

(51) International Patent Classification:  
G06F 9/50 (2006.01)(21) International Application Number:  
PCT/CN2015/090571(22) International Filing Date:  
24 September 2015 (24.09.2015)

(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): ZHENG, Xiao [CN/CN]; No. 880 Zi Xing Road, Zizhu Science Park, MinHang, Shanghai 200241 (CN). DONG, Yao Zu [CN/CN]; Apt. 101 (1009), Building #5, Yanping Road 123, Shanghai 200042 (CN). ZHANG, Yulei [CN/CN]; No. 880 Zi Xing Road, Zizhu Science Park, Minhang, Shanghai 200241 (CN).

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

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

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

Published:

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

(54) Title: APPARATUS AND METHOD FOR PATTERN DRIVEN SELF-ADAPTIVE VIRTUAL GRAPHICS PROCESSOR UNITS

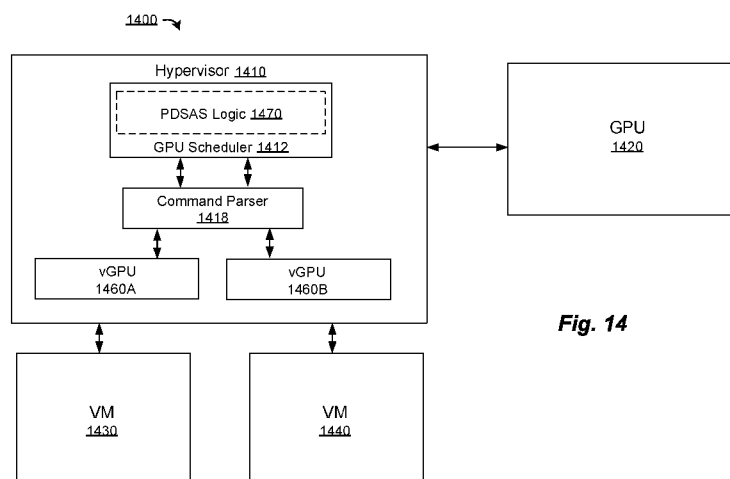


Fig. 14

(57) Abstract: An apparatus and method are described for a pattern driven self-adaptive graphics processing unit (GPU). The apparatus comprises: a graphics processing unit (1420) to process graphics commands and responsively render a plurality of image frames; a hypervisor (1410) to virtualize the GPU to share the GPU among a plurality of virtual machines (VMs) (1430, 1440), the hypervisor subdividing GPU processing for each VM into a plurality of quanta; and scheduling logic (1412) to monitor GPU utilization including GPU busy time and/or GPU idle time for each VM during its allocated quantum and to store utilization data reflecting the GPU utilization during each quantum; the scheduling logic to predict a cost of yielding to a second VM, the scheduling logic to yield the GPU to the second VM if the cost of waiting is greater than the cost of yielding to the second VM.

# APPARATUS AND METHOD FOR PATTERN DRIVEN SELF-ADAPTIVE VIRTUAL GRAPHICS PROCESSOR UNITS

## 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 pattern driven self-adaptive virtual graphics processor units (vGPUs).

### Description of the Related Art

[0002] Various approaches have been developed for graphics virtualization. For example, one approach allows direct assignment of an entire GPU's power to a single user, passing native driver capabilities through the hypervisor without any limitations. A common nomenclature used for this versopm of graphics virtualization is "Direct Graphics Adaptor" (vDGA). Another virtualization approach requires a virtual graphics driver in a virtual machine and uses an API forwarding technique to interface with the graphics hardware. In a more recent approach of sharing a GPU among many concurrent users, each virtual desktop machine maintains a copy of the native graphics driver. On a time sliced basis, an agent in the hypervisor directly assigns the full GPU resource to each virtual machine. Thus, during its time slice, while the virtual machine receives a full dedicated GPU, from the overall system viewpoint several virtual machines share a single GPU.

## BRIEF DESCRIPTION OF THE DRAWINGS

[0003] A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

[0004] **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;

[0005] **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;

[0006] **FIG. 3** is a block diagram of one embodiment of a graphics processor which may be a discreet graphics processing unit, or may be graphics processor integrated with a plurality of processing cores;

- [0007] **FIG. 4** is a block diagram of an embodiment of a graphics-processing engine for a graphics processor;
- [0008] **FIG. 5** is a block diagram of another embodiment of a graphics processor;
- [0009] **FIG. 6** is a block diagram of thread execution logic including an array of processing elements;
- [0010] **FIG. 7** illustrates a graphics processor execution unit instruction format according to an embodiment;
- [0011] **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;
- [0012] **FIG. 9A** is a block diagram illustrating a graphics processor command format according to an embodiment;
- [0013] **FIG. 9B** is a block diagram illustrating a graphics processor command sequence according to an embodiment;
- [0014] **FIG. 10** illustrates exemplary graphics software architecture for a data processing system according to an embodiment;
- [0015] **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;
- [0016] **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;
- [0017] **FIG. 13** illustrates exemplary GPU resource utilization under a round-robin scheduler;
- [0018] **FIG. 14** illustrates one embodiment of a GPU scheduler which implements a pattern driven self-adaptive scheme (PDSAS);
- [0019] **FIG. 15** illustrates an exemplary possibility curve of idleness from an exemplary 3D workload;
- [0020] **FIG. 16** illustrates an exemplary distribution curve of possible wait times;
- [0021] **FIGS. 17A-B** illustrate a method in accordance with one embodiment of the invention;
- [0022] **FIG. 18** illustrates the operation of one embodiment of the invention for different time values, switch cost values, and threshold values; and
- [0023] **FIG. 19** illustrates an exemplary probability cure indicating probabilities at different time values and including a balance coefficient.

## DETAILED DESCRIPTION

[0024] 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

#### [0025] System Overview

[0026] **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.

[0027] 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.

[0028] 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).

**[0029]** 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.

**[0030]** 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.

**[0031]** 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.

**[0032]** 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.

**[0033]** **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.

**[0034]** 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.

**[0035]** 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).

**[0036]** 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.

**[0037]** 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.

**[0038]** 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.

**[0039]** 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.

**[0040]** 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.

**[0041]** **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.

**[0042]** 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.

**[0043]** 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.

**[0044]** 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 subsystem 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.

[0045] 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.

[0046] 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.

[0047] **3D/Media Processing**

[0048] **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.

[0049] 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.

**[0050]** 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.

**[0051]** 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).

**[0052]** 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.

[0053] 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.

[0054] 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.

[0055] **Execution Units**

[0056] **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.

[0057] 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.

[0058] 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.

[0059] 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.

[0060] **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.

[0061] 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.

**[0062]** 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).

**[0063]** 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.

**[0064]** 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.

**[0065]** 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.

**[0066]** 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.

**[0067]** 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.

**[0068]** 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.

**[0069]** **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.

**[0070]** 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.

**[0071]** 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.

**[0072]** 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.

**[0073]** 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.

**[0074]** 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.

**[0075]** 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.

**[0076]** 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.

**[0077]** **Graphics Pipeline**

[0078] **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.

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

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

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

[0082] In some embodiments, graphics pipeline 820 includes tessellation components to perform hardware-accelerated tessellation of 3D objects. In some embodiments, a programmable hull shader 811 configures the tessellation operations. A programmable domain shader 817 provides back-end evaluation of tessellation output. A tessellator 813 operates at the direction of hull shader 811 and contains special purpose logic to generate a set of detailed geometric objects based on a coarse

geometric model that is provided as input to graphics pipeline 820. In some embodiments, if tessellation is not used, tessellation components 811, 813, 817 can be bypassed.

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

**[0084]** Before rasterization, a clipper 829 processes vertex data. The clipper 829 may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In some embodiments, a rasterizer 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.

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

**[0086]** In some embodiments, render output pipeline 870 contains a rasterizer and depth test component 873 that converts vertex-based objects into an associated pixel-based representation. In some embodiments, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache 878 and depth cache 879 are also available in some embodiments. A pixel operations component 877 performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine 841, or

substituted at display time by the display controller 843 using overlay display planes. In some embodiments, a shared L3 cache 875 is available to all graphics components, allowing the sharing of data without the use of main system memory.

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

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

[0089] In some embodiments, graphics pipeline 820 and media pipeline 830 are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL) and Open Computing Language (OpenCL) from the Khronos Group, the Direct3D library from the Microsoft Corporation, or support may be provided to both OpenGL and D3D. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

[0090] **Graphics Pipeline Programming**

[0091] **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.

[0092] 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.

[0093] 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.

[0094] 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.

**[0095]** 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 912 is required immediately before a pipeline switch via the pipeline select command 913.

**[0096]** 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.

**[0097]** 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.

**[0098]** 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.

**[0099]** 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.

**[00100]** 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.

**[00101]** 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.

**[00102]** 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.

**[00103]** In some embodiments, media pipeline 924 is configured in a similar manner as the 3D pipeline 922. A set of media pipeline state commands 940 are

dispatched or placed into in a command queue before the media object commands 942. In some embodiments, media pipeline state commands 940 include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. In some embodiments, media pipeline state commands 940 also support the use one or more pointers to “indirect” state elements that contain a batch of state settings.

**[00104]** 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.

**[00105]      Graphics Software Architecture**

**[00106]** **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.

**[00107]** 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.

**[00108]** 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.

[00109] 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.

[00110] **IP Core Implementations**

[00111] 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.

[00112] **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.

**[00113]** 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 3<sup>rd</sup> 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.

**[00114]** **Figure 12** is a block diagram illustrating an exemplary system on a chip integrated circuit 1200 that may be fabricated using one or more IP cores, according to an embodiment. The exemplary integrated circuit includes one or more application processors 1205 (e.g., CPUs), at least one graphics processor 1210, and may additionally include an image processor 1215 and/or a video processor 1220, any of which may be a modular IP core from the same or multiple different design facilities. The integrated circuit includes peripheral or bus logic including a USB controller 1225, UART controller 1230, an SPI/SDIO controller 1235, and an I<sup>2</sup>S/I<sup>2</sup>C 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.

[00115] 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 PATTERN DRIVEN,  
SELF-ADAPTIVE VIRTUAL GRAPHICS PROCESSOR UNITS

[00116] In current graphics processor unit (GPU) virtualization implementations, the virtual machine monitor (VMM), particularly the virtual GPU (vGPU) device model, traps and emulates the guest access to privileged GPU resources for security and multiplexing, while passing through CPU access of performance-critical resources (e.g., such as CPU access to graphics memory). GPU commands, once submitted, are directly executed (in the GPU) without VMM intervention. As a result, close to native performance is achieved.

[00117] In architectures where the GPU executes workloads from multiple vGPU Guests, switching from one of vGPU Guest to the next vGPU Guest will cause a GPU context switch and the required hardware level ring engines context to be saved/restored. Unlike the CPU, whose context is very lightweight, the GPU context is very heavy. Consequently, the GPU content switch takes significantly more time than a CPU context switch. Profiling data shows that a GPU context switch may take 100~300 microseconds (us), while a CPU context switch may take just a few hundred nanoseconds (ns).

[00118] Therefore, the GPU scheduler policy will not trigger context switching as frequently as in the CPU. Typically, it occurs every several milliseconds (ms). A typical GPU scheduler provides a quantum-based policy to schedule the vGPU instances periodically. Note that the period here may be fixed or weighted or even variable per additional policies. Each vGPU, once scheduled in, is provided a quantum in which to execute the commands (CMDs) prepared by the CPU until the quantum is consumed, and/or until the vGPU is blocked (such as due to a semaphore, waiting for events, etc). While this policy performs well on the CPU side, it poses additional challenges on the GPU side, because the execution of the vGPU instance relies on the available CMDs prepared by CPU. That is to say, a vGPU instance (VM) may have the quantum to run the GPU CMDs, but there may be no CMDs ready at a given point in time (i.e., the GPU has to wait for the CMDs from the CPU to be ready, which may take time). Therefore, the GPU may remain in an idle status, until the next CMD is available, during which the GPU cycles are wasted. **Figure 13** illustrates an example case where a GPU utilization

rate of VM1 is around 20%, during which approximately 80% of the GPU resources are wasted waiting for the next available commands.

[00119] One option is to immediately schedule out the vGPU instance upon detecting that there are no available CMDs. However, because the context switch cost in the GPU is so high compared with the CPU, it may lead to frequent GPU context switches with very high switching cost. It is therefore a challenge for the GPU scheduler to determine the set of conditions in which to yield (schedule out) a vGPU instance without available CMDs.

[00120] The embodiments of the invention described herein implement a Pattern Driven, Self-Adaptive, Scheme (PDSAS) for efficient yielding. As illustrated in **Figure 14**, in one embodiment, the PDSAS logic 1470 is built within or on top of an existing GPU scheduler 1412 and, in fact, may be implemented with any existing GPU scheduler.

#### A. Virtual Graphics Processing Overview

[00121] Additional details of the exemplary embodiment 1400 shown in **Figure 14** will now be provided, followed (in Section B) by a detailed description of operations performed by PDSAS logic 1470. The illustrated embodiment includes multiple VMs, e.g., VM 1430 and VM 1440, managed by hypervisor 1410 (sometimes referred to as a virtual machine monitor (VMM)), which has access to a full array of GPU features in GPU 1420. A virtual GPU (vGPU) 1460A-B may access the full functionality provided by GPU hardware 1420 based on the GPU virtualization technology. In various embodiments, hypervisor 1410 may track, manage resources and lifecycles of one or more vGPUs. Although only two vGPUs 1460A-B are shown in **Figure 14**, hypervisor 1410 may include other vGPUs. In some embodiments, vGPUs may be interactive with native GPU drivers. VM 1430 or VM 1440 may then access the full array of GPU features through vGPUs 1460A-B. The vGPU context may be switched per quantum or event. In some embodiments, the context switch may happen per GPU render engine (e.g., a 3D render engine or blitter render engine). The periodic switching allows multiple VMs to share a physical GPU 1420 in a manner that is transparent to the workloads of the VMs 1430, 1440.

