, camera sensors, RGB sensors, microphones, etc.), display devices (such as output components including display screens, display areas, display projectors, etc.), user/context-awareness components and/or identification/verification sensors/devices (such as biometric sensors/detectors, scanners, etc.), memory or storage devices, databases, and/or data sources (such as data storage devices, hard drives, solid-state drives, hard disks, memory cards or devices, memory circuits, etc.), communication channels or networks (e.g., Cloud network, the Internet, intranet, cellular network, proximity networks, such as Bluetooth, Bluetooth low energy (BLE), Bluetooth Smart, Wi-Fi proximity, Radio Frequency Identification (RFID), Near Field Communication (NFC), Body Area Network (BAN), etc.), wireless or wired communications and relevant protocols (e.g., Wi-Fi®, WiMAX, Ethernet, etc.), connectivity and location management techniques, software applications/websites, (e.g., social and/or business networking websites, etc., business applications, games and other entertainment applications, etc.), programming languages, etc., while ensuring compatibility with changing technologies, parameters, protocols, standards, etc.

Throughout this document, terms like "logic", "component", "module", "framework", "engine", "mechanism", and the like, may be referenced interchangeably and include, by way of example, software, hardware, and/or any combination of software and hardware, such as firmware. In one example, "logic" may refer to or include a software component that is capable of working with one or more of an operating system (e.g., operating system 1306), a graphics driver (e.g., graphics driver 1316), etc., of a computing device, such as computing device 1300. In another example, "logic" may refer to or include a hardware component that is capable of being physically installed along with or as part of one or more system hardware elements, such as an application processor (e.g., CPU 1312), a graphics processor (e.g., GPU 1314), etc., of a computing device, such as computing device 1300. In yet another embodiment, "logic" may refer to or include a firmware component that is capable of being part of system firmware, such as firmware of an application processor (e.g., CPU 1312) or a graphics processor (e.g., GPU 1314), etc., of a computing device, such as computing device 1300.

Further, any use of a particular brand, word, term, phrase, name, and/or acronym, such as "GPU", "GPU domain", "GPGPU", "CPU", "CPU domain", "graphics driver", "workload", "application", "frame", "work unit", "draw", "dispatch", "API", "hardware", "software", "agent", "graphics driver", "kernel mode graphics driver", "user-mode driver", "UMD", "user-mode driver framework", "UMDF", "3D graphics API", "triple buffered constant buffer" or "TBCB", "virtual reality" or "VR", "map" or "mapping", "unmap" or "unmapping", "swap" or "swapping", "swap execution", "swap command", "memory block", "current data", "next data",

"$GPU_{current}$", "$CPU_{current}$", "CPU_{next}", "command buffer", "command list", "data structure", etc., should not be read to limit embodiments to software or devices that carry that label in products or in literature external to this document.

It is contemplated that any number and type of components may be added to and/or removed from buffer mechanism 1310 to facilitate various embodiments including adding, removing, and/or enhancing certain features. For brevity, clarity, and ease of understanding of buffer mechanism 1310, many of the standard and/or known components, such as those of a computing device, are not shown or discussed here. It is contemplated that embodiments, as described herein, are not limited to any particular technology, topology, system, architecture, and/or standard and are dynamic enough to adopt and adapt to any future changes.

Figures 15A, 15B and **15C** illustrate conventional techniques 1500, 1520, 1540. As illustrated, conventional technique 1500 reflects a crude synchronization model between CPU and GPU, including, in some cases, synchronization fences, and therefore, conventional techniques, such as 1500, 1520, 1540, fail to update head tracking data just before rendering begins. For example, an application either has to flush the GPU pipeline or estimate the amount of time required for rendering which delays operations on the CPU that is error-prone and change with the GPU model. For example, with respect to conventional technique 1500, over timeline 1501, an application queues commands 1503 for performing a couple of tasks A and B, but, as illustrated, processing task A 1505 may continue on for an unknown duration of time and it is during this unknown timespan that it is determined safe to modify data for processing task B 1507, such as delaying head tracking information for tasks B for as late as possible, which leads to errors, such as an error to modify data for task B.

Now, as illustrated with reference to conventional technique 1520, if the application overestimates the unknown duration of time 1501 of conventional technique 1500 of **Figure 15A**, such as either by estimated timing alone or waiting for the GPU to signal completion of processing of task A 1505, the process results in an unwanted GPU stall 1521, where GPU dequeues task A, dequeues fence, until it reaches unwanted stall 1521 and then finally, dequeues B, followed by the CPU updating data and signals fence.

In contrast, with reference to conventional technique 1540, if the application underestimates the unknown duration of time 1501, the process results in unwanted latency 1541 from the CPU updating data and signals fence and the GPU dequeuing the fences before leading to processing tasks B 1507. Further, since the application has merely one chance per frame to signal the fence, the misprediction can be both large and inconsistent between frames.

Figure 15D illustrates transaction sequence 1560 according to one embodiment.

Transaction sequence 1560 may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, etc.), software (such as instructions run on a processing device), or a combination thereof, as facilitated by buffer mechanism 1310 and/or swap logic 1320 of **Figure 13**. The processes of transaction sequence 1560 are illustrated in linear sequences for brevity and clarity in presentation; however, it is contemplated that any number of them can be performed in parallel, asynchronously, or in different orders. For brevity, many of the details discussed with reference to the preceding figures may not be discussed or repeated hereafter.

In embodiment, transaction sequence 1560 does not require any CPU-GPU synchronization and thus, a TBCB may be updated either periodically or opportunistically. For example over time 1561, an application may run a thread polling, for example, a head tracking position upon reaching a period of time, such as 1ms, updating the TBCP. In this case, the position may be maximum 1ms old when the relevant GPU performs buffer swap, just before rendering view-dependent draw calls. For example, with commands being queued 1563 and tasks A and B to be processed 1565, 1567, the period may be smartly adjusted such that to, for example, find the best balance between low-latency and low CPU overhead. For example, even in a worse case scenario, a full updated period of latency, such as unwanted latency 1571, may be significantly shorter as shown between one of CPU updating of data and swapping of buffer 1569 and GPU dequeuing the swap.

Figure 16 illustrates a transaction sequence 1610, 1620, 1630 according to one embodiment. Transaction sequence 1610, 1620, 1630 may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, etc.), software (such as instructions run on a processing device), or a combination thereof, as facilitated by buffer mechanism 1310 and/or swap execution logic 1320 of **Figure 13**. The processes of transaction sequence 1610, 1620, 1630 are illustrated in linear sequences for brevity and clarity in presentation; however, it is contemplated that any number of them can be performed in parallel, asynchronously, or in different orders. For brevity, many of the details discussed with reference to the preceding figures may not be discussed or repeated hereafter.

Transaction sequence 1610, 1620, 1630 begins with generation of a TBCB having the illustrated three memory blocks, such as memory block A 1601, memory block B 1603, and memory block C 1605, and at map stage 1610, initial configuration (map stage) 1610 of these memory blocks 1601, 1603, 1605 is performed such that two memory blocks 1603, 1605 are mapped to and visible from a CPU, while one memory block 1601 is mapped to and visible from a GPU as facilitated by mapping logic 1405 of **Figure 14**. In one embodiment, map stage 1610

sets up the mapping operation as follows: $GPU_{current}$ (data, pointer) 1611 is mapped to memory block A 1601, $CPU_{current}$ (data, pointer) 1613 is mapped to memory block B 1603, and CPU_{next} (data, pointer) 1615 is mapped to memory block C 1615.

In one embodiment, this mapping allows the application a location to route the data and commands and how much time the CPU may be willing or allowed to spend on certain processing assignments. For example, in one embodiment, memory block 1603 mapped to $CPU_{current}$ 1613 may be capable of caching certain data that is to be processed right away. Similarly, memory block 1605 mapped to CPU_{next} 1615 may be capable of caching future data that is to be processed next after the current data of memory block 1603 is processed or, in other words, the "next data" is expected to become "current data" in future. Finally, memory block 1601 mapped to $GPU_{current}$ 1611 is capable of caching data that is ready and assigned to the GPU to be processed by the GPU.

Transaction sequence 1610, 1620, 1630 further provides for a call for unmapping of memory blocks 1601-1605 at unmap stage 1620 as facilitated by unmapping logic 1407 of **Figure 14**, such as after the data has been updated in memory block C 1605. In one embodiment, an unmapping operation being performed at unmapping stage 1620 includes swapping of $CPU_{current}$ 1613 and CPU_{next} 1615 pointers such that the recently updated memory block C 1605 becomes "current", while the "current" memory block B 1603 becomes a target block for the "next" update.

Continuing with transaction sequence 1610, 1620, 1630, once the mapping operation is completed and a swap command is placed in a command stream at swap stage 1630 as facilitated by swapping logic 1407 of **Figure 14**, swap execution logic 1320 of **Figure 13** is then triggered to facilitate the GPU to process or execute the swap command as shown at swap stage 1630. In contrast with the conventional techniques, this swap command is executed on the GPU (as opposed to the CPU) upon inserting the swap command into the command stream. As illustrated, in one embodiment, this execution of the swap command facilitates $GPU_{current}$ 1611 to target $CPU_{current}$ 1613 data at memory block C 1605 as being pointed by $CPU_{current}$ 1613, causing memory block C 1605 to become visible to the GPU. This way, the GPU then accesses and executes the current data. In one embodiment, this swap technique may be implemented as a pointer swap or through actual copying of the data between memory blocks 1601-1605, as desired or necessitated based on available resources, such as hardware limitations of a computing device.

Figure 17 illustrates architectural placements 1700, 1750 according to one embodiment. For brevity, many of the details discussed with reference to the preceding figures may not be

discussed or repeated hereafter. Moreover, it is contemplated and to be noted that embodiments are not limited to any particular placement, setup, arrangement, structure, etc., such as (without limitation) architectural placements 1700, 1750.

In architectural placement 1700, an arrangement of four components, such as application 1701, graphics API/runtime 1703, graphics driver 1316, and GPU 1314, is shown. As illustrated, application 1701 may be used to render a code that uses TBCB, while the graphics API for mapping, unmapping, adding swap commands to command streams, etc., may be provided through graphics API/runtime 1703. It is contemplated that embodiments are not limited to any particular software, hardware, and/or firmware components and thus, the API may be exposed as part of graphics API 1703 or as a graphics vendor extension or a graphics API.

Similarly, as illustrated, map, unmap, and/or adding swap components, such as mapping logic 1405, unmapping logic 1407, swapping logic 1409 of **Figure 14**, may be provided through graphics driver 1316, as shown with respect to **Figure 14**, while a swap component for execution, such as swap execution logic 1320 of **Figure 14**, may be offered through or implemented at GPU 1314, as further illustrated with respect to **Figure 14**. For example, in some embodiments, map and upmap components may be implemented through graphics runtime 1703, in which case, driver 1316 may, instead, only need to expose an access mechanism/logic, such as accessing/interfacing logic 1411 of **Figure 14** for GPU current (read) and CPU current (read/write).

As illustrated, architecture placement 1750 provides another setup where extensions for API for map, unmap, add swap commands are provided through driver 1316 for implementation for map, unmap, and add swap commands. It is contemplated that regardless of placements 1700, 1750 or any other architectural setups, swapping logic 1409 of **Figure 14** may be used to add a swap command to a command stream which may then be responsible for facilitating encoding of the commands to which swap execution logic 1320 of **Figure 13** at GPU 1314 responds by executing the swap as defined by the swap command.

Figures 18A, **18B** and **18C** illustrate methods 1800, 1820, and 1840 facilitating mapping, unmapping, and swapping operations, respectively, according to one embodiment. Methods 1800, 1820, 1840 may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, etc.), software (such as instructions run on a processing device), or a combination thereof, as facilitated by buffer mechanism 1310 and/or swap execution logic 1320 of **Figure 13**. The processes of methods 1800, 1820, and 1840 are illustrated in linear sequences for brevity and clarity in presentation; however, it is contemplated that any number of them can be performed in parallel, asynchronously, or in different orders.

For brevity, many of the details discussed with reference to the preceding figures may not be discussed or repeated hereafter.

Referring first to **Figure 18A**, in one embodiment, method 1800 begins at block 1801 with generation of a TBCB (such as by an application running on or hosted or facilitated by a computing device) having multiple memory blocks, such as memory blocks A, B, and C, that are then mapped to a CPU and/or a GPU, where at least two of the memory blocks, such as memory blocks B and C, are mapped and visible to the CPU, while the remaining one memory block, such as memory block A, is mapped and visible to the GPU, as facilitated by mapping logic 1405 of Figure 114. At block 1803, in one embodiment, CPU_{next} is set to neither $CPU_{current}$ nor $GPU_{current}$ such that $GPU_{current}$ is mapped to a first memory block, such as memory block A, $CPU_{current}$ is mapped to a second memory block, such as memory block B, and CPU_{next} is set to a third memory block, such as memory block C. In one embodiment, at block 1805, the third memory block associated with CPU_{next} is returned to the application for writing. At block 1807, method 1800 ends.

Referring now to **Figure 18B**, in one embodiment, method 1820 begins at block 1821 with initiation of unmapping of the memory blocks as facilitated by unmapping logic 1407 of **Figure 14**, where, at block 1823, $CPU_{current}$ is set to CPU_{next} such that the "next" data at the third memory block, such as memory block C, initially mapped to CPU_{next}, may be accessed and regarded as the "current" data by $CPU_{current}$ so that it may then be processed by the GPU at a later stage. In contrast, CPU_{next} may be set to $CPU_{current}$. At block 1825, method 1820 ends.

Figure 18C illustrates method 1840 relating to a swapping operation as initiated by swapping logic 1409 and executed by swap execution logic 1320 of **Figure 14**. For example, at block 1841, swap is initiated by having inserted a swap command into a command stream as facilitated by swapping logic 1409 of **Figure 14**. Once initiated, at block 1843, swap execution logic 1320 hosted by the GPU 1314 of **Figure 14** may executed the swap, such as by setting $GPU_{current}$ to $CPU_{current}$ by copying the data or performing a pointer switch, such as $GPU_{current}$ now pointing to the third memory block, such as memory block C, now having the "current" data, $CPU_{current}$ now pointing to the third memory block, such as memory block C, to provide or populate the "current" data, and CPU_{next} now pointing to the second memory block, such as memory block B, to gather the "next" data.

Figure 19 illustrates method 1900 facilitating control and data flow for mapping, unmapping, and swapping operations according to one embodiment. Method 1900 may be performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, programmable logic, etc.), software (such as instructions run on a processing device), or a

combination thereof, as facilitated by buffer mechanism 1310 and/or swap execution logic 1320 of **Figure 13**. The processes of method 1900 are illustrated in linear sequences for brevity and clarity in presentation; however, it is contemplated that any number of them can be performed in parallel, asynchronously, or in different orders. For brevity, many of the details discussed with reference to the preceding figures may not be discussed or repeated hereafter.

