

(43) International Publication Date
29 December 2016 (29.12.2016)(51) International Patent Classification:
G06F 12/08 (2006.01)(21) International Application Number:
PCT/CN2015/000463(22) International Filing Date:
26 June 2015 (26.06.2015)

(25) Filing Language: English

(26) Publication Language: English

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

(72) Inventors; and

(71) Applicants (for BZ only): TIAN, Kun [CN/CN]; No. 880 Zi Xing Road, Shanghai Zizhu Science Park, Shanghai, 31 200241 (CN). DONG, Yaozu [CN/CN]; Apt. 10I (1009), Building #5, Yanping Road 123, Shanghai, 31 200042 (CN).

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

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

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

Published:

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

(54) Title: APPARATUS AND METHOD FOR EFFICIENT COMMUNICATION BETWEEN VIRTUAL MACHINES

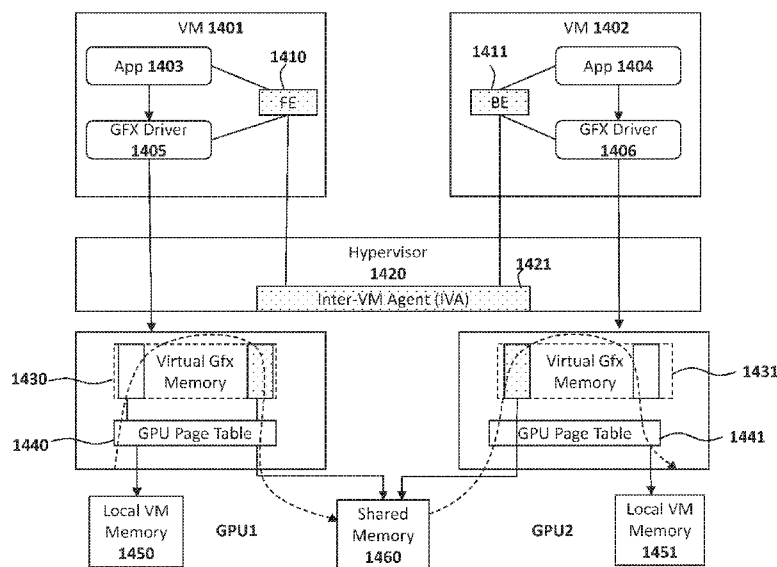


FIG. 14

(57) Abstract: An apparatus and method are described for efficient inter-virtual machine (VM) communication. For example, an apparatus comprises inter-VM communication logic to map a first specified set of device virtual memory addresses of a first VM to a first set of physical memory addresses in a shared system memory and to further map a second specified set of device virtual memory addresses of a second VM to the first set physical memory addresses in the shared system memory.

APPARATUS AND METHOD FOR EFFICIENT
COMMUNICATION BETWEEN VIRTUAL MACHINES

BACKGROUND

Field of the Invention

[0001] This invention relates generally to the field of computer processors. More particularly, the invention relates to an apparatus and method for efficient communication between virtual machines.

Description of the Related Art

[0002] Efficiency of inter-virtual machine (VM) communication is important in today's cloud computing, when multi-VMs are hosted on the same network node working together to fulfill some advanced set of features. For example in Network Function Virtualization (NFV), a firewall VM, a load balancing VM, a router VM, and other networking VMs may run on the same node so a fast inter-VM communication mechanism can bring improved performance.

[0003] Inter-VM communication is also important within a GPU cloud, or GPU as-a-Service (GaaS) which may be used for remote desktop, cloud media, cloud visual analytic, cloud gaming, and high performance computing (HPC). Hosting multiple-VMs on the same node, each fulfilling a specific responsibility, can achieve the best performance compared with running VMs on different nodes. By way of example, streams may be decoded in VM1, post-processed in VM2, and then encoded in VM3. Similarly, ten streams may be decoded in VM1, then five streams encoded individually in VM2/VM3 (encoding is more processor intensive than decoding). By way of another example, 3D rendering may be performed in VM1, encoded and delivered to the remote client in VM2.

[0004] As illustrated in **Figure 13A**, existing inter-VM communication mechanisms include a system memory 1305 which is shared between virtual machines 1301-1302 under the control of the hypervisor 1310. This shared-system-memory architecture avoids unnecessary memory copies from one VM to another and works efficiently for most I/O communication which does not utilize local memory on the I/O device. However, as illustrated in **Figure 13B**, when each VM 1301-1302 includes a local graphics memory 1320-1321, respectively, there is a heavy cost associated with moving data between system memory 1305 and local graphics memories 1320-1321, which could be very inefficient as far as GPU cloud is concerned. Because device local memory such a graphics memory 1320-1321 is generally mapped as uncacheable, reads can be very slow (e.g. reading a 1080p frame could take ~60ms which could

significantly reduce the overall throughput e.g., when talking about 100's FPS). So existing shared-system-memory model can't guarantee fast and efficient inter-VM communication requirement, as far as device local memory is concerned.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

[0006] **FIG. 1** is a block diagram of an embodiment of a computer system with a processor having one or more processor cores and graphics processors;

[0007] **FIG. 2** is a block diagram of one embodiment of a processor having one or more processor cores, an integrated memory controller, and an integrated graphics processor;

[0008] **FIG. 3** is a block diagram of one embodiment of a graphics processor which may be a discrete graphics processing unit, or may be graphics processor integrated with a plurality of processing cores;

[0009] **FIG. 4** is a block diagram of an embodiment of a graphics-processing engine for a graphics processor;

[0010] **FIG. 5** is a block diagram of another embodiment of a graphics processor;

[0011] **FIG. 6** is a block diagram of thread execution logic including an array of processing elements;

[0012] **FIG. 7** illustrates a graphics processor execution unit instruction format according to an embodiment;

[0013] **FIG. 8** is a block diagram of another embodiment of a graphics processor which includes a graphics pipeline, a media pipeline, a display engine, thread execution logic, and a render output pipeline;

[0014] **FIG. 9A** is a block diagram illustrating a graphics processor command format according to an embodiment;

[0015] **FIG. 9B** is a block diagram illustrating a graphics processor command sequence according to an embodiment;

[0016] **FIG. 10** illustrates exemplary graphics software architecture for a data processing system according to an embodiment;

[0017] **FIG. 11** illustrates an exemplary IP core development system that may be used to manufacture an integrated circuit to perform operations according to an embodiment;

[0018] **FIG. 12** illustrates an exemplary system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment;

[0019] FIGS. 13A-B illustrates exemplary systems in which system memory is shared between virtual machines;

[0020] FIG. 14 illustrates one embodiment of an architecture for sharing memory between virtual machines; and

[0021] FIG. 15 illustrates one embodiment of a method for sharing memory between virtual machines.

DETAILED DESCRIPTION

[0022] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the embodiments of the invention described below. It will be apparent, however, to one skilled in the art that the embodiments of the invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form to avoid obscuring the underlying principles of the embodiments of the invention.

EXEMPLARY GRAPHICS PROCESSOR ARCHITECTURES AND DATA TYPES

[0023] System Overview

[0024] 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 one embodiment, the system 100 is a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices.

[0025] 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.

[0026] 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).

[0027] 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.

[0028] 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.

[0029] 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.

[0030] 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.

[0031] **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 **Figure 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.

[0032] 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.

[0033] 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).

[0034] 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.

[0035] 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.

[0036] 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.

[0037] 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.

[0038] 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.

[0039] **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.

[0040] 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.

[0041] 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.

[0042] 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.

[0043] 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.

[0044] 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.

[0045] 3D/Media Processing

[0046] **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 **Figure 3**. Elements of **Figure 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.

[0047] 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.

[0048] 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.

[0049] 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 frame 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).

[0050] 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.

[0051] 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.

[0052] 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.

[0053] Execution Units

[0054] **Figure 5** is a block diagram of another embodiment of a graphics processor 500. Elements of **Figure 5** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0055] 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.

[0056] 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.

[0057] 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.

[0058] **Figure 6** illustrates thread execution logic 600 including an array of processing elements employed in some embodiments of a GPE. Elements of **Figure 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.

[0059] 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.

[0060] 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).

[0061] 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.

[0062] 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.

[0063] 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.

[0064] During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 600 via thread spawning and dispatch logic. In some embodiments, thread execution logic 600 includes a local thread dispatcher 604 that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units 608A-608N. For example, the geometry pipeline (e.g., 536 of **Figure 5**) dispatches vertex processing, tessellation, or geometry processing threads to thread execution logic 600 (**Figure 6**). In some embodiments, thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

[0065] 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.

[0066] 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.

[0067] 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.

[0068] 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.

[0069] 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.

[0070] 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.

[0071] 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.

[0072] 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.

[0073] 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.

[0074] 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.

[0075] **Graphics Pipeline**

[0076] **Figure 8** is a block diagram of another embodiment of a graphics processor 800. Elements of **Figure 8** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0077] 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.

[0078] 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.

[0079] 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.

[0080] 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.

[0081] 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.

[0082] 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.

[0083] 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.

[0084] 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.

[0085] 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.

[0086] 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.

[0087] 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.

[0088] **Graphics Pipeline Programming**

[0089] **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 **Figure 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 **Figure 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.

[0090] 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.

[0091] The flow diagram in **Figure 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.

[0092] 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.

[0093] 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.

[0094] 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.

[0095] 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.

[0096] 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.

[0097] 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.

[0098] 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.

[0099] 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.

[00100] 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.

[00101] 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 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.

[00102] 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.

[00103] **Graphics Software Architecture**

[00104] **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.

[00105] 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.

[00106] In some embodiments, operating system 1020 is a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. When the Direct3D API is in use, the operating system 1020 uses a front-end shader compiler 1024 to compile any shader instructions 1012 in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. In some embodiments, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application 1010.

[00107] 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.

[00108] **IP Core Implementations**

[00109] 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.

[00110] **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. A register transfer level (RTL) design can then be created or synthesized from the simulation model 1100. 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.

[00111] 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.

[00112] **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 I²S/I²C controller 1240. Additionally, the integrated circuit can include a display device

1245 coupled to one or more of a high-definition multimedia interface (HDMI) controller 1250 and a mobile industry processor interface (MIPI) display interface 1255. Storage may be provided by a flash memory subsystem 1260 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller 1265 for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine 1270.