[00122] Much like a single central processing unit (CPU) core may be assigned for a limited time by VM 1430, GPU 1420 may also be assigned for a limited time by VM 1430. Another virtualization model is timesharing, where GPU 1420 or portions of it may be shared by multiple VMs, e.g., VM 1430 and VM 1440, in a multiplexing fashion. Other GPU virtualization models may also be used in other embodiments. In various

embodiments, graphics memory associated with GPU 1420 may be partitioned, and allotted to various vGPUs in hypervisor 1410.

**[00123]** In various embodiments, graphics translation tables (GTTs) may be used by VMs and/or GPU 1420 to map graphics processor memory to system memory or to translate GPU virtual addresses to physical addresses. In some embodiments, hypervisor 1410 may manage graphics memory mapping via shadow GTTs, and the shadow GTTs may be held in a vGPU instance, e.g., vGPU 1460A-B. In various embodiments, each VM may have a corresponding shadow GTT to hold the mapping between graphics memory addresses and physical memory addresses. In some embodiments, the shadow GTT may be shared and maintain the mappings for multiple VMs. In some embodiments, each VM (e.g., VM 1430 and VM 1440), may include both per-process and global GTTs.

**[00124]** Some embodiments may use system memory as graphics memory. System memory may be mapped into multiple virtual address spaces by GPU page tables. Different embodiments may support global graphics memory space and per-process graphics memory address space. The global graphics memory space may be a virtual address space, e.g., 2GB, mapped through a global graphics translation table (GGTT). The lower portion of this address space may be referred to as the “aperture,” accessible by both the GPU 1420 and the CPU (not shown). The upper portion of this address space is called high graphics memory space or hidden graphics memory space, which may be accessed by GPU 1420 only. In various embodiments, shadow global graphics translation tables (SGGTTs) may be used by VM 1430, VM 1440, hypervisor 1410, or GPU 1420 for translating graphics memory addresses to respective system memory addresses based on a global memory address space.

**[00125]** Hypervisor 1410 may use command parser 1418 to detect the potential memory working set of a GPU rendering engine for the commands submitted by VM 1430 or VM 1440. In various embodiments, VM 1430 may have respective command buffers (not shown) to hold commands from different workloads such as 3D workloads and media workloads. Similarly, VM 1440 may have respective command buffers (not shown) to hold commands from these different workloads.

**[00126]** In various embodiments, command parser 1418 may scan a command from a VM and determine if the command contains memory operands. If it does, the command parser 1418 may read the related graphics memory space mappings, e.g., from a GTT for the VM, and then write it into a workload specific portion of the SGGTT. After the whole command buffer of a workload gets scanned, the SGGTT that holds

memory address space mappings associated with this workload may be generated or updated. Additionally, by scanning the to-be-executed commands from VM 1430 or VM 1440, command parser 1418 may also improve the security of GPU operations, such as by mitigating malicious operations.

[00127] In some embodiments, one SGGTT may be generated to hold translations for all workloads from all VMs. In some embodiments, one SGGTT may be generated to hold translations for all workloads, e.g., from one VM only. The workload specific SGGTT portion may be constructed on demand by command parser 1418 to hold the translations for a specific workload (e.g., a 3D workload from VM 1430 or a media workload from VM 1440).

[00128]

#### B. Pattern Driven, Self-Adaptive, Scheme (PDSAS) for Efficient Yielding

[00129] As mentioned, embodiments of the invention implement a Pattern Driven, Self-Adaptive, Scheme (PDSAS) for efficient yielding. As illustrated in **Figure 14**, in one embodiment, the PDSAS logic 1470 is built within or on top of an existing GPU scheduler 1412 and, in fact, may be implemented with any existing GPU scheduler.

[00130] One embodiment of the invention monitors activity and generates a profile pattern of GPU execution status during runtime, including how long the GPU remains in an idle status for a given VM. In one embodiment, this is accomplished by determining the difference from the time when last CMDs are completed until the time when a new CMD is available. These past execution patterns of the VM may then be used to predict the behavior of the VM, and extend the GPU scheduler 1412 to make better decisions on whether to yield the GPU or not, thereby improving efficiency.

[00131] To perform these operations, one embodiment of the PDSAS logic 1470 stamps the execution of a batch of GPU commands per VM, to determine the pattern of how the GPU resources are used. Particularly, the PDSAS logic 1470 gathers statistics on the time duration from the point when the GPU completes the execution of previously submitted CMDs (likely in batch format) where there is initially no available CMDs to submit, to the point when the new CMDs are available.

[00132] In one embodiment, the profiling data includes GPU busy time and/or GPU idle time for each VM, as illustrated in **Figure 13**. In this example, each VM is provided with a quantum of 15ms. The GPU resource utilization of VM1 within its quantum is approximately 20% (i.e., with significant idle periods) and the GPU resource utilization of VM2 within its quantum is approximately 80% (with significantly fewer idle periods).

[00133] The CPU scheduler and GPU scheduler may be agnostic. A VM, at one time, may have four scheduling possibilities: 1) be scheduled-in by the CPU scheduler only, 2) be scheduled-in by the GPU scheduler only, 3) be scheduled-in by both the CPU & GPU schedulers only, 4) Not scheduled by either the GPU or CPU scheduler.

[00134] In one embodiment, the PDSAS logic 1470 then performs an analysis of the determined utilization pattern to arrive at a distribution curve of possible waiting times (i.e., GPU idle occurrences per VM). One embodiment applies the function  $P(t)$  where  $t$  is the waiting duration, and  $P(t)$  is the probability (e.g., 0.01%). One example of such a probability distribution curve for  $P(t)$  is illustrated in **Figure 15** which highlights points on the curve for the average time ( $T_{\text{average}}$ ) and  $T_{80\%}$  (i.e., meaning that there is an 80% probability that the next CMDs will arrive within the designated time period). This example is derived from a 3Dmark workload.

[00135] Once the probability curve is determined, the PDSAS logic 1470 can calculate and predict the cost of waiting ("W-Cost") under a certain probability. In one embodiment, the PDSAS logic 1470 simply performs its evaluation using the average waiting time  $T_{\text{average}}$ :  $\text{W-Cost} = T_{\text{average}}$  where  $T_{\text{average}}$  is calculated as:

$$50\% = \int_0^{T_{\text{average}}} P(t) dt$$

[00136] In one embodiment, as the data is sampled within a small time interval such as 10ns (0.01ms), the above formula becomes a discrete function as below:

$$50\% = \sum_{n=0}^{n=N} P(n)$$

$$N = T_{\text{average}}/0.01$$

[00137] Of course, other values for W-Cost may also be used. For example, in one embodiment, W-Cost is set to the maximum possibility waiting duration for the next commands ( $T_0$ ) as indicated in **Figure 16**. In one embodiment, W-Cost is a variable which is dynamically adjusted during runtime in accordance with a specified policy (some examples of which are discussed below).

[00138] The sampling techniques described above may last a certain time such as 10 minutes, for example. In one embodiment, the pattern in the sample period may be equally used and/or different weights may be used to distinguish between the most current data and the old data (e.g., with more weight being applied to the more current data). For example, the latest 1 minute of sampling data may be considered to have the best heuristics and, as such, may have a weight of 80%, while the data for the

remaining 9 minutes may have a weight of 20%. In such an implementation, Final  $P(t)$  =  $P_a(t) * 0.8 + P_b(t) * 0.2$  where  $P(t)$  is the distribution curve of possible wait time. Of course, the underlying principles of the invention are not limited to any particular manner of determining weights and/or time periods within which to apply the weights.

**[00139]** One embodiment of the PDSAS logic 1470 enhances the GPU scheduler 1412 by predicting the cost and gain of yielding the vGPU instance. For example, the PDSAS logic 1470 may perform the above evaluation and then compare the gain and loss for the two approaches. In one embodiment, if the cost of waiting is greater than or equal to the cost of switching, then the PDSAS logic 1470 yields the vGPU. That is:

If  $W\text{-Cost} \geq SW\text{-Cost (switch cost)} + \text{Thres0}$ , then yield the vGPU.

As mentioned, the  $W\text{-Cost}$  is the time waiting with the GPU idle if the GPU is not yielded. The  $SW\text{-Cost}$  is the switch cost, and  $\text{Thres0}$  is a minimal bar of gain to yield the GPU. As mentioned, in one embodiment, the  $W\text{-Cost}$  is the average cost to wait for the vGPU to get the new CMD to run, and  $SW\text{-Cost}$  is the cost of a vGPU context switch (e.g., 0.3ms), and  $\text{Thres0}$  is a threshold value which may be set to keep the policy flexible and conservative.

**[00140]** In one embodiment, with the PDSAS logic 1470 implementing the above scheme, the GPU utilization is improved and therefore the total system GPU throughput is improved. However, it may introduce the unbalanced GPU resource allocation between VMs in some cases. That is, a particular VM may be allocated with fewer resources than it should be allocated even in a long time point of view. In one embodiment, this issue is addressed by accounting for the usage of each GPU for each VM, and extending the scheduler to raise the priority of the VM, which will then have more chances to be the next VM to be scheduled in, and/or be given longer quantum if it is scheduled in and has CMDs to execute. Different scheduler policies may be used, such as a priority-based scheduler, or round-robin scheduler or any combination thereof. The approach described above can be used to make a VM with lower resource allocation have a higher probability of being scheduled in on the next cycle.

**[00141]** A method in accordance with one embodiment of the invention is illustrated in **Figures 17A-B**. The method is subdivided into those operations performed by vCPU execution (**Figure 17A**) and those operations performed by the GPU scheduler (**Figure 17B**). At 1701, CPU of the VM begins execution of a GPU workload and, at 1702, the GPU workload generates a sequence of commands which are placed in a command buffer 1705. If within a sampling period, determined at 1703, the  $P(t)$  curve is updated and sampling is ended at 1704.

[00142] Turning to **Figure 17B**, at 1706, the GPU scheduler selects the next vGPU for execution (e.g., for a given quantum such as 15ms). If within the quantum, determined at 1707, then the vGPU is currently executing a workload at 1708. When vGPU execution is complete at 1709, a determination is made as to whether new commands are available from the command buffer at 1710. If so, then the process returns to 1707. If not, then sampling is started at 1711. If there is a pending workload in other VMs, determined at 1712, then the PDSAS logic is applied at 1713. If not, then the process returns to 1707 for the current vGPU instance. A determination is made at 1714 as to whether to yield to another vGPU instance (e.g., using the PDSAS techniques described herein). If so, then the process returns to 1706. If not, then the process returns to 1707 for the current vGPU instance.

[00143] Thus, some embodiments of the invention described herein define policies to calculate WCost compared with SW-Cost and Thres0. In some architectures, it has been determined that SW-cost is approximately 0.1~0.3ms, sampled at runtime. There are also several ways in which Thres0 may be selected. In one embodiment, Thres0 = 0. In another embodiment, Thres0 is a fixed value such as 0.2ms as a barrier to conservative scheduling (such as for an RT VM/vGPU), or -0.1ms as a positive factor to motivate scheduling. In yet another embodiment, the value is determined from the next vGPU's average GPU execution time (i.e., just as idleness may be determined for VMs, a busy state may also be determined for VMs). Here, for example, the next VM means next Guest scheduler will schedule in. For example, for the next vGPU with a 3D workload running, if the average GPU busy time is 0.5ms per each command buffer, then Thres0 may be set to 0.5ms. A VM would like to switch to the next VM with total switch cost = SW-Cost + 0.5ms = 0.8ms, which means it requires an average wait for 0.8ms until GPU ownership is switched again (i.e., for a 2 VM implementation). It should be noted, however, that the underlying principles of the invention are not limited to any particular manner of calculating Thres0.

[00144] Three exemplary policies will now be described (1)  $W\text{-Cost} = T_{\text{average}}$ , (2)  $W\text{-Cost} = T_{80\%}$ , and (3)  $W\text{-Cost} = (1-D) * T_{\text{average}}$  (where D is a balance coefficient as described below).

#### 1. $W\text{-Cost} = T_{\text{average}}$

[00145] In this embodiment, the cost is not yielding the GPU and waiting  $T_{\text{average}}$  time with the GPU idle. If the switch cost, SW-cost = 0.3, and Thred0 is the next vGPU's workload average execution time of 0.5ms, then the total cost is 0.8ms. In this policy, as long as the vGPU keeps the same workload, W-Cost may remain relatively

consistent, since a workload may have a fixed pattern. So if  $T_{\text{average}} > 0.8\text{ms}$ , the current vGPU may yield the GPU upon each CMDs completion.

[00146] As depicted in the flowchart in **Figure 17a-b**, the advantage of this policy is that we can maximize the usage of GPU cycles, but it may impact vGPU1 (the one yields). In other embodiment, the vGPU1 workload is high priority and should not be impacted even if we yield the GPU to vGPU2, as if the physical GPU can be scheduled back to vGPU1 in time (before the average wait time is due for example). This is extremely important for those high priority or realtime workloads (e.g., live streaming media, video conferencing, etc). The disadvantage is that it is always trying to keep the GPU for one VM or always choosing to yield the GPU on each cycle. The VM with small  $T_{\text{average}}$  and small Thred0 value (indicating it has busy workload) will then tend to occupy most of GPUs resources. It requires the scheduler's help to balance the GPU resources between each of the VMs.