Method 1900 begins at block 1901 with initiation of a CPU thread, such as CPU thread A, and, at block 1903, adding render commands not needed latest data which, at block 1907, is shown in communication with adding render commands needing latest data, leading to an end at block 1909. At block 1905, in one embodiment, a swap command is added to a command queue, such as command queue 1911, which is delivered to and ready by a read command, such as read command 1913, as facilitated by GPU 1314.

At block 1915, in one embodiment, a decision is made as to whether a command swap is to be performed. If not, the rendered command is processed at block 1917. If yes, however, a swap is executed at block 1919, such as by setting $GPU_{current}$ to $CPU_{current}$ at block 1921, wherein block 1923 is shown as reading data from block 1921. As previously discussed, new data may be obtained at block 1923 through another CPU thread, such as CPU thread B, at block 1925, to contribute to and continue with one or more mapping operations at block 1923. Further, as illustrated, map at block 1923 writes to CPU_{next} at block 1925 and then, proceeds to block 1931 which writes new data to memory, such as memory 1933, and then proceeds to unmap at block 1927. As illustrated, unmap at 1927 reads from CPU_{next} at block 1925 and writes to $CPU_{current}$ at block 1921, before proceeding to block 1929 (wait until next update due).

In one embodiment, once the unmapping operation is performed, method 1900 may simply wait for the next update to occur or come due at block 1929. In another embodiment, upon unmapping, method 1900 continues with initiating the swap operation at block 1921 by setting $GPU_{current}$ to $CPU_{current}$. As illustrated, unmap at block 1927 writes to data at block 1921 and proceeds to block 1929 (wait until next update due). Referring back to the mapping operation at block 1923, method 1900 may continue at block 1931 with writing of new data to memory, such as memory 1933, and subsequently, processing of the render command at block 1917.

References to "one embodiment", "an embodiment", "example embodiment", "various embodiments", etc., indicate that the embodiment(s) so described may include particular features, structures, or characteristics, but not every embodiment necessarily includes the particular features, structures, or characteristics. Further, some embodiments may have some, all, or none of the features described for other embodiments.

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

In the following description and claims, the term "coupled" along with its derivatives, may be used. "Coupled" is used to indicate that two or more elements co-operate or interact with each other, but they may or may not have intervening physical or electrical components between them.

As used in the claims, unless otherwise specified the use of the ordinal adjectives "first", "second", "third", etc., to describe a common element, merely indicate that different instances of like elements are being referred to, and are not intended to imply that the elements so described must be in a given sequence, either temporally, spatially, in ranking, or in any other manner.

The following clauses and/or examples pertain to further embodiments or examples. Specifics in the examples may be used anywhere in one or more embodiments. The various features of the different embodiments or examples may be variously combined with some features included and others excluded to suit a variety of different applications. Examples may include subject matter such as a method, means for performing acts of the method, at least one machine-readable medium including instructions that, when performed by a machine cause the machine to performs acts of the method, or of an apparatus or system for facilitating hybrid communication according to embodiments and examples described herein.

Some embodiments pertain to Example 1 that includes an apparatus to facilitate efficient processing of graphics data using triple buffered constant buffers at computing devices, comprising: detection/reception logic to detect generation of a multi-block buffer by an application to perform data processing at a graphics processor of the apparatus; mapping logic to map a first memory block of the multi-block buffer to the graphics processor, wherein the mapping logic is further to map a second memory block and a third memory block of the multi-block buffer to an application processor; and swap execution logic to execute a swap operation to facilitate the graphics processor to process a current data set associated with the application processor.

Example 2 includes the subject matter of Example 1, further comprising: swapping logic to initiate the swap operation by inserting a swap command in a command stream capable of being executed by the graphics processor; and unmapping logic to unmap the second memory block and the third memory block to facilitate the initiation of the swap operation, wherein

unmapping includes exchanging a next data set of the third memory block with the current data set of the second memory block.

Example 3 includes the subject matter of Example 1 or 2, wherein first, second, and third memory blocks of the multi-block buffer to store data including the current data set, wherein a first rate at which the data is generated is independent of a second rate at which the data is consumed.

Example 4 includes the subject matter of Example 1, wherein the command stream includes a plurality of graphics commands capable of consuming the data, wherein the multi-block buffer to facilitate generation of the plurality of graphics commands asynchronous from the generation of the data capable of being consumed by the plurality of graphics commands.

Example 5 includes the subject matter of Example 1, wherein the graphics processor to facilitate consumption of the data using a memory block of the multi-block buffer, and wherein the application processor to facilitate writing of the data to one or more memory blocks of the multi-block buffer, wherein the application processor is further to facilitate overwriting of one or more portions of the data in a least recently updated memory block of the one or more memory blocks.

Example 6 includes the subject matter of Example 1 or 2, wherein the swapping operation comprises exchanging a most recently updated memory block of the second and third memory blocks associated with the application processor with the first memory block mapped to be consumed by the graphics processor by switching pointers associated with the most recently updated memory block and the first memory block.

Example 7 includes the subject matter of Example 6, wherein exchanging is further performed by changing a virtual-to-physical memory mapping of the most recently updated memory block and the first memory block.

Example 8 includes the subject matter of Example 6, wherein exchanging comprises copying the most recently updated memory block with the first memory block.

Some embodiments pertain to Example 9 that includes a method for facilitating efficient processing of graphics data using triple buffered constant buffers at computing devices, comprising: detecting generation of a multi-block buffer by an application to perform data processing at a graphics processor of a computing device; mapping a first memory block of the multi-block buffer to the graphics processor, wherein mapping further includes mapping a second memory block and a third memory block of the multi-block buffer to an application processor; and executing a swap operation to facilitate the graphics processor to process a current data set associated with the application processor.

Example 10 includes the subject matter of Example 9, further comprising: initiating the swap operation by inserting a swap command in a command stream capable of being executed by the graphics processor; and unmapping the second memory block and the third memory block to facilitate the initiation of the swap operation, wherein unmapping includes exchanging a next data set of the third memory block with the current data set of the second memory block.

Example 11 includes the subject matter of Example 9 or 10, wherein first, second, and third memory blocks of the multi-block buffer to store data including the current data set, wherein a first rate at which the data is generated is independent of a second rate at which the data is consumed.

Example 12 includes the subject matter of Example 9, wherein the command stream includes a plurality of graphics commands capable of consuming the data, wherein the multi-block buffer to facilitate generation of the plurality of graphics commands asynchronous from the generation of the data capable of being consumed by the plurality of graphics commands.

Example 13 includes the subject matter of Example 9, wherein the graphics processor to facilitate consumption of the data using a memory block of the multi-block buffer, and wherein the application processor to facilitate writing of the data to one or more memory blocks of the multi-block buffer, wherein the application processor is further to facilitate overwriting of one or more portions of the data in a least recently updated memory block of the one or more memory blocks.

Example 14 includes the subject matter of Example 9 or 10, wherein the swapping operation comprises exchanging a most recently updated memory block of the second and third memory blocks associated with the application processor with the first memory block mapped to be consumed by the graphics processor by switching pointers associated with the most recently updated memory block and the first memory block.

Example 15 includes the subject matter of Example 14, wherein exchanging is further performed by changing a virtual-to-physical memory mapping of the most recently updated memory block and the first memory block.

Example 16 includes the subject matter of Example 14, wherein exchanging comprises copying the most recently updated memory block with the first memory block.

Some embodiments pertain to Example 17 includes a system comprising a storage device having instructions, and a processor to execute the instructions to facilitate a mechanism to perform one or more operations comprising: detecting generation of a multi-block buffer by an application to perform data processing at a graphics processor of a computing device; mapping a first memory block of the multi-block buffer to the graphics processor, wherein mapping further

includes mapping a second memory block and a third memory block of the multi-block buffer to an application processor; and executing a swap operation to facilitate the graphics processor to process a current data set associated with the application processor.

Example 18 includes the subject matter of Example 17, wherein the one or more operations further comprise: initiating the swap operation by inserting a swap command in a command stream capable of being executed by the graphics processor; and unmapping the second memory block and the third memory block to facilitate the initiation of the swap operation, wherein unmapping includes exchanging a next data set of the third memory block with the current data set of the second memory block.

Example 19 includes the subject matter of Example 17 or 18, wherein first, second, and third memory blocks of the multi-block buffer to store data including the current data set, wherein a first rate at which the data is generated is independent of a second rate at which the data is consumed.

Example 20 includes the subject matter of Example 17, wherein the command stream includes a plurality of graphics commands capable of consuming the data, wherein the multi-block buffer to facilitate generation of the plurality of graphics commands asynchronous from the generation of the data capable of being consumed by the plurality of graphics commands.

Example 21 includes the subject matter of Example 17, wherein the graphics processor to facilitate consumption of the data using a memory block of the multi-block buffer, and wherein the application processor to facilitate writing of the data to one or more memory blocks of the multi-block buffer, wherein the application processor is further to facilitate overwriting of one or more portions of the data in a least recently updated memory block of the one or more memory blocks.

Example 22 includes the subject matter of Example 17 or 18, wherein the swapping operation comprises exchanging a most recently updated memory block of the second and third memory blocks associated with the application processor with the first memory block mapped to be consumed by the graphics processor by switching pointers associated with the most recently updated memory block and the first memory block.

Example 23 includes the subject matter of Example 22, wherein exchanging is further performed by changing a virtual-to-physical memory mapping of the most recently updated memory block and the first memory block.

Example 24 includes the subject matter of Example 22, wherein exchanging comprises copying the most recently updated memory block with the first memory block.

Some embodiments pertain to Example 25 includes an apparatus comprising: means for detecting generation of a multi-block buffer by an application to perform data processing at a graphics processor of a computing device; means for mapping a first memory block of the multi-block buffer to the graphics processor, wherein mapping further includes mapping a second memory block and a third memory block of the multi-block buffer to an application processor; and means for executing a swap operation to facilitate the graphics processor to process a current data set associated with the application processor.

Example 26 includes the subject matter of Example 25, further comprising: means for initiating the swap operation by inserting a swap command in a command stream capable of being executed by the graphics processor; and means for unmapping the second memory block and the third memory block to facilitate the initiation of the swap operation, wherein unmapping includes exchanging a next data set of the third memory block with the current data set of the second memory block.

Example 27 includes the subject matter of Example 25 or 26, wherein first, second, and third memory blocks of the multi-block buffer to store data including the current data set, wherein a first rate at which the data is generated is independent of a second rate at which the data is consumed.

Example 28 includes the subject matter of Example 25, wherein the command stream includes a plurality of graphics commands capable of consuming the data, wherein the multi-block buffer to facilitate generation of the plurality of graphics commands asynchronous from the generation of the data capable of being consumed by the plurality of graphics commands.

Example 29 includes the subject matter of Example 25, wherein the graphics processor to facilitate consumption of the data using a memory block of the multi-block buffer, and wherein the application processor to facilitate writing of the data to one or more memory blocks of the multi-block buffer, wherein the application processor is further to facilitate overwriting of one or more portions of the data in a least recently updated memory block of the one or more memory blocks.

Example 30 includes the subject matter of Example 25 or 26, wherein the swapping operation comprises exchanging a most recently updated memory block of the second and third memory blocks associated with the application processor with the first memory block mapped to be consumed by the graphics processor by switching pointers associated with the most recently updated memory block and the first memory block.

Example 31 includes the subject matter of Example 30, wherein exchanging is further performed by changing a virtual-to-physical memory mapping of the most recently updated memory block and the first memory block.

Example 32 includes the subject matter of Example 30, wherein exchanging comprises copying the most recently updated memory block with the first memory block.

Example 33 includes at least one non-transitory or tangible machine-readable medium comprising a plurality of instructions, when executed on a computing device, to implement or perform a method as claimed in any of claims or examples 9-16.

Example 34 includes at least one machine-readable medium comprising a plurality of instructions, when executed on a computing device, to implement or perform a method as claimed in any of claims or examples 9-16.

Example 35 includes a system comprising a mechanism to implement or perform a method as claimed in any of claims or examples 9-16.

Example 36 includes an apparatus comprising means for performing a method as claimed in any of claims or examples 9-16.

Example 37 includes a computing device arranged to implement or perform a method as claimed in any of claims or examples 9-16.

Example 38 includes a communications device arranged to implement or perform a method as claimed in any of claims or examples 9-16.

Example 39 includes at least one machine-readable medium comprising a plurality of instructions, when executed on a computing device, to implement or perform a method or realize an apparatus as claimed in any preceding claims.

Example 40 includes at least one non-transitory or tangible machine-readable medium comprising a plurality of instructions, when executed on a computing device, to implement or perform a method or realize an apparatus as claimed in any preceding claims.

Example 41 includes a system comprising a mechanism to implement or perform a method or realize an apparatus as claimed in any preceding claims.

Example 42 includes an apparatus comprising means to perform a method as claimed in any preceding claims.

Example 43 includes a computing device arranged to implement or perform a method or realize an apparatus as claimed in any preceding claims.

Example 44 includes a communications device arranged to implement or perform a method or realize an apparatus as claimed in any preceding claims.

The drawings and the forgoing description give examples of embodiments. Those skilled

in the art will appreciate that one or more of the described elements may well be combined into a single functional element. Alternatively, certain elements may be split into multiple functional elements. Elements from one embodiment may be added to another embodiment. For example, orders of processes described herein may be changed and are not limited to the manner described herein. Moreover, the actions of any flow diagram need not be implemented in the order shown; nor do all of the acts necessarily need to be performed. Also, those acts that are not dependent on other acts may be performed in parallel with the other acts. The scope of embodiments is by no means limited by these specific examples. Numerous variations, whether explicitly given in the specification or not, such as differences in structure, dimension, and use of material, are possible. The scope of embodiments is at least as broad as given by the following claims.

CLAIMS

What is claimed is:

1. An apparatus to facilitating efficient processing of graphics data using triple buffered constant buffers at computing devices, comprising:

detection/reception logic to detect generation of a multi-block buffer by an application to perform data processing at a graphics processor of the apparatus;

mapping logic to map a first memory block of the multi-block buffer to the graphics processor, wherein the mapping logic is further to map a second memory block and a third memory block of the multi-block buffer to an application processor; and

swap execution logic to execute a swap operation to facilitate the graphics processor to process a current data set associated with the application processor.

2. The apparatus of claim 1, further comprising:

swapping logic to initiate the swap operation by inserting a swap command in a command stream capable of being executed by the graphics processor; and

unmapping logic to unmap the second memory block and the third memory block to facilitate the initiation of the swap operation, wherein unmapping includes exchanging a next data set of the third memory block with the current data set of the second memory block.

3. The apparatus of claim 1 or 2, wherein first, second, and third memory blocks of the multi-block buffer to store data including the current data set, wherein a first rate at which the data is generated is independent of a second rate at which the data is consumed.