[00113] 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.

APPARATUS AND METHOD FOR EFFICIENT COMMUNICATION BETWEEN VIRTUAL MACHINES

[00114] As illustrated in **Figure 13A**, existing inter-VM communication mechanisms include a system memory 1305 which is shared between virtual machines (VMs) 1301-1302 under the control of the hypervisor 1310. This architecture avoids unnecessary memory copies from one VM to another and works efficiently for most I/O communication which does not utilize local memory on the I/O device. However, as illustrated in **Figure 13B**, when each VM 1301-1302 includes a local graphics memory 1320-1321, respectively, there is a heavy cost associated with moving data between system memory 1305 and local graphics memories 1320-1321, which may become very inefficient as far as the GPU cloud is concerned. Because a device local memory such a graphics memory 1320-1321 is generally mapped as uncacheable, reads can be very slow. (e.g. reading a 1080p frame could take ~60ms which could significantly reduce the overall throughput e.g., when talking about 100's FPS). So existing shared-system-memory model can't guarantee fast and efficient inter-VM communication requirement, as far as device local memory is concerned.

[00115] The embodiments of the invention utilize device-level memory sharing to provide fast inter-VM communication for virtualization usages with local memory on devices GPU devices and I/O devices. For example, for a GPU implementation, these embodiments include a fast inter-VM communication mechanism with device-level memory sharing to avoid unnecessary copy overhead between the system memory and graphics memory. While the embodiments described below focus on an implementation in which the device is a GPU (e.g., for a GPU cloud), the underlying principles of the invention may be implemented on any system which includes device-level memory sharing, and not restricted to the virtualization usages

[00116] **Figure 14** illustrates an exemplary embodiment which includes a first virtual machine (VM) 1401 and a second VM 1402 which efficiently exchange data through a shared

memory 1460 using the techniques described herein. In the illustrated embodiment, VM 1401 includes a graphics application 1403 which utilizes a graphics driver 1405 and a front end (FE) driver component 1410. Similarly, VM 1402 includes a graphics application 1404 which also includes a graphics driver 1406 and a back end (BE) driver component 1411. In some embodiments below, the first VM 1401 is referred to as the “source” VM and the second VM is referred to as the “destination” VM (i.e., because the first VM 1401 acts a data source for the second VM 1402).

[00117] As illustrated, the source VM 1401 includes a virtual graphics memory 1430 mapped to a local VM physical memory 1450 using a first GPU page table 1440 and the destination VM includes a virtual graphics memory 1431 mapped to a local VM memory 1451 using a second GPU page table 1441. In addition, in one embodiment, the GPU page tables 1440-1441 map the virtual graphics memories 1430-1431 to a system memory 1460 for sharing memory pages between the first VM 1401 and the second VM 1402.

[00118] In one embodiment, a hypervisor 1420 controls the GPU page tables 1440-1441, either from two physical GPUs (in GPU pass-through scenario), or from two virtual GPUs (in GPU sharing scenario), to map to the same pages in system memory 1460 as specified by the source VM and the destination VM. Then the shared system memory 1460, mapped to different graphics memory addresses in the two VMs 1401-1402, is directly used as the render output in the source VM 1401 and the render input in the destination VM 1402, thus removing all unnecessary CPU copies to achieve high performance.

[00119] In the embodiment shown in **Figure 14**, the FE/BE driver pair 1410/1411 is installed in source/destination VMs 1401/1402 individually, interfacing with the applications 1403/1404, graphics drivers 1405/1406 and an Inter-VM Agent (IVA) 1421 to initialize and manage the sharing in device local memory 1450/1451. Once the inter-VM communication mechanism described herein is established, the shared system memory 1460 is directly used as the render output by the application 1403 in the source VM 1401, and then directly taken as render input to the application 1404 in destination VM 1402, completely removing all unnecessary CPU copies in the path.

[00120] Modern GPUs support the concept of virtual memory, which can be mapped to either device local memory 1450-1451 or system memory 1460 pages, through a GPU page table 1440-1441. Based on this feature, the application 1403-1404 or graphics driver 1405-1406 may program the GPU to take either device local memory or system memory as the graphics memory operand in a unified manner. For example, in some processor architectures (e.g., such as Intel Processor Graphics), there is no device local memory, i.e., all graphics virtual memory is backed by a system memory.

[00121] **Figure 14** illustrates an embodiment which is based on such a system, where graphics virtual memory 1430-1431 is backed by VM system memory 1460. The FE component 1410 and BE component 1411 can be implemented as part of the graphics drivers 1405 and 1406, respectively, or maintained as separate components (as illustrated). In one embodiment, the FE 1410 and BE 1411 each request a range of free virtual memory addresses from the graphics drivers 1405-1406 (which could be different in each VM) and then provide virtual memory address and system memory pages to the IVA 1421 in the hypervisor 1420.

[00122] In one embodiment, the IVA 1421 will then configure the GPU page tables 1440-1441 to map the specified virtual memory addresses to the shared system memory 1460, for each GPU independently. The FE component 1410 and BE component 1411 may initially communicate to negotiate whether to establish such a device-level communication channel. The IVA 1421 then determines whether the two VMs 1401-1402 are qualified to establish an inter-VM channel in such a manner, e.g., based on whether the two VMs 1401-1402 run on the same physical node, and/or based on GPU functionalities. In one embodiment, the IVA 1421 employs a handshake protocol to detect whether two VMs 1401-1402 are qualified to use the mechanism. If the two VMs are in different nodes, for example, the IVA 1421 determines that they are not qualified. In one embodiment, the IVA 1421 may implement a dynamic switch between fast communication (device level sharing) and slow communication (network based) at run-time.

[00123] The embodiments of the invention described herein may be applied to both integrated GPUs and discrete GPUs. In fact, these embodiments may be applied to any GPU as long as the GPU provides a mechanism to map system memory into graphics virtual memory address space.

[00124] A method in accordance with one embodiment of the invention for establishing a fast inter-VM channel is illustrated in **Figure 15**. The method may be implemented within the context of the system architecture illustrated in **Figure 14**, but is not limited to any particular system architecture.

[00125] The left side of **Figure 15** applies to the front end (FE) and the right side applies to the back end (BE). At 1501 a determination is made as to whether the BE agrees to use device-level sharing. If not, then at 1530, an existing inter-VM communication mechanism is implemented (e.g., shared memory based or network-based). If the BE agrees to device-level sharing, then at 1502, the FE requests the application to allocate system memory pages. At 1503, the FE requests the graphics driver to allocate empty virtual graphics memory addresses, and at 1504, the FE requests the IVA to set up the mapping to be used for sharing as described herein. If the IVA determines that the FE/BE sharing is qualified (e.g., if they are on the same node), then at 1506, the IVA configures the GPU page table to implement device-level sharing.

At 1507, the FE identifies the allocated virtual graphics memory addresses to the application and, at 1508, the application starts rendering output into the specified virtual graphics memory locations. By way of example, and not limitation, the application in the VM with the FE may perform 3D rendering, the results of which may be encoded and delivered to the application in the VM with the BE using the inter-VM communication techniques described herein. The VM with the BE may then further process/encode the 3D rendering results.

[00126] The operations on the right, performed with respect to the BE, may occur simultaneously with the operations on the left, performed with respect to the FE. At 1521, a determination is made as to whether the FE agrees to use device-level sharing. If not then at 1530, an existing inter-VM communication mechanism is implemented (e.g., shared memory based or network-based). If the FE agrees to device-level sharing, then at 1522, the BE requests the graphics driver to allocate empty virtual graphics memory addresses. At 1523, the BE requests the IVA to set up the mapping to be used for sharing as described herein. If the IVA determines that the FE/BE sharing is qualified (e.g., if they are on the same node), then at 1525, the IVA configures the GPU page table to implement device-level sharing. At 1526, the BE identifies the allocated virtual graphics memory addresses to the application and, at 1508, the application begins utilizing the specified virtual graphics memory as render input.

[00127] In one embodiment, event notification is still implemented in the same manner as existing inter-VM communication mechanisms, either through a hypervisor-specific event channel, or through a virtual network protocol.

[00128] In addition, with respect to live migration, in one embodiment, FE 1410/BE 1411 can be notified to dynamically switch between a device-level channel and a network based channel when one VM is migrated to another node. In another embodiment, the IVA is made aware of the live migration and may, for example, update some status bit(s) in the shared memory page to provide an indication of the migration. Subsequently, when the FE/BE attempts to issue a transaction, they will recognize the status bit(s) and then negotiate to use the network based channel themselves.

[00129] The embodiments of the invention may be implemented on any type of GPU which provides a mechanism for mapping system memory into its graphics memory address space. These embodiments may also be implemented using any GPU virtualization techniques, including GPU passthrough techniques (e.g., each VM receives a dedicated GPU such as with an Intel VT-d implementation) and GPU sharing (e.g., multiple VMs shares a single GPU, either through full emulated vGPU such as Intel GVT-g, or through API-level forwarding). These implementations may be used as long as the IVA is provided with the ability to control the physical GPU page table.

[00130] Embodiments of the invention may include various steps, which have been described above. The steps may be embodied in machine-executable instructions which may be used to cause a general-purpose or special-purpose processor to perform the steps. Alternatively, these steps may be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

[00131] As described herein, instructions may refer to specific configurations of hardware such as application specific integrated circuits (ASICs) configured to perform certain operations or having a predetermined functionality or software instructions stored in memory embodied in a non-transitory computer readable medium. Thus, the techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices (e.g., an end station, a network element, etc.). Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer machine-readable media, such as non-transitory computer machine-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer machine-readable communication media (e.g., electrical, optical, acoustical or other form of propagated signals – such as carrier waves, infrared signals, digital signals, etc.).

[00132] In addition, such electronic devices typically include a set of one or more processors coupled to one or more other components, such as one or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage device of a given electronic device typically stores code and/or data for execution on the set of one or more processors of that electronic device. Of course, one or more parts of an embodiment of the invention may be implemented using different combinations of software, firmware, and/or hardware. Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the invention may be practiced without some of these specific details. In certain instances, well known structures and functions were not described in elaborate detail in order to avoid obscuring the subject matter of the present invention. Accordingly, the scope and spirit of the invention should be judged in terms of the claims which follow.