## 2. W-Cost = $T_{80\%}$

[00147] As described above,  $T_{80\%}$  means 80% of the next commands will arrive within this duration. The below formula may be used to determine  $T_{80\%}$ . While the value of 80% is used in this exemplary embodiment, various other percentages may be used while still complying with the underlying principles of the invention (e.g., 90%, 95%, 99%,  $T_{\text{max}}$ , etc).

$$\int_0^{T_{80\%}} P(t)dt = 80\%$$

[00148] In one embodiment, this implementation has the same effect as described in the  $W\text{-Cost} = T_{\text{average}}$  policy above with the primary difference being that  $W\text{-Cost}$  may be a larger value if  $T_{80\%} > T_{\text{average}}$ . **Figure 18** illustrates an exemplary embodiment for  $W\text{-Cost} = T_{80\%}$  using the same VM1 and VM2 workloads as described above. As described in **Figure 18**, VM1 is more readily scheduled to VM2 due to a larger  $W\text{-Cost}$  value. As illustrated, the switching in 1801 is similar to that discussed above. However, the switching in 1802 is very different, as the GPU alternates processing CMDs from each of VM1 and VM2, given the larger  $W\text{-Cost}$  value. In one embodiment, the idle time in each CMD buffer may not be large enough and, as a result, the workload itself may be slightly impacted/delayed. Consequently, this implementation is more suitable for non-realtime workloads. However, as illustrated at 1802, this implementation may result in an improved overall GPU utilization rate.

[00149] Two other examples of  $W\text{-Cost}$  are  $W\text{-Cost} = \text{Max}$  and  $W\text{-Cost} = 0$ . In the first case,  $W\text{-Cost} = \text{Max}$ , the GPU will be yielded immediately. In the second case,  $W\text{-Cost} = 0$ , the GPU will be yielded immediately.

Cost=0, the GPU will not yield during the VM quantum.

[00150] **Figure 19** illustrates an exemplary probability curve indicating probabilities at different time values and including a balance coefficient.

### 3. **W-Cost = (1-D)\* T<sub>average</sub>**

[00151] In this embodiment, D is a balance coefficient between (-1.0 ~ 1.0). A negative D value means the PDSAS logic 1470 will more apt to yield GPU execution time. And a positive D value means the PDSAS logic 1470 will more apt to keep stay within its GPU quantum (e.g., 15ms). The D factor is added in order to influence in-balance GPU resource allocation among multiple VMs. D is adjusted in runtime according to each VM's GPU utilization rate. It is already know that Policy#1 or Policy#2 may cause some VM to take most of the GPU time without the scheduler's help.

[00152] The following is a 2 VM example. The coefficient D is calculated with  $D = 1 - VM\% * TotalVM$ . In this example, VM% means the current VM's allocated percentage of GPU time. TotalVM = 2 for a 2 VM guest case. If VM% is 30% (where it is supposed to be 50% for 2 VM case),  $D = 0.4 * T_{average}$ . W-cost =  $0.6 * T_{average}$ . Using these techniques, the VM (who got 30% of GPU time in past) is apt to allocate more GPU resources due to the small W-cost introduced.

[00153] It should be noted that the techniques described above for predicting idleness for the next CMDs may also be used to predict when interrupts arrive. In this implementation, the IRQ handler may wait for that time to combine the incoming interrupts, processing them together. This may essentially reduce the total trap numbers injected to Guest VMs.

[00154] In this detailed description, reference is made to the accompanying drawings, which form a part hereof, wherein like numerals designate like parts throughout, and in which is shown by way of illustration embodiments that may be practiced. It is to be understood that other embodiments may be utilized and structural or logical changes may be made without departing from the scope of the present disclosure. Therefore, the following detailed description is not to be taken in a limiting sense, and the scope of embodiments is defined by the appended claims and their equivalents.

[00155] Various operations may be described as multiple discrete actions or operations in turn, in a manner that is most helpful in understanding the claimed subject matter. However, the order of description should not be construed as to imply that

these operations are necessarily order dependent. In particular, these operations may not be performed in the order of presentation. Operations described may be performed in a different order than the described embodiment. Various additional operations may be performed and/or described operations may be omitted in additional embodiments.

**[00156]** For the purposes of the present disclosure, the phrase “A and/or B” means (A), (B), or (A and B). For the purposes of the present disclosure, the phrase “A, B, and/or C” means (A), (B), (C), (A and B), (A and C), (B and C), or (A, B, and C). Where the disclosure recites “a” or “a first” element or the equivalent thereof, such disclosure includes one or more such elements, neither requiring nor excluding two or more such elements. Further, ordinal indicators (e.g., first, second, or third) for identified elements are used to distinguish between the elements, and do not indicate or imply a required or limited number of such elements, nor do they indicate a particular position or order of such elements unless otherwise specifically stated.

**[00157]** Reference in the description to one embodiment or an embodiment means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The description may use the phrases “in one embodiment,” “in another embodiment,” “in some embodiments,” “in embodiments,” “in various embodiments,” or the like, which may each refer to one or more of the same or different embodiments. Furthermore, the terms “comprising,” “including,” “having,” and the like, as used with respect to embodiments of the present disclosure, are synonymous.

**[00158]** In embodiments, the term “engine” or “module” or “logic” may refer to, be part of, or include an application specific integrated circuit (ASIC), an electronic circuit, a processor (shared, dedicated, or group), and/or memory (shared, dedicated, or group) that execute one or more software or firmware programs, a combinational logic circuit, and/or other suitable components that provide the described functionality. In embodiments, an engine or a module may be implemented in firmware, hardware, software, or any combination of firmware, hardware, and software. 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.

**[00159]** 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.).

**[00160]** 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. A apparatus comprising:  
a graphics processing unit (GPU) to process graphics commands and responsively render a plurality of image frames;  
a hypervisor to virtualize the GPU to share the GPU among a plurality of virtual machines (VMs), the hypervisor subdividing GPU processing for each VM into a plurality of quanta; and  
scheduling logic to monitor GPU utilization including GPU busy time and/or GPU idle time for each VM during its allocated quantum and to store utilization data reflecting the GPU utilization during each quantum;  
the scheduling logic to predict a cost of waiting within a given quantum of a first VM based on the utilization data and to further predict a cost of yielding to a second VM, the scheduling logic to yield the GPU to the second VM if the cost of waiting is greater than the cost of yielding to the second VM.
2. The apparatus as in claim 1 wherein the scheduling logic is to yield the GPU to the second VM if the cost of waiting is greater than the sum of the cost of yielding to the second VM and a specified threshold value.
3. The apparatus as in claim 2 wherein the threshold value comprises a value of 0 or greater.
4. The apparatus as in claim 1 wherein to monitor the GPU busy time and/or GPU idle time, the scheduling logic stamps comments within a set of GPU commands to be executed per VM within each quantum.
5. The apparatus as in claim 1 wherein the scheduling logic is configured to store the utilization data reflecting GPU utilization as a probability curve reflecting probabilities a new command will be received within a given quantum for different time periods.
6. The apparatus as in claim 5 wherein at least one time period comprises an average amount of time within which a new command is expected based on the probability curve.

7. The apparatus as in claim 5 wherein at least one time period is associated with a specified percentage comprising a likelihood that a command will be received by within the designated time period.

8. The apparatus as in claim 1 wherein the scheduling logic is configured to determine whether commands are pending for the second VM and to yield the GPU to the second VM only if commands are pending for the second VM.

9. The apparatus as in claim 2 wherein the scheduling logic dynamically adjusts the specified threshold value in response to workload conditions on each of the VMs.

10. A method comprising:  
selecting a first virtual machine (VM) to execute commands on a graphics processor unit (GPU);  
monitoring GPU utilization data including GPU busy time and/or GPU idle time for the first VM during one or more allocated quanta;  
storing utilization data reflecting the GPU utilization;  
predicting a cost of waiting within a first quantum of a first VM based on the utilization data and a cost of yielding to a second VM; and  
yielding the GPU to the second VM if the cost of waiting is greater than the cost of yielding to the second VM.

11. The method as in claim 10 wherein the GPU is to be yielded to the second VM if the cost of waiting is greater than the sum of the cost of yielding to the second VM and a specified threshold value.

12. The method as in claim 11 wherein the threshold value comprises a value of 0 or greater.

13. The method as in claim 10 wherein to monitor the GPU busy time and/or GPU idle time, stamping a set of GPU commands to be executed per VM within each quantum.

14. The method as in claim 10 further comprising storing the utilization data reflecting GPU utilization as a probability curve reflecting probabilities a new command will be received within a given quantum for different time periods.

15. The method as in claim 14 wherein at least one time period comprises an average amount of time within which a new command is expected based on the probability curve.

16. The method as in claim 14 wherein at least one time period is associated with a specified percentage comprising a likelihood that a command will be received by within the designated time period.

17. The method as in claim 10 further comprising:  
determining whether commands are pending for the second VM and yielding the GPU to the second VM only if commands are pending for the second VM.

18. The method as in claim 2 further comprising:  
dynamically adjusting the specified threshold value in response to workload conditions on each of the VMs.

19. A system comprising:  
a memory to store data and program code;  
a central processing unit (CPU) comprising an instruction cache for caching a portion of the program code and a data cache for caching a portion of the data, the CPU further comprising execution logic to execute at least some of the program code and responsively process at least some of the data, at least a portion of the program code comprising graphics commands;  
a graphics processing unit (GPU) to process the graphics commands and responsively render a plurality of image frames;  
a graphics processing unit (GPU) to process graphics commands and responsively render a plurality of image frames;  
a hypervisor to virtualize the GPU to share the GPU among a plurality of virtual machines (VMs), the hypervisor subdividing GPU processing for each VM into a plurality of quanta; and

scheduling logic to monitor GPU utilization including GPU busy time and/or GPU idle time for each VM during its allocated quantum and to store utilization data reflecting the GPU utilization during each quantum;

the scheduling logic to predict a cost of waiting within a given quantum of a first VM based on the utilization data and to further predict a cost of yielding to a second VM, the scheduling logic to yield the GPU to the second VM if the cost of waiting is greater than the cost of yielding to the second VM.

20. The apparatus as in claim 19 wherein the scheduling logic is to yield the GPU to the second VM if the cost of waiting is greater than the sum of the cost of yielding to the second VM and a specified threshold value.

21. The apparatus as in claim 20 wherein the threshold value comprises a value of 0 or greater.

22. The apparatus as in claim 19 wherein to monitor the GPU busy time and/or GPU idle time, the scheduling logic stamps comments within a set of GPU commands to be executed per VM within each quantum.

23. The apparatus as in claim 19 wherein the scheduling logic is configured to store the utilization data reflecting GPU utilization as a probability curve reflecting probabilities a new command will be received within a given quantum for different time periods.

24. The apparatus as in claim 23 wherein at least one time period comprises an average amount of time within which a new command is expected based on the probability curve.

25. The apparatus as in claim 23 wherein at least one time period is associated with a specified percentage comprising a likelihood that a command will be received by within the designated time period.