4. The apparatus of claim 1, wherein the command stream includes a plurality of graphics commands capable of consuming the data, wherein the multi-block buffer to facilitate generation of the plurality of graphics commands asynchronous from the generation of the data capable of being consumed by the plurality of graphics commands.

5. The apparatus of claim 1, wherein the graphics processor to facilitate consumption of the data using a memory block of the multi-block buffer, and wherein the application processor to facilitate writing of the data to one or more memory blocks of the multi-block buffer, wherein the application processor is further to facilitate overwriting of one or more portions of the data in a least recently updated memory block of the one or more memory blocks.

6. The apparatus of claim 1 or 2, wherein the swapping operation comprises exchanging a most recently updated memory block of the second and third memory blocks associated with the application processor with the first memory block mapped to be consumed by the graphics processor by switching pointers associated with the most recently updated memory block and the first memory block.

7. The apparatus of claim 6, wherein exchanging is further performed by changing a virtual-to-physical memory mapping of the most recently updated memory block and the first memory block.

8. The apparatus of claim 6, wherein exchanging comprises copying the most recently updated memory block with the first memory block.

9. A method for facilitating efficient processing of graphics data using triple buffered constant buffers at computing devices, comprising:

detecting generation of a multi-block buffer by an application to perform data processing at a graphics processor of a computing device;

mapping a first memory block of the multi-block buffer to the graphics processor, wherein mapping further includes mapping a second memory block and a third memory block of the multi-block buffer to an application processor; and

executing a swap operation to facilitate the graphics processor to process a current data set associated with the application processor.

10. The method of claim 9, further comprising:

initiating the swap operation by inserting a swap command in a command stream capable of being executed by the graphics processor; and

unmapping the second memory block and the third memory block to facilitate the initiation of the swap operation, wherein unmapping includes exchanging a next data set of the third memory block with the current data set of the second memory block.

11. The method of claim 9, wherein first, second, and third memory blocks of the multi-block buffer to store data including the current data set, wherein a first rate at which the data is generated is independent of a second rate at which the data is consumed.

12. The method of claim 9, wherein the command stream includes a plurality of graphics commands capable of consuming the data, wherein the multi-block buffer to facilitate generation of the plurality of graphics commands asynchronous from the generation of the data capable of being consumed by the plurality of graphics commands.

13. The method of claim 9, wherein the graphics processor to facilitate consumption of the data using a memory block of the multi-block buffer, and wherein the application processor to facilitate writing of the data to one or more memory blocks of the multi-block buffer, wherein the application processor is further to facilitate overwriting of one or more portions of the data in a least recently updated memory block of the one or more memory blocks.

14. The method of claim 9, wherein the swapping operation comprises exchanging a most recently updated memory block of the second and third memory blocks associated with the application processor with the first memory block mapped to be consumed by the graphics processor by switching pointers associated with the most recently updated memory block and the first memory block.

15. The method of claim 13, wherein exchanging is further performed by changing a virtual-to-physical memory mapping of the most recently updated memory block and the first memory block.

16. The method of claim 13, wherein exchanging comprises copying the most recently updated memory block with the first memory block.

17. At least one machine-readable medium comprising a plurality of instructions, when executed on a computing device, to implement or perform a method as claimed in any of claims 9-16.

18. A system comprising a mechanism to implement or perform a method as claimed in any of claims 9-16.

19. An apparatus comprising means for performing a method as claimed in any of claims 9-16.

20. A computing device arranged to implement or perform a method as claimed in any of claims 9-16.

21. A communications device arranged to implement or perform a method as claimed in any of claims 9-16.

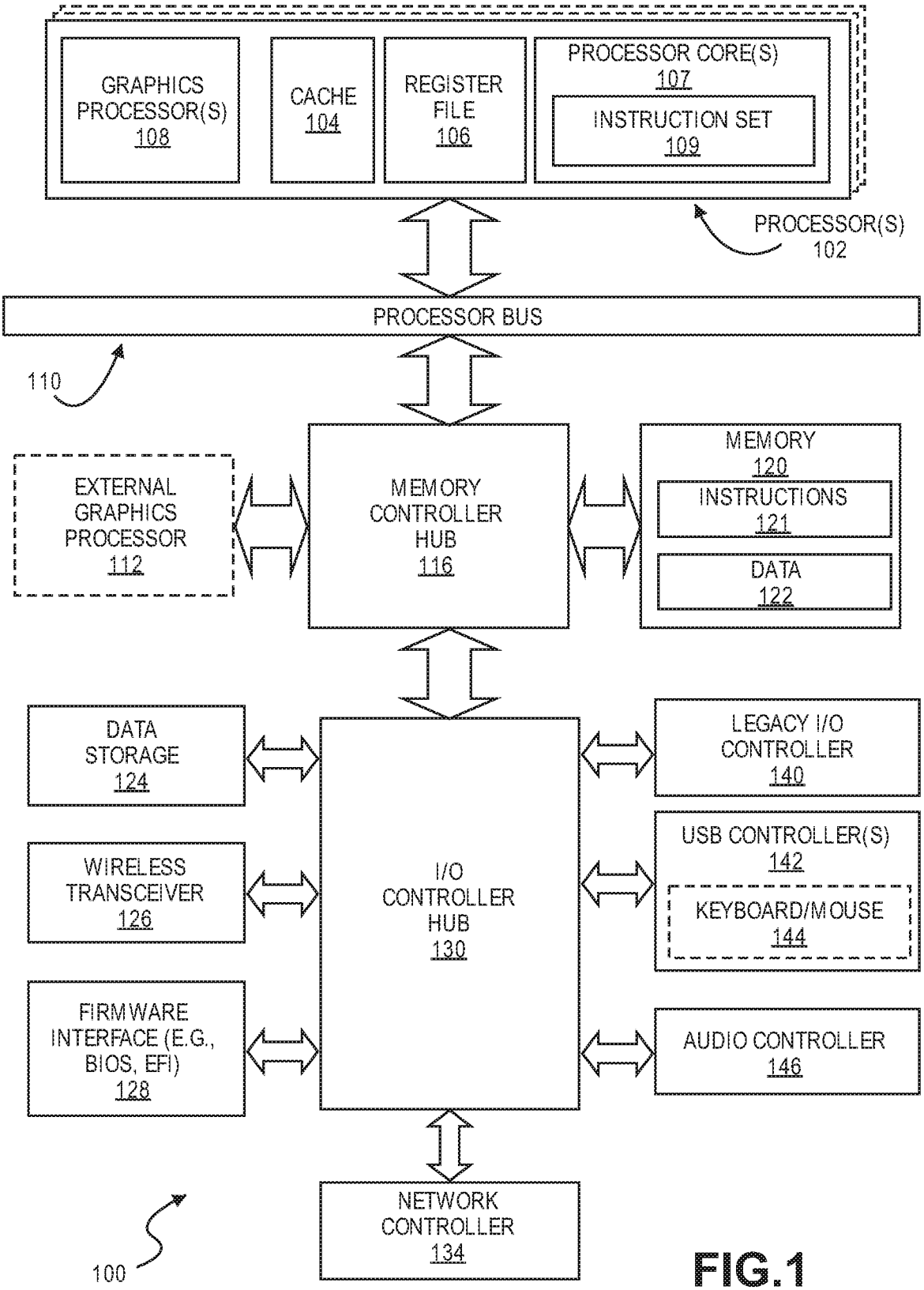

GRAPHICS PROCESSOR(S) 108
CACHE 104
REGISTER FILE 106
PROCESSOR CORE(S) 107
INSTRUCTION SET 109
PROCESSOR(S) 102
PROCESSOR BUS
110
EXTERNAL GRAPHICS PROCESSOR 112
MEMORY CONTROLLER HUB 116
MEMORY 120
INSTRUCTIONS 121
DATA 122
DATA STORAGE 124
WIRELESS TRANSCEIVER 126
FIRMWARE INTERFACE (E.G., BIOS, EFI)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

FIG.1

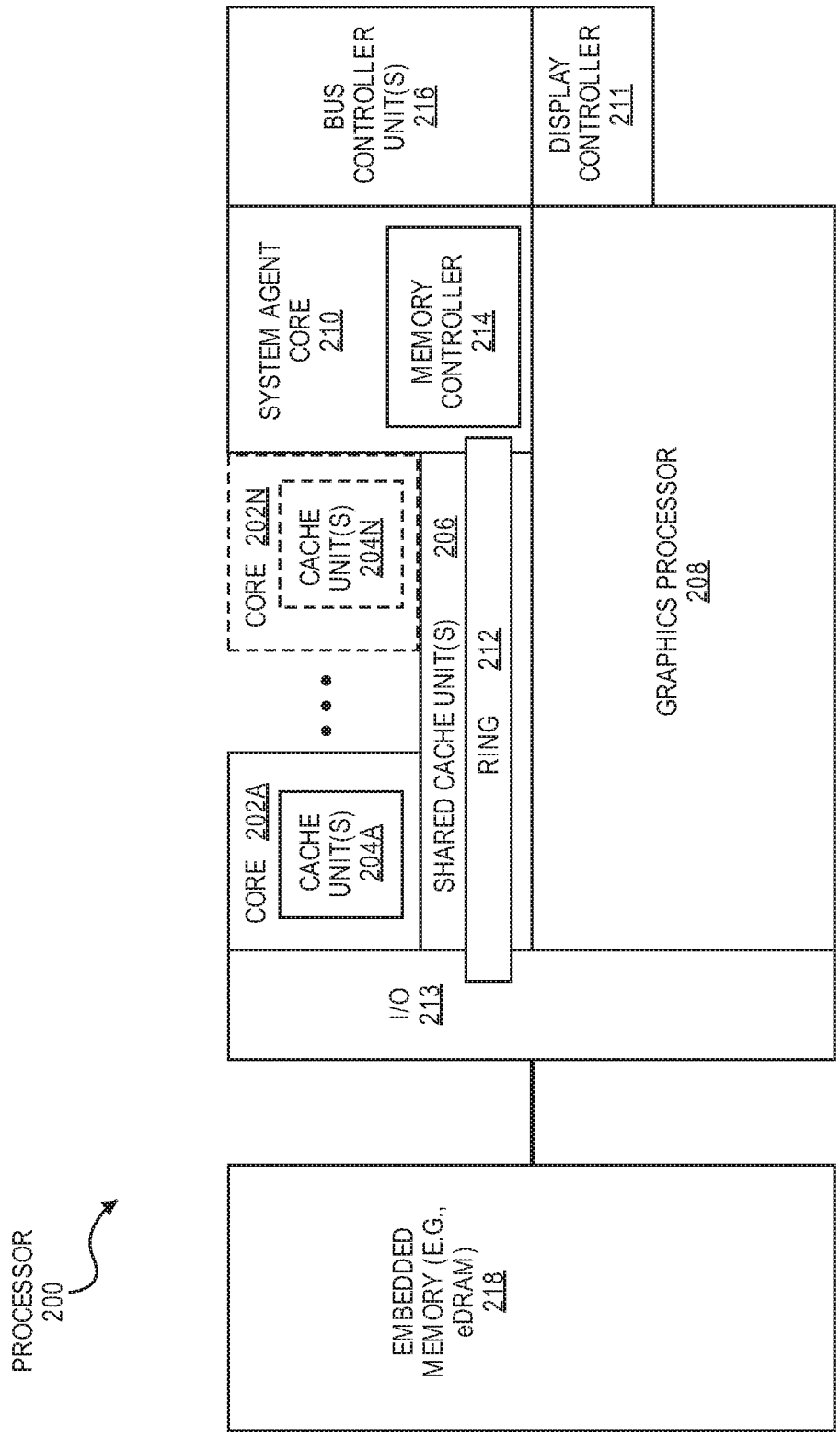

PROCESSOR 200
EMBEDDED MEMORY (E.G., eDRAM) 218
I/O 213
CORE 202A
CACHE UNIT(S) 204A
CORE 202N
CACHE UNIT(S) 204N
SHARED CACHE UNIT(S) 206
RING 212
GRAPHICS PROCESSOR 208
SYSTEM AGENT CORE 210
MEMORY CONTROLLER 214
BUS CONTROLLER UNIT(S) 216
DISPLAY CONTROLLER 211

FIG. 2

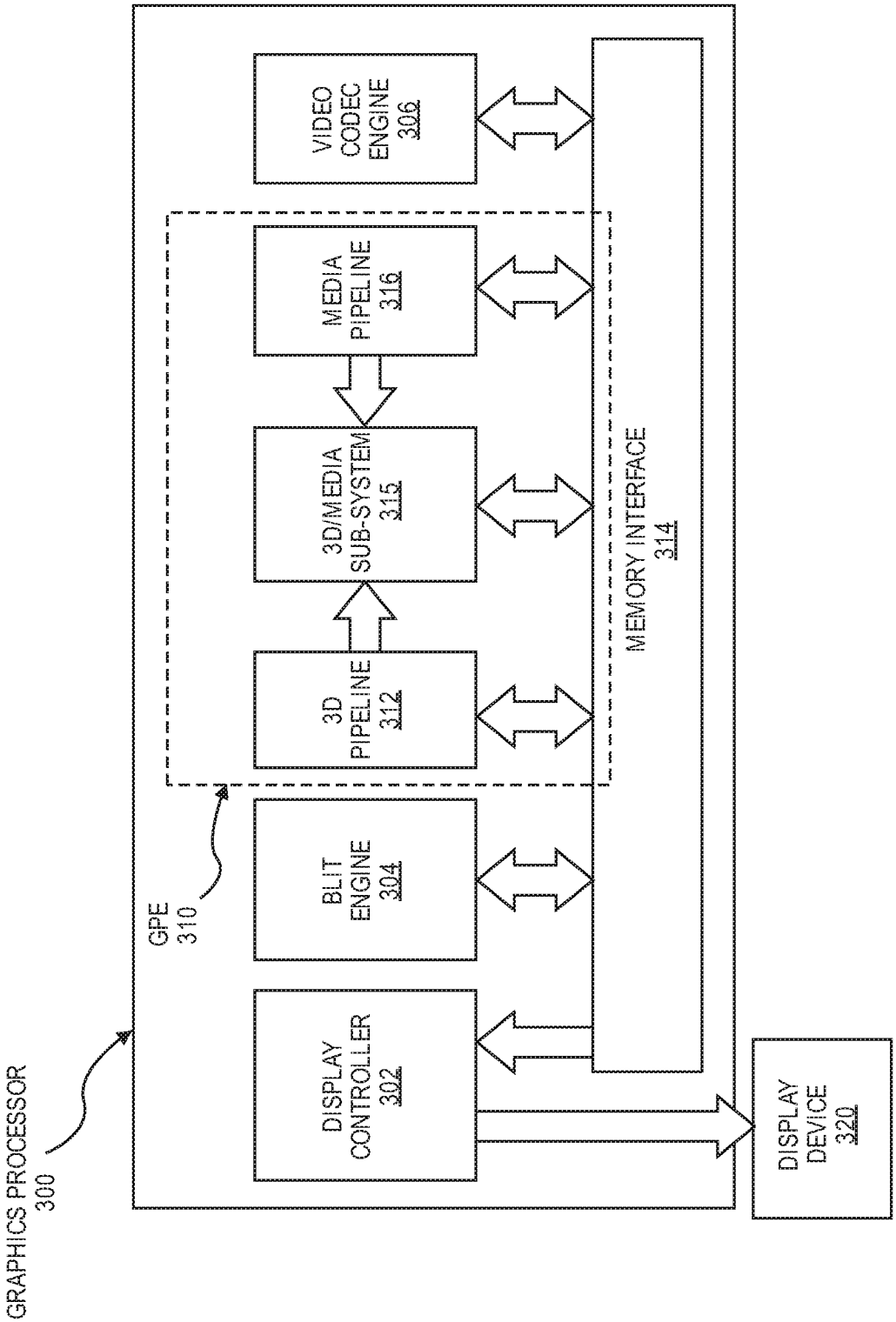

GRAPHICS PROCESSOR 300
GPE 310
DISPLAY CONTROLLER 302
BLIT ENGINE 304
3D PIPELINE 312
3D/MEDIA SUB-SYSTEM 315
MEDIA PIPELINE 316
VIDEO CODEC ENGINE 306
MEMORY INTERFACE 314
DISPLAY DEVICE 320

