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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
04 January 2018 (04.01.2018)

WIPO | PCT

(10) International Publication Number
WO 2018/004939 A1

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

(21) International Application Number:
PCT/US2017/035069

(22) International Filing Date:
30 May 2017 (30.05.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
15/201,163 01 July 2016 (01.07.2016) US

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

(72) Inventors: **APODACA, Michael**; 1563 Ballou Cir, Folsom, California 95630 (US). **CIMINI, David M.**; 5611 Bench Court, Orangevale, California 95662 (US). **RAOUX, Thomas F.**; 707 Continental Circle, Apt. 1210, Mountain View, California 94040 (US). **GHOSH, Somnath**; 445 Bradley Ave., San Jose, California 95128 (US). **MUKHERJEE, Uddipan**; 440 Oak Grove Drive, Apt. 103, Santa Clara, California 95054 (US). **BOSE, Debraj**; 116 Holly Terrace, Sunnyvale, California 94086 (US). **DEKA, Sthiti**; 2200 Juliette Lane, Santa Clara, California 95054 (US). **GEVIM, Yohai**; Ner Halaya 2, 40293 Beit Yanai (IL).

(74) Agent: **AGHEVLI, Ramin**; Alpine Technology Law Group LLC, 9249 S. Broadway, Unit 200-201, Highlands Ranch, Colorado 80129 (US).

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

(54) Title: REDUCING MEMORY LATENCY IN GRAPHICS OPERATIONS

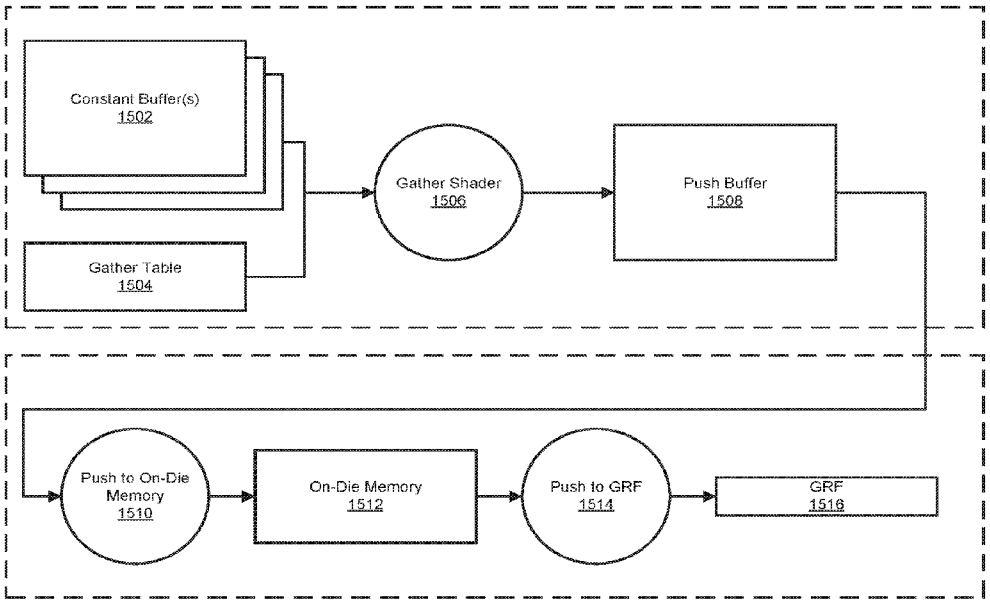


FIG. 15

(57) Abstract: Methods and apparatus relating to reducing memory latency in graphics operations are described. In an embodiment, uniform data is transferred from a buffer to a General Register File (GRF) of a processor based at least in part on information stored in a gather table. The uniform data comprises data that is uniform across a plurality of primitives in a graphics operation. Other embodiments are also disclosed and claimed.

WO 2018/004939 A1

[Continued on next page]

(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).

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

Published:

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

REDUCING MEMORY LATENCY IN GRAPHICS OPERATIONS

RELATED APPLICATIONS

This application claims priority under 35 U.S.C. § 365(c) to US Application No. 15/201,163 filed on July 1, 2016, entitled REDUCING MEMORY LATENCY IN GRAPHICS OPERATIONS. The entire disclosure of this document is incorporated by reference herein for all purposes.

FIELD

The present disclosure generally relates to the field of electronics. More particularly, some embodiments relate to techniques for reducing memory latency in graphics operations.

BACKGROUND

Most computing systems tend to include one or more graphics processors to execute graphics (or even general-purpose) instructions. Moreover, in three-dimensional (3D) graphics, some graphics programs (sometimes called shader programs or more generally "shaders") run at different frequencies, such as per-vertex, per-pixel, etc. These programs may fetch data from memory, such as matrices and texture data. Frequently, the data fetched is uniform across all primitives in a drawing operation. These may be referred to as "constants" (or uniform data) in 3D graphics. Fetching constants from memory within graphics programs causes stalls (or slowdowns) in execution. These stalls are mostly hidden by thread switching and caches. However, when there is an insufficient number of threads to hide the latency to memory, or there is an insufficient amount of cache to hold all the constants, these stalls can degrade the performance of the graphics operations.

BRIEF DESCRIPTION OF THE DRAWINGS

The detailed description is provided with reference to the accompanying figures. In the figures, the left-most digit(s) of a reference number identifies the figure in which the reference number first appears. The use of the same reference numbers in different figures indicates similar or identical items.

Figs. 1 and 10 illustrate block diagrams of embodiments of computing systems, which may be utilized to implement various embodiments discussed herein.

Figs. 2-6, 8, 13, and 14 illustrate various components of processers in accordance with some embodiments.

Fig. 7 illustrates graphics core instruction formats, according to some embodiments.

Figs. 9A and 9B illustrate graphics processor command format and sequence, respectively, according to some embodiments.

Fig. 11 illustrates a diagram of IP core development according to an embodiment.

Fig. 12 illustrates components of a System on Chip (SoC or SOC) integrated circuit, according to an embodiment.

Fig. 15 illustrates flow of uniform data to General Register File (GRF) using shader-based gathering, according to an embodiment.

Fig. 16 illustrates sample pseudocode for a gather shader program, according to an embodiment.

Fig. 17 illustrates a sequence of Graphics Processing Unite (GPU) commands for executing gather shaders using batch processing, according to an embodiment.

Fig. 18 illustrates flow for batching gather shaders, according to an embodiment.

Fig. 19 illustrates a sequence of GPU commands for batching gather shaders, according to an embodiment.

Fig. 20 illustrates flow of Uniform data to GRF using draw shader, according to an embodiment.

Fig. 21 illustrates a sample pseudocode for a draw shader program, according to an embodiment.

DETAILED DESCRIPTION

In the following description, numerous specific details are set forth in order to provide a thorough understanding of various embodiments. However, various embodiments may be practiced without the specific details. In other instances, well-known methods, procedures, components, and circuits have not been described in detail so as not to obscure the particular embodiments. Further, various aspects of embodiments may be performed using various means, such as integrated semiconductor circuits ("hardware"), computer-readable instructions organized into one or more programs ("software"), or some combination of hardware and software. For the purposes of this disclosure reference to "logic" shall mean either hardware, software, firmware, or some combination thereof.

Some embodiments provide techniques for reducing memory latency in graphics operations. In one or more embodiments, memory fetches from high frequency graphics (or GPU (Graphics Processing Unit)) programs for uniform data is reduced. Instead, a graphics program may execute per-draw (or per-batch) to pre-fetch data from memory. The data may then be stored in such a way that it can be used within the higher frequency programs with minimal latency. Multiple solutions will be discussed herein which would be applicable in

both current and future GPUs. As discussed herein, "uniform data" generally refers to data that is uniform (or constant) across all or a plurality of primitives in a graphics operation/draw.

Further, some embodiments may be applied in computing systems that include one or more processors (e.g., with one or more processor cores), such as those discussed with reference to Figs. 1-21, including for example mobile computing devices, e.g., a smartphone, tablet, UMPC (Ultra-Mobile Personal Computer), laptop computer, Ultrabook™ computing device, wearable devices (such as a smart watch or smart glasses), etc.

System Overview

Fig. 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.

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

In some embodiments, the one or more processors 102 each include one or more processor cores 107 to process instructions which, when executed, perform operations for system and user software. In some embodiments, each of the one or more processor cores 107 is configured to process a specific instruction set 109. In some embodiments, instruction set 109 may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a Very Long Instruction Word (VLIW). Multiple processor cores 107 may each process a different instruction set 109, which may include instructions to facilitate the emulation of other instruction sets. Processor core 107 may also include other processing devices, such a Digital Signal Processor (DSP).

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

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

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

In some embodiments, ICH 130 enables peripherals to connect to memory device 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller 146, a firmware interface 128, a wireless transceiver 126 (e.g., Wi-Fi, Bluetooth), a data storage device 124 (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller 140 for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 142 connect input devices, such as keyboard and mouse 144 combinations. A network controller 134 may also couple with ICH 130. In some embodiments, a high-performance network controller (not shown) couples with 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.

Fig. 2 is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-202N, an integrated memory controller 214, and an integrated graphics processor 208. Those elements of **Fig. 2** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. Processor 200 can include additional cores up to and including additional core 202N represented by the dashed lined boxes. Each of processor cores 202A-202N includes one or more internal cache units 204A-204N. In some embodiments each processor core also has access to one or more shared cached units 206.

The internal cache units 204A-204N and shared cache units 206 represent a cache memory hierarchy within the processor 200. The cache memory hierarchy may include at least one level of instruction and data cache within each processor core and one or more levels of shared mid-level cache, such as a Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the LLC. In some embodiments, cache coherency logic maintains coherency between the various cache units 206 and 204A-204N.

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

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

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

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

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

In some embodiments, processor cores 202A-202N are homogenous cores executing the same instruction set architecture. In another embodiment, processor cores 202A-202N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores 202A-202N 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.

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

interface 314 to access memory. Memory interface 314 can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

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

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

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

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

In some embodiments, 3D/Media subsystem 315 includes logic for executing threads spawned by 3D pipeline 312 and media pipeline 316. In one embodiment, the pipelines send

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

Graphics Processing Engine

Fig. 4 is a block diagram of a graphics processing engine 410 of a graphics processor in accordance with some embodiments. In one embodiment, the graphics processing engine (GPE) 410 is a version of the GPE 310 shown in **Fig. 3**. Elements of **Fig. 4** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. For example, the 3D pipeline 312 and media pipeline 316 of **Fig. 3** are illustrated. The media pipeline 316 is optional in some embodiments of the GPE 410 and may not be explicitly included within the GPE 410. For example, in at least one embodiment, a separate media and/or image processor is coupled to the GPE 410.

In some embodiments, GPE 410 couples with or includes a command streamer 403, which provides a command stream to the 3D pipeline 312 and/or media pipelines 316. In some embodiments, command streamer 403 is coupled with 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 312 and/or media pipeline 316. The commands are directives fetched from a ring buffer, which stores commands for the 3D pipeline 312 and media pipeline 316. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The commands for the 3D pipeline 312 can also include references to data stored in memory, such as but not limited to vertex and geometry data for the 3D pipeline 312 and/or image data and memory objects for the media pipeline 316. The 3D pipeline 312 and media pipeline 316 process the commands and data by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to a graphics core array 414.

In various embodiments the 3D pipeline 312 can execute one or more shader programs, such as vertex shaders, geometry shaders, pixel shaders, fragment shaders, compute shaders, or other shader programs, by processing the instructions and dispatching execution threads to the graphics core array 414. The graphics core array 414 provides a unified block of

execution resources. Multi-purpose execution logic (e.g., execution units) within the graphic core array 414 includes support for various 3D API shader languages and can execute multiple simultaneous execution threads associated with multiple shaders.

In some embodiments the graphics core array 414 also includes execution logic to perform media functions, such as video and/or image processing. In one embodiment, the execution units additionally include general-purpose logic that is programmable to perform parallel general purpose computational operations, in addition to graphics processing operations. The general purpose logic can perform processing operations in parallel or in conjunction with general purpose logic within the processor core(s) 107 of **Fig. 1** or core 202A-202N as in **Fig. 2**.

Output data generated by threads executing on the graphics core array 414 can output data to memory in a unified return buffer (URB) 418. The URB 418 can store data for multiple threads. In some embodiments the URB 418 may be used to send data between different threads executing on the graphics core array 414. In some embodiments the URB 418 may additionally be used for synchronization between threads on the graphics core array and fixed function logic within the shared function logic 420.

In some embodiments, graphics core array 414 is scalable, such that the array includes a variable number of graphics cores, each having a variable number of execution units based on the target power and performance level of GPE 410. In one embodiment the execution resources are dynamically scalable, such that execution resources may be enabled or disabled as needed.

The graphics core array 414 couples with shared function logic 420 that includes multiple resources that are shared between the graphics cores in the graphics core array. The shared functions within the shared function logic 420 are hardware logic units that provide specialized supplemental functionality to the graphics core array 414. In various embodiments, shared function logic 420 includes but is not limited to sampler 421, math 422, and inter-thread communication (ITC) 423 logic. Additionally, some embodiments implement one or more cache(s) 425 within the shared function logic 420. A shared function is implemented where the demand for a given specialized function is insufficient for inclusion within the graphics core array 414. Instead a single instantiation of that specialized function is implemented as a stand-alone entity in the shared function logic 420 and shared among the execution resources within the graphics core array 414. The precise set of functions that are shared between the graphics core array 414 and included within the graphics core array 414 varies between embodiments.

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

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

In some embodiments, graphics processor 500 receives batches of commands via ring interconnect 502. The incoming commands are interpreted by a command streamer 503 in the pipeline front-end 504. In some embodiments, graphics processor 500 includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) 580A-580N. For 3D geometry processing commands, command streamer 503 supplies commands to geometry pipeline 536. For at least some media processing commands, command streamer 503 supplies the commands to a video front end 534, which couples with a media engine 537. In some embodiments, media engine 537 includes a Video Quality Engine (VQE) 530 for video and image post-processing and a multi-format encode/decode (MFX) 533 engine to provide hardware-accelerated media data encode and decode. In some embodiments, geometry pipeline 536 and media engine 537 each generate execution threads for the thread execution resources provided by at least one graphics core 580A.

In some embodiments, graphics processor 500 includes scalable thread execution resources featuring modular cores 580A-580N (sometimes referred to as core slices), each having multiple sub-cores 550A-550N, 560A-560N (sometimes referred to as core sub-slices). In some embodiments, graphics processor 500 can have any number of graphics cores 580A through 580N. In some embodiments, graphics processor 500 includes a graphics core 580A having at least a first sub-core 550A and a second 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.

Execution Units

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

In some embodiments, thread execution logic 600 includes a shader processor 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 scalable execution unit array can dynamically scale by enabling or disabling one or more execution units (e.g., any of execution unit 608A, 608B, 608C, 608D, through 608N-1 and 608N) based on the computational requirements of a workload. 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 units 608A-608N. In some embodiments, each execution unit (e.g. 608A) is a stand-alone programmable general purpose computational unit that is capable of executing multiple simultaneous hardware threads while processing multiple data elements in parallel for each thread. In various embodiments, the array of execution units 608A-608N is scalable to include any number individual execution units.