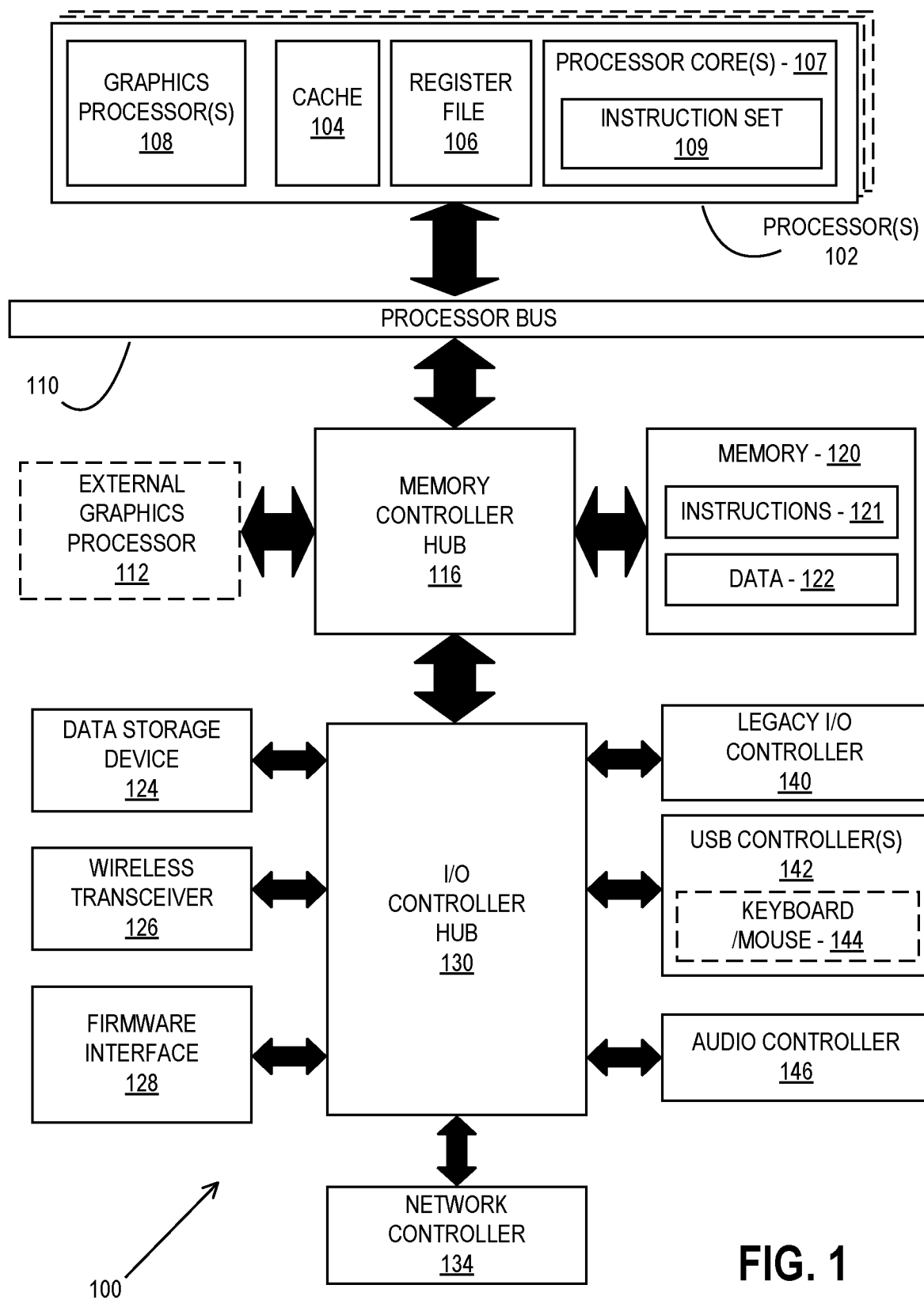


FIG. 1

PROCESSOR 200

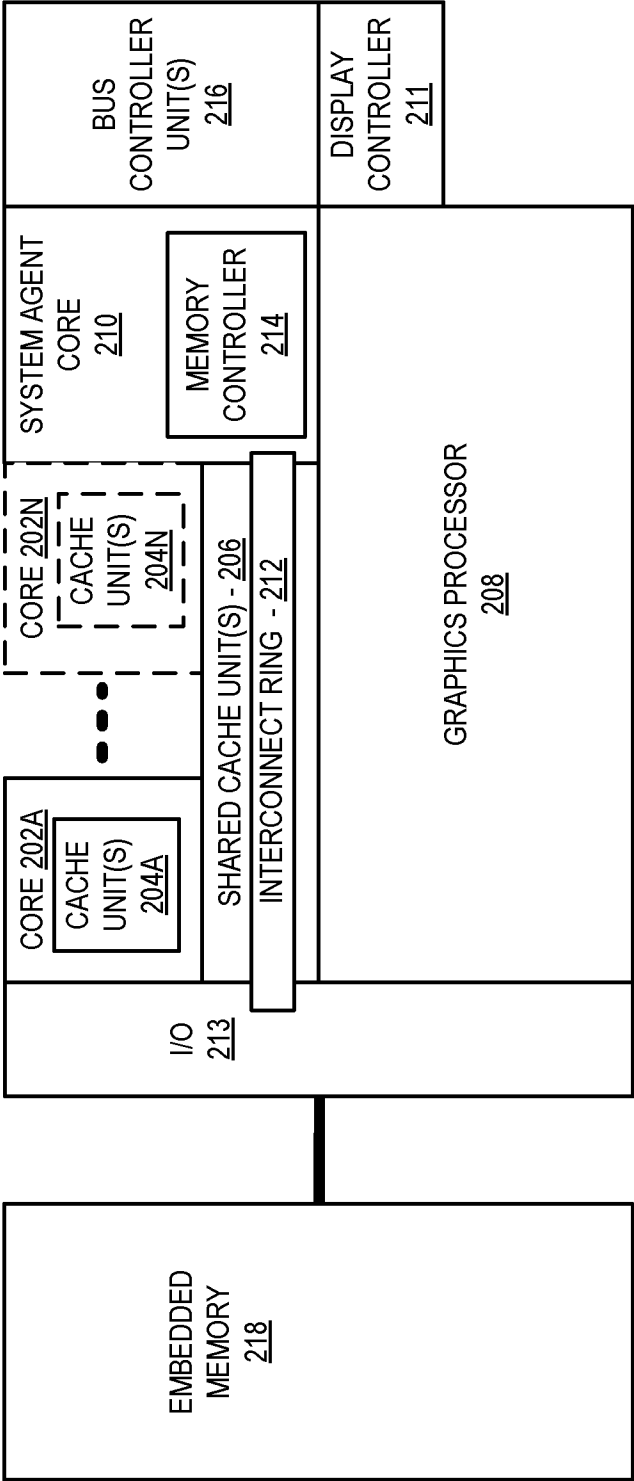
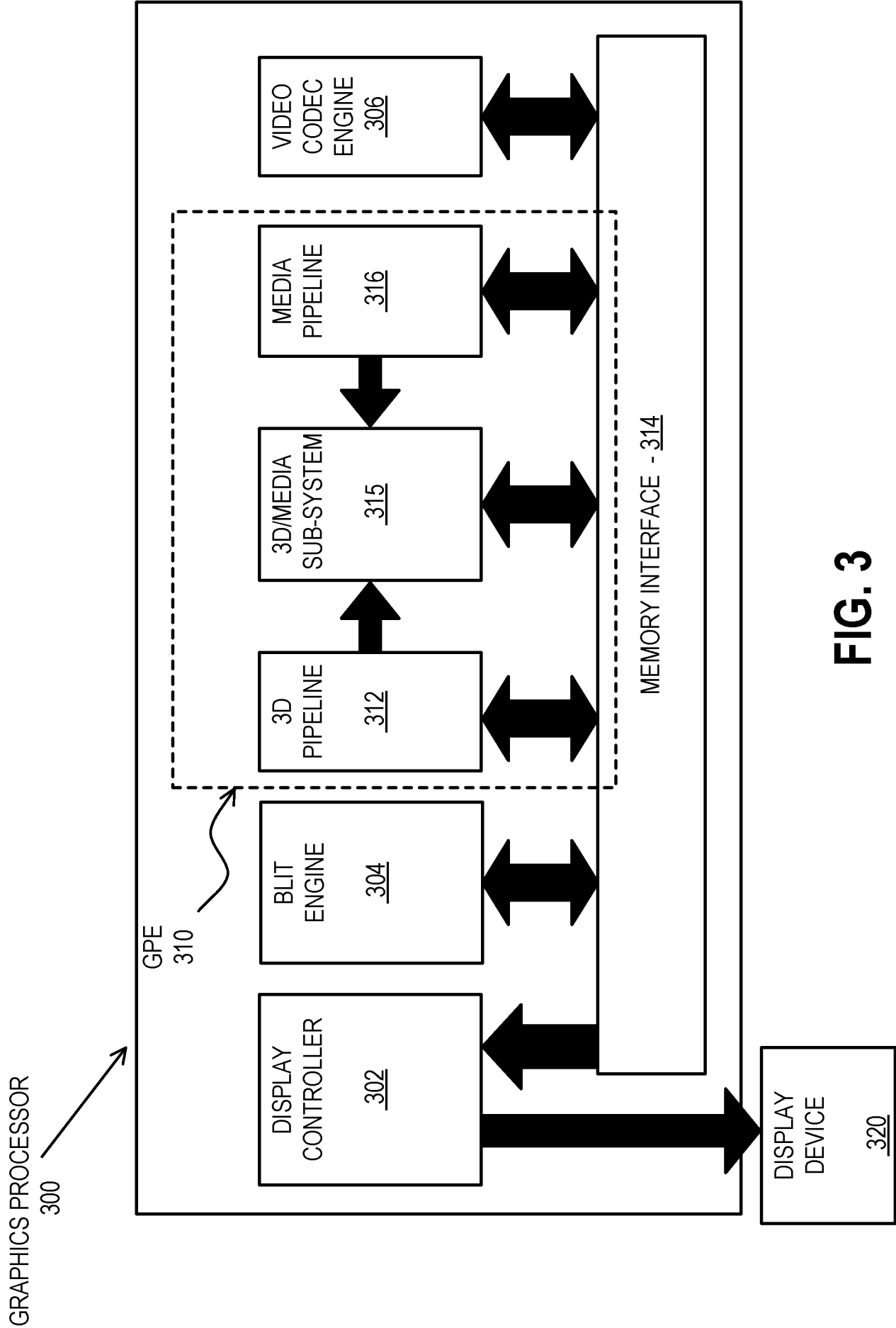