FIG. 3

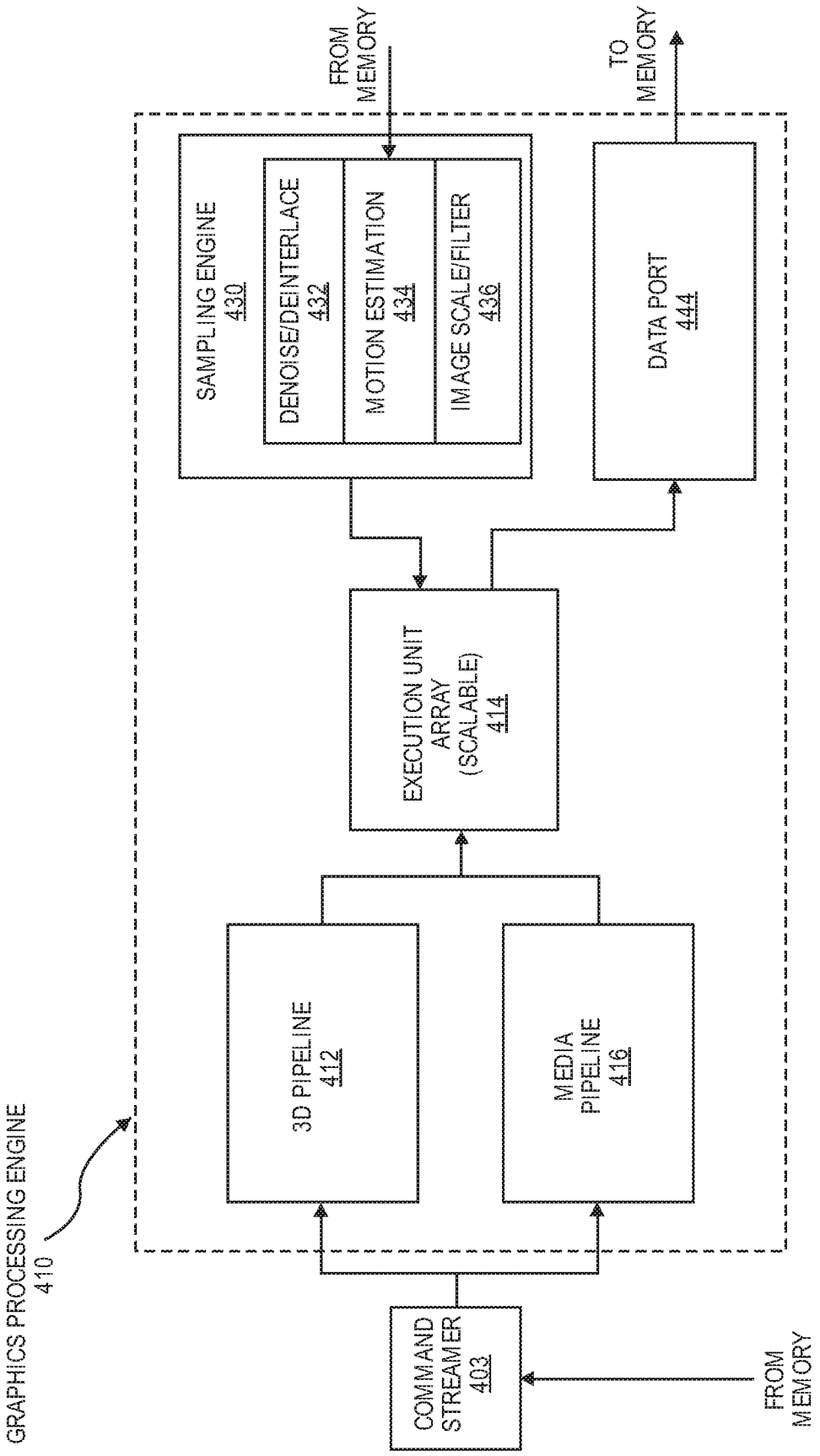

GRAPHICS PROCESSING ENGINE
410
COMMAND STREAMER
403
FROM MEMORY
3D PIPELINE
412
MEDIA PIPELINE
416
EXECUTION UNIT ARRAY (SCALABLE)
414
SAMPLING ENGINE
430
DENOISE/DEINTERLACE
432
MOTION ESTIMATION
434
IMAGE SCALE/FILTER
436
DATA PORT
444
FROM MEMORY
TO MEMORY

FIG. 4

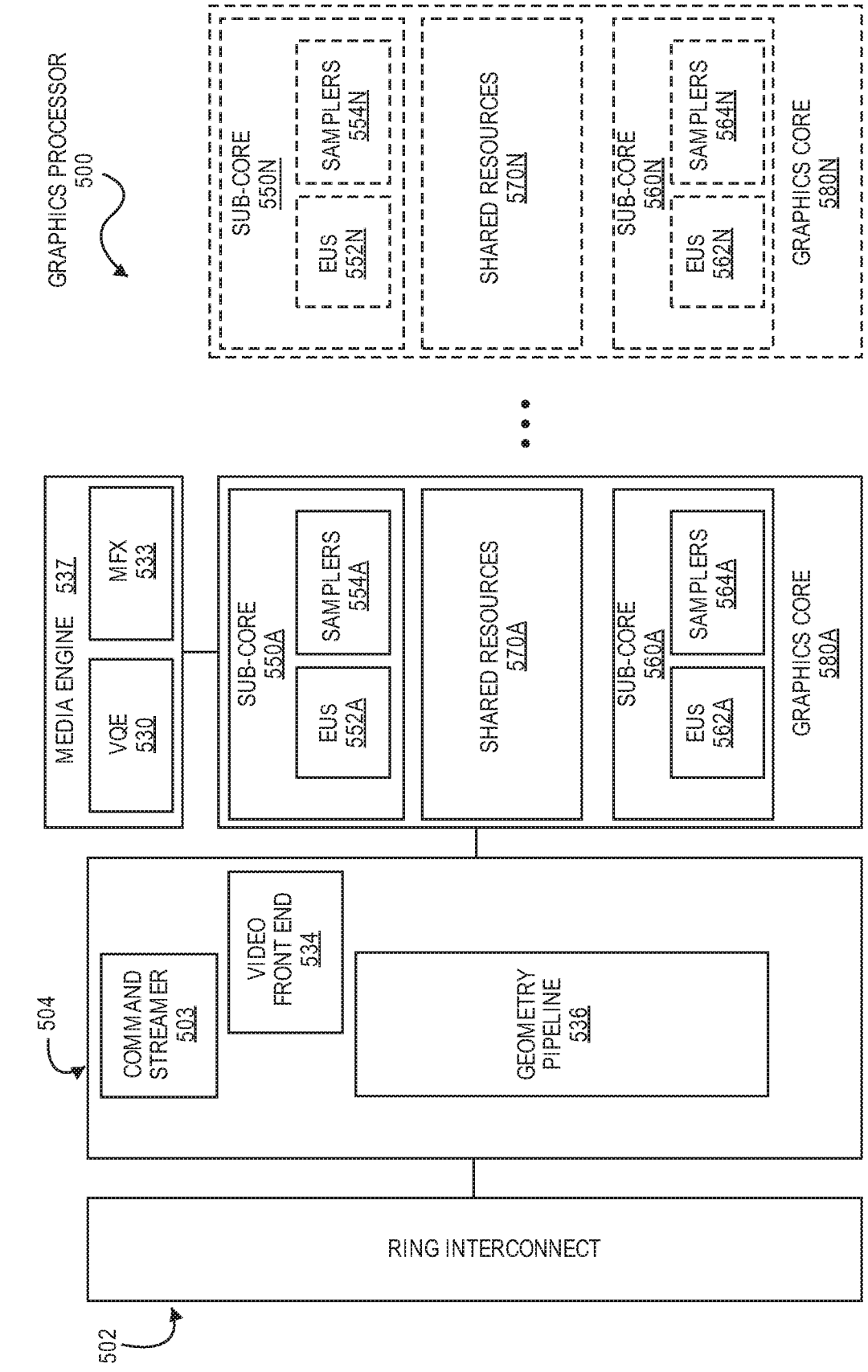

GRAPHICS PROCESSOR
500
MEDIA ENGINE 537
VQE 530
MFX 533
SUB-CORE 550A
EUS 552A
SAMPLERS 554A
SHARED RESOURCES 570A
SUB-CORE 560A
EUS 562A
SAMPLERS 564A
GRAPHICS CORE 580A
SUB-CORE 550N
EUS 552N
SAMPLERS 554N
SHARED RESOURCES 570N
SUB-CORE 560N
EUS 562N
SAMPLERS 564N
GRAPHICS CORE 580N
504
COMMAND STREAMER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

FIG. 5

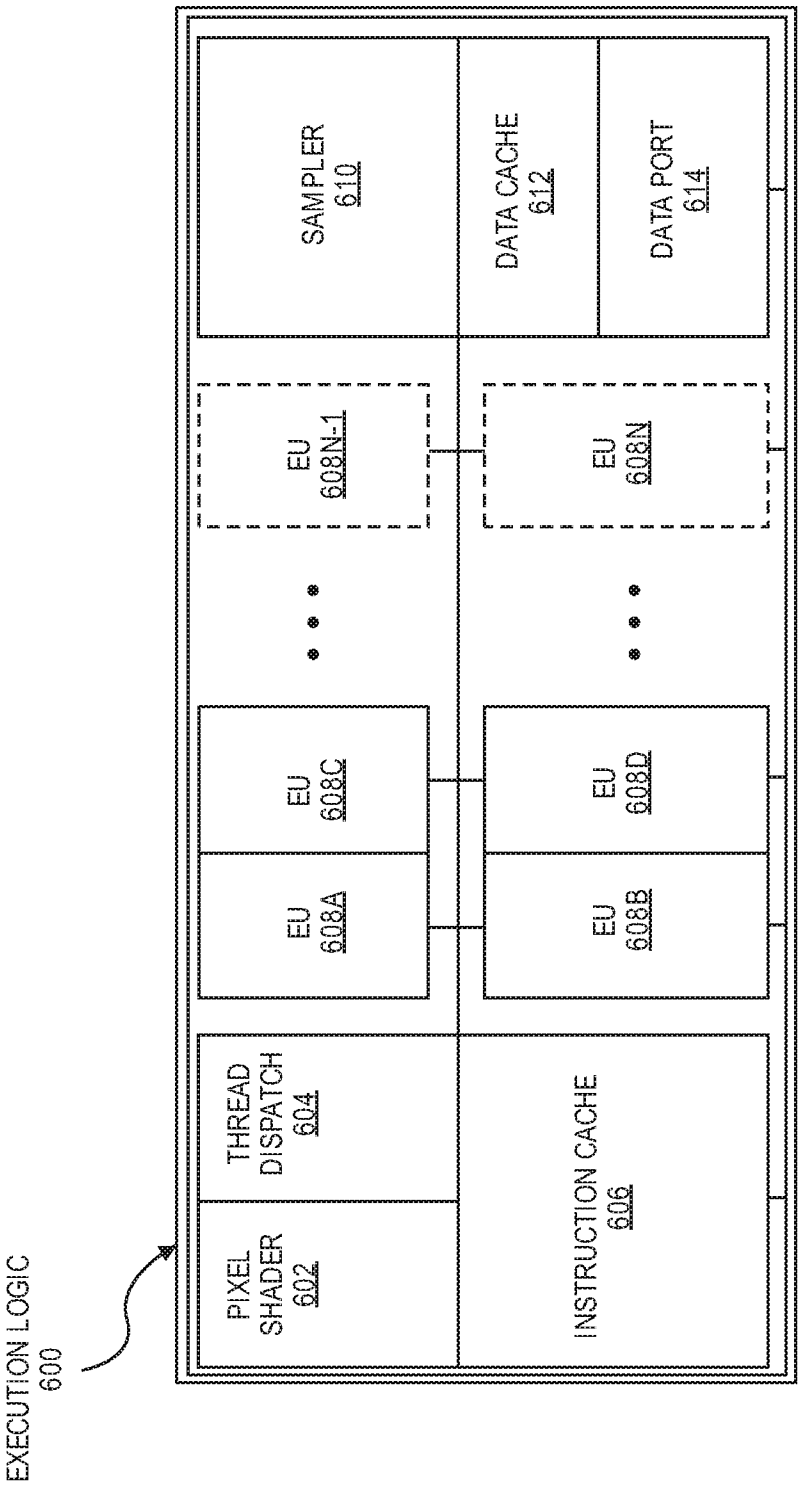

EXECUTION LOGIC
600
PIXEL SHADER
602
THREAD DISPATCH
604
INSTRUCTION CACHE
606
EU
608A
EU
608C
EU
608N-1
EU
608B
EU
608D
EU
608N
SAMPLER
610
DATA CACHE
612
DATA PORT
614