CLAIMS

What is claimed is:

1. An apparatus comprising:
inter-VM communication logic to map a first specified set of device virtual memory addresses of a first VM to a first set of physical memory addresses in a shared system memory and to further map a second specified set of device virtual memory addresses of a second VM to the first set physical memory addresses in the shared system memory.
2. The apparatus as in claim 1 further comprising:
a first page table associated with the first VM to be updated by the inter-VM communication logic to include the mapping between the first specified set of device virtual memory addresses of the first VM to the first set physical memory addresses; and
a second page table associated with the second VM to be updated by the inter-VM communication logic to include the mapping between the second specified set of device virtual memory addresses of the second VM to the first set physical memory addresses.
3. The apparatus as in claim 2 wherein the first VM comprises a first graphics driver to execute graphics commands responsive to a first graphics application and wherein the second VM comprises a second graphics driver to implement graphics commands responsive to a second graphics application, wherein the first page table comprises a first graphics processing unit (GPU) page table and the second page table comprises a second GPU page table.
4. The apparatus as in claim 3 further comprising:
a front end (FE) component associated with the first graphics driver to request the inter-VM communication logic to establish the mappings in the first page table for the first VM; and
a back end (BE) component associated with the second graphics driver to request the inter-VM communication logic to establish the mappings in the second page table for the second VM.
5. The apparatus as in claim 4 wherein the FE is configured to request a first range of non-present device virtual memory addresses from the first graphics driver and the BE is configured to request a second range of non-present device virtual memory addresses from the second graphics driver.

6. The apparatus as in claim 5 wherein the FE and the BE are configured to provide the first range and the second range to the inter-VM communication logic, which responsively establishes the mappings using the first range and the second range within the first page table and the second page table, respectively.

7. The apparatus as in claim 6 wherein the FE and the BE are further configured to provide an indication of a system memory address range to be used by the inter-VM communication logic to establish the mappings in the first page table and the second page table.

8. The apparatus as in claim 7 wherein the inter-VM communication logic is configured to determine whether the first VM and the second VM are qualified to establish an inter-VM channel through the system memory using the first and second page tables and establishes the mappings in the first and second page tables only upon a determination that the first VM and the second VM are qualified.

9. The apparatus as in claim 8 wherein the inter-VM communication logic determines that the first VM and the second VM are qualified if the first VM and the second VM are running on a common physical node.

10. The apparatus as in claim 1 wherein the first VM comprises a source VM configured to perform a first set of graphics operations and write results in the shared system memory and wherein the second VM comprises a destination VM configured to perform a second set of graphics operations using the results as input.

11. The apparatus as in claim 10 wherein the graphics operations comprise graphics rendering operations.

12. A method for efficient virtual machine (VM) communication comprising:
sharing a system memory between a first virtual machine (VM) and a second VM;
mapping a first specified set of device virtual memory addresses of the first VM to a first set of physical memory addresses in the shared system memory; and
mapping a second specified set of device virtual memory addresses of the second VM to the first set physical memory addresses in the shared system memory.

13. The method as in claim 12 further comprising:

updating a first page table associated with the first VM to include the mapping between the first specified set of device virtual memory addresses of the first VM to the first set physical memory addresses; and

updating a second page table associated with the second VM to include the mapping between the second specified set of device virtual memory addresses of the second VM to the first set physical memory addresses.

14. The method as in claim 13 wherein the first VM comprises a first graphics driver to execute graphics commands responsive to a first graphics application and wherein the second VM comprises a second graphics driver to implement graphics commands responsive to a second graphics application, wherein the first page table comprises a first graphics processing unit (GPU) page table and the second page table comprises a second GPU page table.

15. The method as in claim 14 further comprising:
requesting the mappings in the first page table for the first VM by a front end (FE) component associated with the first graphics driver; and
requesting the mappings in the second page table for the second VM by a back end (BE) component associated with the second graphics driver.

16. The method as in claim 15 wherein the FE is configured to request a first range of non-present device virtual memory addresses from the first graphics driver and the BE is configured to request a second range of non-present device virtual memory addresses from the second graphics driver.

17. The method as in claim 16 wherein the FE and the BE are configured to provide the first range and the second range, the method further comprising:
responsively establishing the mappings using the first range and the second range within the first page table and the second page table, respectively.

18. The method as in claim 17 wherein the FE and the BE are further configured to provide an indication of a system memory address range to be used to establish the mappings in the first page table and the second page table.

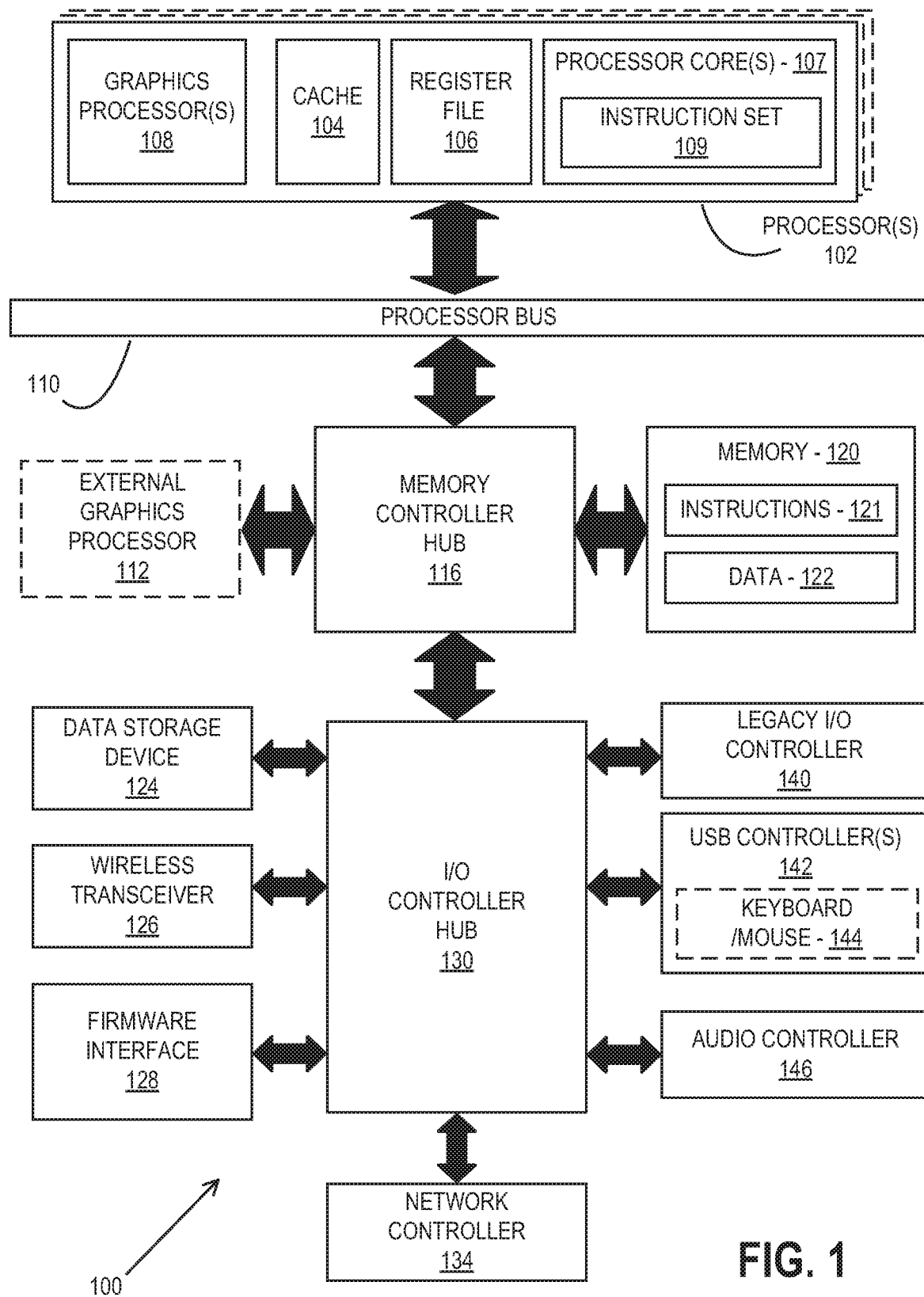
19. The method as in claim 18 further comprising:

determining whether the first VM and the second VM are qualified to establish an inter-VM channel through the system memory using the first and second page tables and establishes the mappings in the first and second page tables only upon a determination that the first VM and the second VM are qualified.

20. The method as in claim 19 wherein determining further comprises determining that the first VM and the second VM are qualified if the first VM and the second VM are running on a common physical node.

21. The method as in claim 12 wherein the first VM comprises a source VM configured to perform a first set of graphics operations and write results in the shared system memory and wherein the second VM comprises a destination VM configured to perform a second set of graphics operations using the results as input.