FIG. 2



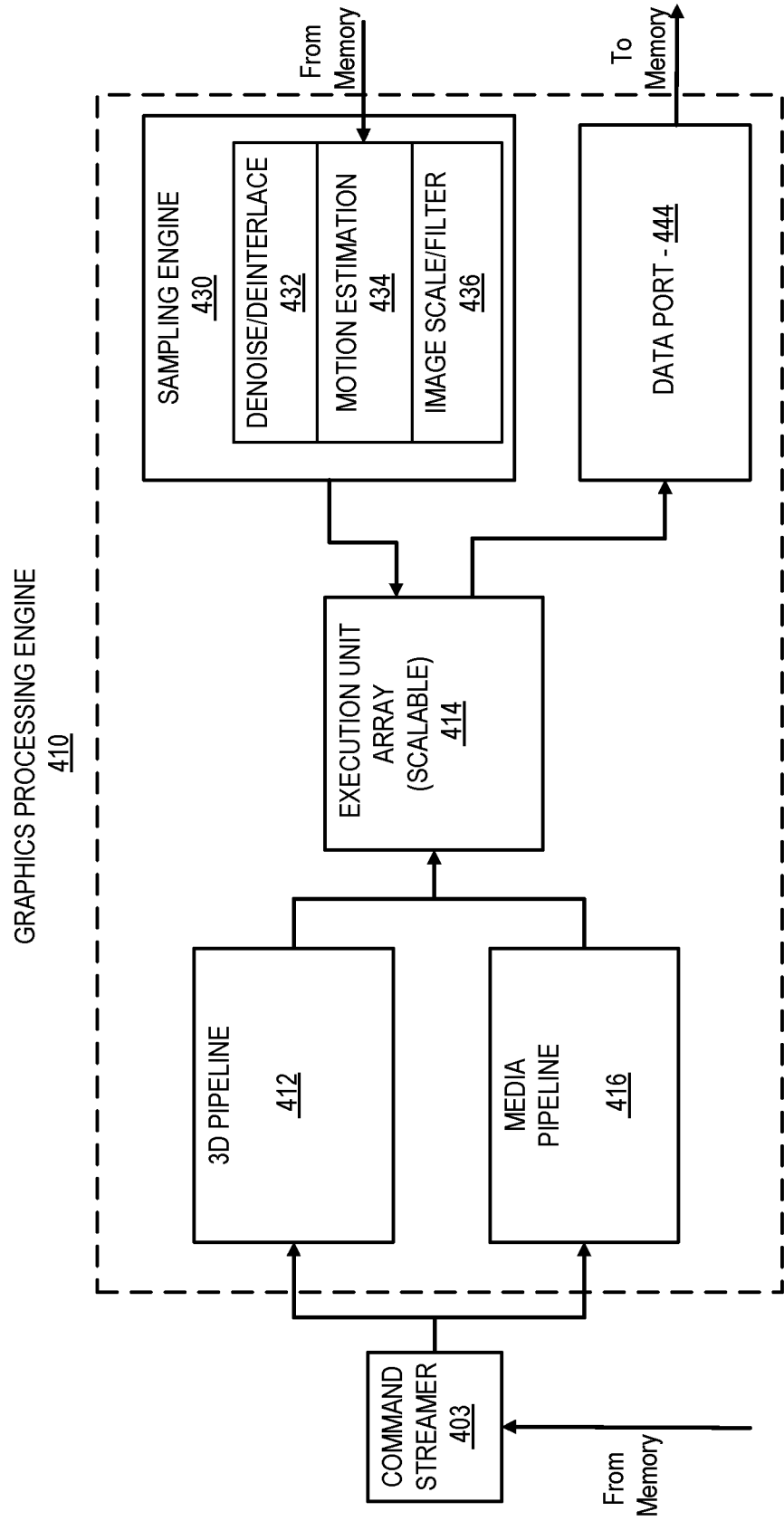


FIG. 4

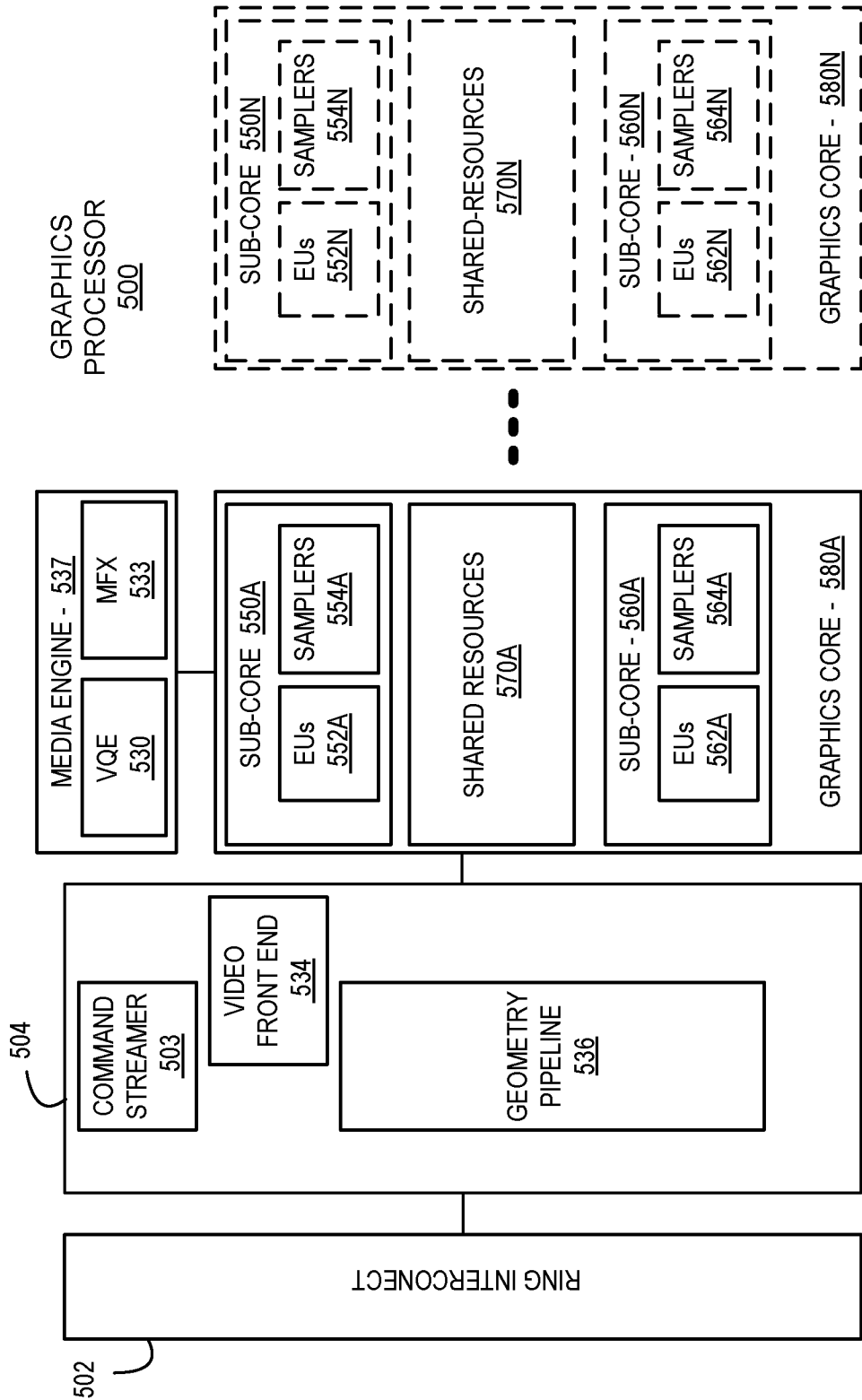


FIG. 5

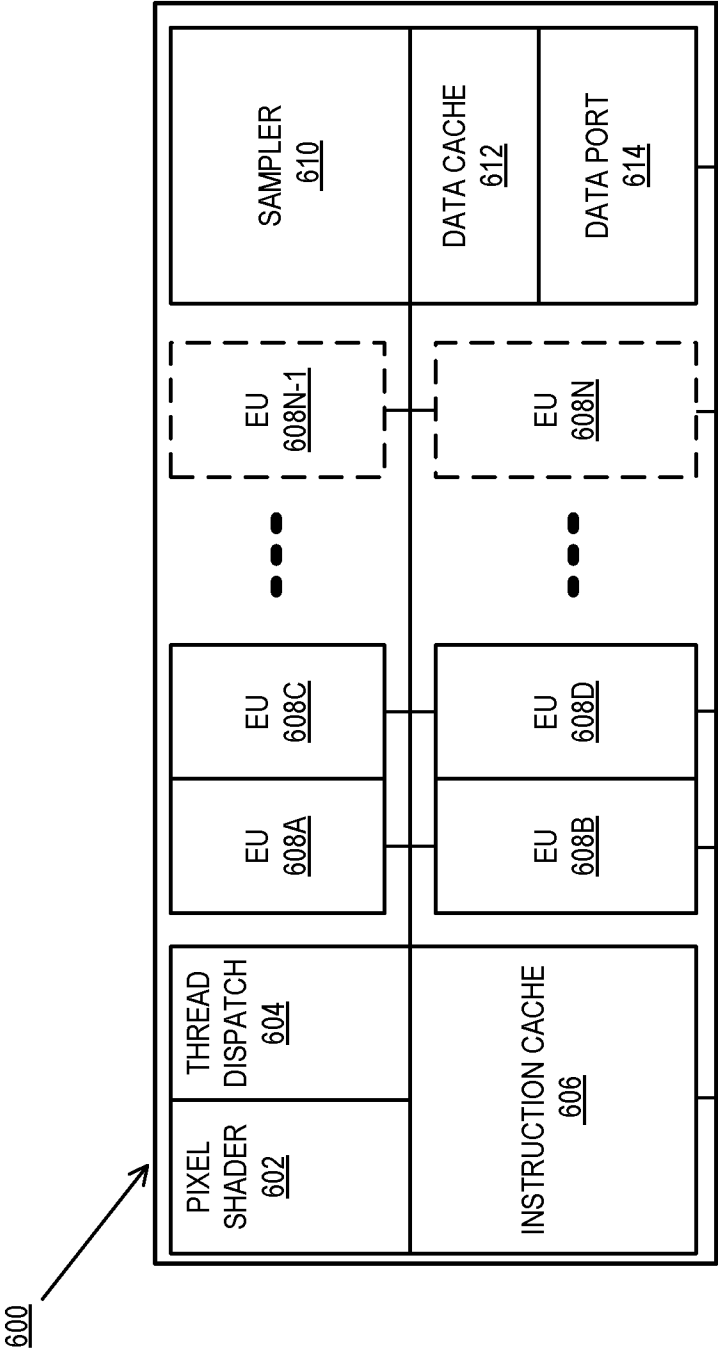


FIG. 6

GRAPHICS CORE INSTRUCTION  
FORMATS

700

128-BIT INSTRUCTION

710



64-BIT COMPACT  
INSTRUCTION

730



OPCODE DECODE

740

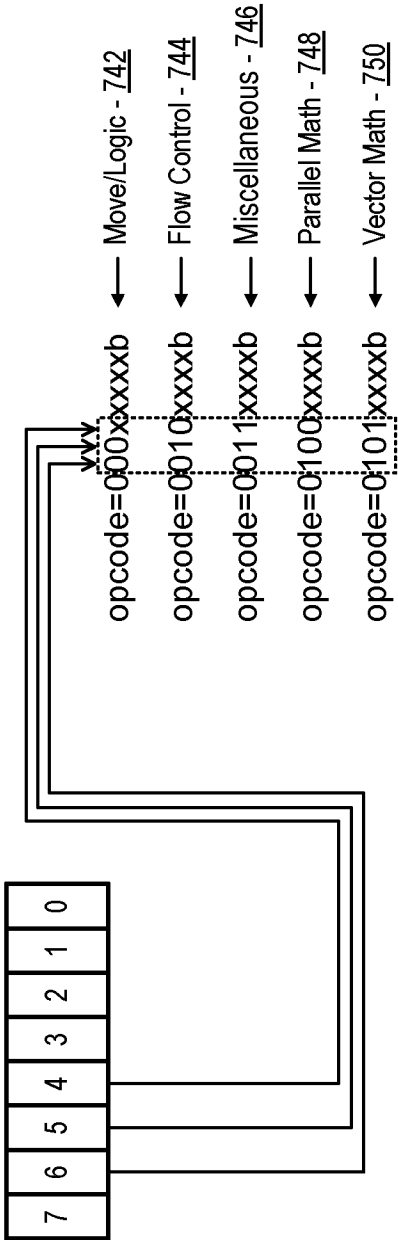


FIG. 7

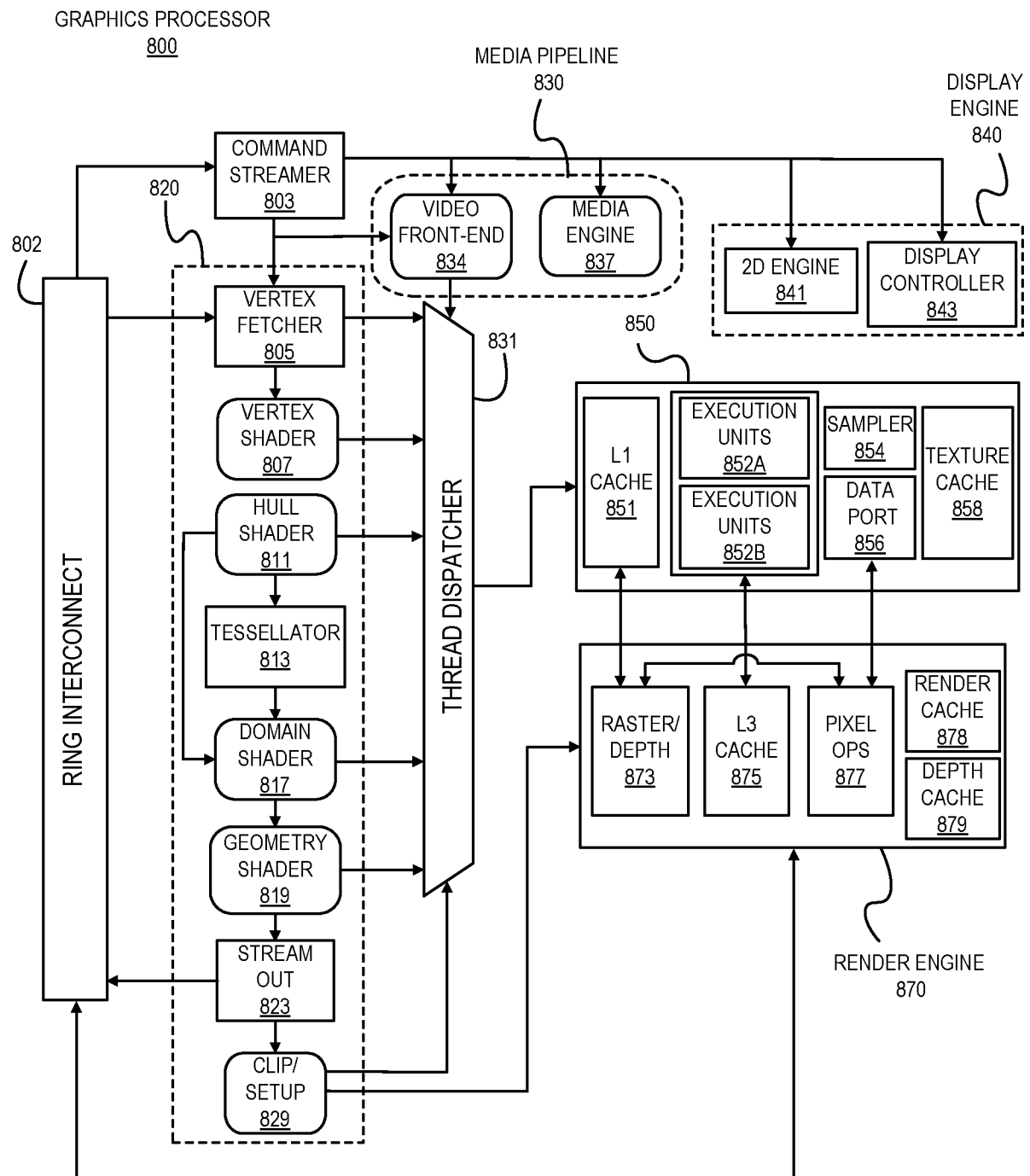
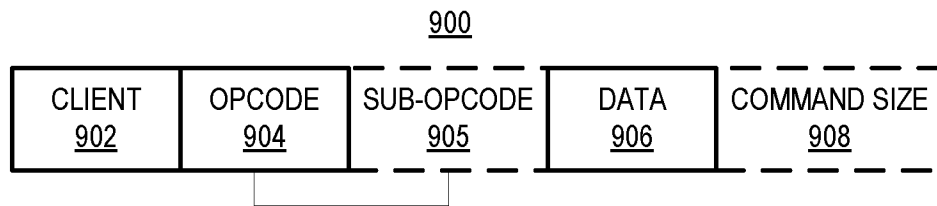
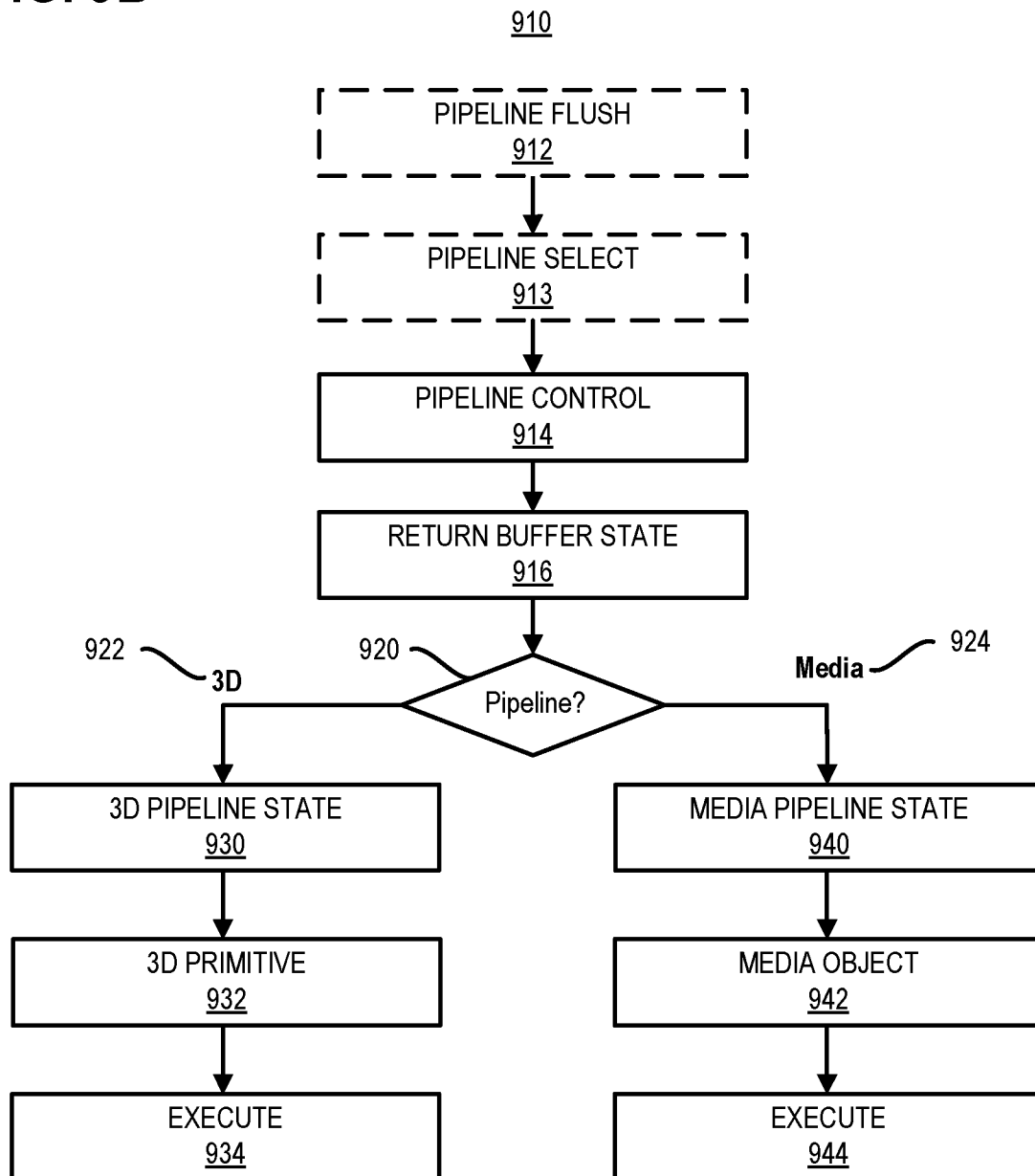
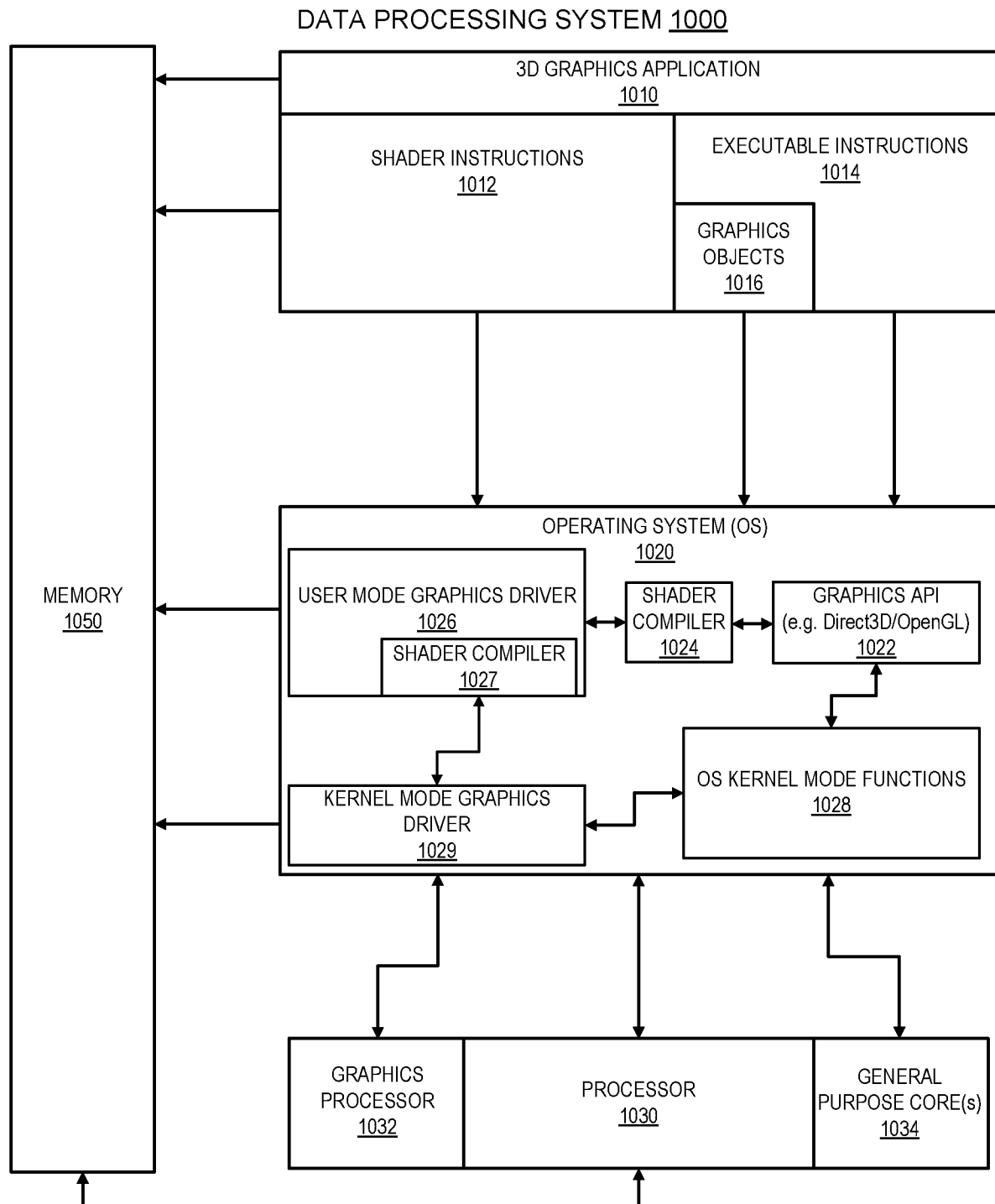