FIG. 6

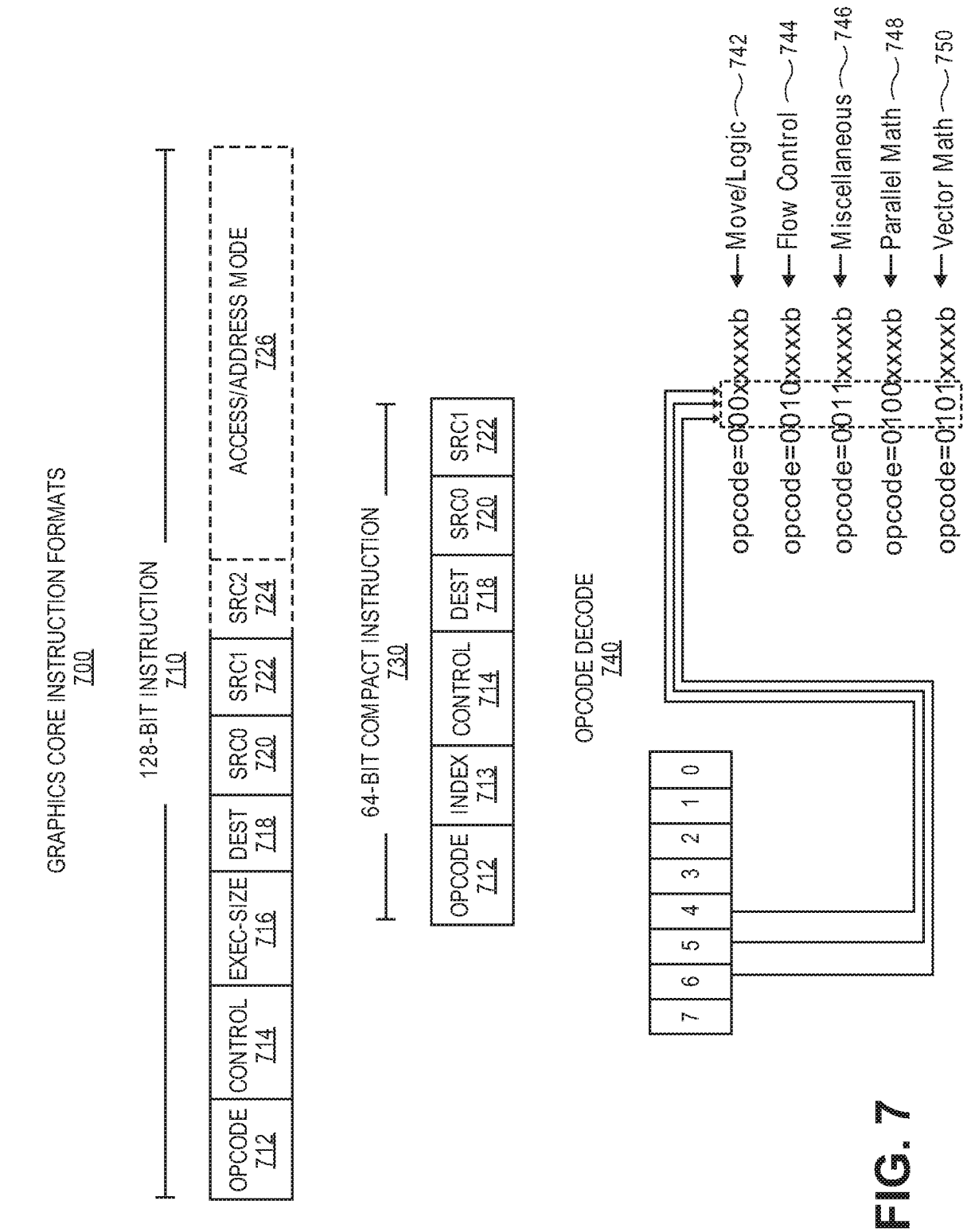

GRAPHICS CORE INSTRUCTION FORMATS
Z00
128-BIT INSTRUCTION
Z10
OPCODE Z12
CONTROL Z14
EXEC-SIZE Z16
DEST Z18
SRC0 Z20
SRC1 Z22
SRC2 Z24
ACCESS/ADDRESS MODE Z26
64-BIT COMPACT INSTRUCTION
Z30
OPCODE Z12
INDEX Z13
CONTROL Z14
DEST Z18
SRC0 Z20
SRC1 Z22
OPCODE DECODE
Z40
7 6 5 4 3 2 1 0
opcode=000xxxxb
opcode=001xxxxb
opcode=010xxxxb
opcode=011xxxxb
opcode=100xxxxb
opcode=101xxxxb
Move/Logic 742
Flow Control 744
Miscellaneous 746
Parallel Math 748
Vector Math 750

FIG. 7

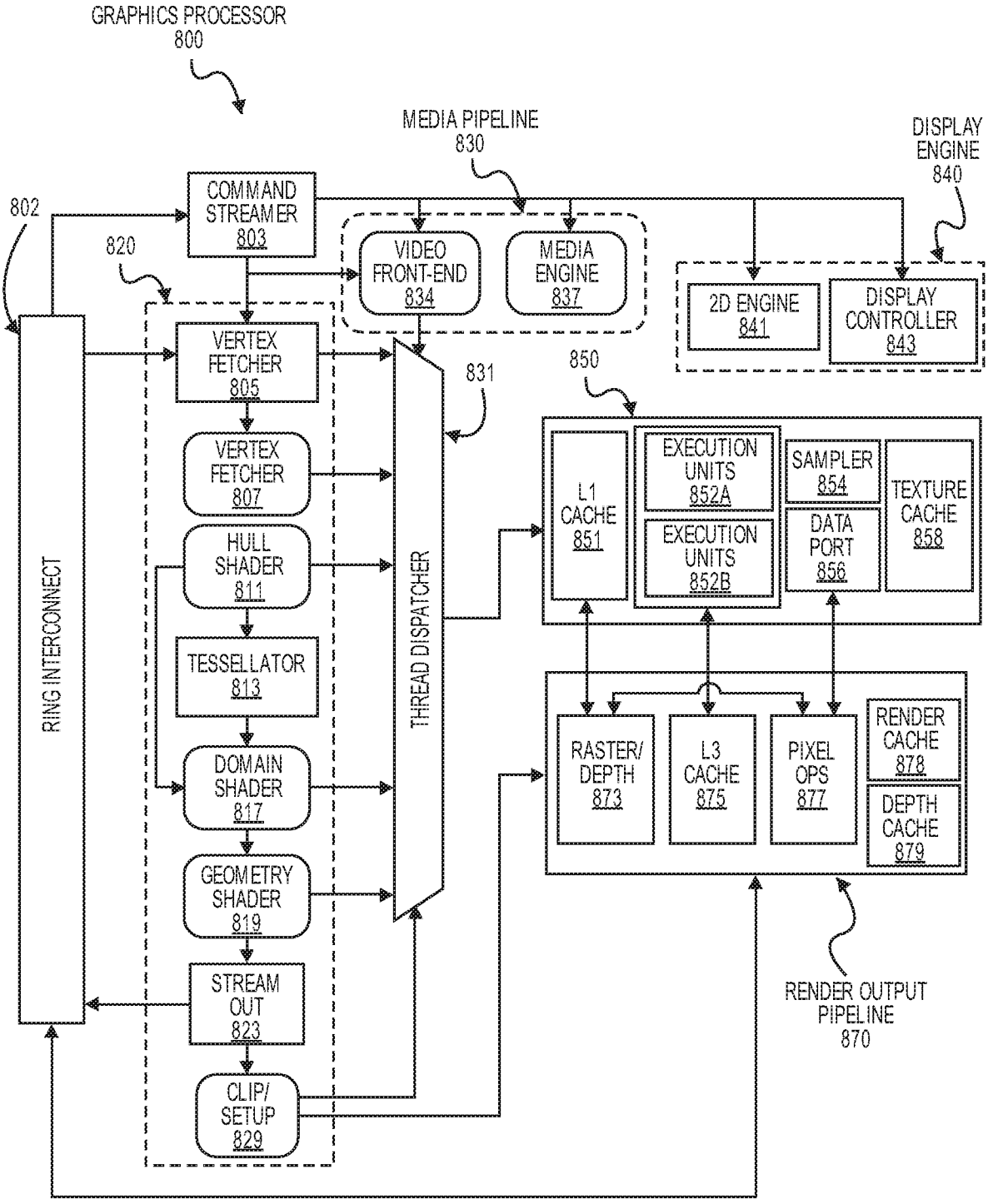

GRAPHICS PROCESSOR
800
MEDIA PIPELINE
830
DISPLAY ENGINE
840
802
820
COMMAND STREAMER
803
VIDEO FRONT-END
834
MEDIA ENGINE
837
2D ENGINE
841
DISPLAY CONTROLLER
843
VERTEX FETCHER
805
VERTEX FETCHER
807
HULL SHADER
811
TESSELLATOR
813
DOMAIN SHADER
817
GEOMETRY SHADER
819
STREAM OUT
823
CLIP/ SETUP
829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

FIG. 8

FIG. 9A

GRAPHICS PROCESSOR COMMAND FORMAT
900

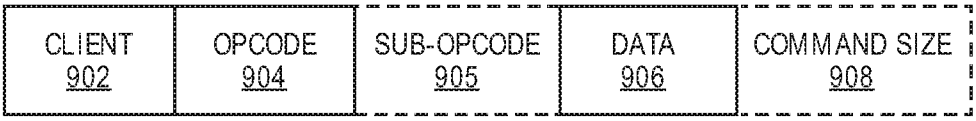
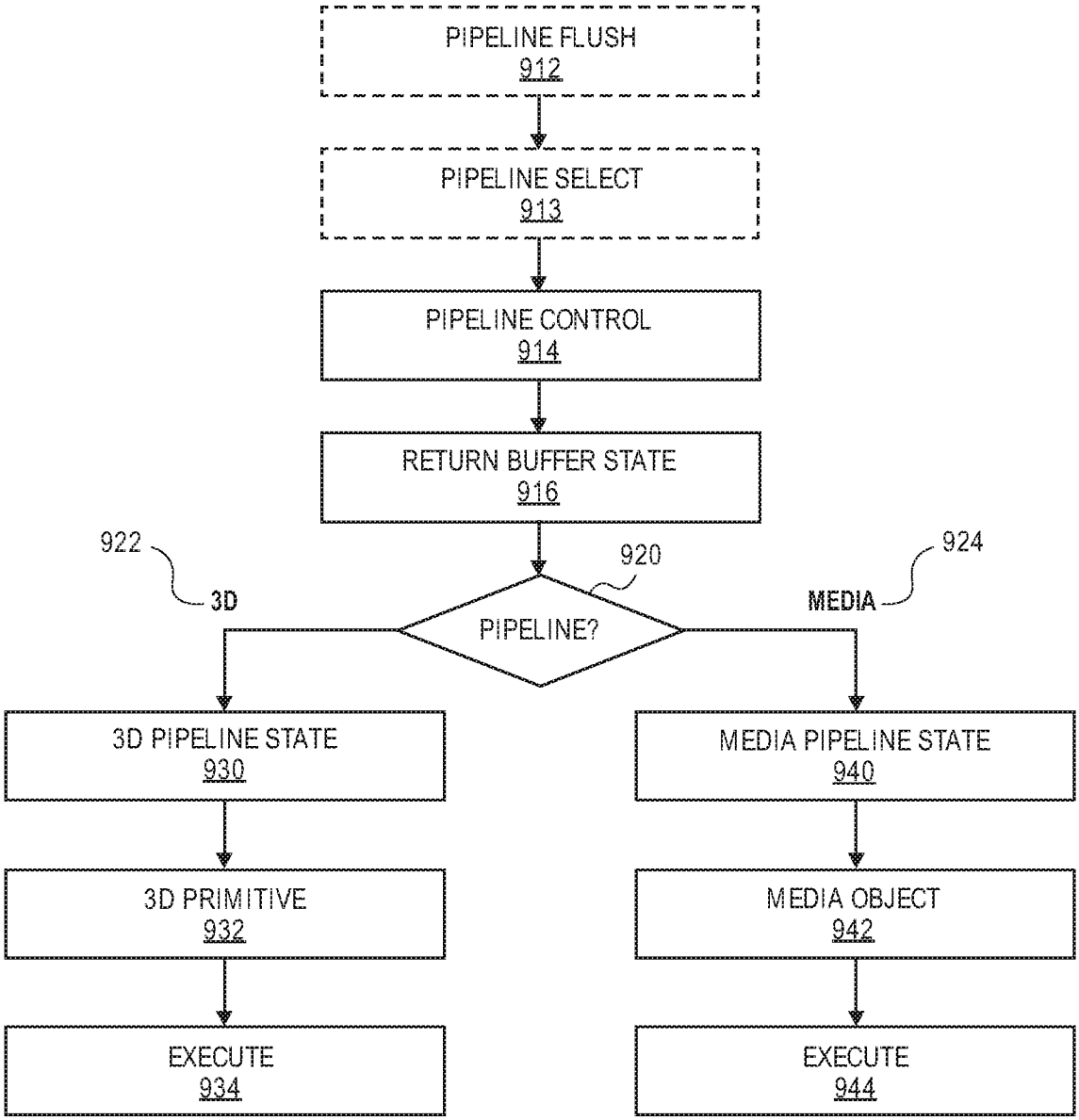