In some embodiments, the execution units 608A-608N are primarily used to execute shader programs. A shader processor 602 can process the various shader programs and dispatch execution threads associated with the shader programs via a thread dispatcher 604. In one embodiment the thread dispatcher includes logic to arbitrate thread initiation requests from the graphics and media pipelines and instantiate the requested threads on one or more execution unit in the execution units 608A-608N. For example, the geometry pipeline (e.g., 536 of **Fig. 5**) can dispatch vertex, tessellation, or geometry shaders to the thread execution logic 600 (**Fig. 6**) for processing. In some embodiments, thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

In some embodiments, the execution units 608A-608N support an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal

translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders). Each of the execution units 608A-608N is capable of multi-issue single instruction multiple data (SIMD) execution and multi-threaded operation enables an efficient execution environment in the face of higher latency memory accesses. Each hardware thread within each execution unit has a dedicated high-bandwidth register file and associated independent thread-state. Execution is multi-issue per clock to pipelines capable of integer, single and double precision floating point operations, SIMD branch capability, logical operations, transcendental operations, and other miscellaneous operations. While waiting for data from memory or one of the shared functions, dependency logic within the execution units 608A-608N causes a waiting thread to sleep until the requested data has been returned. While the waiting thread is sleeping, hardware resources may be devoted to processing other threads. For example, during a delay associated with a vertex shader operation, an execution unit can perform operations for a pixel shader, fragment shader, or another type of shader program, including a different vertex shader.

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

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

One or more internal instruction caches (e.g., 606) are included in the thread execution logic 600 to cache thread instructions for the execution units. In some embodiments, one or more data caches (e.g., 612) are included to cache thread data during thread execution. In

some embodiments, a sampler 610 is included to provide texture sampling for 3D operations and media sampling for media operations. In some embodiments, sampler 610 includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 600 via thread spawning and dispatch logic. Once a group of geometric objects has been processed and rasterized into pixel data, pixel processor logic (e.g., pixel shader logic, fragment shader logic, etc.) within the shader processor 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, a pixel shader or fragment shader calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. In some embodiments, pixel processor logic within the shader processor 602 then executes an application programming interface (API)-supplied pixel or fragment shader program. To execute the shader program, the shader processor 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 the sampler 610 to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

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

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

In some embodiments, the graphics processor execution units natively support instructions in a 128-bit instruction 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 instruction 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 instruction format 710.

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

Some execution unit instructions have up to three operands including two source operands, src0 720, src1 722, and one destination 718. In some embodiments, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 724), where the instruction opcode 712 determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

In some embodiments, the 128-bit instruction format 710 includes an access/address mode field 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.

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

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 directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed based on an address register value and an address immediate field in the instruction.

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

Graphics Pipeline

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

In some embodiments, graphics processor 800 includes a graphics pipeline 820, a media pipeline 830, a display engine 840, thread execution logic 850, and a render output pipeline 870. In some embodiments, graphics processor 800 is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor 800 via a ring interconnect 802. In

some embodiments, ring interconnect 802 couples graphics processor 800 to other processing components, such as other graphics processors or general-purpose processors. Commands from ring interconnect 802 are interpreted by a command streamer 803, which supplies instructions to individual components of graphics pipeline 820 or media pipeline 830.

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

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

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

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

Before rasterization, a clipper 829 processes vertex data. The clipper 829 may be a fixed function clipper or a programmable clipper having clipping and geometry shader

functions. In some embodiments, a rasterizer and depth test component 873 in the render output pipeline 870 dispatches pixel shaders to convert the geometric objects into their per pixel representations. In some embodiments, pixel shader logic is included in thread execution logic 850. In some embodiments, an application can bypass the rasterizer and depth test component 873 and access un-rasterized vertex data via a stream out unit 823.

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

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

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

In some embodiments, graphics processor 800 includes a display engine 840. In some embodiments, display engine 840 is external to processor 800 and couples with the graphics processor via the ring interconnect 802, or some other interconnect bus or fabric. In some embodiments, display engine 840 includes a 2D engine 841 and a display controller 843. In some embodiments, display engine 840 contains special purpose logic capable of operating independently of the 3D pipeline. In some embodiments, display controller 843 couples with

a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

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

Graphics Pipeline Programming

Fig. 9A is a block diagram illustrating a graphics processor command format 900 according to some embodiments. **Fig. 9B** is a block diagram illustrating a graphics processor command sequence 910 according to an embodiment. The solid lined boxes in **Fig. 9A** illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format 900 of **Fig. 9A** includes data fields to identify a target client 902 of the command, a command operation code (opcode) 904, and the relevant data 906 for the command. A sub-opcode 905 and a command size 908 are also included in some commands.

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

some embodiments, the command parser automatically determines the size of at least some of the commands based on the command opcode. In some embodiments commands are aligned via multiples of a double word.

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

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

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

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

In some embodiments, return buffer state commands 916 are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the

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

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

The commands to configure 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 on the particular 3D API in use. In some embodiments, 3D pipeline state 930 commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

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

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

In some embodiments, the graphics processor command sequence 910 follows the media pipeline 924 path when performing media operations. In general, the specific use and

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

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

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

Graphics Software Architecture

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

In some embodiments, 3D graphics application 1010 contains one or more shader programs including shader instructions 1012. The shader language instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL

Shader Language (GLSL). The application also includes executable instructions 1014 in a machine language suitable for execution by the general-purpose processor core 1034. The application also includes graphics objects 1016 defined by vertex data.

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

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

IP Core Implementations

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

may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

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

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

Exemplary System on a Chip Integrated Circuit

Figs. 12-14 illustrated exemplary integrated circuits and associated graphics processors that may be fabricated using one or more IP cores, according to various embodiments described herein. In addition to what is illustrated, other logic and circuits may be included, including additional graphics processors/cores, peripheral interface controllers, or general purpose processor cores.

Fig. 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.

Exemplary integrated circuit 1200 includes one or more application processor(s) 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. Integrated circuit 1200 includes peripheral or bus logic including a USB controller 1225, UART controller 1230, an SPI/SDIO controller 1235, and an I^2S/I^2C controller 1240. Additionally, the integrated circuit can include a display device 1245 coupled to one or more of a high-definition multimedia interface (HDMI) controller 1250 and a mobile industry processor interface (MIPI) display interface 1255. Storage may be provided by a flash memory subsystem 1260 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller 1265 for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine 1270.

Fig. 13 is a block diagram illustrating an exemplary graphics processor 1310 of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor 1310 can be a variant of the graphics processor 1210 of **Fig. 12**. Graphics processor 1310 includes a vertex processor 1305 and one or more fragment processor(s) 1315A1315N (e.g., 1315A, 1315B, 1315C, 1315D, through 1315N-1, and 1315N). Graphics processor 1310 can execute different shader programs via separate logic, such that the vertex processor 1305 is optimized to execute operations for vertex shader programs, while the one or more fragment processor(s) 1315A-1315N execute fragment (e.g., pixel) shading operations for fragment or pixel shader programs. The vertex processor 1305 performs the vertex processing stage of the 3D graphics pipeline and generates primitives and vertex data. The fragment processor(s) 1315A-1315N use the primitive and vertex data generated by the vertex processor 1305 to produce a framebuffer that is displayed on a display device. In one embodiment, the fragment processor(s) 1315A-1315N are optimized to execute fragment shader programs as provided for in the OpenGL API, which may be used to perform similar operations as a pixel shader program as provided for in the Direct 3D API.

Graphics processor 1310 additionally includes one or more memory management units (MMUs) 1320A-1320B, cache(s) 1325A-1325B, and circuit interconnect(s) 1330A-1330B. The one or more MMU(s) 1320A-1320B provide for virtual to physical address mapping for integrated circuit 1310, including for the vertex processor 1305 and/or fragment processor(s) 1315A-1315N, which may reference vertex or image/texture data stored in memory, in addition to vertex or image/texture data stored in the one or more cache(s) 1325A-1325B. In one embodiment the one or more MMU(s) 1325A-1325B may be synchronized with other

MMUs within the system, including one or more MMUs associated with the one or more application processor(s) 1205, image processor 1215, and/or video processor 1220 of **Fig. 12**, such that each processor 1205-1220 can participate in a shared or unified virtual memory system. The one or more circuit interconnect(s) 1330A-1330B enable graphics processor 1310 to interface with other IP cores within the SoC, either via an internal bus of the SoC or via a direct connection, according to embodiments.

Fig. 14 is a block diagram illustrating an additional exemplary graphics processor 1410 of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor 1410 can be a variant of the graphics processor 1210 of **Fig. 12**. Graphics processor 1410 includes the one or more MMU(s) 1320A-1320B, caches 1325A-1325B, and circuit interconnects 1330A-1330B of the integrated circuit 1300 of **Fig. 13**.

Graphics processor 1410 includes one or more shader core(s) 1415A-1415N (e.g., 1415A, 1415B, 1415C, 1415D, 1415E, 1415F, through 1315N-1, and 1315N), which provides for a unified shader core architecture in which a single core or type or core can execute all types of programmable shader code, including shader program code to implement vertex shaders, fragment shaders, and/or compute shaders. The exact number of shader cores present can vary among embodiments and implementations. Additionally, graphics processor 1410 includes an inter-core task manager 1405, which acts as a thread dispatcher to dispatch execution threads to one or more shader cores 1415A-1415N and a tiling unit 1418 to accelerate tiling operations for tile-based rendering, in which rendering operations for a scene are subdivided in image space, for example to exploit local spatial coherence within a scene or to optimize use of internal caches.

As mentioned above, when there is an insufficient number of threads to hide the latency to memory, or there is an insufficient amount of cache to hold all the constants, these stalls can degrade the performance of the graphics operations. Most GPUs attempt to solve these problem by having a large number of parallel threads executing the same program in order to hide the latency, as well as local caches to store and reuse the uniform/constants data after it has been read from memory. However, smaller GPU have fewer threads available to hide the latency and less area to afford caches large enough to maintain the uniform data. By contrast, the embodiments described herein do not rely on having a large number of threads or a large amount of cache and therefore scale better across small and large GPUs.

Moreover, some GPUs may utilize a fixed-function hardware-based solution for pre-gathering uniform data from multiple memory locations and storing them into a single, contiguous memory buffer. Upon the GPU receiving a drawing operation, that buffer is then

copied into the GPU's Unified Return Buffer (URB) (an on-die local memory for storing data passed between program stages) or equivalent. Upon thread dispatch for a program, the URB data is pushed into the Execution Unit's (EU) General Register File (GRF) for immediate use by the thread running the program. The first part, the gathering of uniform data into a buffer, may be performed using a GPU unit called the Resource Streamer. This unit can be inefficient and limited in programmability. Hence some implementations fail to utilize the full bandwidth of the GPU.

By contrast, the techniques described herein may utilize the full bandwidth of the GPU and are fully programmable. The second part, the pushing of memory into the URB and then into the GRF, is a method referred to as Push Constants. This method (or a subset of this method) may be used in one or more embodiments. For example, one or more embodiments utilize two variations. The first embodiment, referred to herein as "shader-based gather", takes advantage of features in some GPUs. The second idea, referred to herein as "draw shaders", extends the capability of the GPU in future hardware.

More particularly, **Fig. 15** illustrates flow of uniform data to GRF using shader-based gathering, according to an embodiment. More particularly, shader-based gathering utilizes features of the GPU by executing a GPU program that gathers the uniform data from memory and stores the results into a (e.g., contiguous) memory buffer. The GPU program may execute once per-Draw (e.g., for 3D) operation or per-Dispatch (e.g., for GPGPU or General Purpose GPU) operation. Multiple of these GPU programs can also be executed in batches to increase the performance of the GPU. For example, these batches can gather all the uniform data for an entire render pass, therefore eliminating the additional memory bandwidth for reading uniform data for Tile-Based Rendering (TBR) GPUs. The resulting buffers can then be pushed into the GRF, as described below.

Referring to Fig. 15, constant buffer(s) 1502 (e.g., containing the uniform data) and gather table 1504 (e.g., describing one or more locations of the uniform data in the constant buffer(s) and the order in which each portion of the uniform data should be pushed into the GRF) are used by gather shader logic 1506 to perform gather shader operation(s). The gather table 1504 is created by the GPU shader compiler in an embodiment. The gather shader logic 1506 reads the uniform data from the constant buffers 1502 and stores the results in the push buffer 1508, based on information stored in the gather table 1504. In an embodiment, software (e.g., logic) performs operations discussed with reference to items 1502-1508.

Logic 1510 pushes/transfers the uniform data in the (e.g., contiguous) buffer 1508 to on-die memory 1512 (e.g., where "on-die" memory refers to memory on the same die as the logic and/or the processor EU that executes the operation(s)). In an embodiment, a fixed-

function hardware unit (e.g., 1510) copies the data from the push buffer 1508 to the on-die memory 1512. The on-die memory 1512 (such as the URB) stores the uniform data that should be pushed to the GRF, e.g., when a thread is dispatched to the EU. In an embodiment, a fixed function hardware unit (e.g., logic 1514) copies the data from the on-die memory 1512 to the GRF 1516 when the thread is dispatched. The GRF data may then be used by the thread when executing the program on the EU. In an embodiment, hardware (e.g., logic) performs operations discussed with reference to items 1510-1516.

Fig. 16 illustrates sample pseudocode for a gather shader program, according to an embodiment. More particularly, one or more lines of code are described below:

Line 1: 'cbPerObject' contains the Gather Table

Line 8: 'ConstantBuffers' are the application supplied Constant Buffers, containing uniform data

Line 10: 'BufferPushed' is the output buffer that will be pushed into the GRF

Line 12: 'BITS' function extracts data from the compressed Gather Table

Line 20: for each 16Byte uniform value that we want to gather

Line 22: Get the Gather Table value for the current uniform

Line 23: Extract the index into the application's buffer

Line 24: Extract the index to the application's buffer

Line 25: Read the 16Byte value from memory

Line 26: Store the 16Byte value to the output buffer

For simplicity of illustration, this program only reads 16Byte values. However, it is possible to gather variable sized values by extending the gather table 1504 to include the size.

Fig. 17 illustrates a sequence of GPU commands for executing gather shaders using batch processing, according to an embodiment. More particularly, a second-level batch buffer 1702 executes a series of GPGPU commands that execute the gather shader for each shader stage, for each draw operation. A first-level batch buffer 1704 executes a series of draw operations, preceded by requests to push data into the GRF. As shown in Fig. 17, a plurality of constant buffer/gather table sets may be present, as well as push buffers.

As discussed herein, PIPE_SELECT determines whether the machine is configured for 3D or Media/GPGPU operations (see, e.g., Fig. 9B). GPGPU_WALKER is an instruction to execute a compute shader. BBS or Batch Buffer Start is an instruction for submitting commands to the command streamer. 3DSTATE_VS is an instruction for programming the

state of the vertex shader unit, which can include the pointer to the vertex shader kernel. 3DSTATE_PS may be the same as VS except for the Pixel Shader. CONSTANT_BUFFER_VS is an instruction to fetch data from the push buffer to be preloaded into the GRF when the vertex shader program thread is spawned. CONSTANT_BUFFER_PS may be the same as VS except for the Pixel Shader. 3DPRIM is an instruction to submit primitive(s) to the 3D pipeline.