22. The method as in claim 21 wherein the graphics operations comprise graphics rendering operations.



PROCESSOR 200

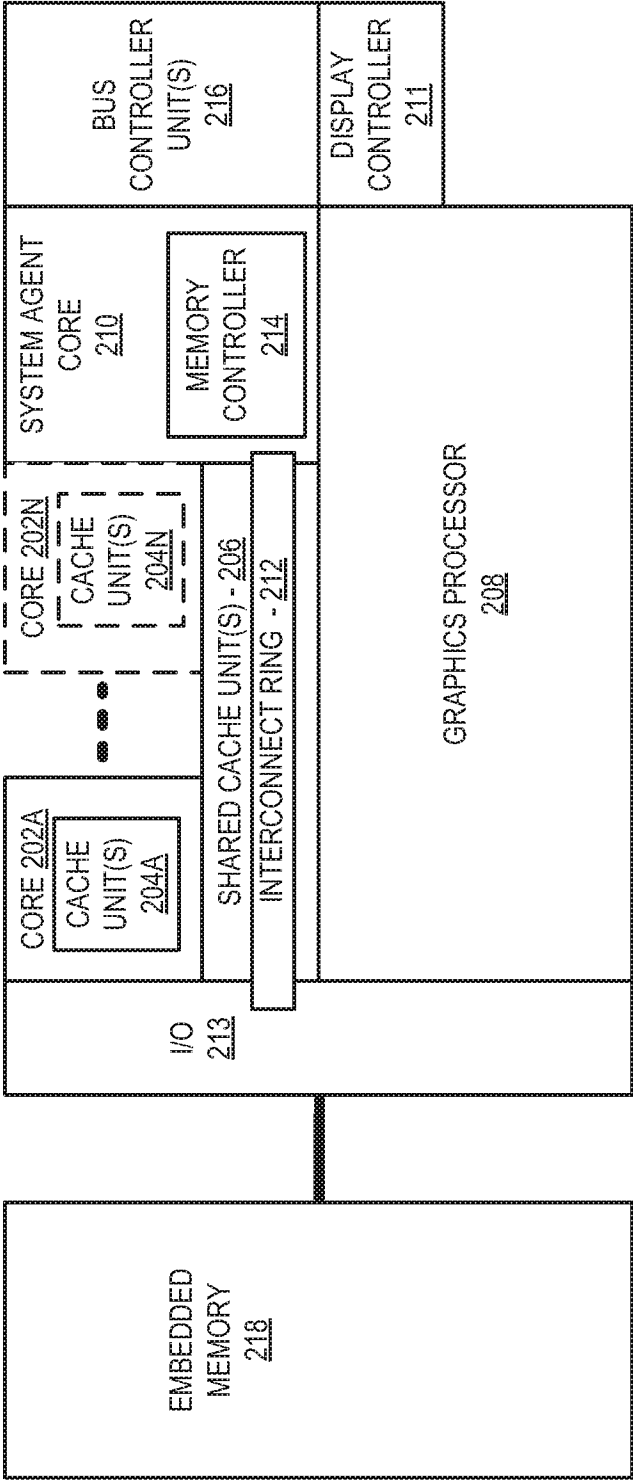


FIG. 2

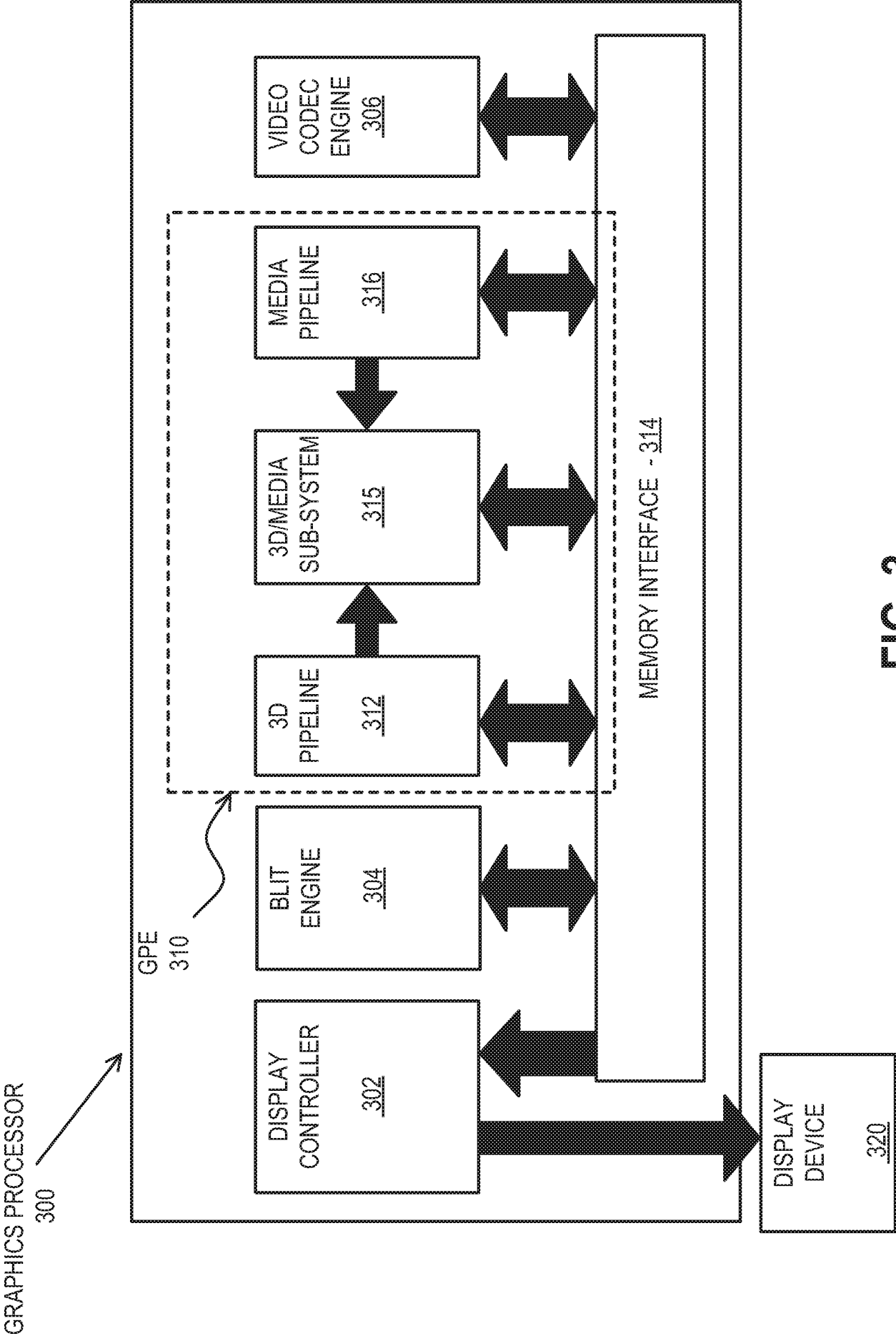


FIG. 3