FIG. 9B

GRAPHICS PROCESSOR COMMAND SEQUENCE
910



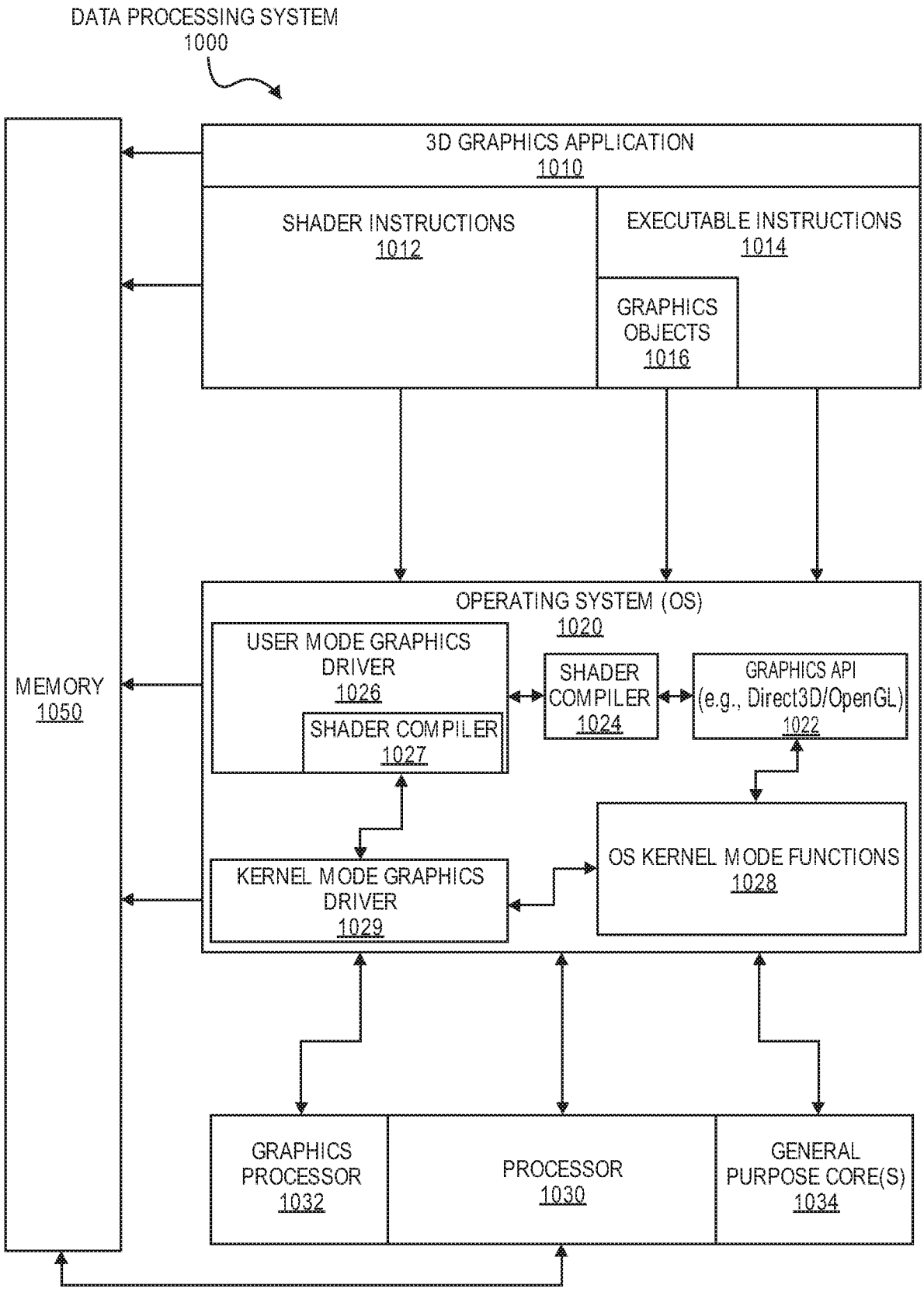

DATA PROCESSING SYSTEM
1000
3D GRAPHICS APPLICATION
1010
SHADER INSTRUCTIONS
1012
EXECUTABLE INSTRUCTIONS
1014
GRAPHICS OBJECTS
1016
MEMORY
1050
OPERATING SYSTEM (OS)
1020
USER MODE GRAPHICS DRIVER
1026
SHADER COMPILER
1027
SHADER COMPILER
1024
GRAPHICS API
(e.g., Direct3D/OpenGL)
1022
OS KERNEL MODE FUNCTIONS
1028
KERNEL MODE GRAPHICS DRIVER
1029
GRAPHICS PROCESSOR
1032
PROCESSOR
1030
GENERAL PURPOSE CORE(S)
1034

FIG. 10

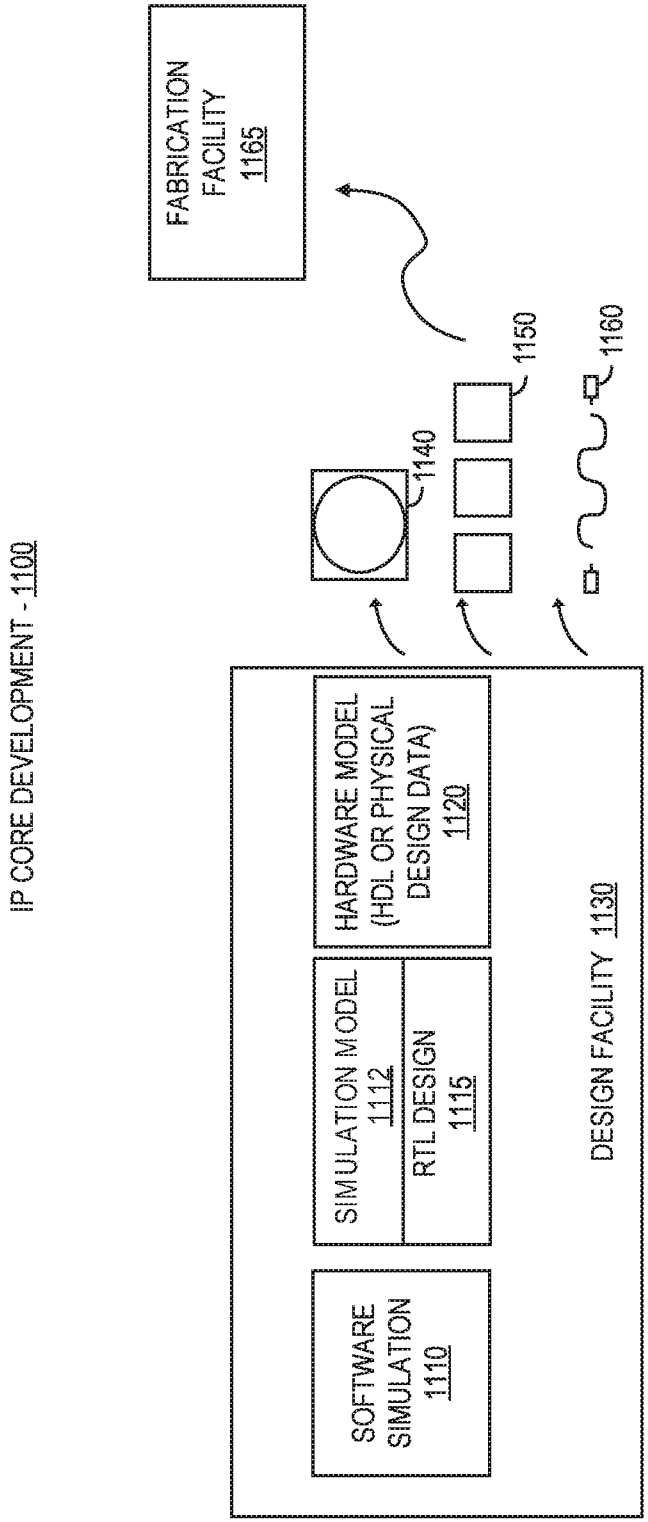

IP CORE DEVELOPMENT - 1100
FABRICATION FACILITY 1165
1140
1150
1160
SOFTWARE SIMULATION 1110
SIMULATION MODEL 1112
RTL DESIGN 1115
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA) 1120
DESIGN FACILITY 1130

FIG. 11

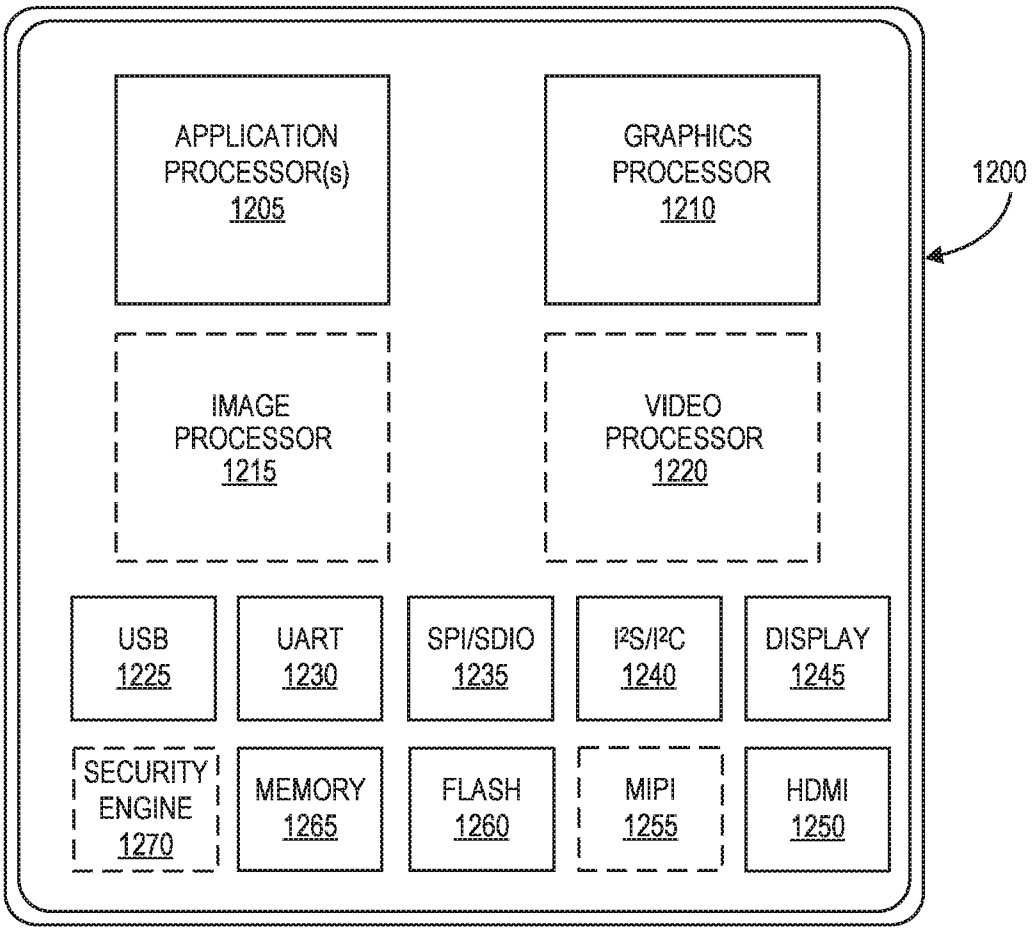

APPLICATION PROCESSOR(s) 1205
GRAPHICS PROCESSOR 1210
1200
IMAGE PROCESSOR 1215
VIDEO PROCESSOR 1220
USB 1225
UART 1230
SPI/SDIO 1235
I²S/I²C 1240
DISPLAY 1245
SECURITY ENGINE 1270
MEMORY 1265
FLASH 1260
MIPI 1255
HDMI 1250

FIG. 12

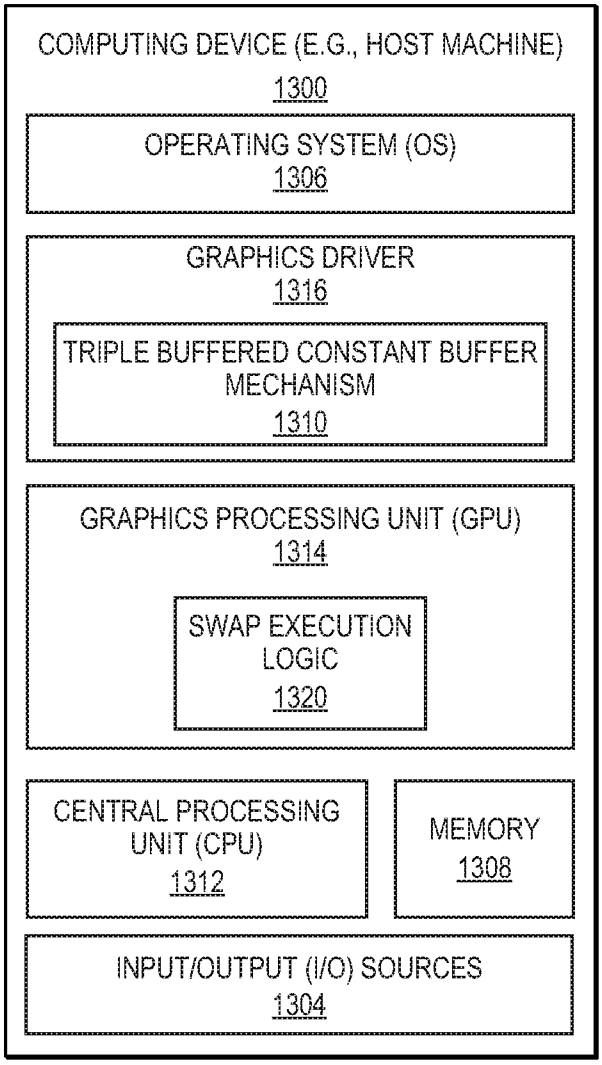

COMPUTING DEVICE (E.G., HOST MACHINE)
1300
OPERATING SYSTEM (OS)
1306
GRAPHICS DRIVER
1316
TRIPLE BUFFERED CONSTANT BUFFER MECHANISM
1310
GRAPHICS PROCESSING UNIT (GPU)
1314
SWAP EXECUTION LOGIC
1320
CENTRAL PROCESSING UNIT (CPU)
1312
MEMORY
1308
INPUT/OUTPUT (I/O) SOURCES
1304

FIG. 13

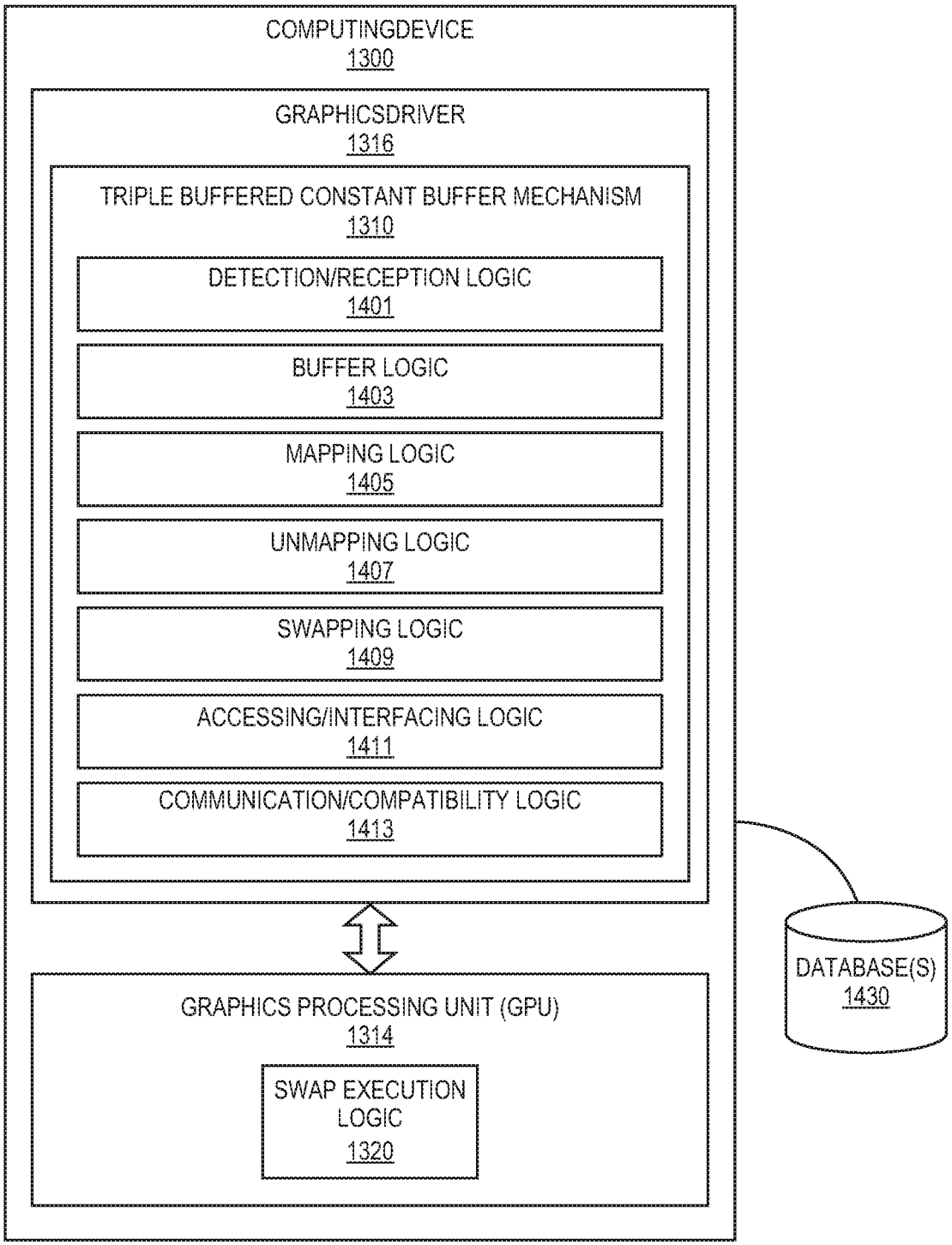

COMPUTINGDEVICE
1300
GRAPHICSDRIVER
1316
TRIPLE BUFFERED CONSTANT BUFFER MECHANISM
1310
DETECTION/RECEPTION LOGIC
1401
BUFFER LOGIC
1403
MAPPING LOGIC
1405
UNMAPPING LOGIC
1407
SWAPPING LOGIC
1409
ACCESSING/INTERFACING LOGIC
1411
COMMUNICATION/COMPATIBILITY LOGIC
1413
GRAPHICS PROCESSING UNIT (GPU)
1314
SWAP EXECUTION LOGIC
1320
DATABASE(S)
1430