Fig. 18 illustrates flow for batching gather shaders of Fig. 17, according to an embodiment. In one embodiments, one or more of the operations of Fig. 17 are performed by a driver (or logic) discussed herein. At operation 1802, the sequence is executed for each draw or dispatch request sent to the driver that is to perform the batch gather shader operations. At an operation 1804, it is determined whether the contents of the source constant buffer are dependent on a previous GPU operation. At operation 1806, if there are any dependencies, the gather operation follows that GPU operation (e.g., closing/terminating any pending/existing batch). At operation 1808, it is determined whether there is an existing open batch for appending new gather operations. If not, operation 1810 opens a new batch immediately before the draw or dispatch operation. At an operation 1812, the gather operation is appended to the batch.

Fig. 19 illustrates a sequence of GPU commands for batching gather shaders (e.g., prior to a TBR render pass), according to an embodiment. A second-level batch buffer 1902 executes a series of GPGPU commands that execute the gather shader for each shader stage, for each draw operation, for the entire render pass. A first-level batch buffer 1904 executes a series of per-tile batch buffers, containing per-tile draw operations, preceded by requests to push data into the GRF.

Generally, draw shaders are GPU programs that are executed once per-Draw operation. This could be a program that is executed before a vertex shader program; or it could be implemented as coarse vertex shaders. One advantage of draw shaders is that it does not require an additional memory buffer for storing the uniform data. The draw shader would read all the uniform data and pass the data to the following higher frequency programs, e.g., through the URB. Draw shaders could also be exposed to applications. One downside of draw shaders is that they only apply to 3D operations and they do not solve the TBR bandwidth issue.

Fig. 20 illustrates flow of Uniform data to GRF using draw shader, according to an embodiment. A constant buffer memory 2002 contains the uniform data. A draw shader logic 2004 that reads the uniform data from the constant buffer(s) 2002 and stores the results in on-die memory 2006. The on-die memory 2006 (such as the URB) stores the uniform data

that should be pushed into the GRF when a thread is dispatched to the EU. Logic 2008 (e.g., fixed function hardware unit) copies the data from the on-die memory to the GRF 2010 when the thread is dispatched. The GRF may then be used by the thread when executing the program on the EU. In an embodiment, software (e.g., logic) performs operations discussed with reference to items 2002 and 2004. Also, in one embodiment, hardware (e.g., logic) performs operations discussed with reference to items 2006-2010.

Fig. 21 illustrates a sample pseudocode for a draw shader program, according to an embodiment. More particularly, one or more lines of code are described below:

Line 1: 'cbPerObject' contains the uniform values across the entire draw operation

Line 7: 'DR_OUTPUT' is the data structure of the data generated by the draw shader

Line 17: 'VS_INPUT' is the per-vertex data structure input into the Vertex Shader; this may be generated by fixed-function hardware, called Vertex Fetch

Line 27: the draw shader function; executes once per Draw call

Lines 31-34: reads the uniform data from memory; outputs the data to the Vertex Shader

Line 37: the Vertex Shader function; executes once per Vertex. Inputs to the Vertex Shader include the output from the draw shader and the output from Vertex Fetch

Lines 41-42: uses both the per-vertex data and the per-draw data to calculate the vertex position and normal

The following examples pertain to further embodiments. Example 1 may optionally include an apparatus comprising: first logic, at least a portion of which is in hardware, to transfer uniform data from a buffer to a General Register File (GRF) of a processor based at least in part on information stored in a gather table, wherein the uniform data is to comprise data that is uniform across a plurality of primitives in a graphics operation. Example 2 may optionally include the apparatus of example 1, wherein the buffer is to comprise a Unified Return Buffer (URB) of the processor. Example 3 may optionally include the apparatus of example 2, wherein the first logic is to transfer the uniform data from the URB to the GRF. Example 4 may optionally include the apparatus of example 2, wherein the transferred uniform data is to be passed to one or more higher frequency operations. Example 5 may optionally include the apparatus of example 2, wherein the URB is to store data corresponding to a plurality of threads executing on the processor. Example 6 may optionally include the apparatus of example 1, wherein the information stored in the gather table is to

comprise: information corresponding to one or more locations of one or more portions of the uniform data in one or more constant buffers and an order in which the one or more portions of the stored uniform data are to be transferred to the GRF. Example 7 may optionally include the apparatus of example 1, wherein on-die memory is to comprise the buffer. Example 8 may optionally include the apparatus of example 1, comprising second logic to gather the uniform data from a plurality of locations in memory and to store the gathered uniform data in the buffer. Example 9 may optionally include the apparatus of example 8, wherein the second logic is to perform the gather and store operations in batches. Example 10 may optionally include the apparatus of example 8, wherein the batches are to cause gathering of all uniform data for an entire render pass. Example 11 may optionally include the apparatus of example 8, wherein the second logic is to perform the gather and store operations once per draw for three-dimensional graphics operations. Example 12 may optionally include the apparatus of example 8, wherein the second logic is to perform the gather and store operations once per dispatch for general purpose operations. Example 13 may optionally include the apparatus of example 1, wherein the buffer is to store a plurality of portions of the uniform data contiguously. Example 14 may optionally include the apparatus of example 1, wherein the processor is to comprise one or more of: a Graphics Processing Unit (GPU) and a processor core. Example 15 may optionally include the apparatus of example 14, wherein the GPU is to comprise one or more graphics processing cores. Example 16 may optionally include the apparatus of example 1, wherein the processor is to comprise one or more processor cores. Example 17 may optionally include the apparatus of example 1, wherein the processor is to comprise at least a portion of the buffer. Example 18 may optionally include the apparatus of example 1, wherein the processor and the buffer are on a single integrated circuit die.

Example 19 may optionally include a method comprising: transferring uniform data from a buffer to a General Register File (GRF) of a processor based at least in part on information stored in a gather table, wherein the uniform data comprises data that is uniform across a plurality of primitives in a graphics operation. Example 20 may optionally include the method of example 19, wherein the buffer comprises a Unified Return Buffer (URB) of the processor. Example 21 may optionally include the method of example 20, further comprising transferring the uniform data from the URB to the GRF. Example 22 may optionally include the method of example 20, further comprising passing the transferred uniform data to one or more higher frequency operations. Example 23 may optionally include the method of example 20, further comprising the URB storing data corresponding to a plurality of threads executing on the processor. Example 24 may optionally include the

method of example 19, wherein the information stored in the gather table comprises: information corresponding to one or more locations of one or more portions of the uniform data in one or more constant buffers and an order in which the one or more portions of the stored uniform data are to be transferred to the GRF. Example 25 may optionally include the method of example 19, further comprising gathering the uniform data from a plurality of locations in memory and to storing the gathered uniform data in the buffer. Example 26 may optionally include the method of example 25, further comprising performing the gather and store operations in batches. Example 27 may optionally include the method of example 26, wherein the batches are to cause gathering of all uniform data for an entire render pass. Example 28 may optionally include the method of example 25, further comprising performing the gather and store operations once per draw for three-dimensional graphics operations. Example 29 may optionally include the method of example 25, further comprising performing the gather and store operations once per dispatch for general purpose operations.

Example 30 includes one more computer-readable medium comprising one or more instructions that when executed on at least one processor configure the at least one processor to perform one or more operations to: transfer uniform data from a buffer to a General Register File (GRF) of the processor based at least in part on information stored in a gather table, wherein the uniform data comprises data that is uniform across a plurality of primitives in a graphics operation. Example 31 may optionally include the one or more computer-readable medium of example 30, wherein the buffer comprises a Unified Return Buffer (URB) of the processor. Example 32 may optionally include the one or more computer-readable medium of example 30, wherein the information stored in the gather table comprises: information corresponding to one or more locations of one or more portions of the uniform data in one or more constant buffers and an order in which the one or more portions of the stored uniform data are to be transferred to the GRF.

Example 33 may optionally include an apparatus comprising means to perform a method as set forth in any preceding example. Example 34 comprises machine-readable storage including machine-readable instructions, when executed, to implement a method or realize an apparatus as set forth in any preceding example.

In various embodiments, the operations discussed herein, e.g., with reference to Figs. 1-21, may be implemented as hardware (e.g., logic circuitry), software, firmware, or combinations thereof, which may be provided as a computer program product, e.g., including a tangible (e.g., non-transitory) machine-readable or computer-readable medium having stored thereon instructions (or software procedures) used to program a computer to perform

a process discussed herein. The machine-readable medium may include a storage device such as those discussed with respect to Figs. 1-21.

Additionally, such computer-readable media may be downloaded as a computer program product, wherein the program may be transferred from a remote computer (e.g., a server) to a requesting computer (e.g., a client) by way of data signals provided in a carrier wave or other propagation medium via a communication link (e.g., a bus, a modem, or a network connection).

Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, and/or characteristic described in connection with the embodiment may be included in at least an implementation. The appearances of the phrase "in one embodiment" in various places in the specification may or may not be all referring to the same embodiment.

Also, in the description and claims, the terms "coupled" and "connected," along with their derivatives, may be used. In some embodiments, "connected" may be used to indicate that two or more elements are in direct physical or electrical contact with each other. "Coupled" may mean that two or more elements are in direct physical or electrical contact. However, "coupled" may also mean that two or more elements may not be in direct contact with each other, but may still cooperate or interact with each other.

Thus, although embodiments have been described in language specific to structural features and/or methodological acts, it is to be understood that claimed subject matter may not be limited to the specific features or acts described. Rather, the specific features and acts are disclosed as sample forms of implementing the claimed subject matter.

CLAIMS

1. An apparatus comprising:

first logic, at least a portion of which is in hardware, to transfer uniform data from a buffer to a General Register File (GRF) of a processor based at least in part on information stored in a gather table,

wherein the uniform data is to comprise data that is uniform across a plurality of primitives in a graphics operation.

2. The apparatus of claim 1, wherein the buffer is to comprise a Unified Return Buffer (URB) of the processor.

3. The apparatus of claim 2, wherein the first logic is to transfer the uniform data from the URB to the GRF.

4. The apparatus of claim 2, wherein the transferred uniform data is to be passed to one or more higher frequency operations.

5. The apparatus of claim 2, wherein the URB is to store data corresponding to a plurality of threads executing on the processor.

6. The apparatus of claim 1, wherein the information stored in the gather table is to comprise: information corresponding to one or more locations of one or more portions of the uniform data in one or more constant buffers and an order in which the one or more portions of the stored uniform data are to be transferred to the GRF.

7. The apparatus of claim 1, wherein on-die memory is to comprise the buffer.

8. The apparatus of claim 1, comprising second logic to gather the uniform data from a plurality of locations in memory and to store the gathered uniform data in the buffer.

9. The apparatus of claim 8, wherein the second logic is to perform the gather and store operations in batches.

10. The apparatus of claim 9, wherein the batches are to cause gathering of all uniform data for an entire render pass.

11. The apparatus of claim 8, wherein the second logic is to perform the gather and store operations once per draw for three-dimensional graphics operations.

12. The apparatus of claim 8, wherein the second logic is to perform the gather and store operations once per dispatch for general purpose operations.

13. The apparatus of claim 1, wherein the buffer is to store a plurality of portions of the uniform data contiguously.

14. The apparatus of claim 1, wherein the processor is to comprise one or more of: a Graphics Processing Unit (GPU) and a processor core.

15. The apparatus of claim 14, wherein the GPU is to comprise one or more graphics processing cores.

16. The apparatus of claim 1, wherein the processor is to comprise one or more processor cores.

17. The apparatus of claim 1, wherein the processor is to comprise at least a portion of the buffer.

18. The apparatus of claim 1, wherein the processor and the buffer are on a single integrated circuit die.

19. A method comprising:

transferring uniform data from a buffer to a General Register File (GRF) of a processor based at least in part on information stored in a gather table,

wherein the uniform data comprises data that is uniform across a plurality of primitives in a graphics operation.

20. The method of claim 19, wherein the buffer comprises a Unified Return Buffer (URB) of the processor.

21. The method of claim 19, wherein the information stored in the gather table comprises: information corresponding to one or more locations of one or more portions of the uniform data in one or more constant buffers and an order in which the one or more portions of the stored uniform data are to be transferred to the GRF.

22. The method of claim 19, further comprising gathering the uniform data from a plurality of locations in memory and to storing the gathered uniform data in the buffer.

23. An apparatus comprising means to perform a method as set forth in any preceding claim.

24. Machine-readable storage including machine-readable instructions, when executed, to implement a method or realize an apparatus as set forth in any preceding claim.

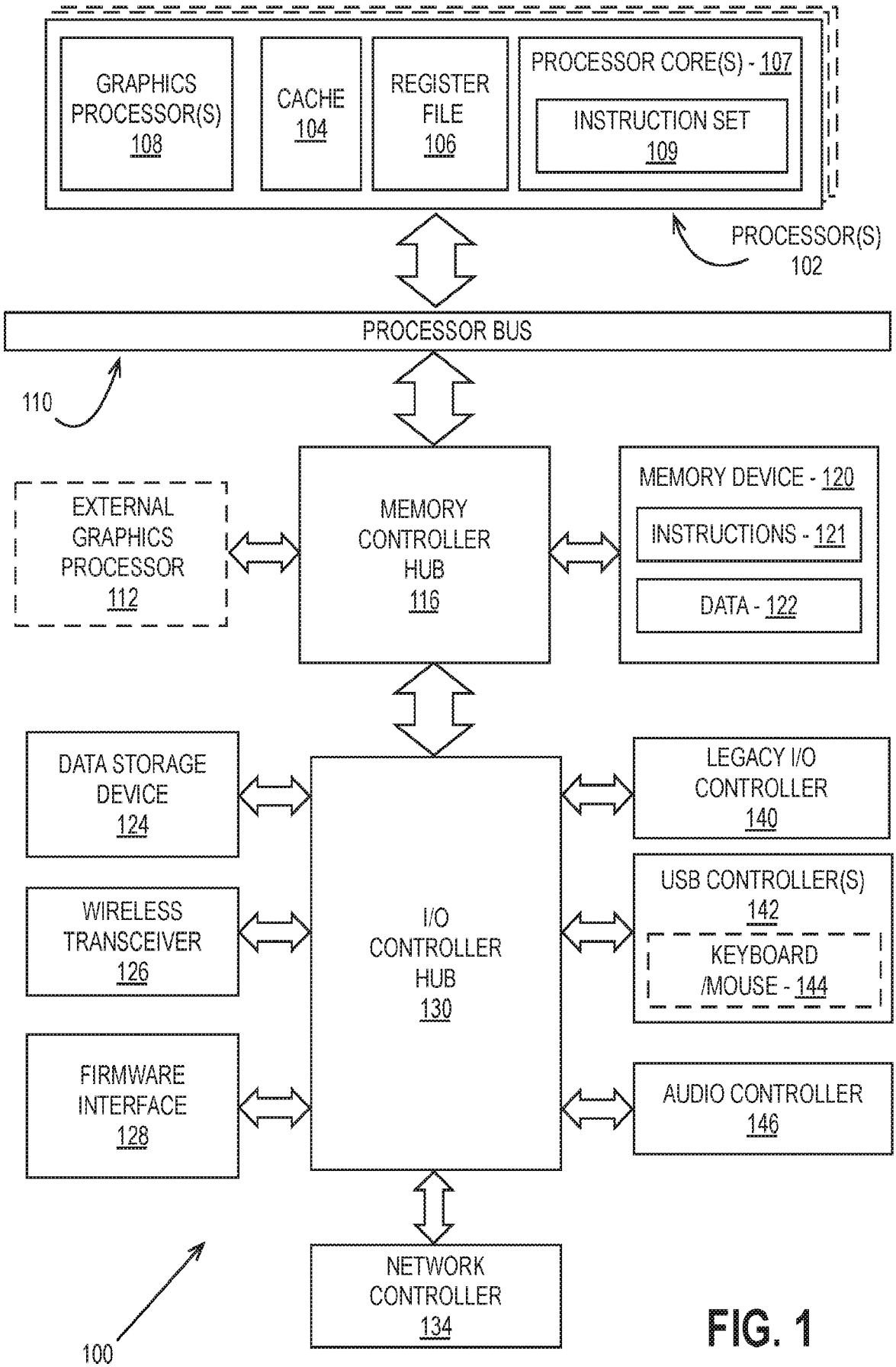

GRAPHICS PROCESSOR(S) 108
CACHE 104
REGISTER FILE 106
PROCESSOR CORE(S) - 107
INSTRUCTION SET 109
PROCESSOR(S) 102
PROCESSOR BUS
110
EXTERNAL GRAPHICS PROCESSOR 112
MEMORY CONTROLLER HUB 116
MEMORY DEVICE - 120
INSTRUCTIONS - 121
DATA - 122
DATA STORAGE DEVICE 124
WIRELESS TRANSCEIVER 126
FIRMWARE INTERFACE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