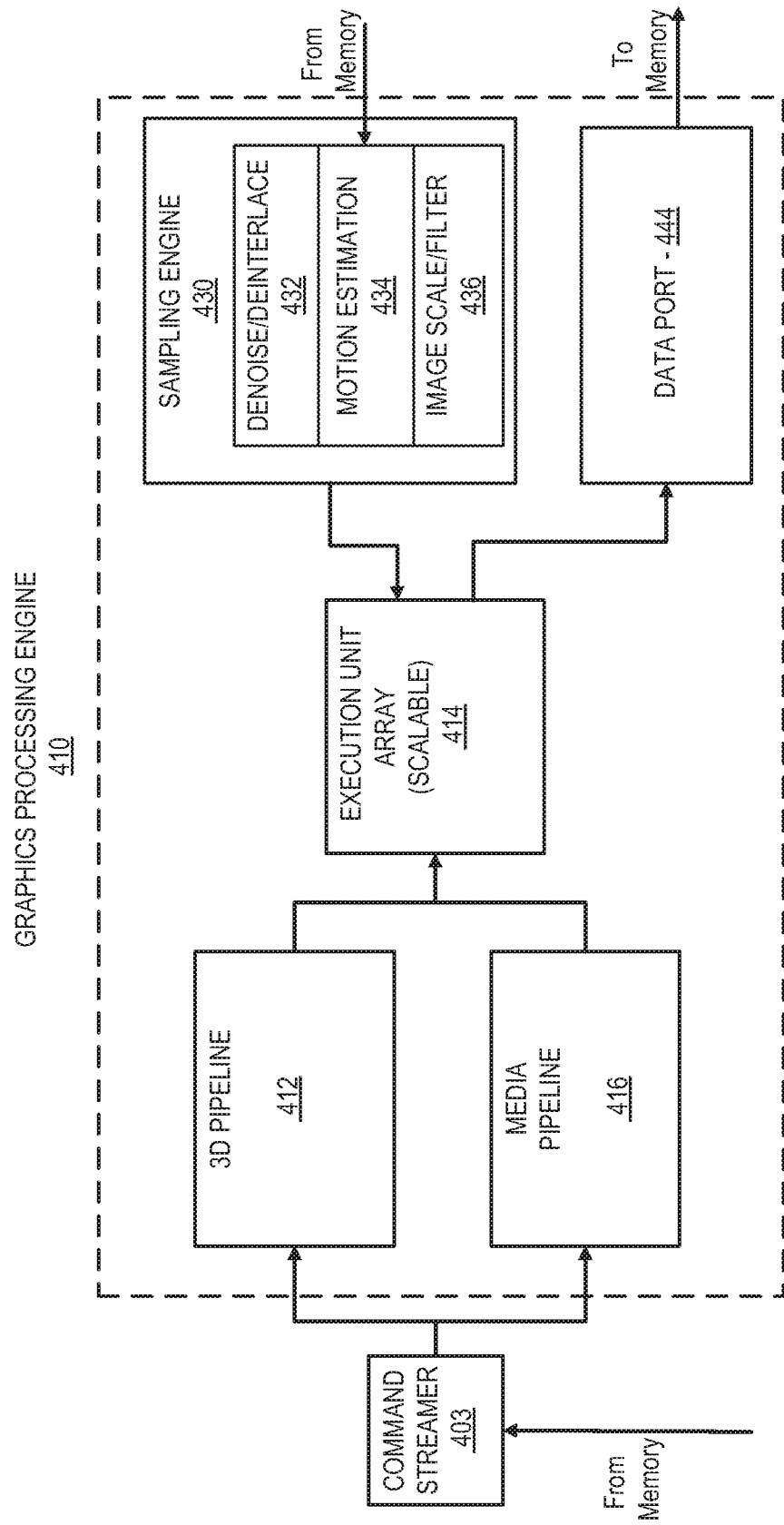


FIG. 4

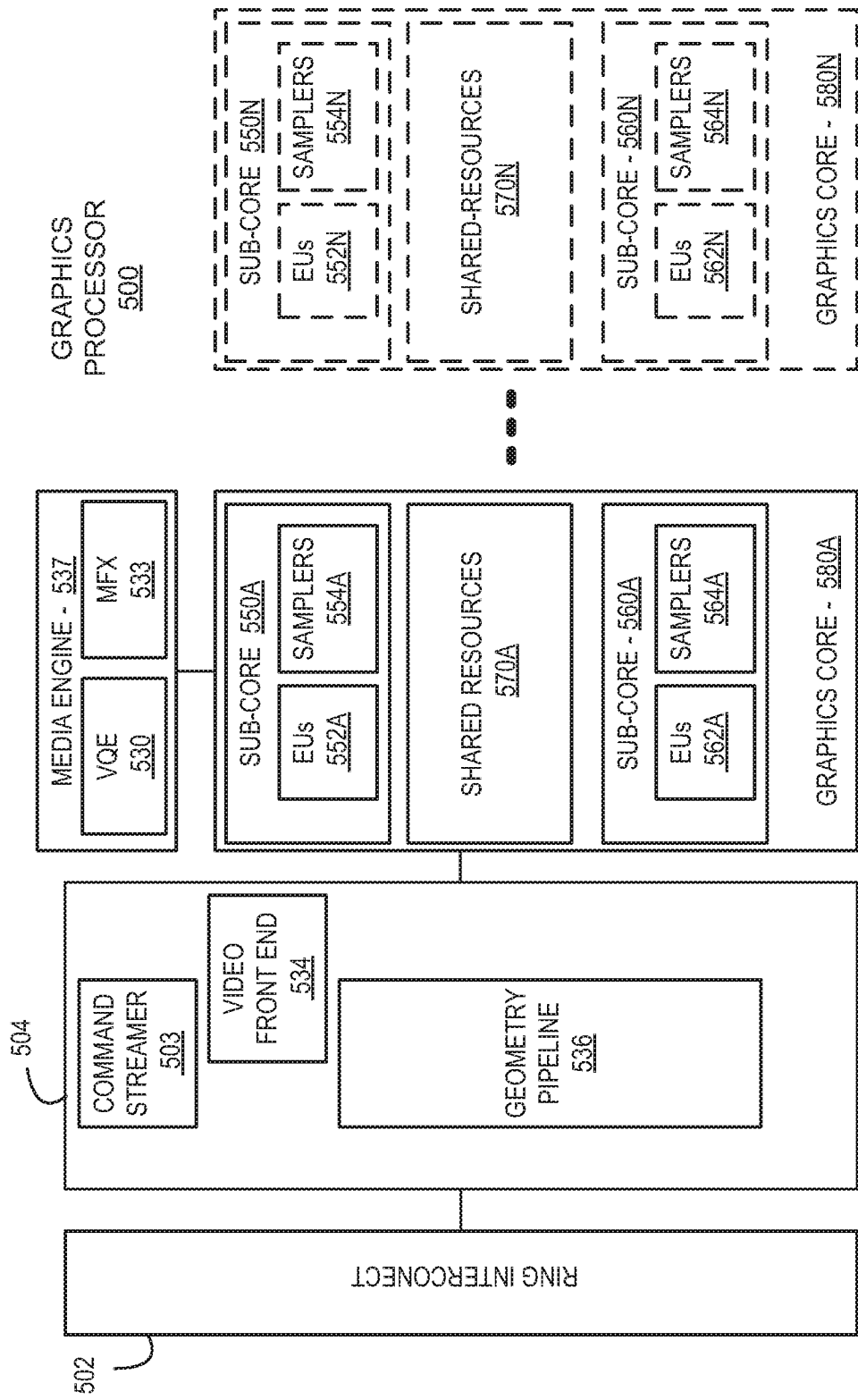


FIG. 5

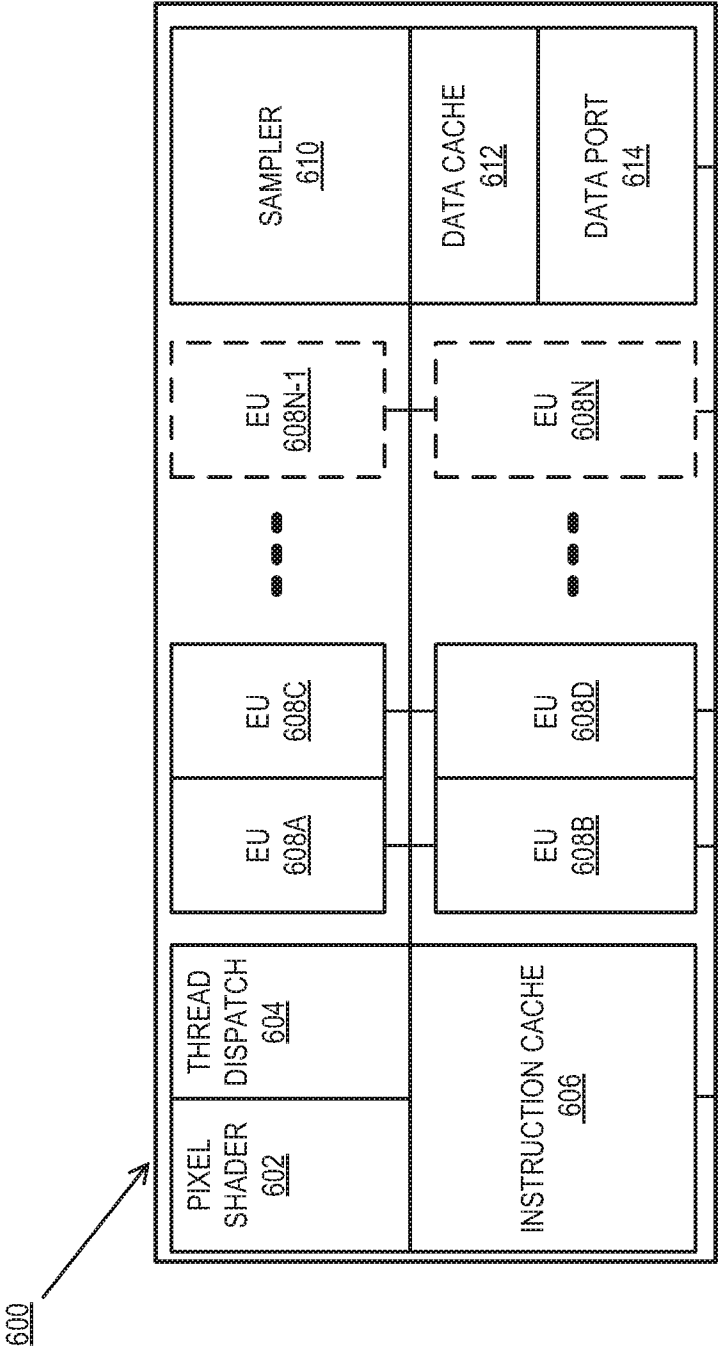


FIG. 6

GRAPHICS CORE INSTRUCTION
FORMATS

700

128-BIT INSTRUCTION

710



64-BIT COMPACT

INSTRUCTION

730