FIG. 8

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

**FIG. 10**

IP CORE DEVELOPMENT - 1100

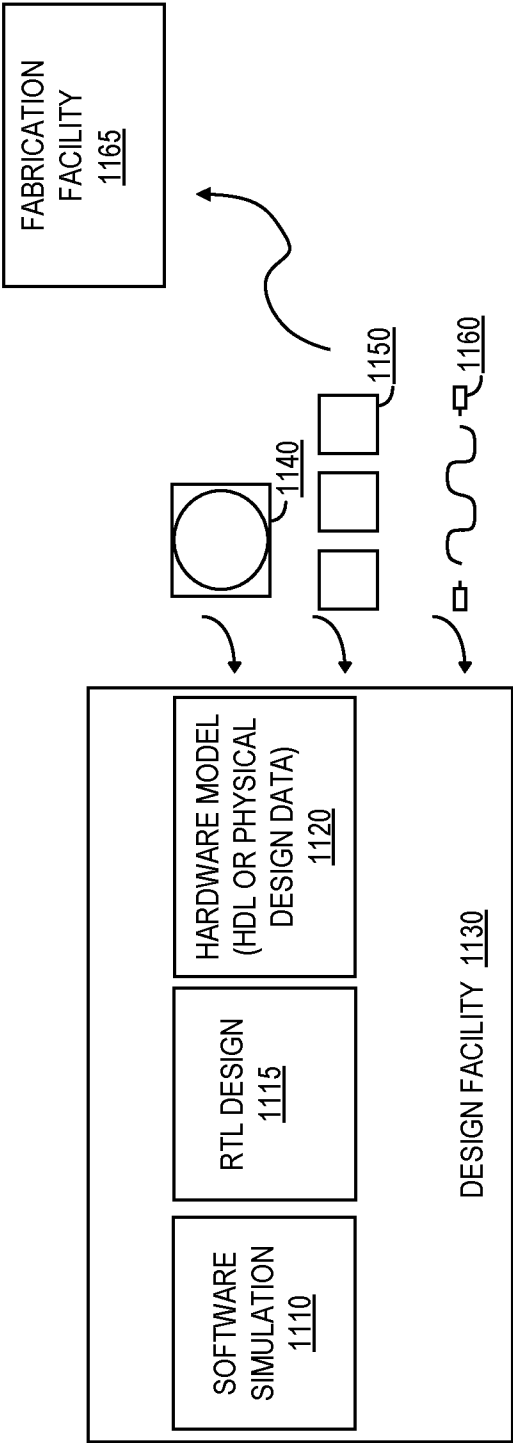
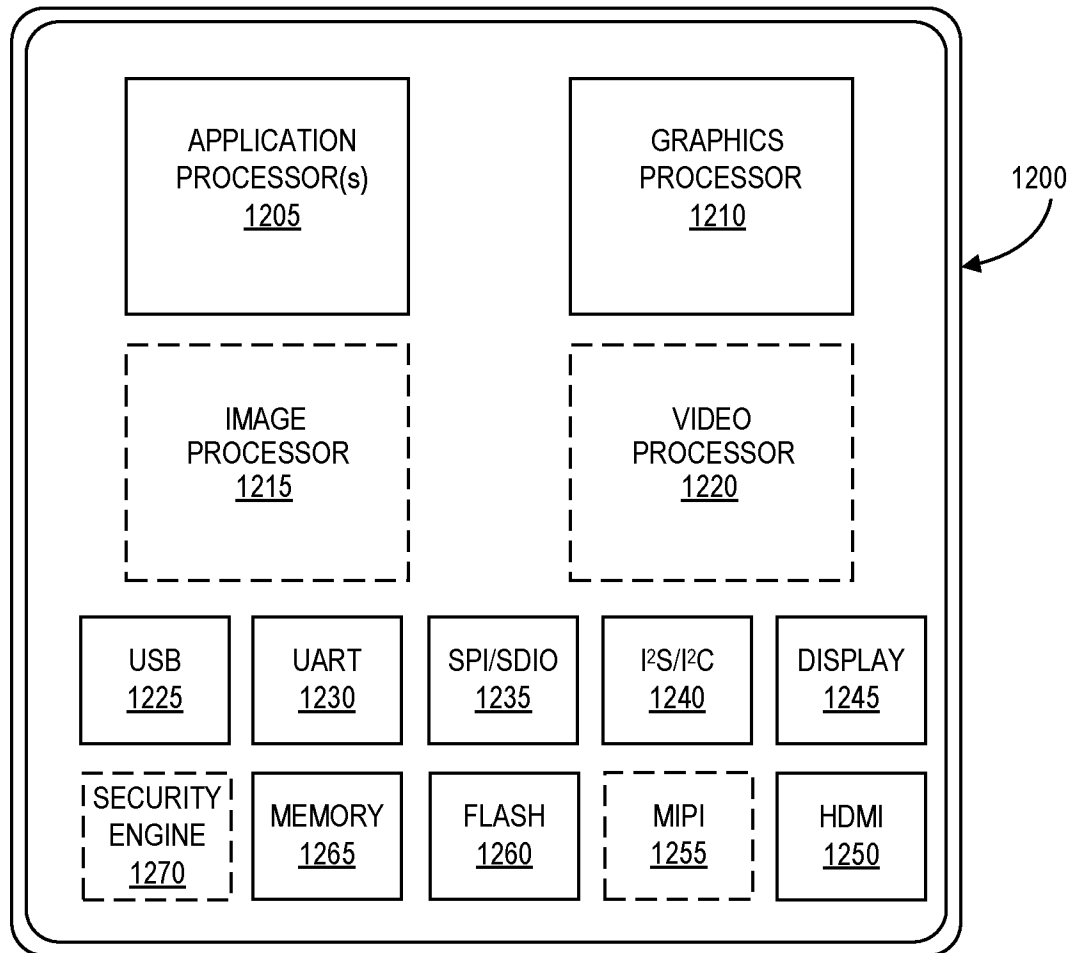