FIG. 1

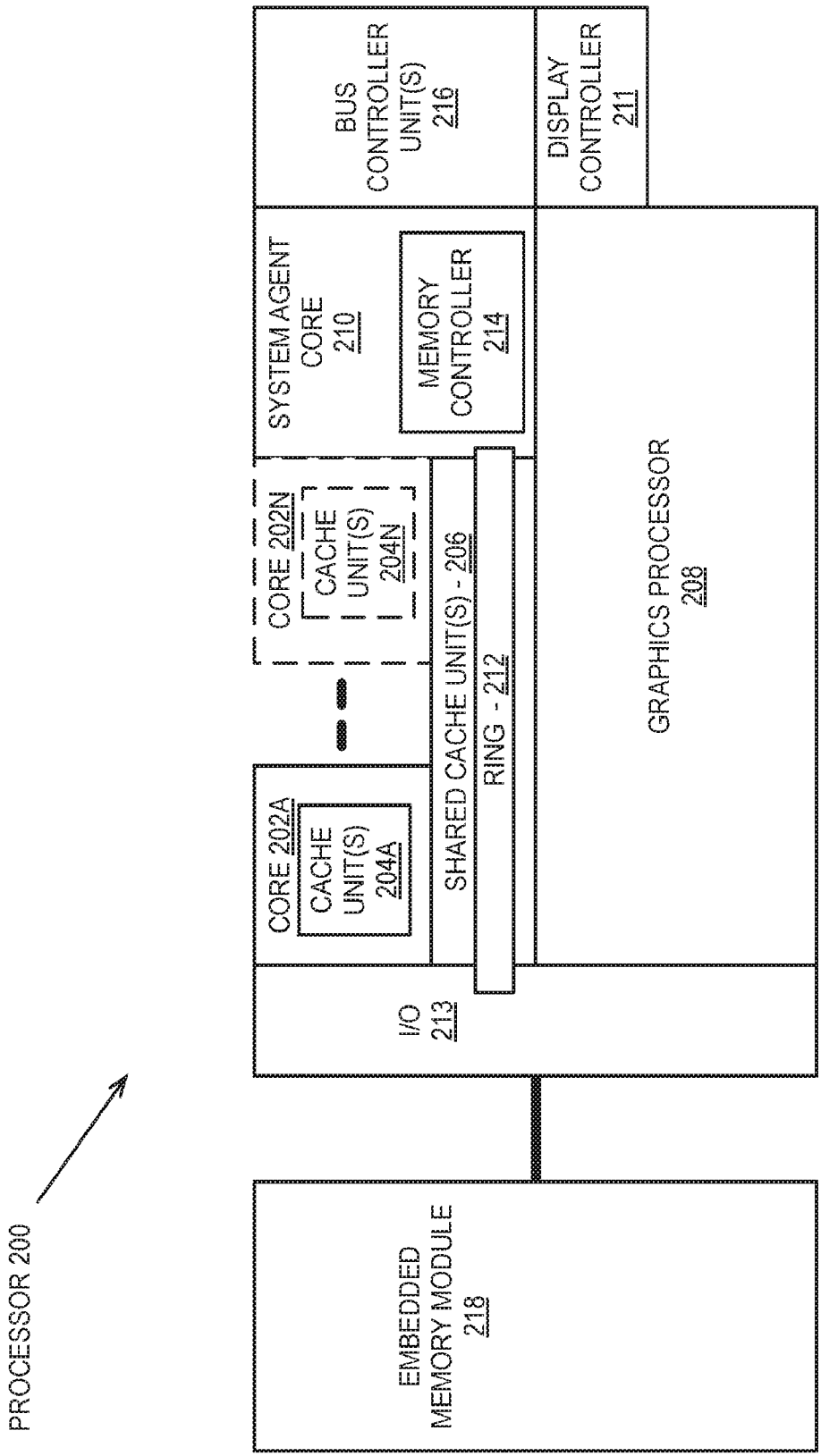

PROCESSOR 200
EMBEDDED MEMORY MODULE 218
I/O 213
CORE 202A
CACHE UNIT(S) 204A
CORE 202N
CACHE UNIT(S) 204N
SHARED CACHE UNIT(S) - 206
RING - 212
GRAPHICS PROCESSOR 208
SYSTEM AGENT CORE 210
MEMORY CONTROLLER 214
BUS CONTROLLER UNIT(S) 216
DISPLAY CONTROLLER 211

FIG. 2

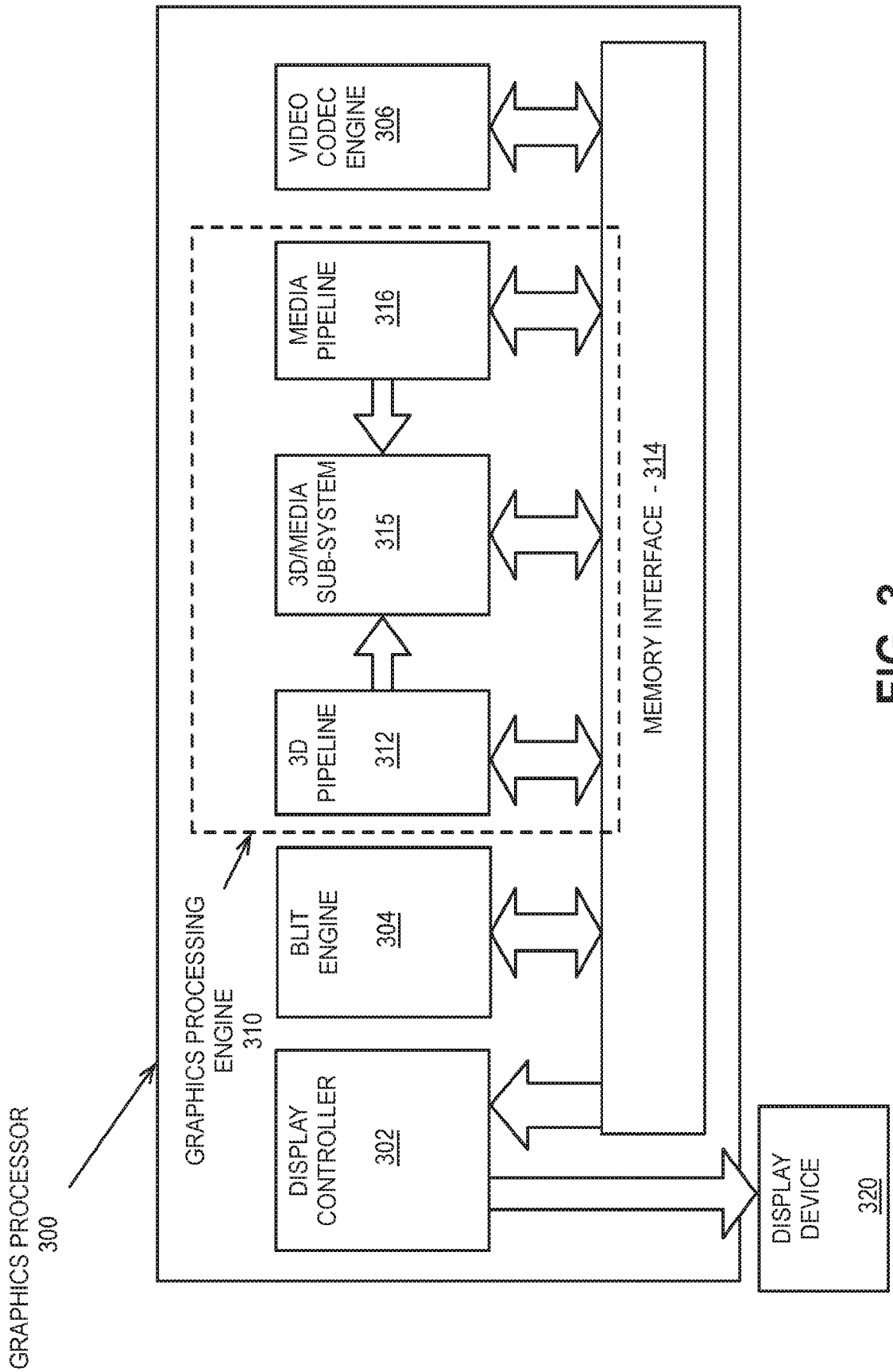

GRAPHICS PROCESSOR
300
GRAPHICS PROCESSING
ENGINE
310
DISPLAY
CONTROLLER
302
BLIT
ENGINE
304
3D
PIPELINE
312
3D/MEDIA
SUB-SYSTEM
315
MEDIA
PIPELINE
316
VIDEO
CODEC
ENGINE
306
MEMORY INTERFACE - 314
DISPLAY
DEVICE
320

FIG. 3

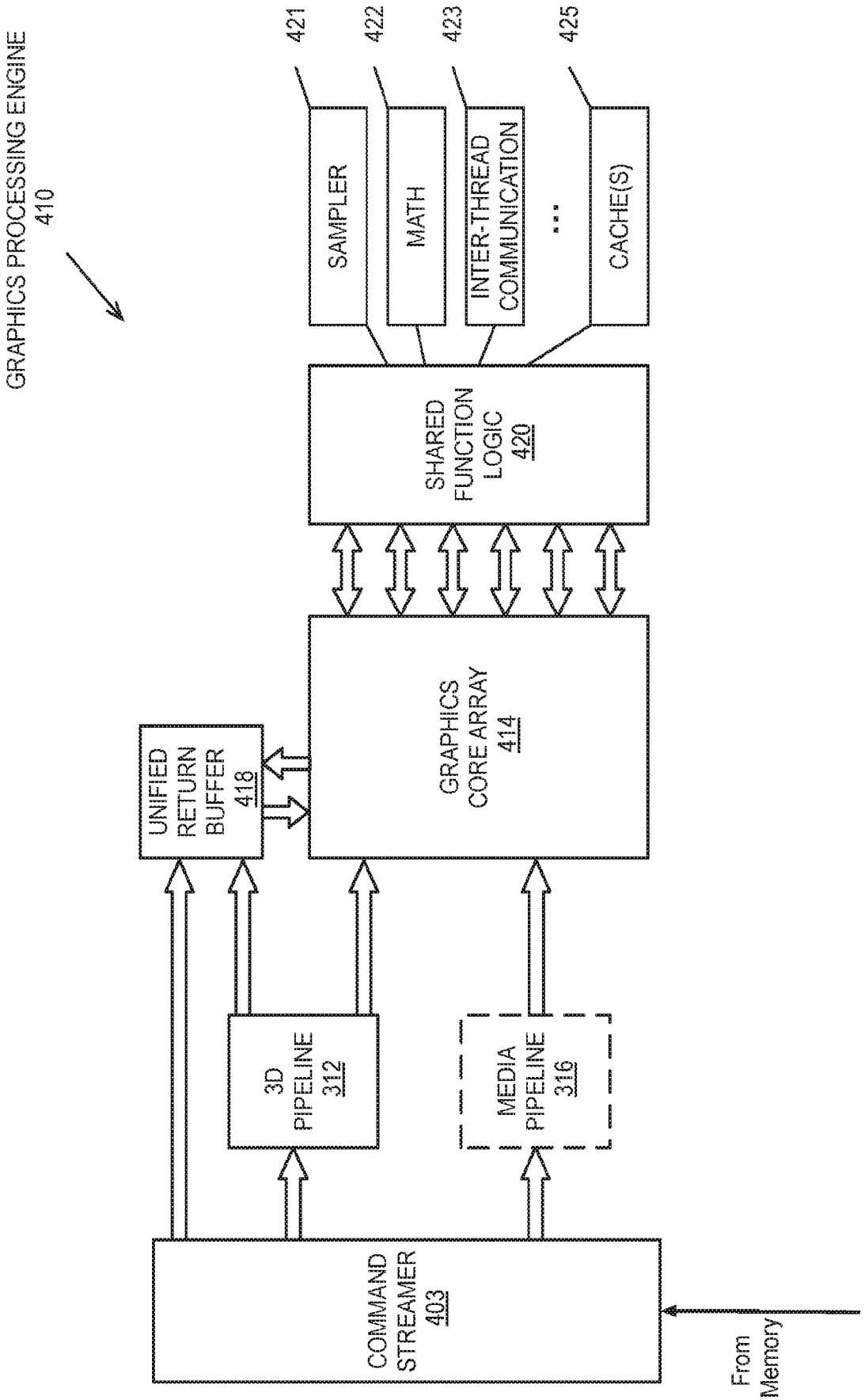

GRAPHICS PROCESSING ENGINE
410
421
422
423
425
SAMPLER
MATH
INTER-THREAD COMMUNICATION
...
CACHE(S)
SHARED FUNCTION LOGIC 420
GRAPHICS CORE ARRAY 414
UNIFIED RETURN BUFFER 418
3D PIPELINE 312
MEDIA PIPELINE 316
COMMAND STREAMER 403
From Memory