OPCODE DECODE

740

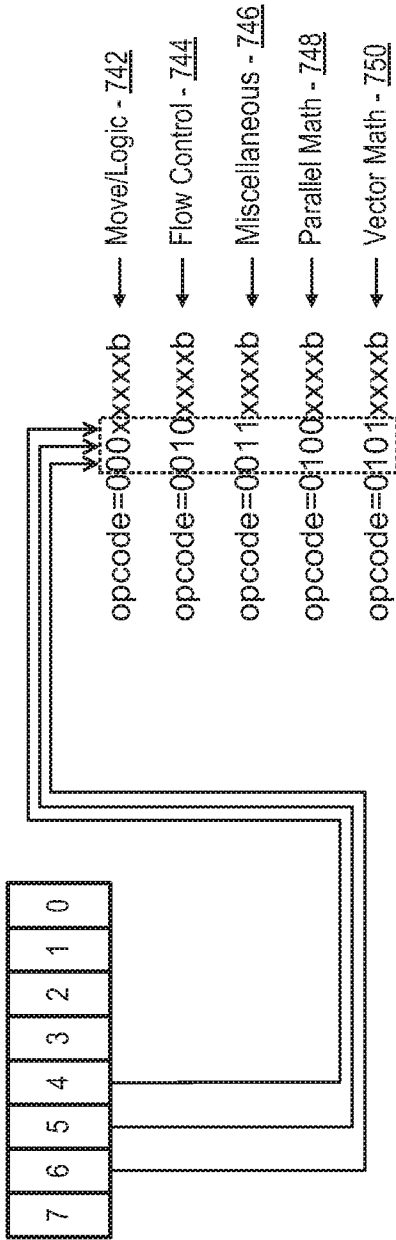


FIG. 7

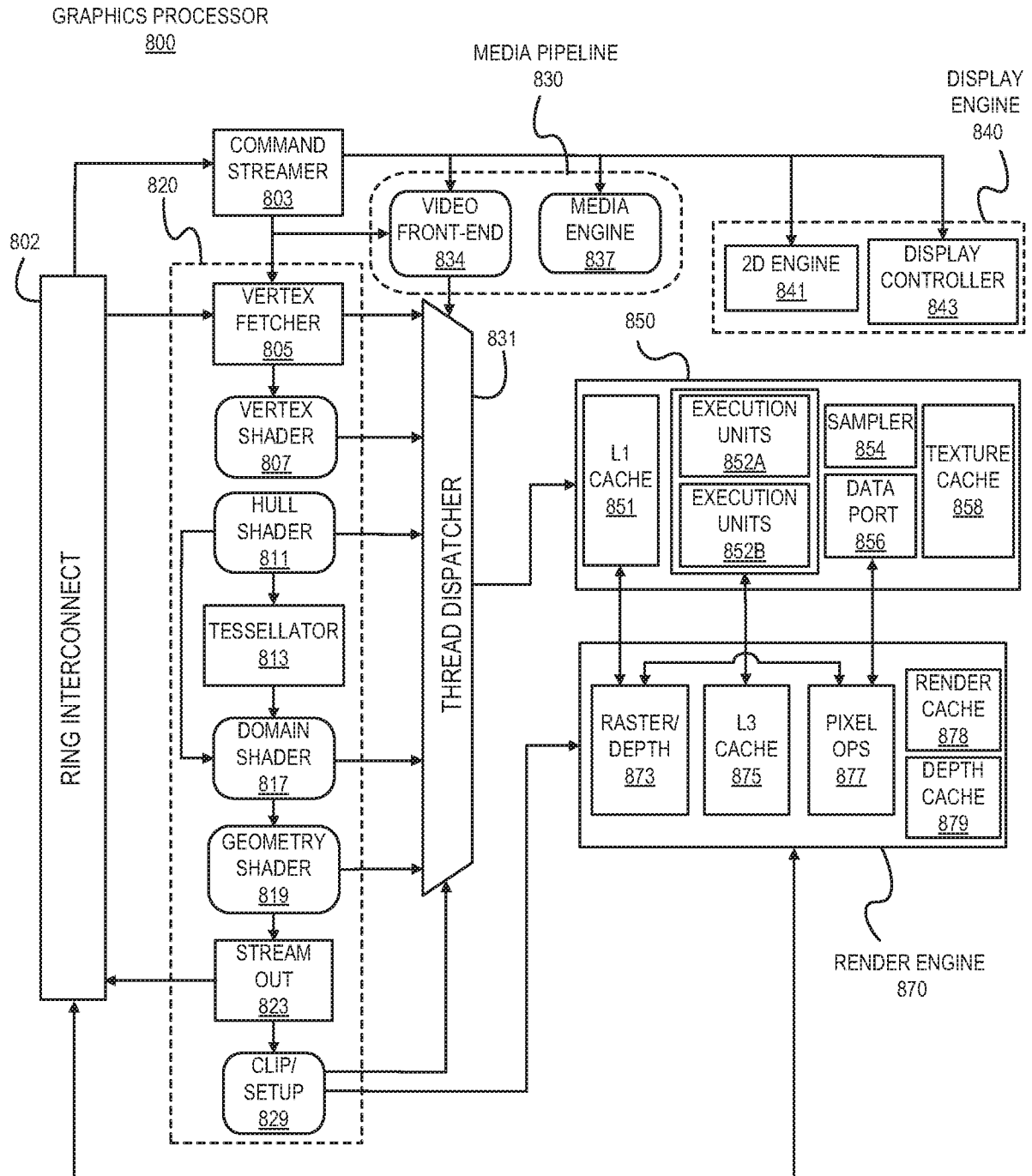
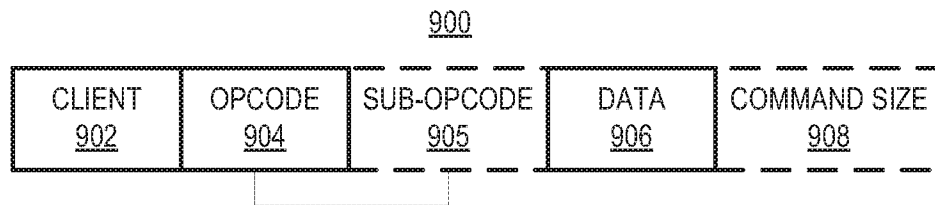
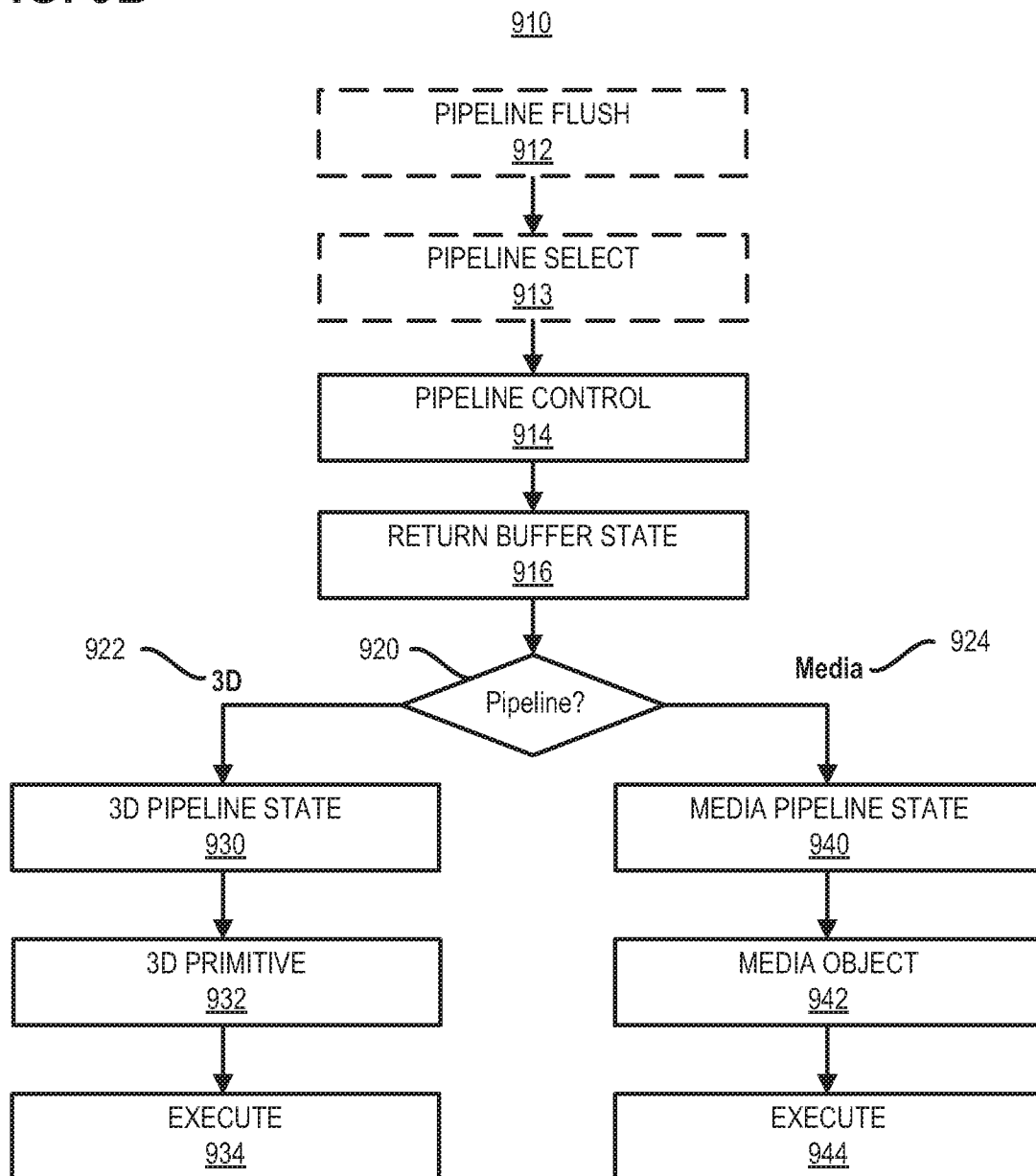
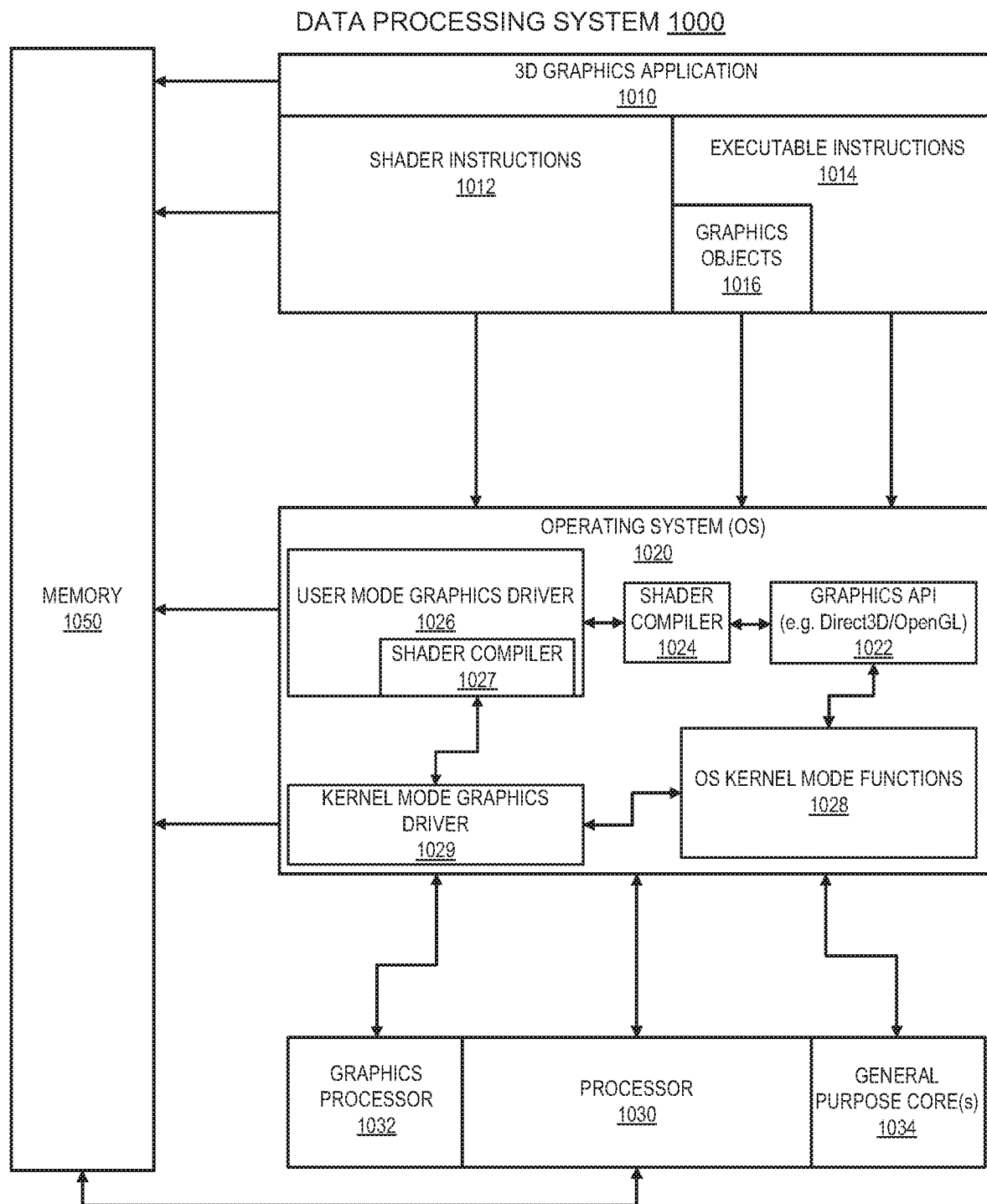