FIG. 11

**FIG. 12**

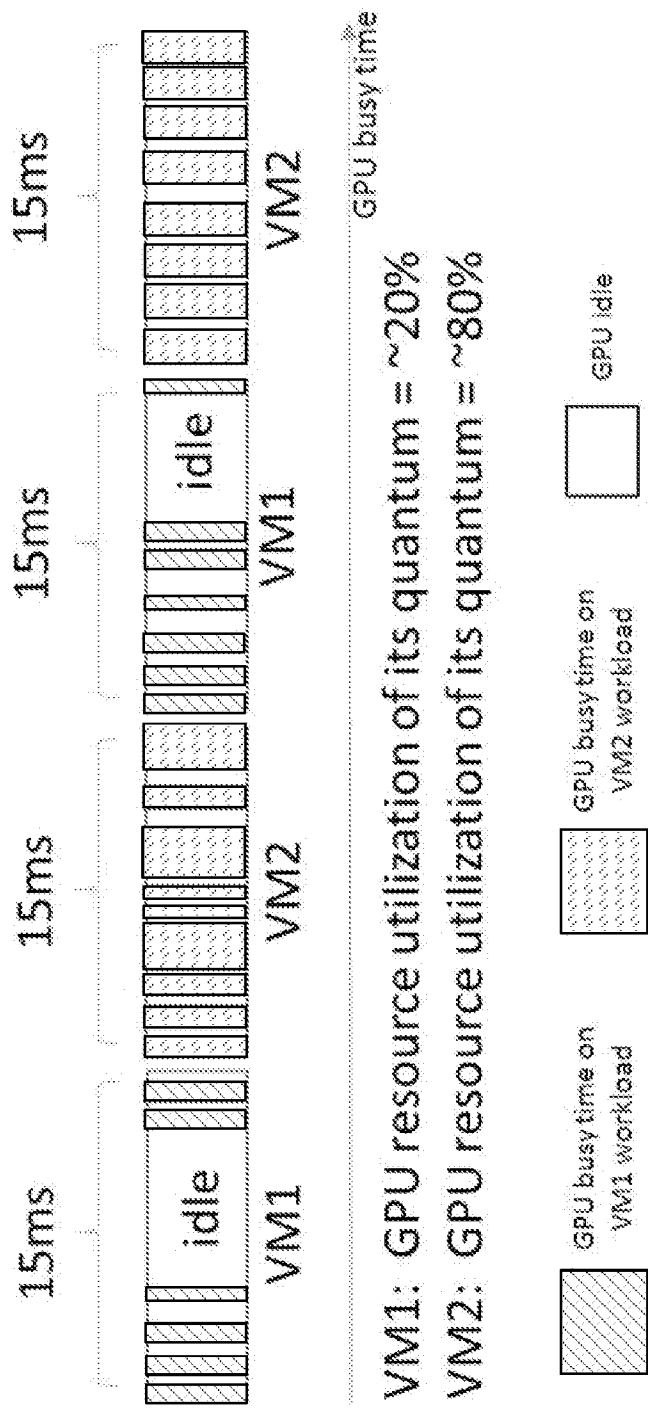


FIG. 13

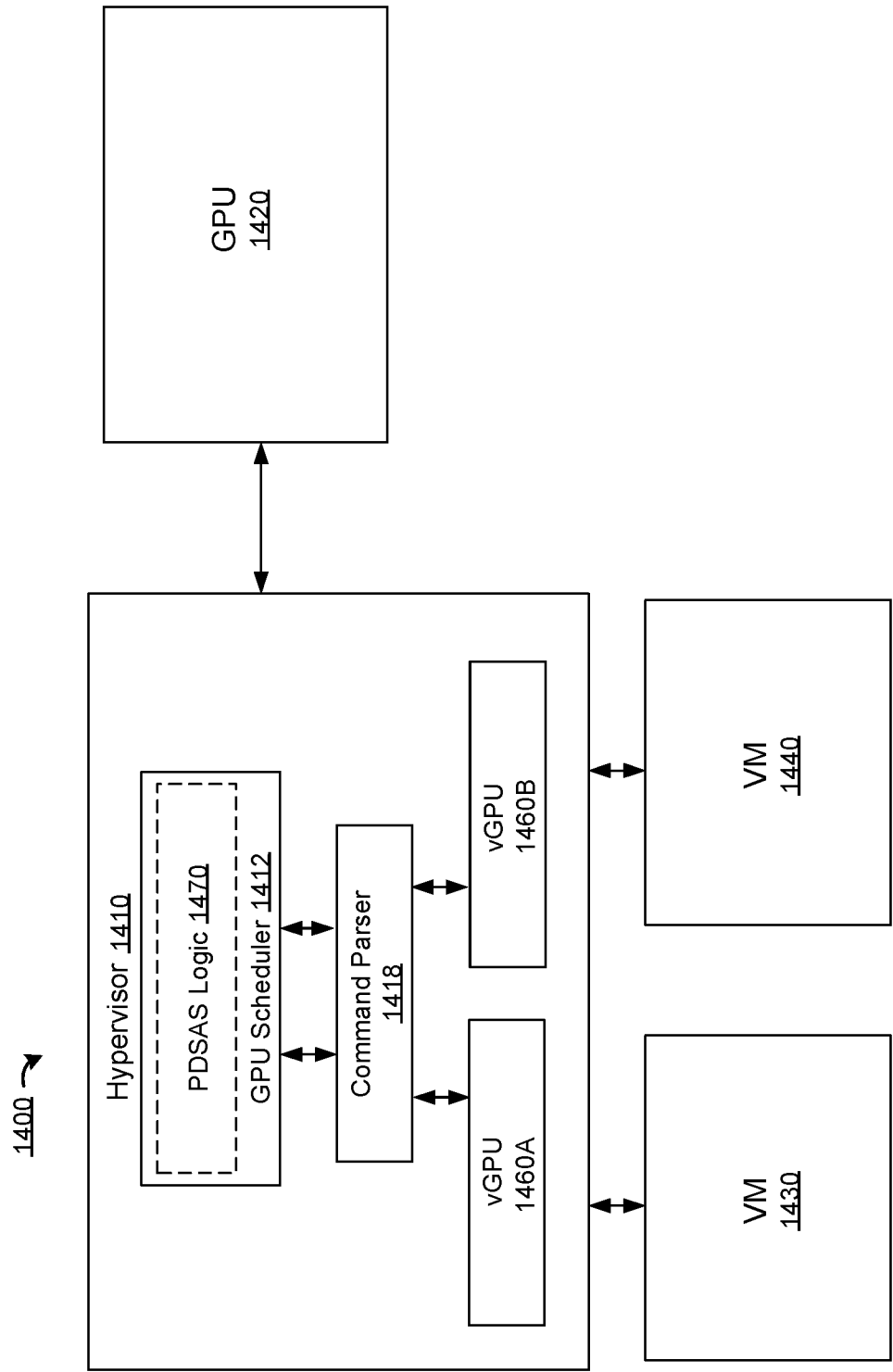


Fig. 14

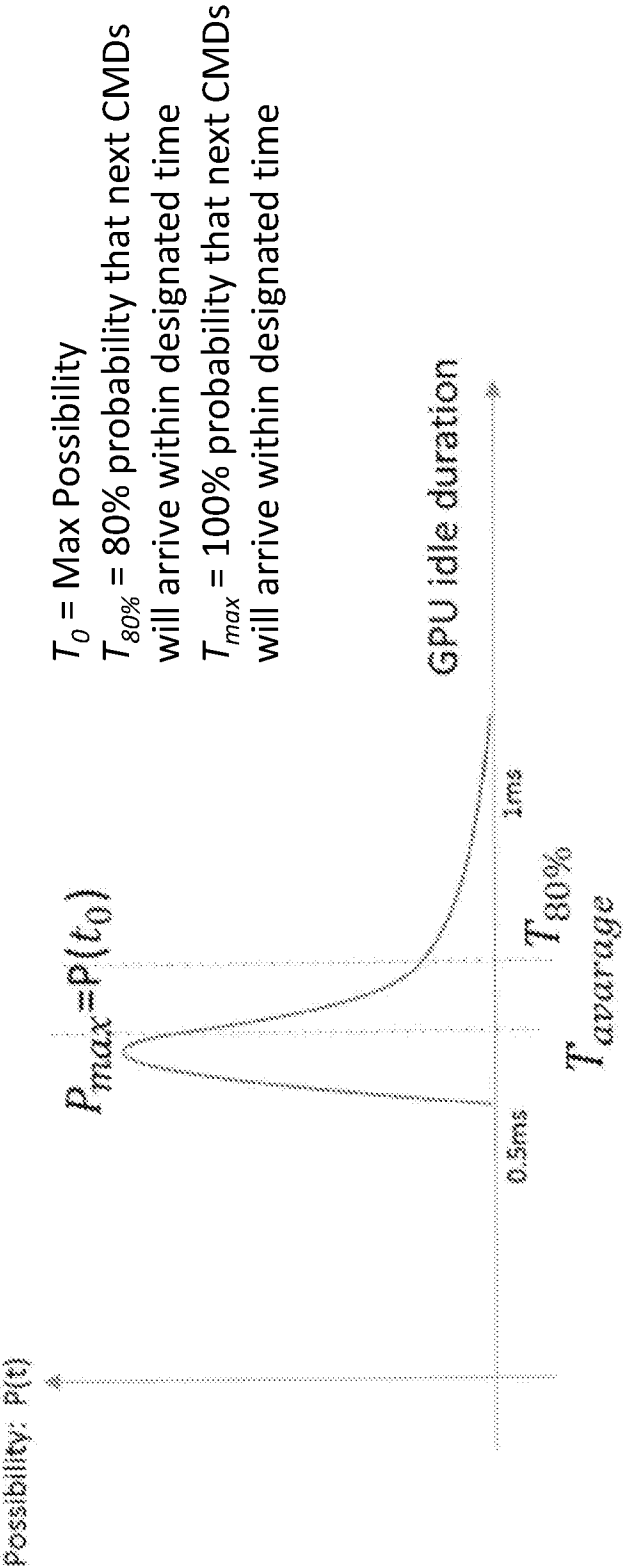


FIG. 15

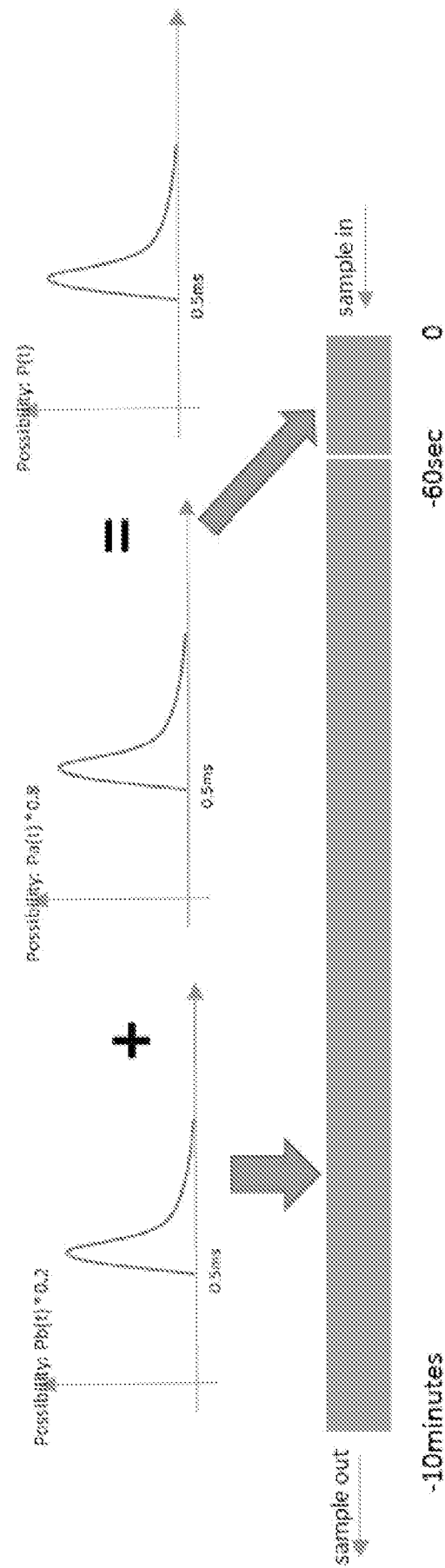


FIG. 16

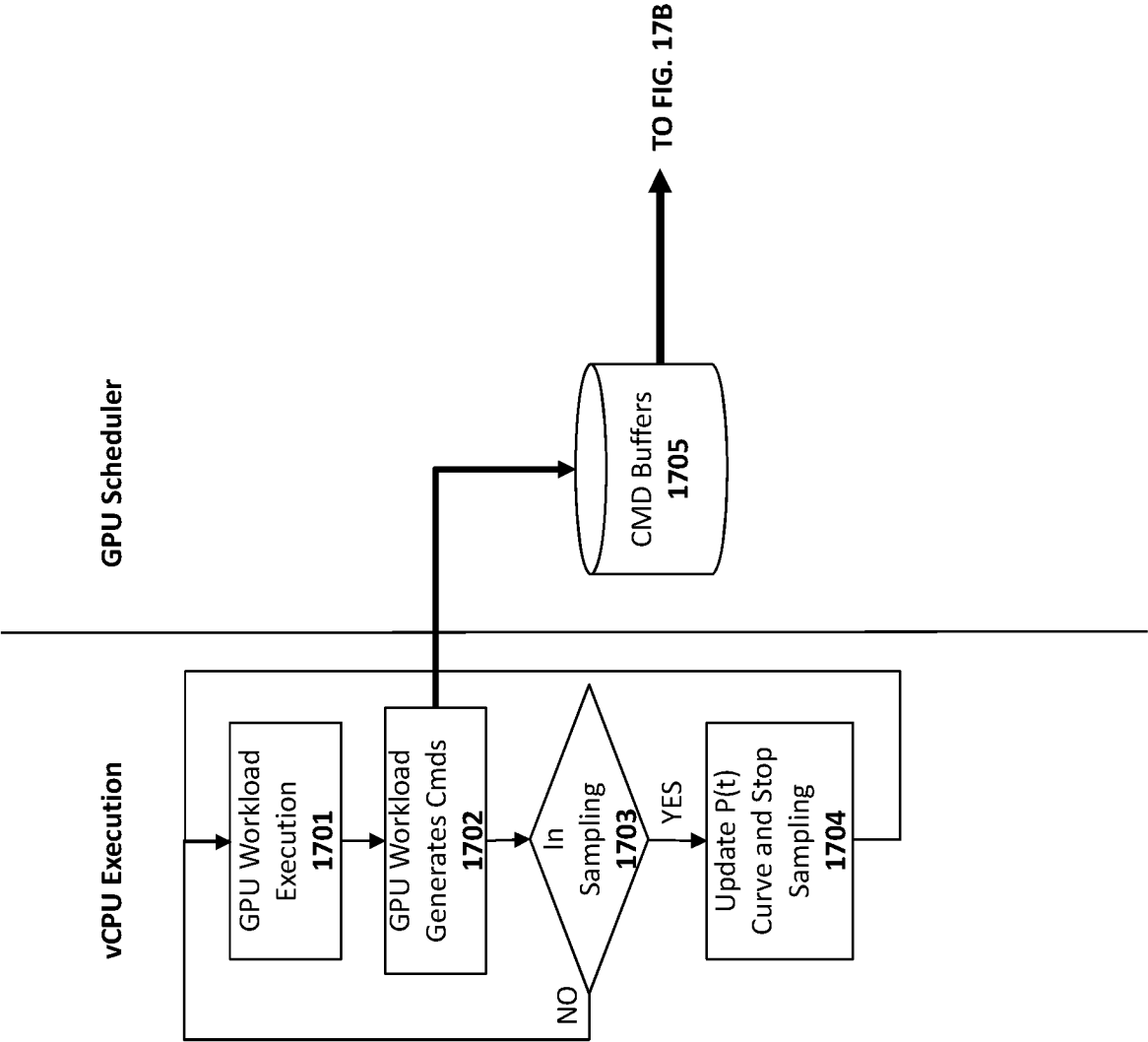
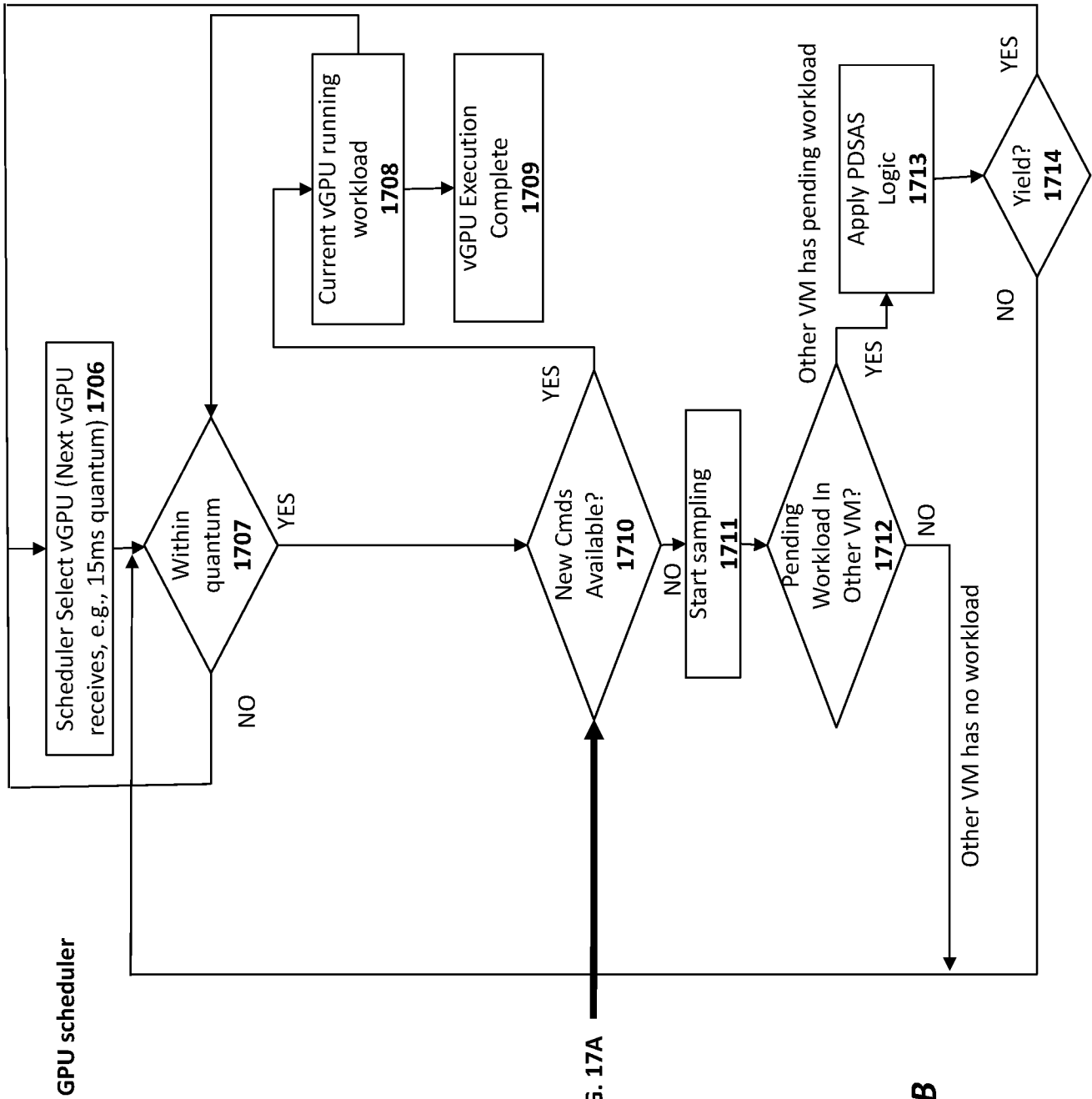


FIG. 17A



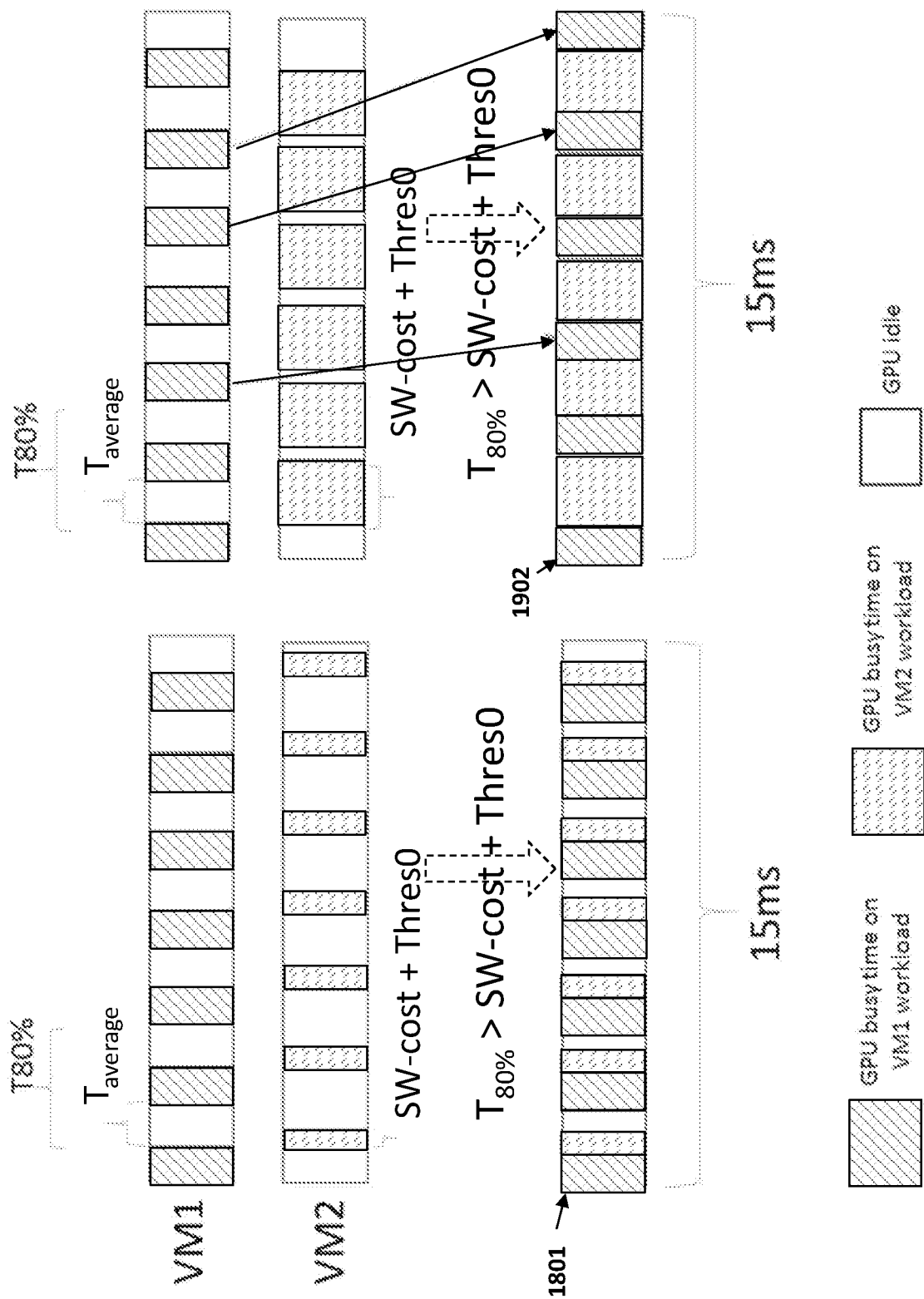


FIG. 18

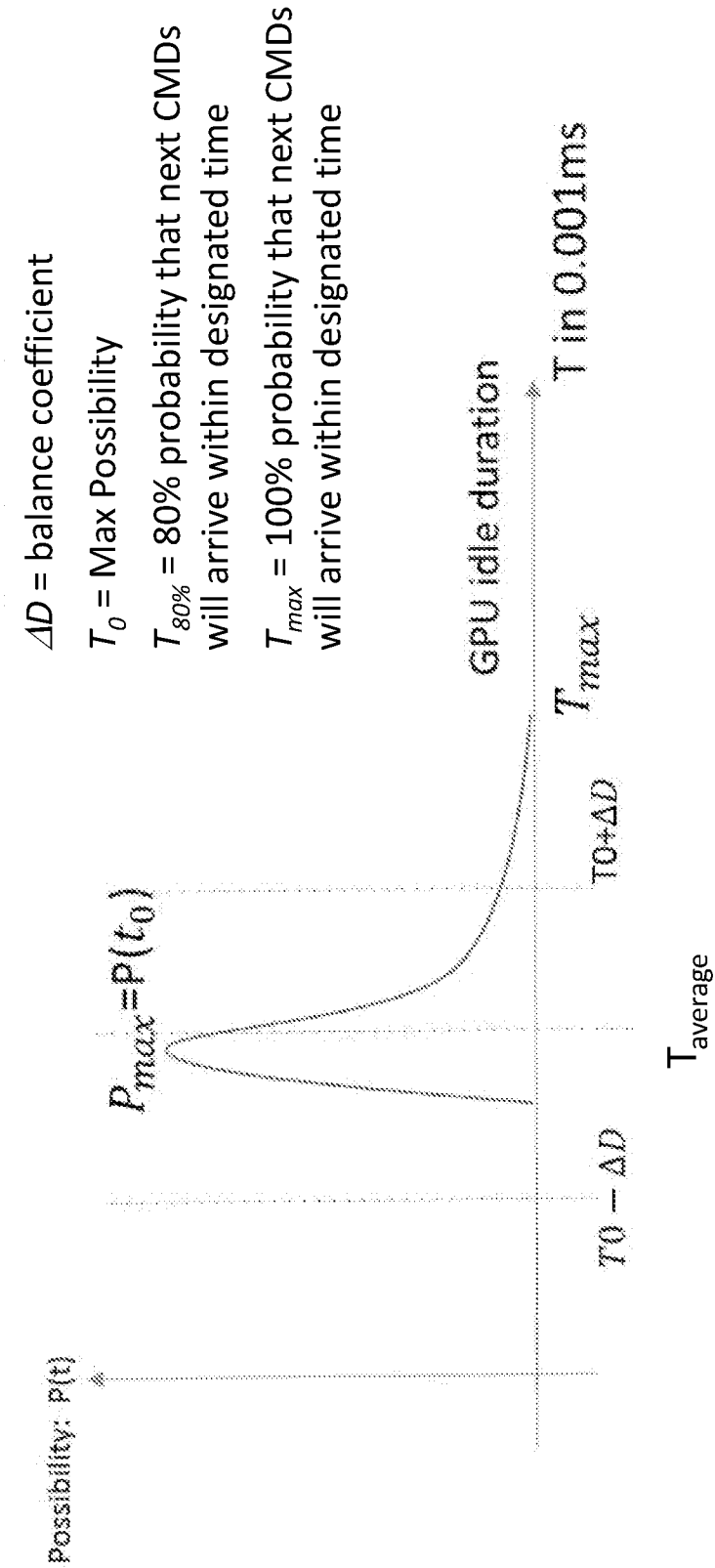


FIG. 19

## INTERNATIONAL SEARCH REPORT

International application No.

PCT/CN2015/090571

**A. CLASSIFICATION OF SUBJECT MATTER**

G06F 9/50(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; H04L

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)

EPODOC,WPI,CNKI,CNPAT: graphics w processing w unit, gpu, shar+, vm, virtual w machine, time, busy, wait, idle

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category* | Citation of document, with indication, where appropriate, of the relevant passages                                                                            | Relevant to claim No. |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|
| A         | CN 104660711 A (HUACUN DATA INFORMATION TECHNOLOGY CO., LTD.) 27 May 2015 (2015-05-27)<br>the abstract, description, paragraphs [0019]-[0033] and figures 1-2 | 1-25                  |
| A         | US 2012084774 A1 (MICROSOFT CORPORATION) 05 April 2012 (2012-04-05)<br>the whole document                                                                     | 1-25                  |
| A         | CN 104216783 A (SHANGHAI JIAO TONG UNIVERSITY) 17 December 2014 (2014-12-17)<br>the whole document                                                            | 1-25                  |

☐ 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

08 June 2016

Date of mailing of the international search report

23 June 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

FENG,Tingting

Facsimile No. (86-10)62019451

Telephone No. (86-10)62413577

**INTERNATIONAL SEARCH REPORT**  
**Information on patent family members**

International application No.

**PCT/CN2015/090571**

| Patent document<br>cited in search report |            |    | Publication date<br>(day/month/year) | Patent family member(s) |            |    | Publication date<br>(day/month/year) |
|-------------------------------------------|------------|----|--------------------------------------|-------------------------|------------|----|--------------------------------------|
| CN                                        | 104660711  | A  | 27 May 2015                          | None                    |            |    |                                      |
| US                                        | 2012084774 | A1 | 05 April 2012                        | EP                      | 2622470    | A2 | 07 August 2013                       |
|                                           |            |    |                                      | CN                      | 102402462  | A  | 04 April 2012                        |
|                                           |            |    |                                      | WO                      | 2012050718 | A2 | 19 April 2012                        |
|                                           |            |    |                                      | JP                      | 2014500535 | A  | 09 January 2014                      |
|                                           |            |    |                                      | HK                      | 1187425    | A0 | 04 April 2014                        |
| CN                                        | 104216783  | A  | 17 December 2014                     | None                    |            |    |                                      |