FIG. 4

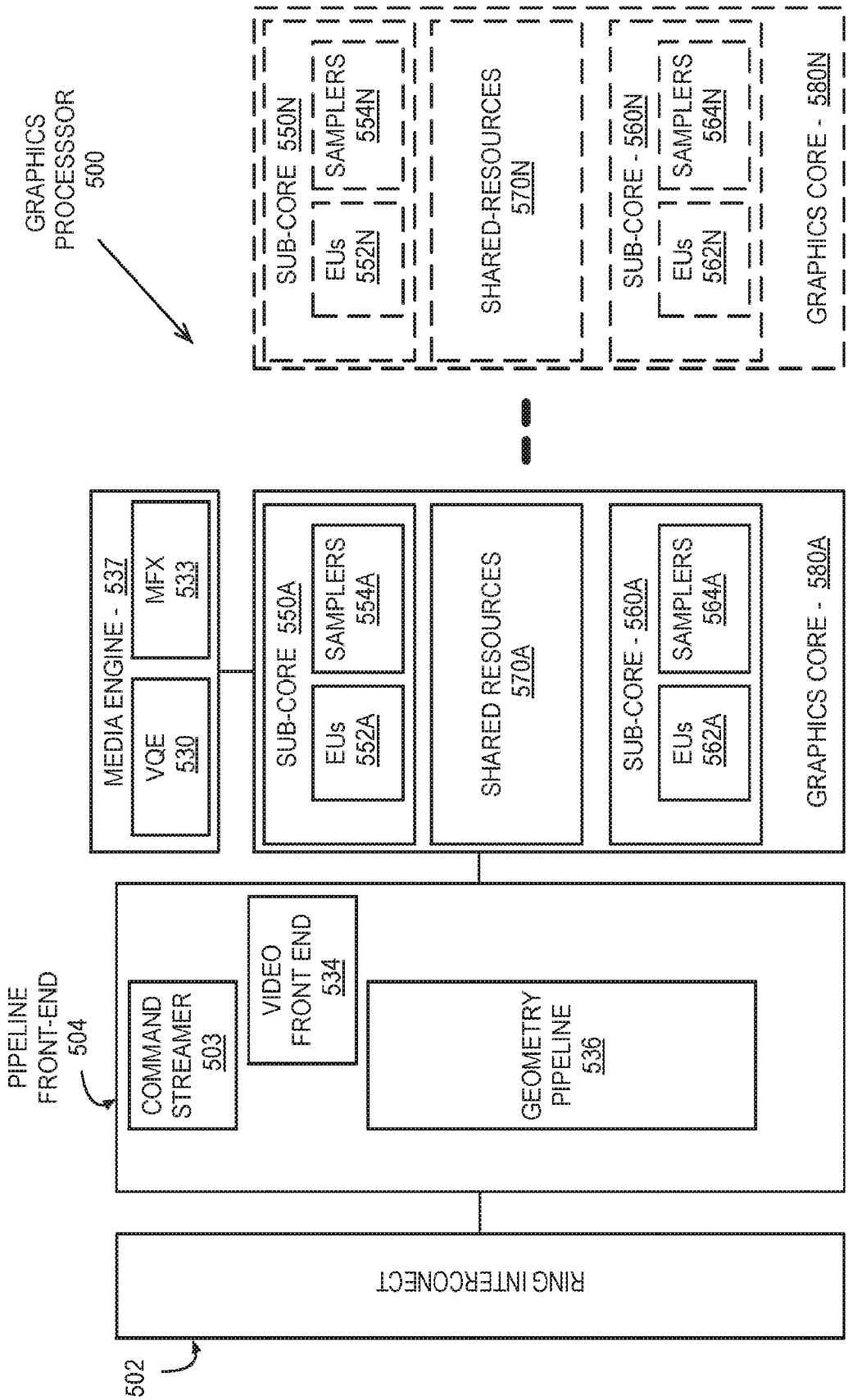

GRAPHICS PROCESSSOR 500
SUB-CORE 550N
EUs 552N
SAMPLERS 554N
SHARED-RESOURCES 570N
SUB-CORE - 560N
EUs 562N
SAMPLERS 564N
GRAPHICS CORE - 580N
MEDIA ENGINE - 537
VQE 530
MFX 533
SUB-CORE 550A
EUs 552A
SAMPLERS 554A
SHARED RESOURCES 570A
SUB-CORE - 560A
EUs 562A
SAMPLERS 564A
GRAPHICS CORE - 580A
PIPELINE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

FIG. 5

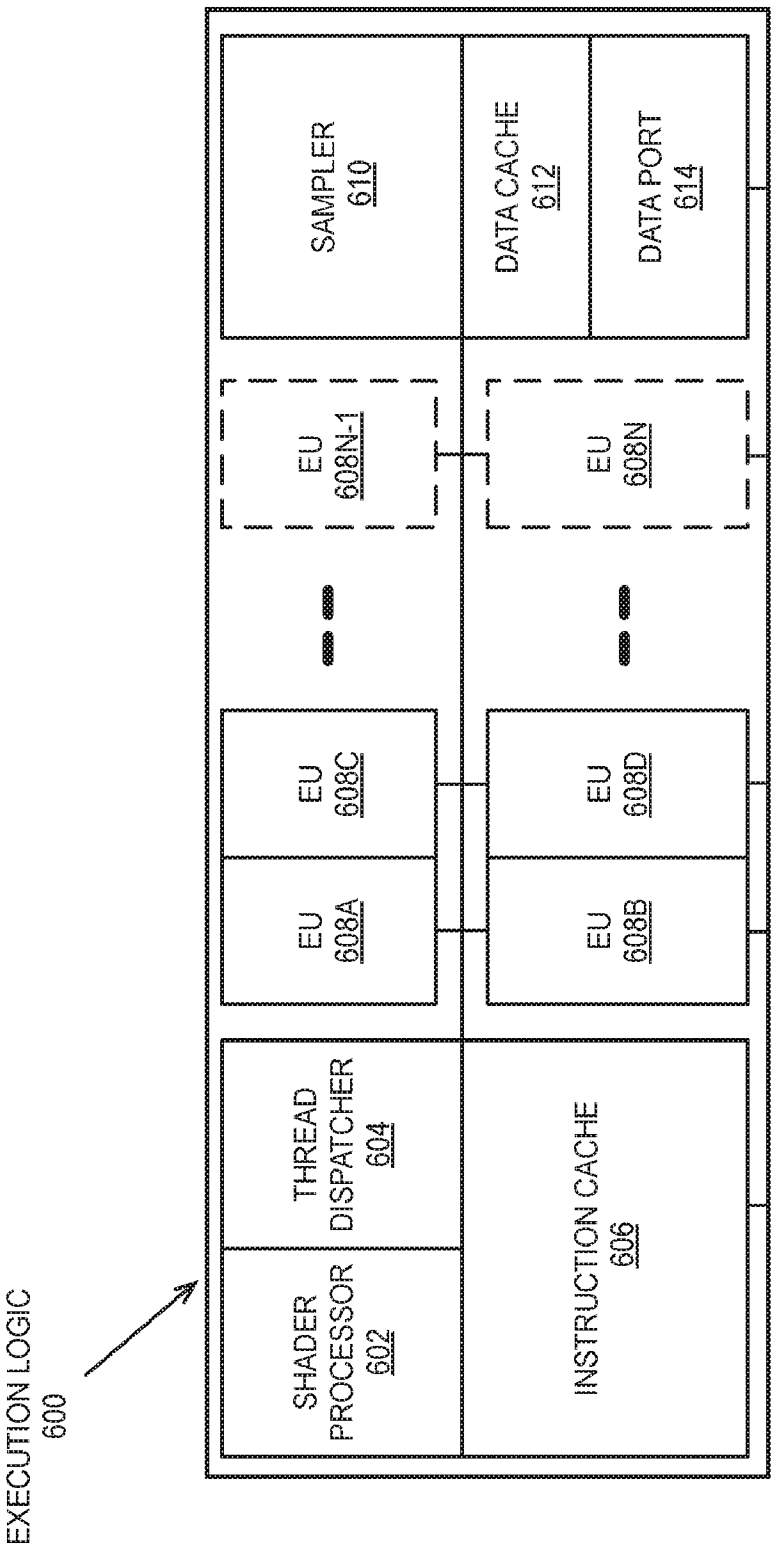

EXECUTION LOGIC
600
SHADER PROCESSOR
602
THREAD DISPATCHER
604
INSTRUCTION CACHE
606
EU
608A
EU
608C
EU
608N-1
EU
608B
EU
608D
EU
608N
SAMPLER
610
DATA CACHE
612
DATA PORT
614

FIG. 6

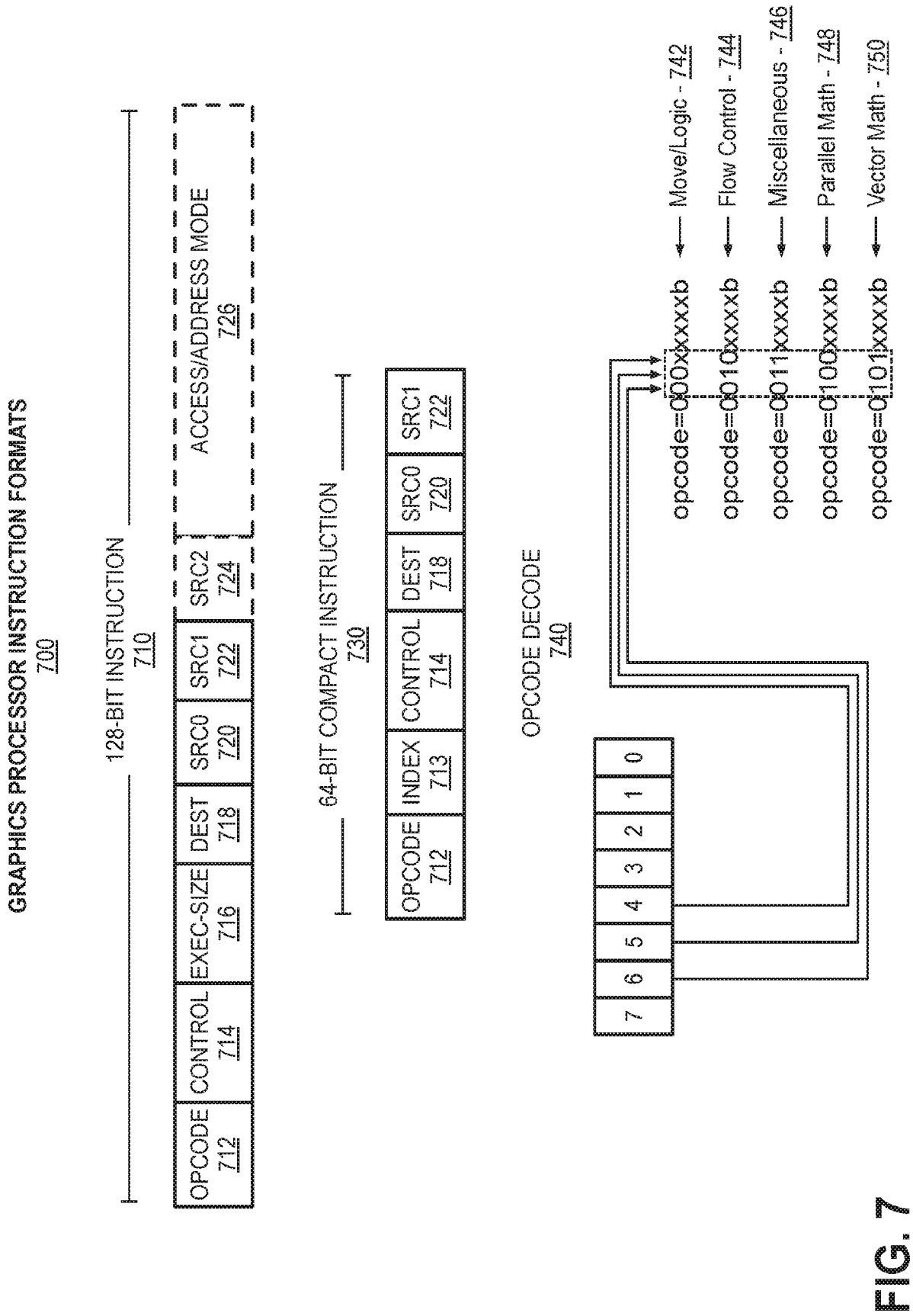

GRAPHICS PROCESSOR INSTRUCTION FORMATS
700
128-BIT INSTRUCTION
710
OPCODE 712
CONTROL 714
EXEC-SIZE 716
DEST 718
SRC0 720
SRC1 722
SRC2 724
ACCESS/ADDRESS MODE 726
64-BIT COMPACT INSTRUCTION
730
OPCODE 712
INDEX 713
CONTROL 714
DEST 718
SRC0 720
SRC1 722
OPCODE DECODE
740
7 6 5 4 3 2 1 0
opcode=0000xxxxb
opcode=0001xxxxb
opcode=0010xxxxb
opcode=0011xxxxb
opcode=0100xxxxb
opcode=0101xxxxb
Move/Logic - 742
Flow Control - 744
Miscellaneous - 746
Parallel Math - 748
Vector Math - 750

FIG. 7

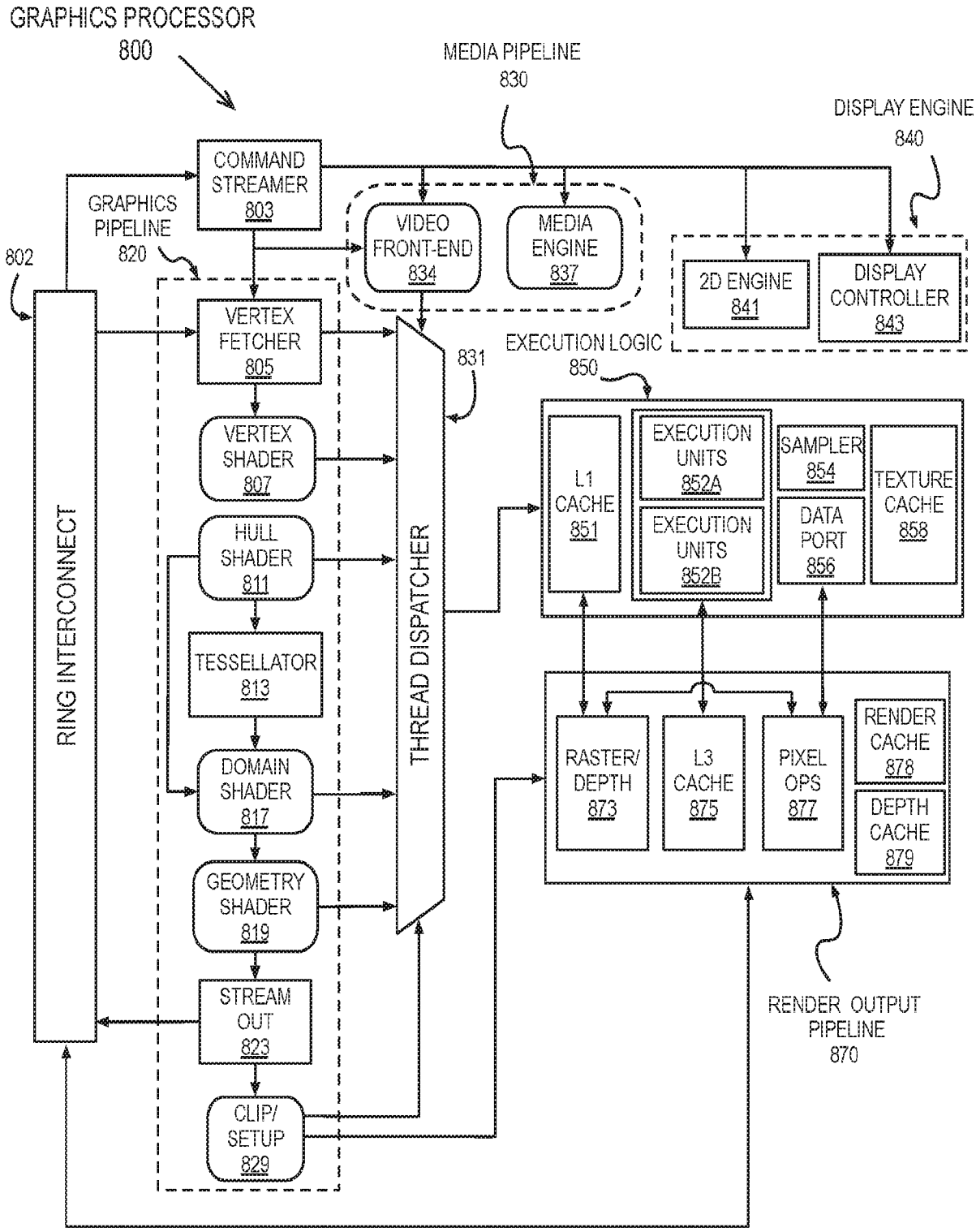

GRAPHICS PROCESSOR
800
MEDIA PIPELINE
830
DISPLAY ENGINE
840
COMMAND STREAMER
803
GRAPHICS PIPELINE
820
802
VIDEO FRONT-END
834
MEDIA ENGINE
837
2D ENGINE
841
DISPLAY CONTROLLER
843
VERTEX FETCHER
805
VERTEX SHADER
807
HULL SHADER
811
TESSELLATOR
813
DOMAIN SHADER
817
GEOMETRY SHADER
819
STREAM OUT
823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