FIG. 8

FIG. 9A**GRAPHICS PROCESSOR COMMAND FORMAT****FIG. 9B****GRAPHICS PROCESSOR COMMAND SEQUENCE**

**FIG. 10**

IP CORE DEVELOPMENT - 1100

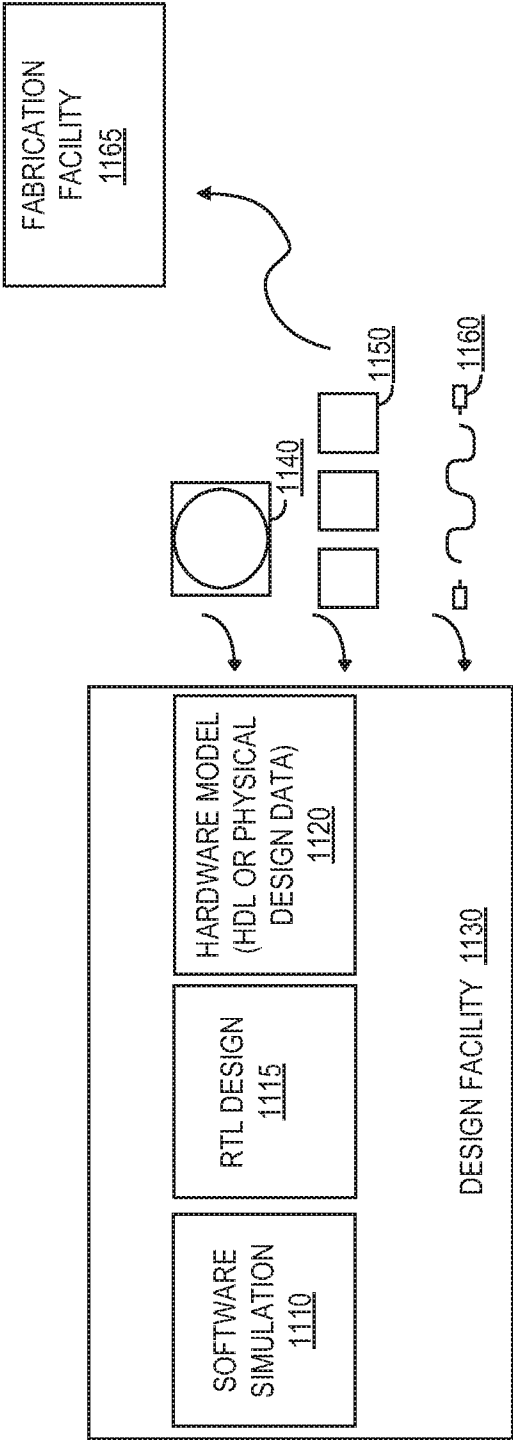
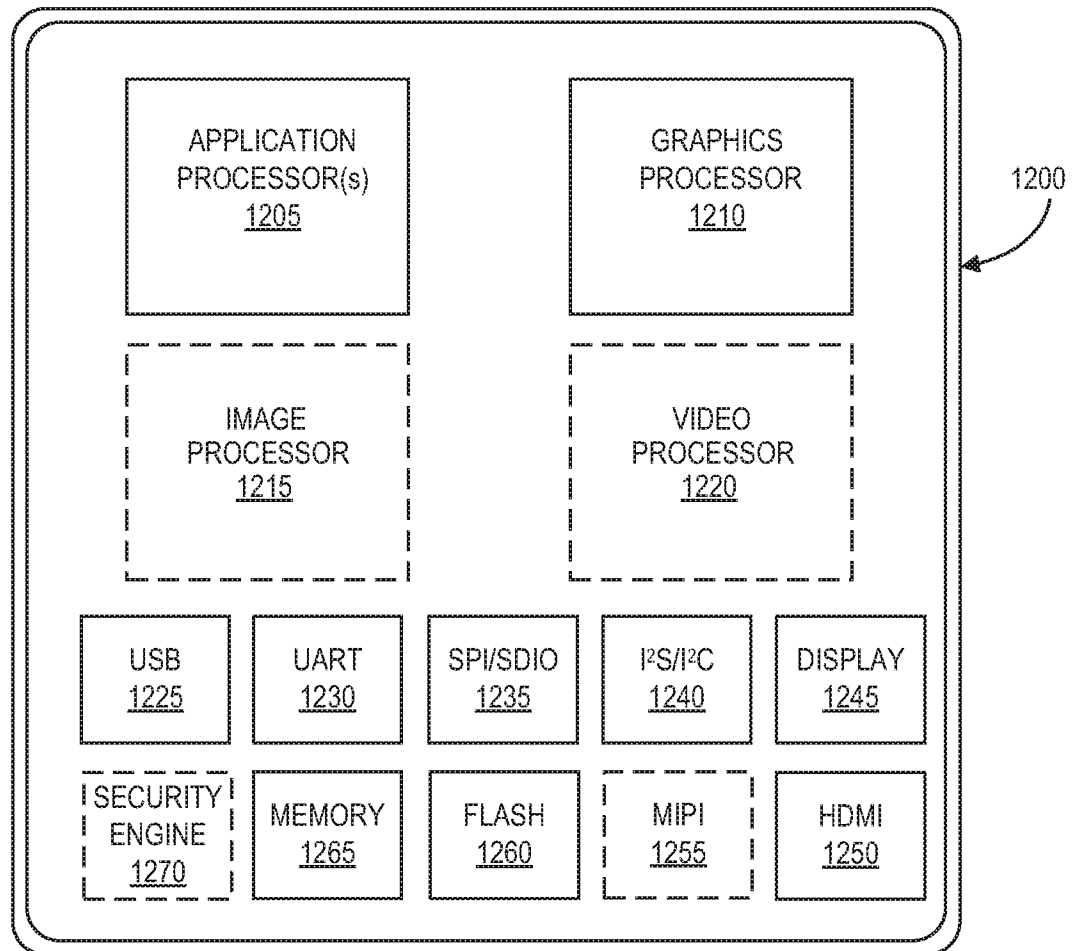
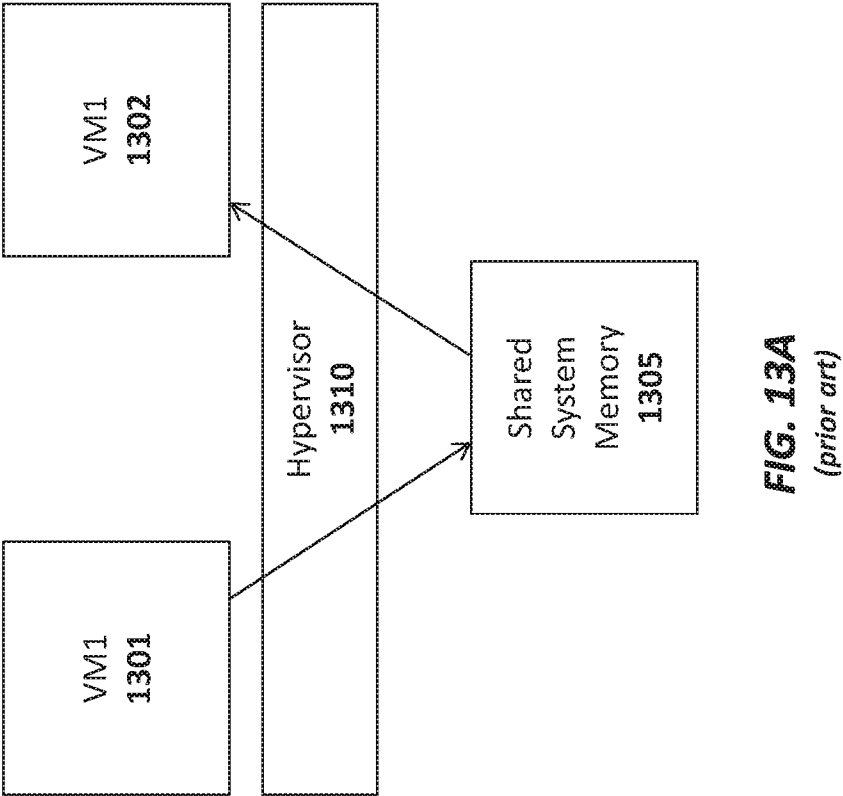


FIG. 11

**FIG. 12**



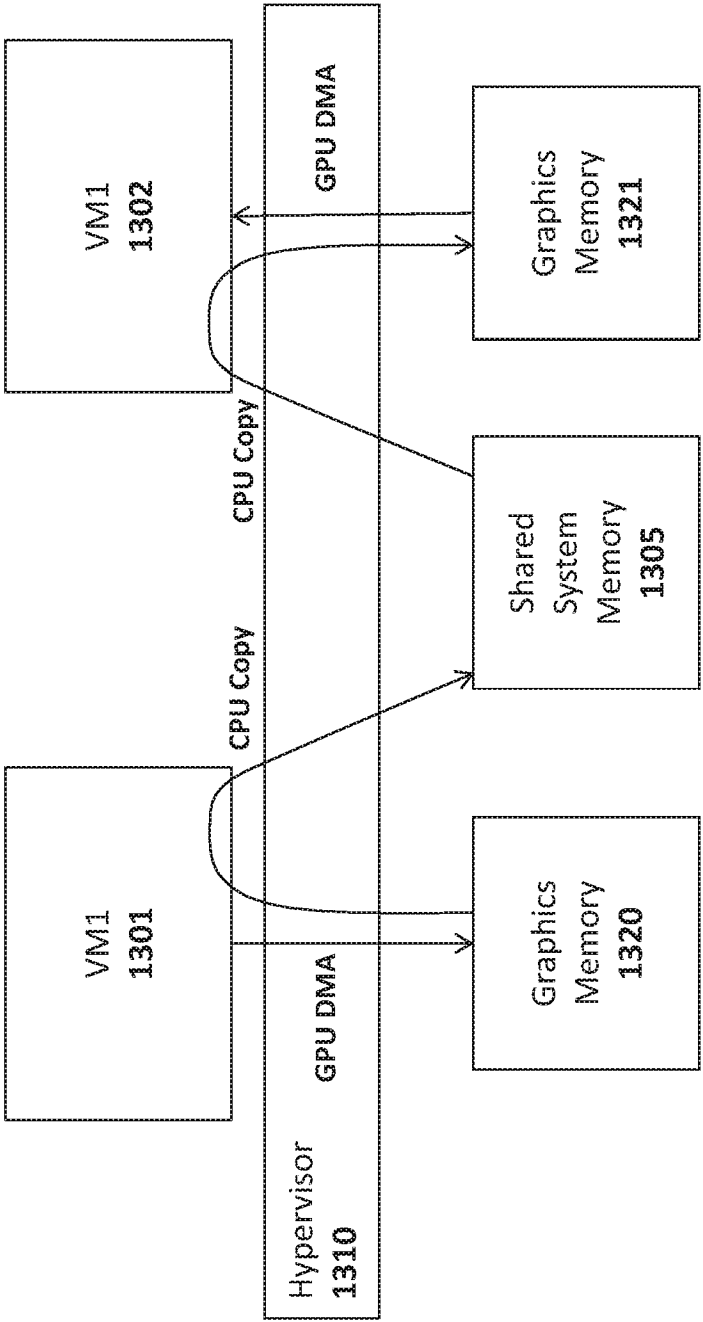


FIG. 13B
(prior art)