FIG. 14

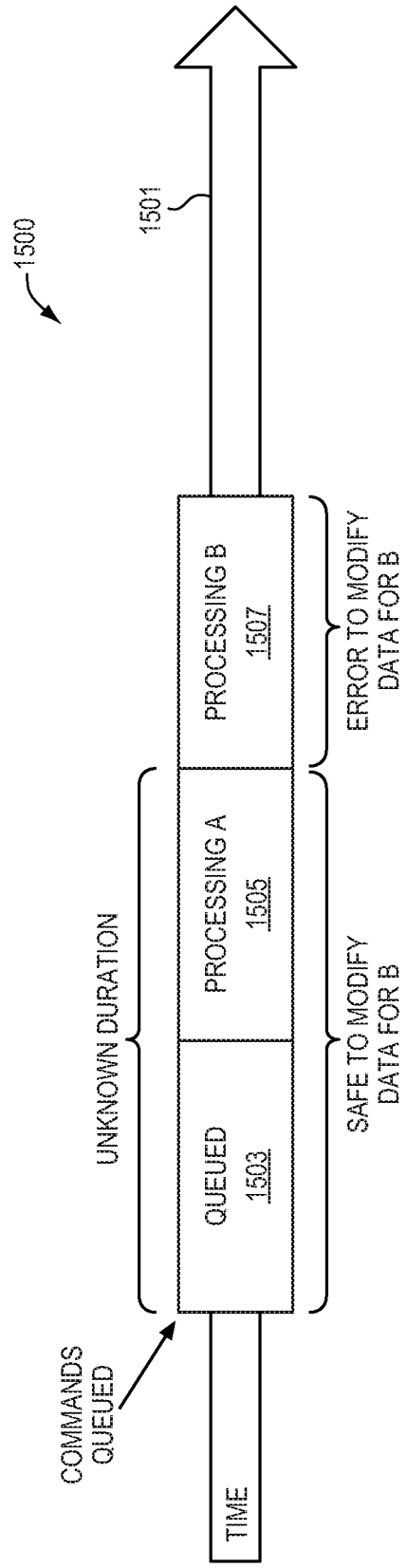


FIG. 15A
PRIOR ART

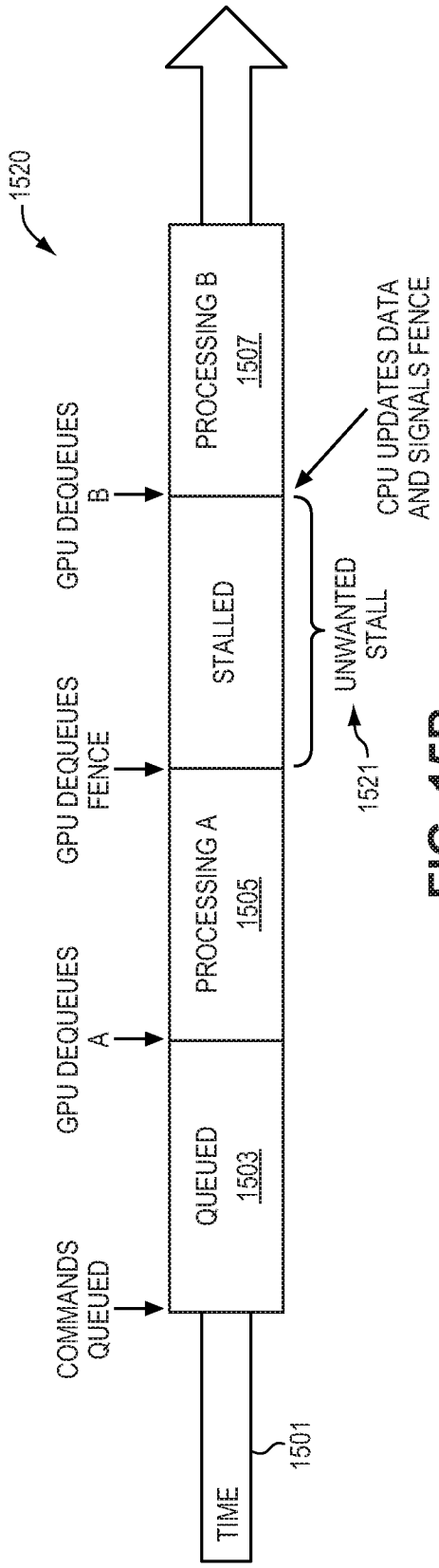


FIG. 15B
PRIOR ART

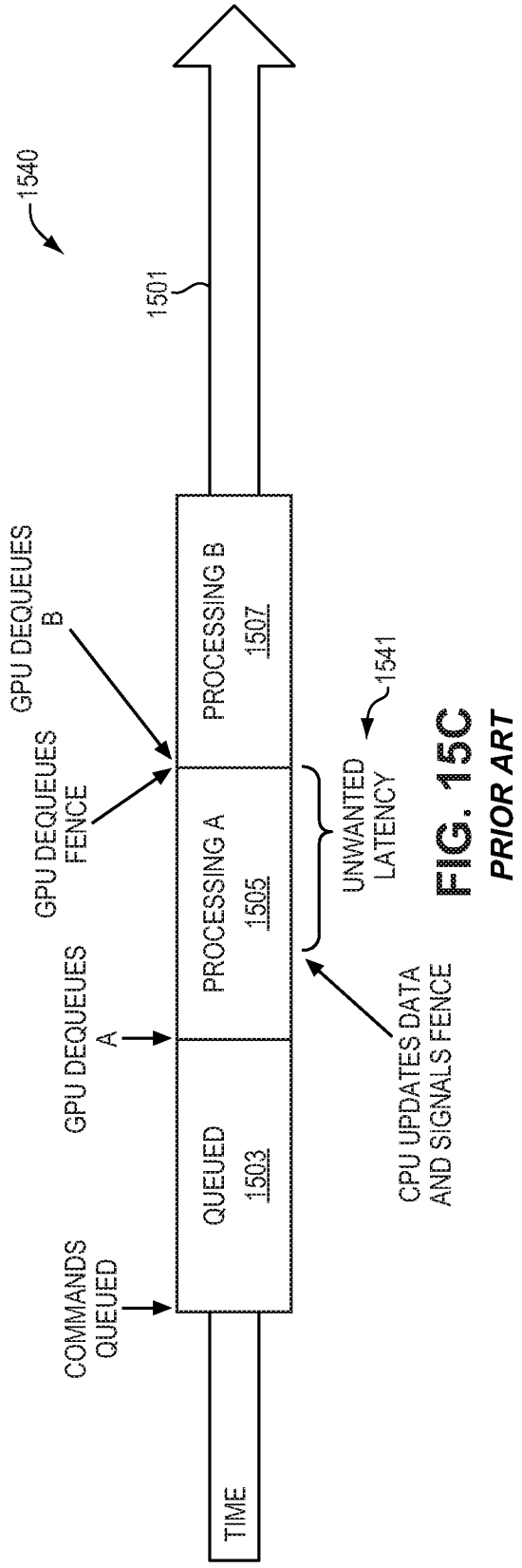


FIG. 15C
PRIOR ART

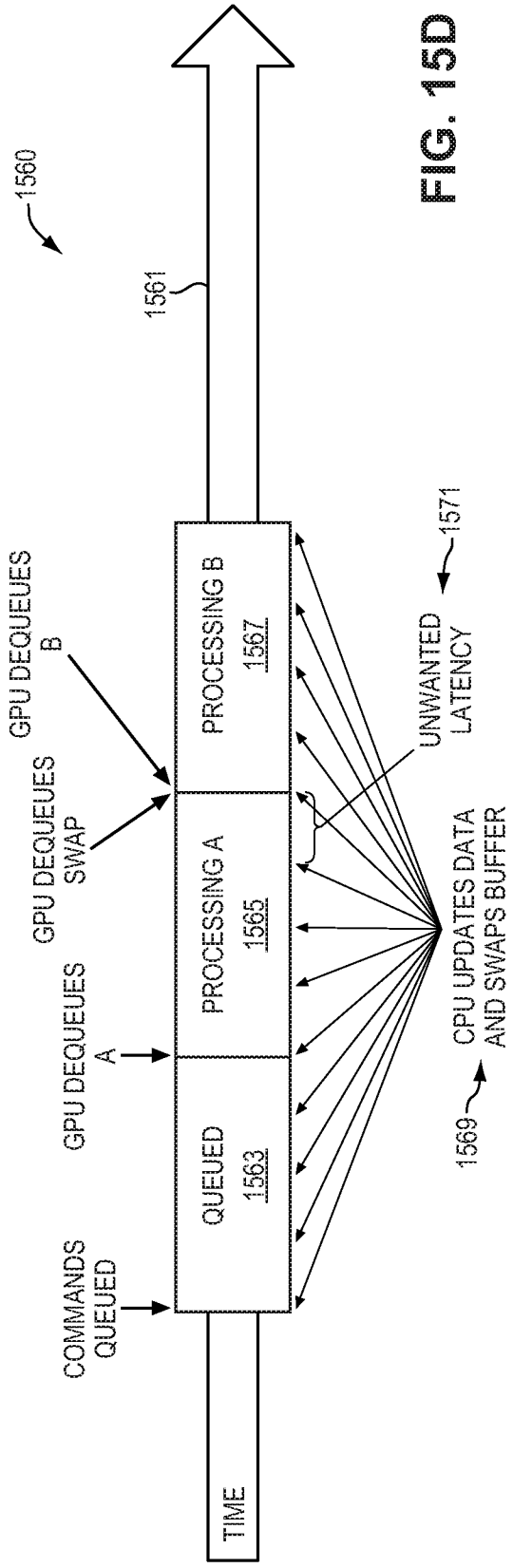


FIG. 15D

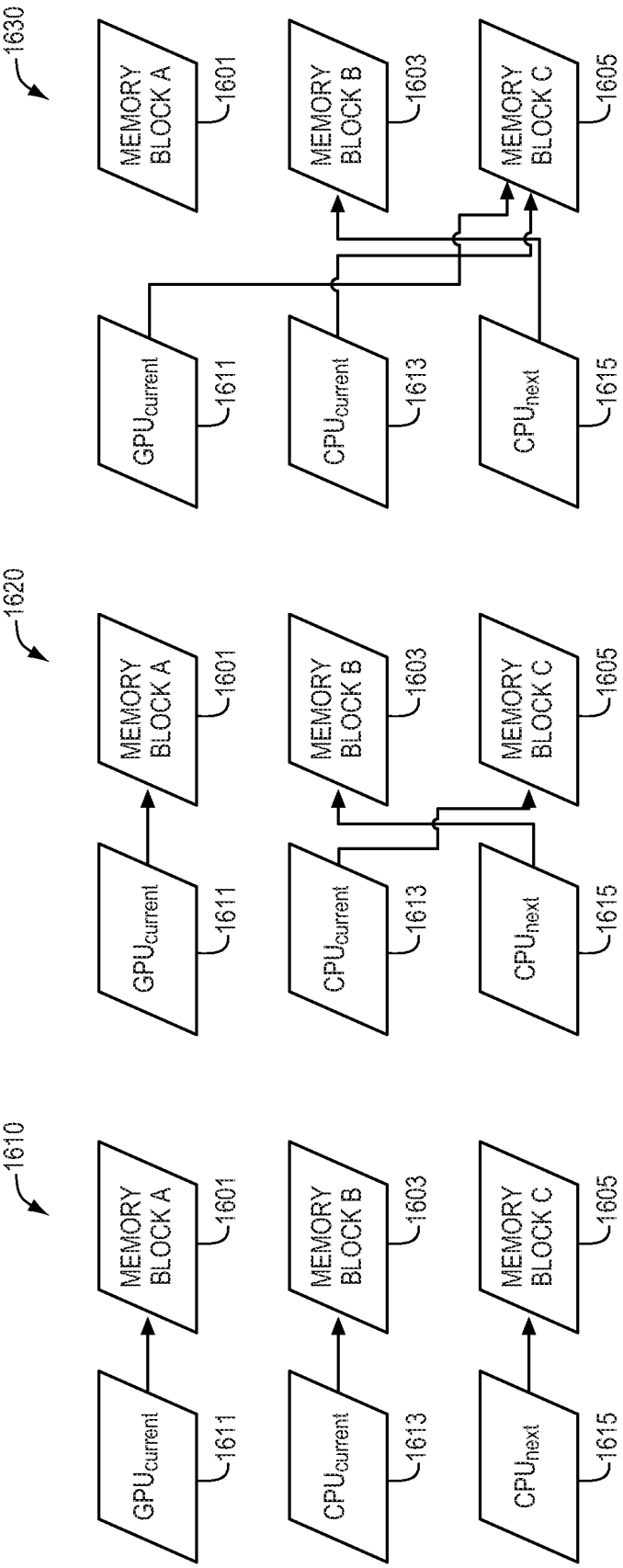

1630
MEMORY BLOCK A
1601
MEMORY BLOCK B
1603
MEMORY BLOCK C
1605
GPUcurrent
1611
CPUcurrent
1613
CPUnext
1615
1620
MEMORY BLOCK A
1601
MEMORY BLOCK B
1603
MEMORY BLOCK C
1605
GPUcurrent
1611
CPUcurrent
1613
CPUnext
1615
1610
MEMORY BLOCK A
1601
MEMORY BLOCK B
1603
MEMORY BLOCK C
1605
GPUcurrent
1611
CPUcurrent
1613
CPUnext
1615