FIG. 8

FIG. 9A GRAPHICS PROCESSOR COMMAND FORMAT
900

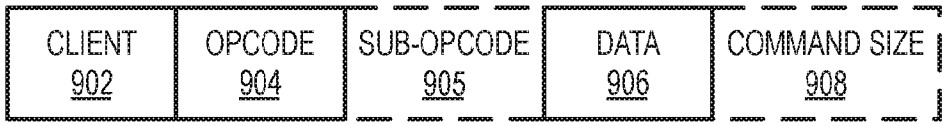
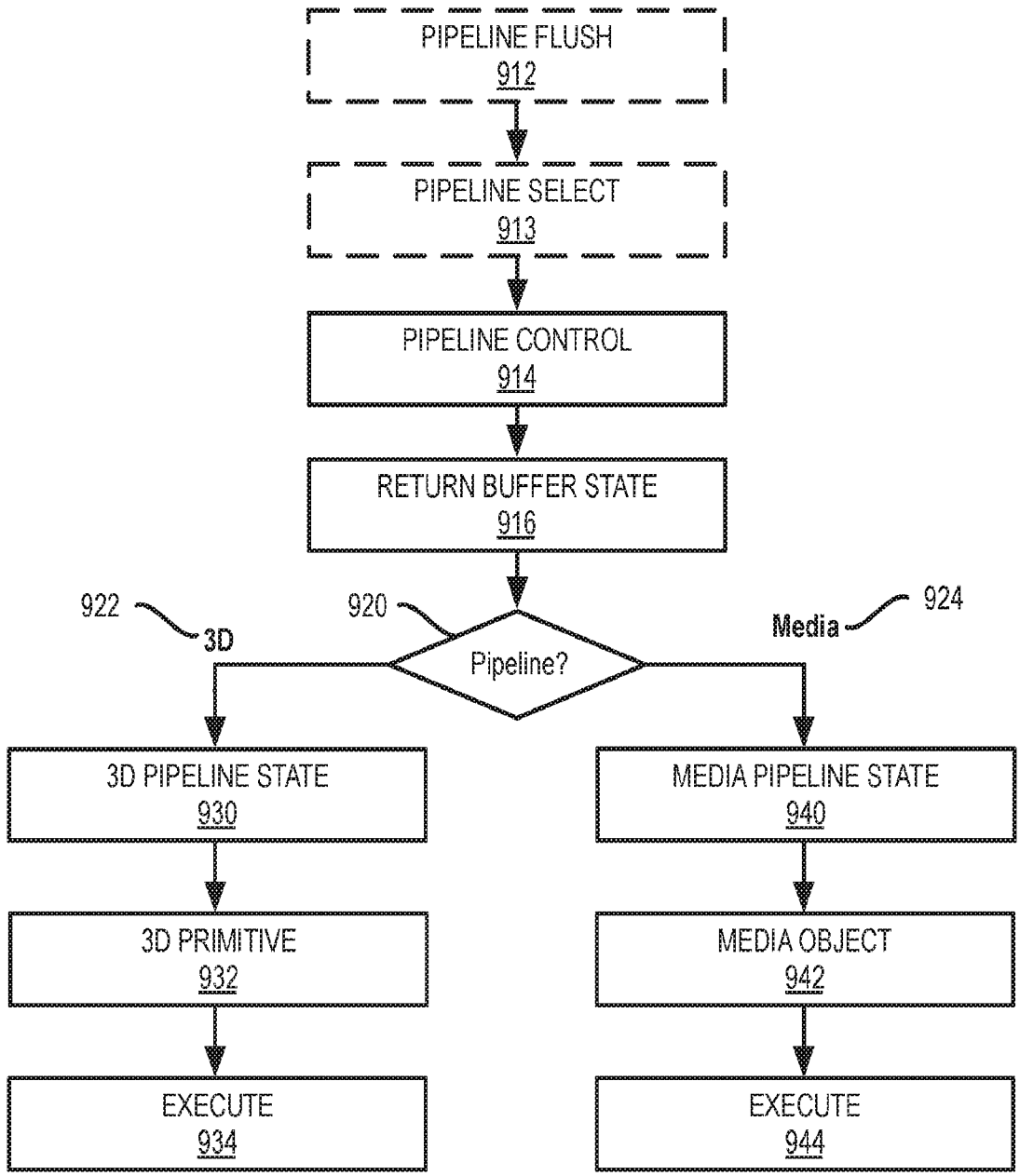


FIG. 9B GRAPHICS PROCESSOR COMMAND SEQUENCE
910



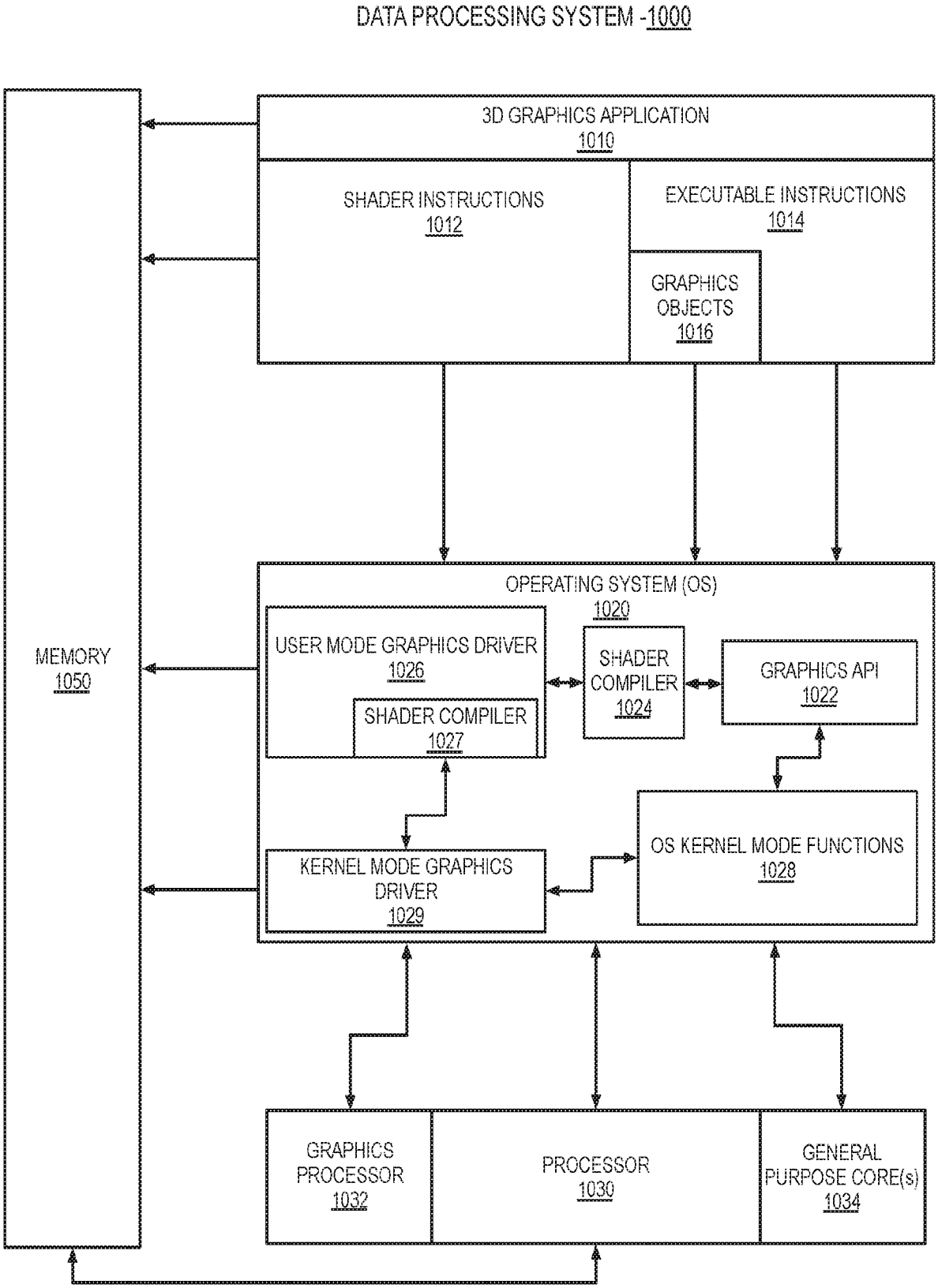

DATA PROCESSING SYSTEM -1000
3D GRAPHICS APPLICATION
1010
SHADER INSTRUCTIONS
1012
EXECUTABLE INSTRUCTIONS
1014
GRAPHICS OBJECTS
1016
MEMORY
1050
OPERATING SYSTEM (OS)
1020
USER MODE GRAPHICS DRIVER
1026
SHADER COMPILER
1027
SHADER COMPILER
1024
GRAPHICS API
1022
KERNEL MODE GRAPHICS DRIVER
1029
OS KERNEL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

FIG. 10

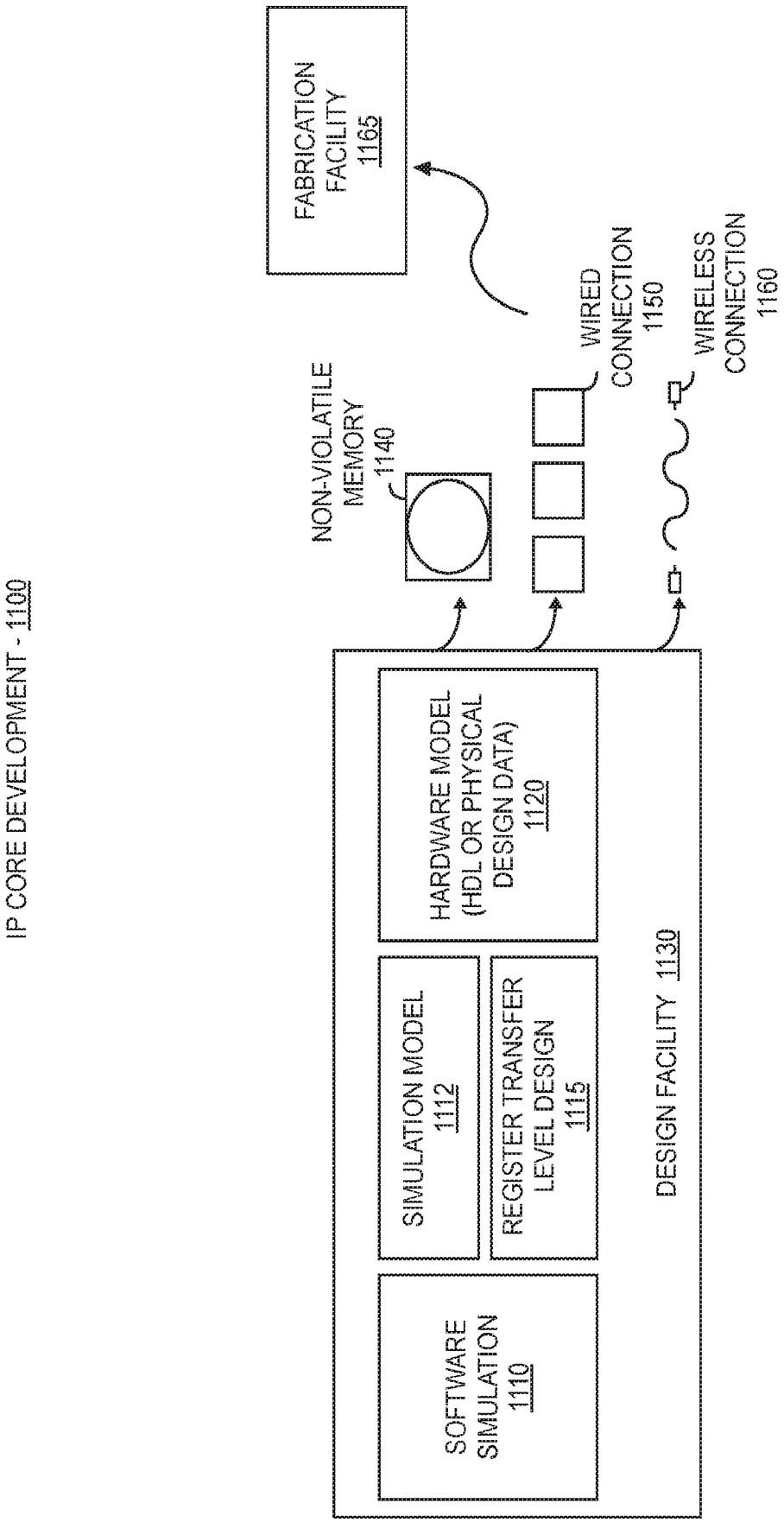

IP CORE DEVELOPMENT - 1100
FABRICATION FACILITY 1165
NON-VIOLATILE MEMORY 1140
WIRED CONNECTION 1150
WIRELESS CONNECTION 1160
SOFTWARE SIMULATION 1110
SIMULATION MODEL 1112
REGISTER TRANSFER LEVEL DESIGN 1115
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA) 1120
DESIGN FACILITY 1130