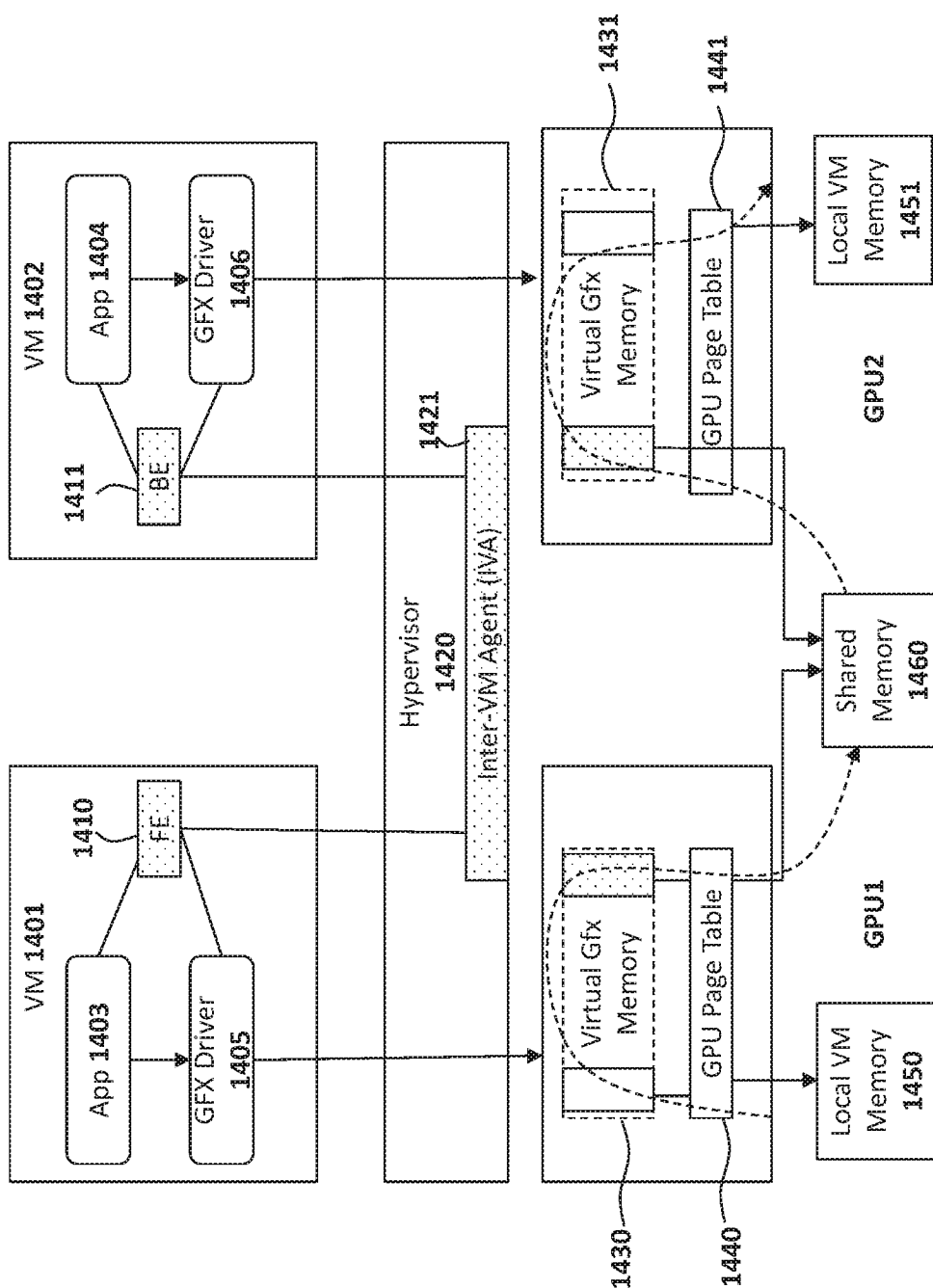


FIG. 14

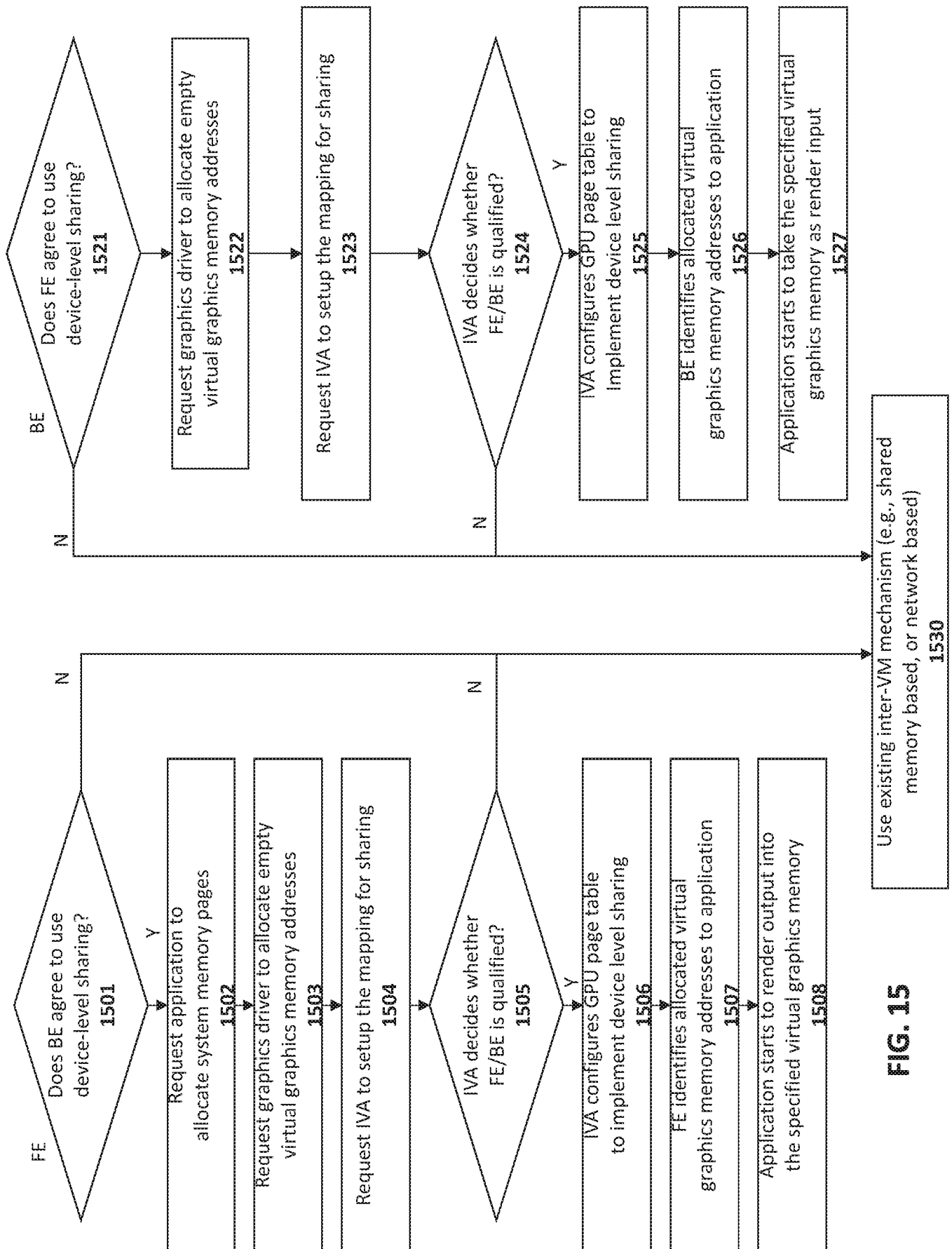


FIG. 15

INTERNATIONAL SEARCH REPORT

International application No.

PCT/CN2015/000463

A. CLASSIFICATION OF SUBJECT MATTER

G06F 12/08(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)

G06F

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

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

CNKI,CNPAT,WPI,EPODOC,IEEE: VM, virtual, machine, share, memory, address, map, GPU, graphics, page table

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2010217916 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 26 August 2010 (2010-08-26) description, paragraphs [0027]-[0029], [0048]-[0055] and figures 3-6	1-22
X	US 2005114855 A1 (BAUMBERGER, DANIEL P.) 26 May 2005 (2005-05-26) description, paragraphs [0017]-[0031] and figures 2-3	1-22
A	US 2012166705 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 28 June 2012 (2012-06-28) the whole document	1-22
A	CN 102707986 A (MICROSOFT CORPORATION) 03 October 2012 (2012-10-03) the whole document	1-22



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

17 February 2016

Date of mailing of the international search report

11 March 2016

Name and mailing address of the ISA/CN

STATE INTELLECTUAL PROPERTY OFFICE OF THE
P.R.CHINA
6, Xitucheng Rd., Jimen Bridge, Haidian District, Beijing
100088, China

Authorized officer

SU, Wen

Facsimile No. (86-10)62019451

Telephone No. (86-10)010-62413821

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.

PCT/CN2015/000463

Patent document cited in search report			Publication date (day/month/year)	Patent family member(s)			Publication date (day/month/year)
US	2010217916	A1	26 August 2010	CN	101819564	A	01 September 2010
US	2005114855	A1	26 May 2005	None			
US	2012166705	A1	28 June 2012	US	2012216185	A1	23 August 2012
				CN	102541619	A	04 July 2012
CN	102707986	A	03 October 2012	US	2012084517	A1	05 April 2012
				EP	2622461	A2	07 August 2013
				WO	2012050724	A2	19 April 2012
				HK	1175002	A0	21 June 2013