FIG. 16

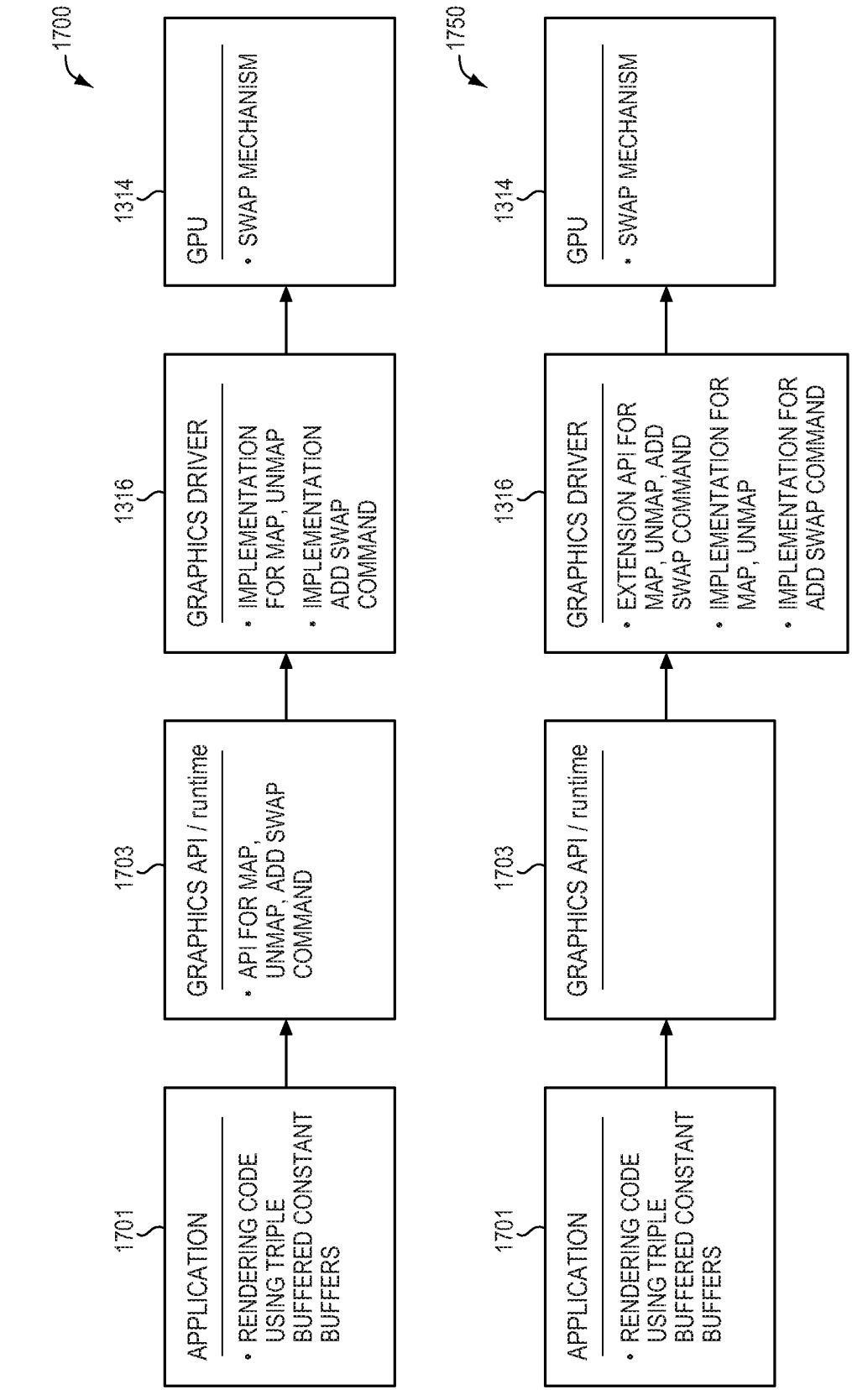

1700
1701
APPLICATION
• RENDERING CODE USING TRIPLE BUFFERED CONSTANT BUFFERS
1703
GRAPHICS API / runtime
• API FOR MAP, UNMAP, ADD SWAP COMMAND
1316
GRAPHICS DRIVER
• IMPLEMENTATION FOR MAP, UNMAP
• IMPLEMENTATION ADD SWAP COMMAND
1314
GPU
• SWAP MECHANISM
1750
1701
APPLICATION
• RENDERING CODE USING TRIPLE BUFFERED CONSTANT BUFFERS
1703
GRAPHICS API / runtime
1316
GRAPHICS DRIVER
• EXTENSION API FOR MAP, UNMAP, ADD SWAP COMMAND
• IMPLEMENTATION FOR MAP, UNMAP
• IMPLEMENTATION FOR ADD SWAP COMMAND
1314
GPU
• SWAP MECHANISM

FIG. 17

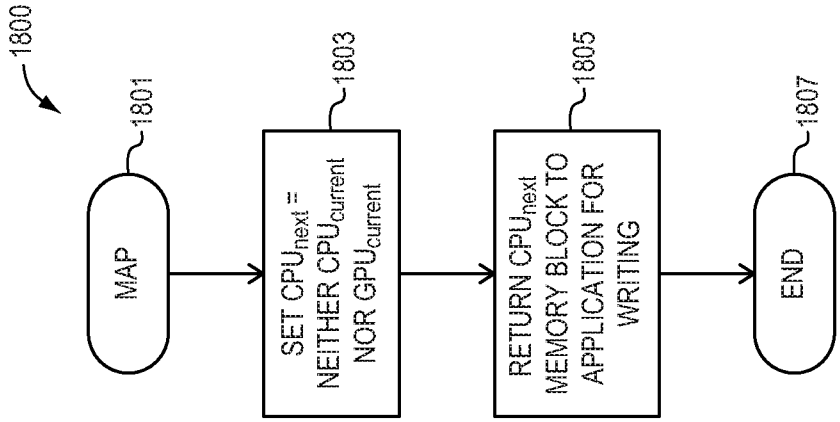


FIG. 18A

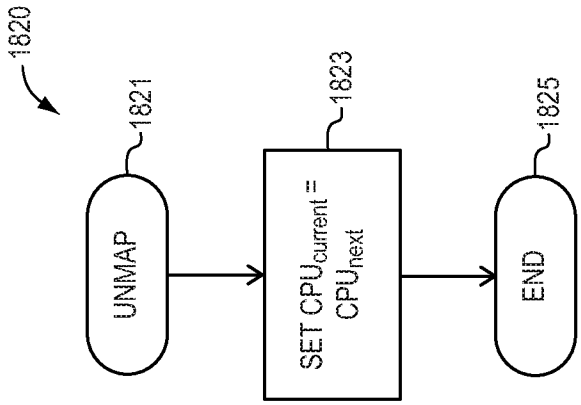


FIG. 18B

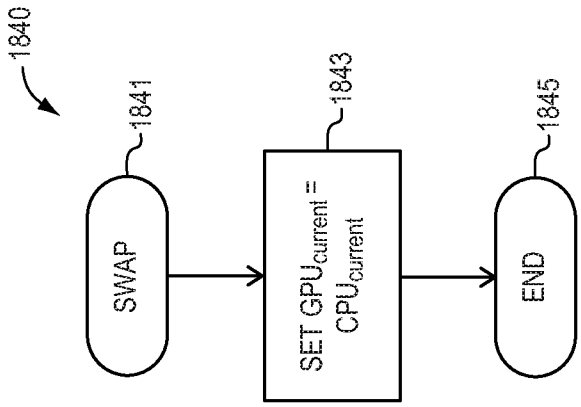


FIG. 18C

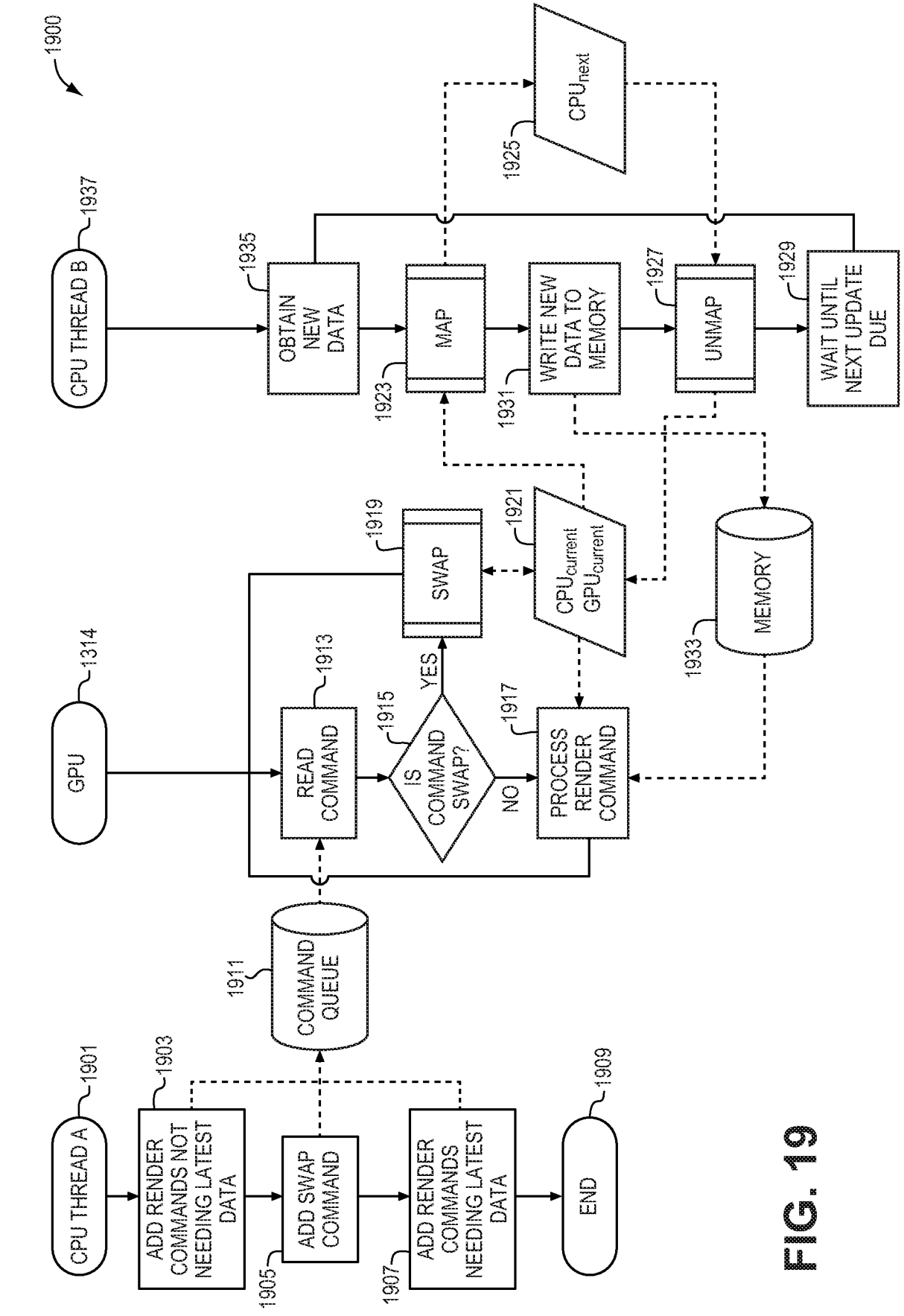

1900
CPU THREAD A
1901
ADD RENDER COMMANDS NOT NEEDING LATEST DATA
1903
ADD SWAP COMMAND
1905
ADD RENDER COMMANDS NEEDING LATEST DATA
1907
END
1909
COMMAND QUEUE
1911
GPU
1314
READ COMMAND
1913
IS COMMAND SWAP?
1915
YES
NO
SWAP
1919
PROCESS RENDER COMMAND
1917
CPUcurrent GPUcurrent
1921
MEMORY
1933
CPU THREAD B
1937
OBTAIN NEW DATA
1935
MAP
1923
WRITE NEW DATA TO MEMORY
1931
UNMAP
1927
WAIT UNTIL NEXT UPDATE DUE
1929
CPUnext
1925

FIG. 19

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/043190

A. CLASSIFICATION OF SUBJECT MATTER

G06T 1/20(2006.01)i, G06T 1/60(2006.01)i, G06F 9/54(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)
G06T 1/20; G06F 12/02; G06F 15/16; G06F 15/80; G09G 5/39; G06T 1/60; G06T 1/00; G06F 9/54

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & keywords: multi-block buffer, swap operation, asynchronous

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2011-0279463 A1 (CHIN-JUNG YANG et al.) 17 November 2011 See paragraphs [0029]-[0036]; claims 1-7; and figures 3A-4B.	1-19
A	US 2014-0104283 A1 (APPLE INC.) 17 April 2014 See paragraphs [0015]-[0025]; and figures 2-3.	1-19
A	US 2013-0021354 A1 (PETER ANTHONY VAN EERD et al.) 24 January 2013 See paragraphs [0041]-[0069]; and figure 2.	1-19
A	US 2015-0006839 A1 (INTEL CORPORATION) 01 January 2015 See paragraphs [0012]-[0020]; and figures 1-2.	1-19
A	US 7821517 B2 (HASSANE S. AZAR et al.) 26 October 2010 See column 8, line 42 - column 9, line 63; and figure 4.	1-19

☐ 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
19 October 2016 (19.10.2016)	**20 October 2016 (20.10.2016)**

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea
Facsimile No. +82-42-481-8578

Authorized officer
AHN, Jeong Hwan
Telephone No. +82-42-481-8633

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2016/043190

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2011-0279463 A1	17/11/2011	CN 102243755 A	16/11/2011
		CN 102243755 B	20/03/2013
		TW 201140493 A	16/11/2011
		TW I452538 B	11/09/2014
		US 8823719 B2	02/09/2014
US 2014-0104283 A1	17/04/2014	US 8937623 B2	20/01/2015
US 2013-0021354 A1	24/01/2013	CA 2782832 A1	19/01/2013
		EP 2549472 A1	23/01/2013
		US 9263003 B2	16/02/2016
US 2015-0006839 A1	01/01/2015	CN 104025037 A	03/09/2014
		EP 2801023 A1	12/11/2014
		JP 2015-505409 A	19/02/2015
		WO 2013-103573 A1	11/07/2013
US 7821517 B2	26/10/2010	CA 2567079 A1	04/05/2007
		CN 100960438 A	09/05/2007
		CN 104767956 A	08/07/2015
		EP 1784021 A2	09/05/2007
		EP 1784021 A3	26/12/2007
		EP 1784021 B1	19/05/2010
		JP 04568711 B2	27/10/2010
		JP 2007-215156 A	23/08/2007
		KR 10-0830286 B1	19/05/2008
		KR 10-2007-0048634 A	09/05/2007
		TW I339795 B	01/04/2011
		US 2007-0103590 A1	10/05/2007
		US 2009-0207178 A1	20/08/2009
		US 7525548 B2	28/04/2009