FIG. 11

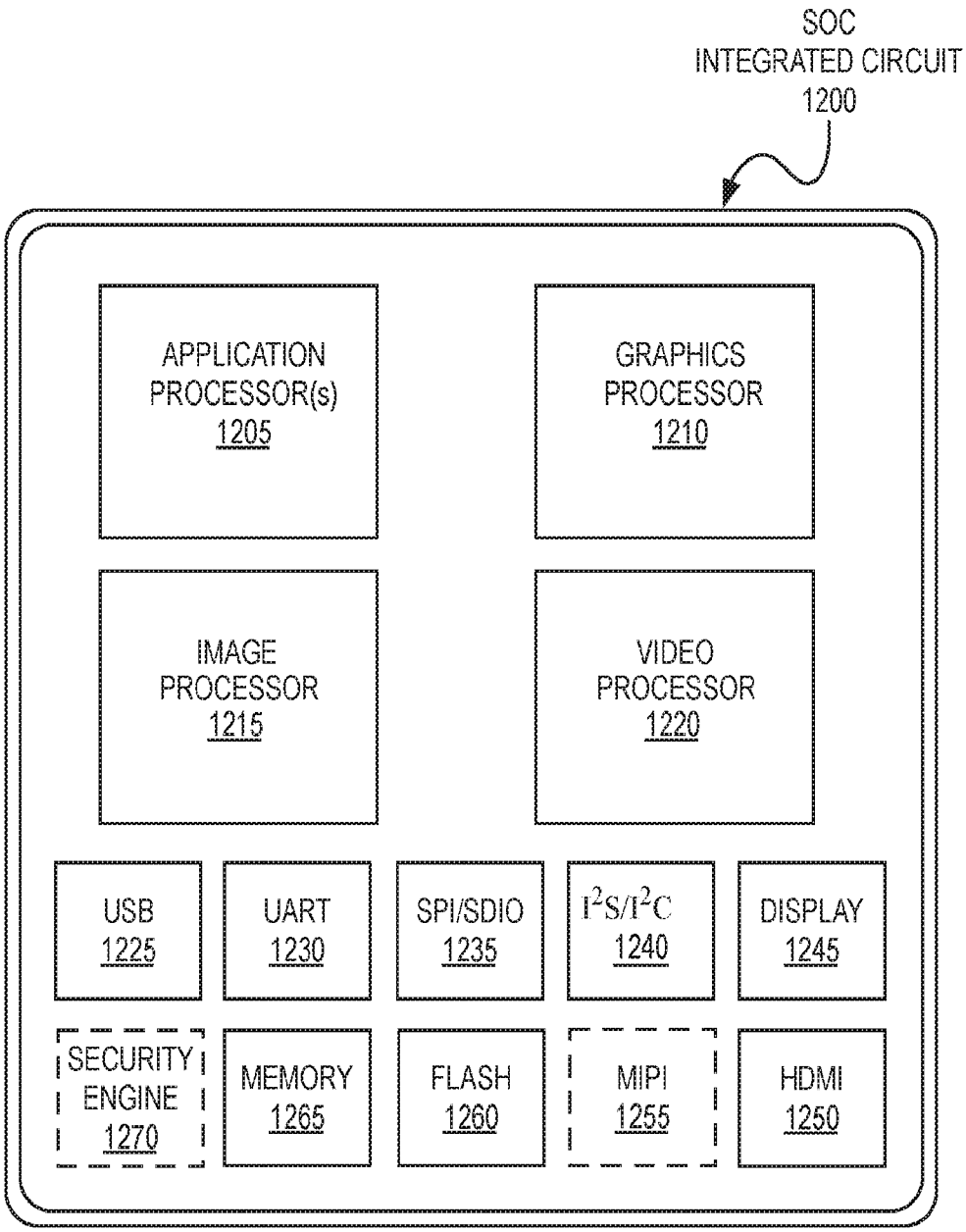

SOC
INTEGRATED CIRCUIT
1200
APPLICATION PROCESSOR(s) 1205
GRAPHICS PROCESSOR 1210
IMAGE PROCESSOR 1215
VIDEO PROCESSOR 1220
USB 1225
UART 1230
SPI/SDIO 1235
I2S/I2C 1240
DISPLAY 1245
SECURITY ENGINE 1270
MEMORY 1265
FLASH 1260
MIPI 1255
HDMI 1250

FIG. 12

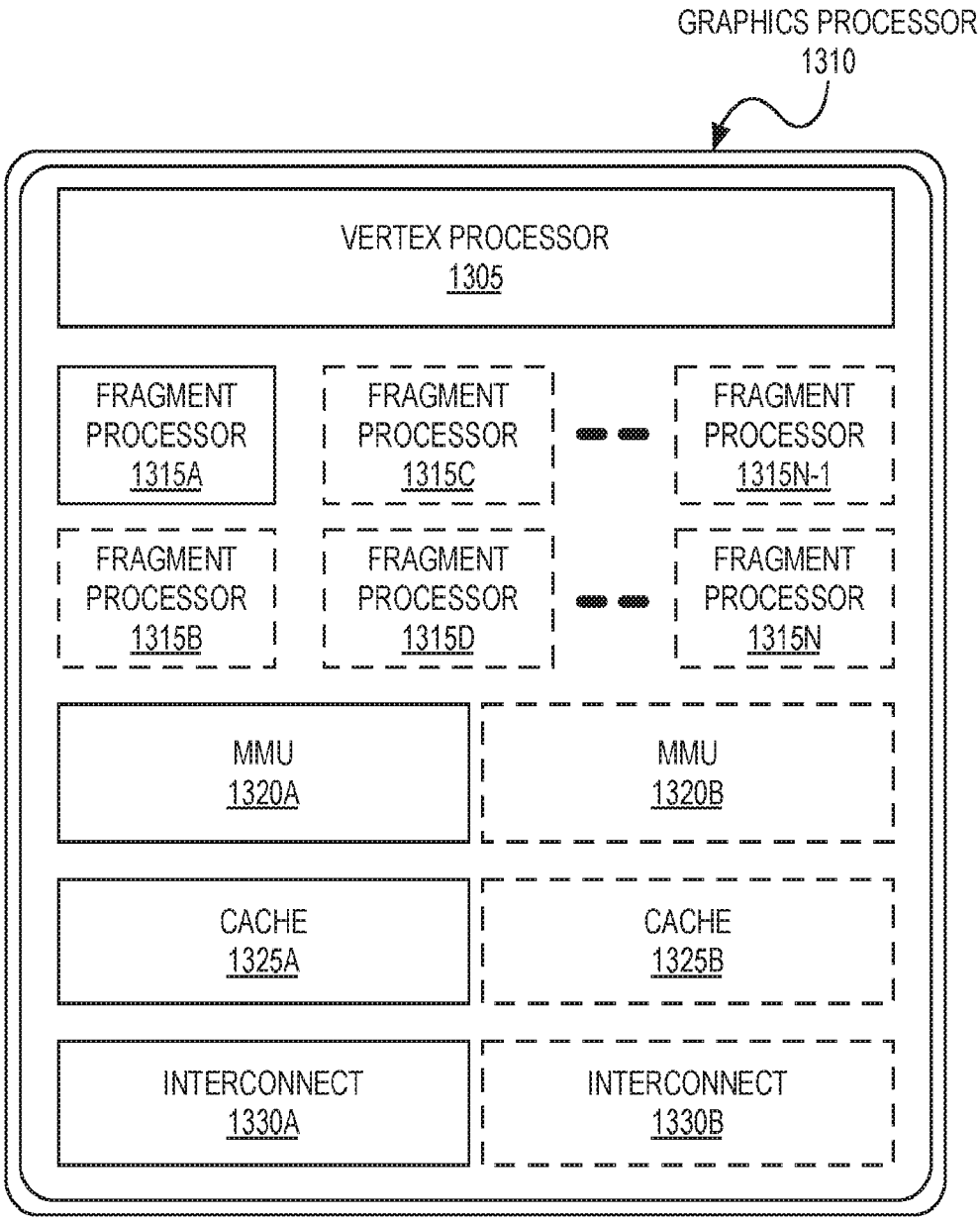

GRAPHICS PROCESSOR
1310
VERTEX PROCESSOR
1305
FRAGMENT PROCESSOR
1315A
FRAGMENT PROCESSOR
1315C
FRAGMENT PROCESSOR
1315N-1
FRAGMENT PROCESSOR
1315B
FRAGMENT PROCESSOR
1315D
FRAGMENT PROCESSOR
1315N
MMU
1320A
MMU
1320B
CACHE
1325A
CACHE
1325B
INTERCONNECT
1330A
INTERCONNECT
1330B

FIG. 13

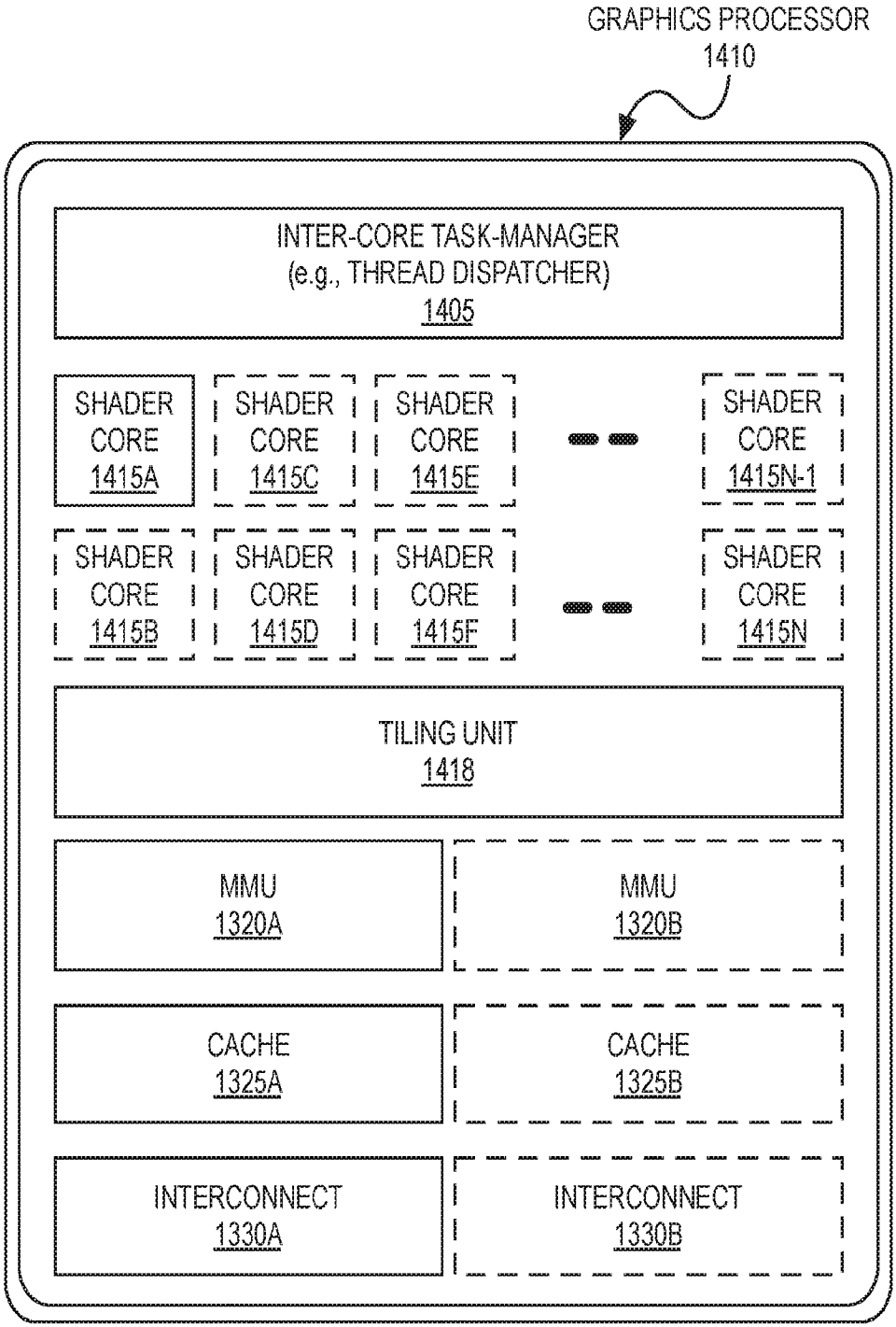

GRAPHICS PROCESSOR
1410
INTER-CORE TASK-MANAGER
(e.g., THREAD DISPATCHER)
1405
SHADER CORE 1415A
SHADER CORE 1415C
SHADER CORE 1415E
SHADER CORE 1415N-1
SHADER CORE 1415B
SHADER CORE 1415D
SHADER CORE 1415F
SHADER CORE 1415N
TILING UNIT
1418
MMU
1320A
MMU
1320B
CACHE
1325A
CACHE
1325B
INTERCONNECT
1330A
INTERCONNECT
1330B

FIG. 14

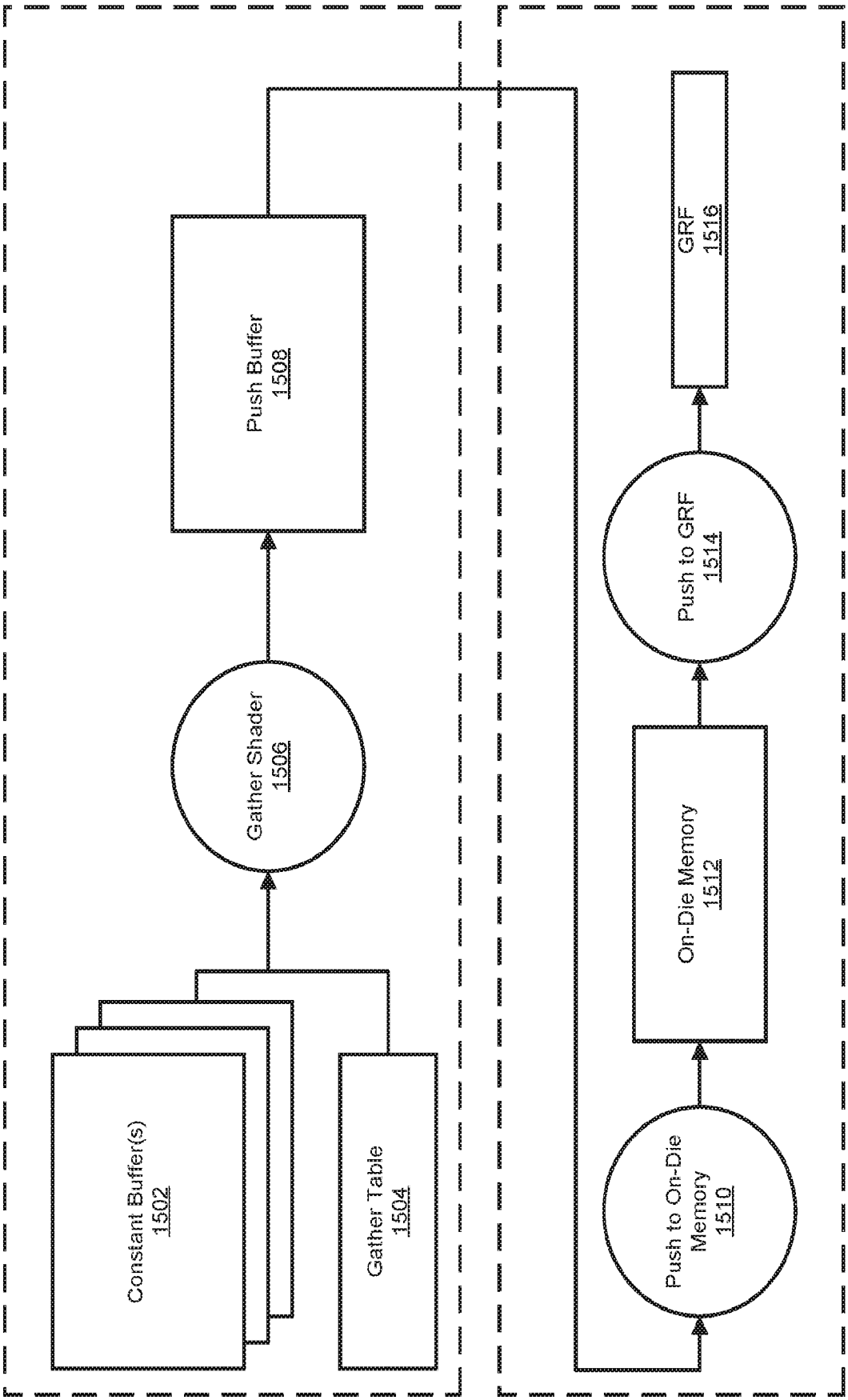

Constant Buffer(s)
1502
Gather Table
1504
Gather Shader
1506
Push Buffer
1508
Push to On-Die Memory
1510
On-Die Memory
1512
Push to GRF
1514
GRF
1516

FIG. 15

```
1  cbuffer cbPerObject : register( b0 )
2  {
3      uint numberOfConstant;
4      // each entry has 16bits for the offset, 16bits for the buffer index
5      uint gatherMap[256];
6  };
7
8  ByteAddressBuffer ConstantBuffers[16] : register(t0)
9
10 RWByteAddressBuffer BufferPushed : register(u0);
11
12 uint BITS(uint data, uint start, uint length)
13 {
14     return (data >> start) & (~(0xFFFFFFFF << length));
15 }
16
17 void VSMain( )
18 {
19     uint offset = 0;
20     for(uint i = 0; i < numberOfConstant; i++)
21     {
22         uint b = gatherMap[i];
23         uint index = BITS(b, 0, 16);
24         uint bufferIndex = BITS(b, 16, 16);
25         uint4 data = ConstantBuffers[bufferIndex].Load4(index*16);
26         BufferPushed.Store4(offset,  data);
27         offset += 16;
28     }
29 }
```

FIG. 16

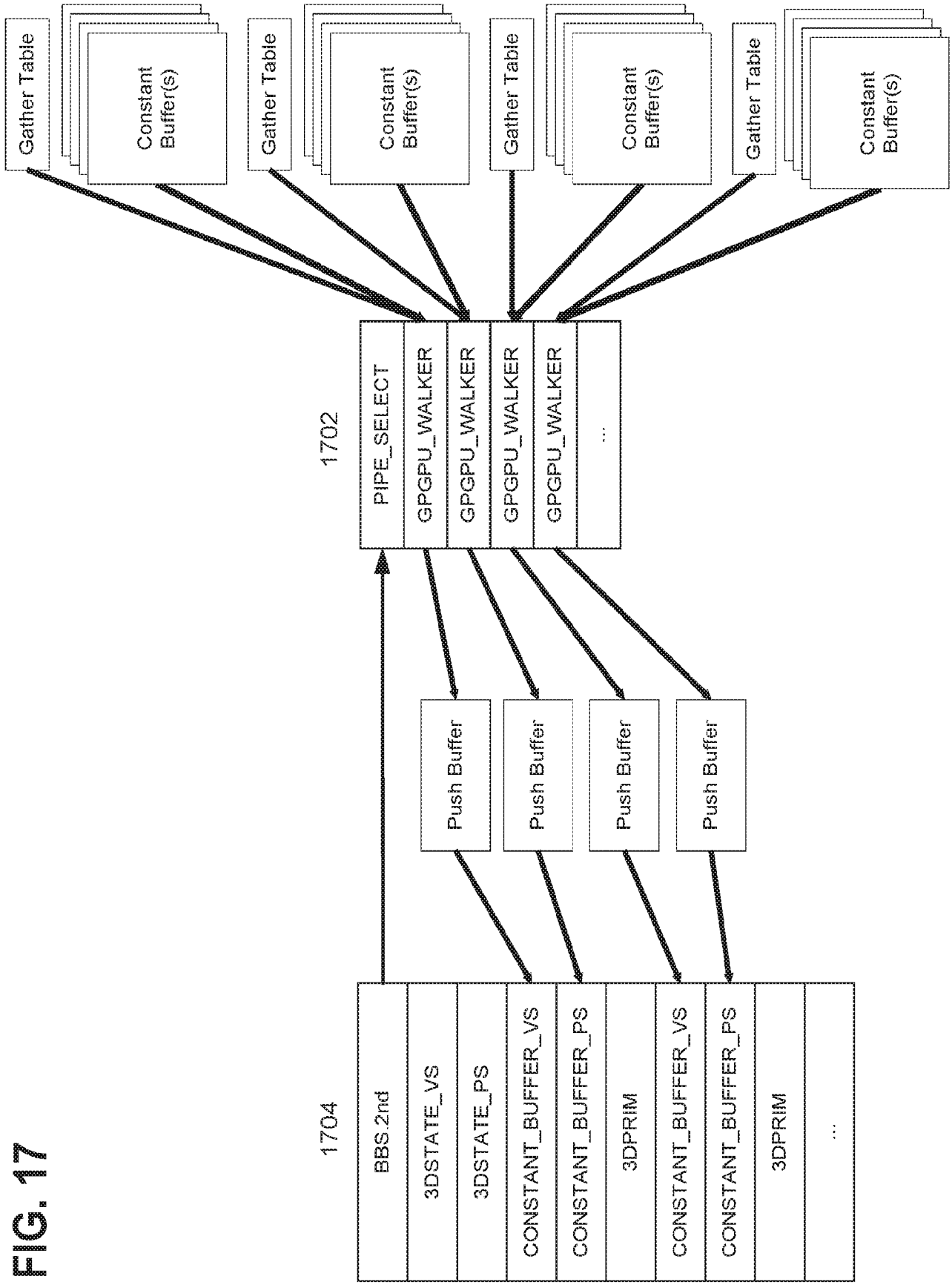
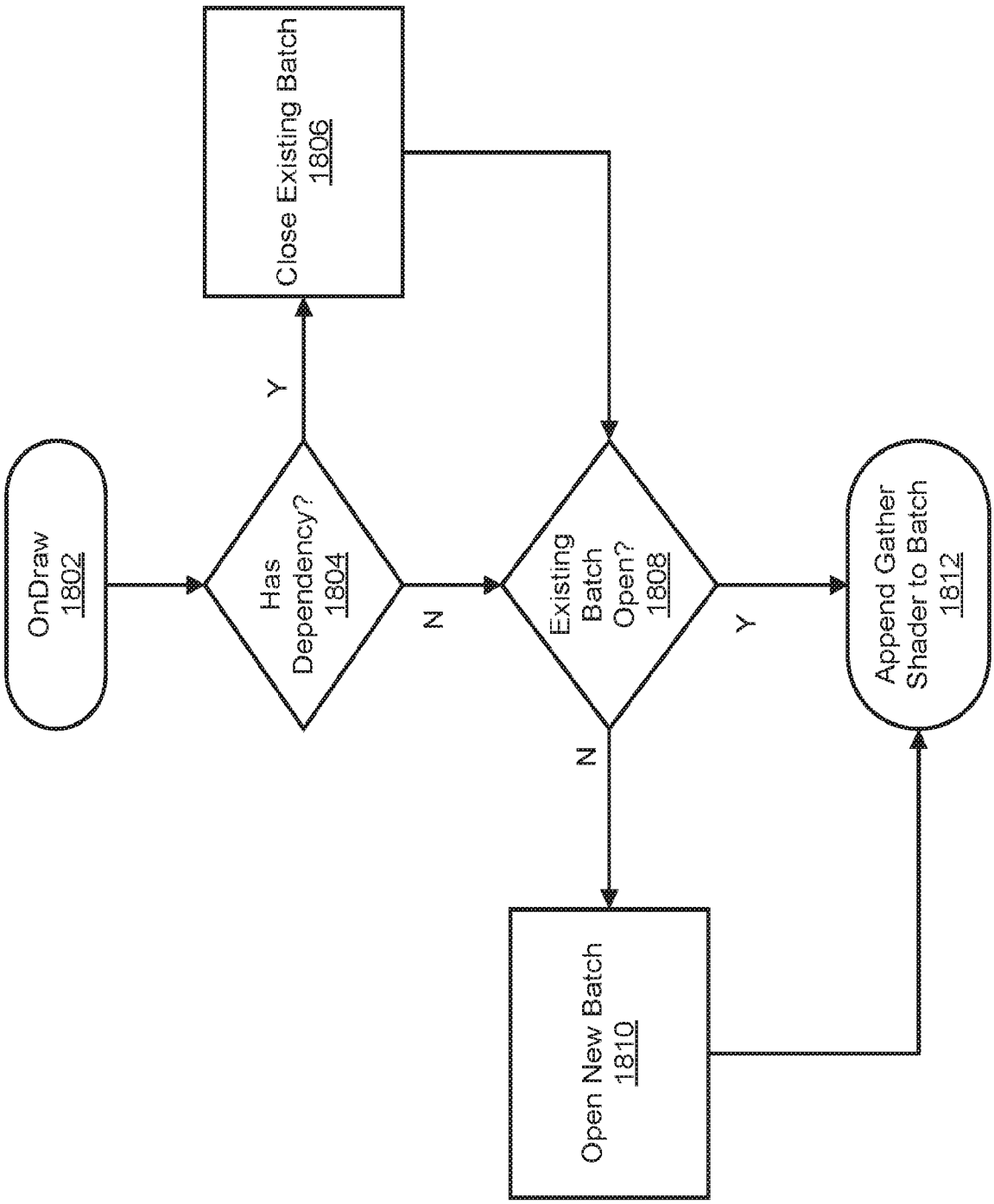

Gather Table
Constant Buffer(s)
Gather Table
Constant Buffer(s)
Gather Table
Constant Buffer(s)
Gather Table
Constant Buffer(s)
1702
PIPE_SELECT
GPGPU_WALKER
GPGPU_WALKER
GPGPU_WALKER
GPGPU_WALKER
...
Push Buffer
Push Buffer
Push Buffer
Push Buffer
1704
BBS.2nd
3DSTATE_VS
3DSTATE_PS
CONSTANT_BUFFER_VS
CONSTANT_BUFFER_PS
3DPRIM
CONSTANT_BUFFER_VS
CONSTANT_BUFFER_PS
3DPRIM
...

FIG. 17

FIG. 18


OnDraw
1802
Has Dependency?
1804
Y
Close Existing Batch
1806
N
Existing Batch Open?
1808
Y
Append Gather Shader to Batch
1812
N
Open New Batch
1810

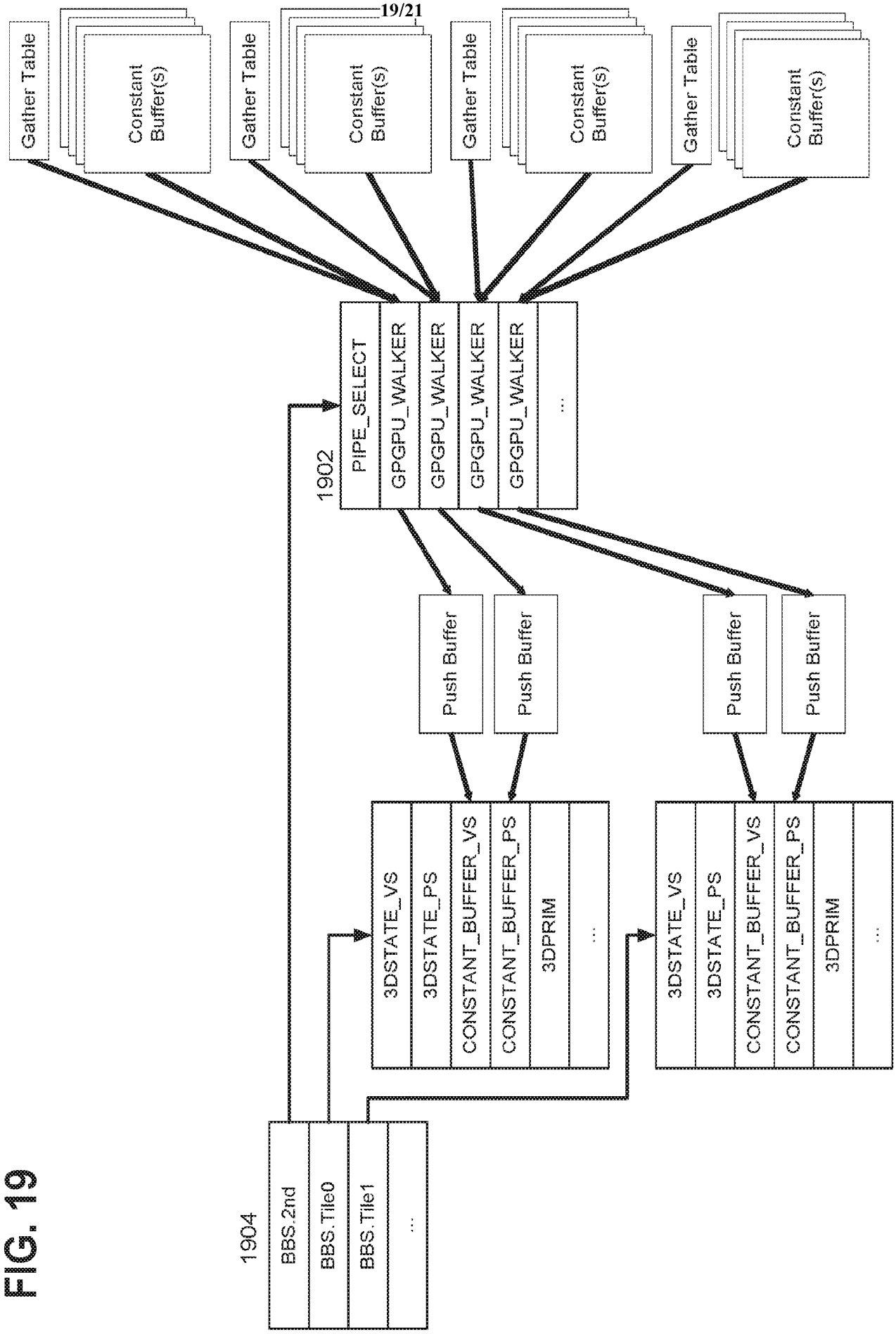

Gather Table
Constant Buffer(s)
Gather Table
Constant Buffer(s)
Gather Table
Constant Buffer(s)
Gather Table
Constant Buffer(s)
1902
PIPE_SELECT
GPGPU_WALKER
GPGPU_WALKER
GPGPU_WALKER
GPGPU_WALKER
...
Push Buffer
Push Buffer
Push Buffer
Push Buffer
3DSTATE_VS
3DSTATE_PS
CONSTANT_BUFFER_VS
CONSTANT_BUFFER_PS
3DPRIM
...
3DSTATE_VS
3DSTATE_PS
CONSTANT_BUFFER_VS
CONSTANT_BUFFER_PS
3DPRIM
...
1904
BBS.2nd
BBS.Tile0
BBS.Tile1
...

FIG. 19

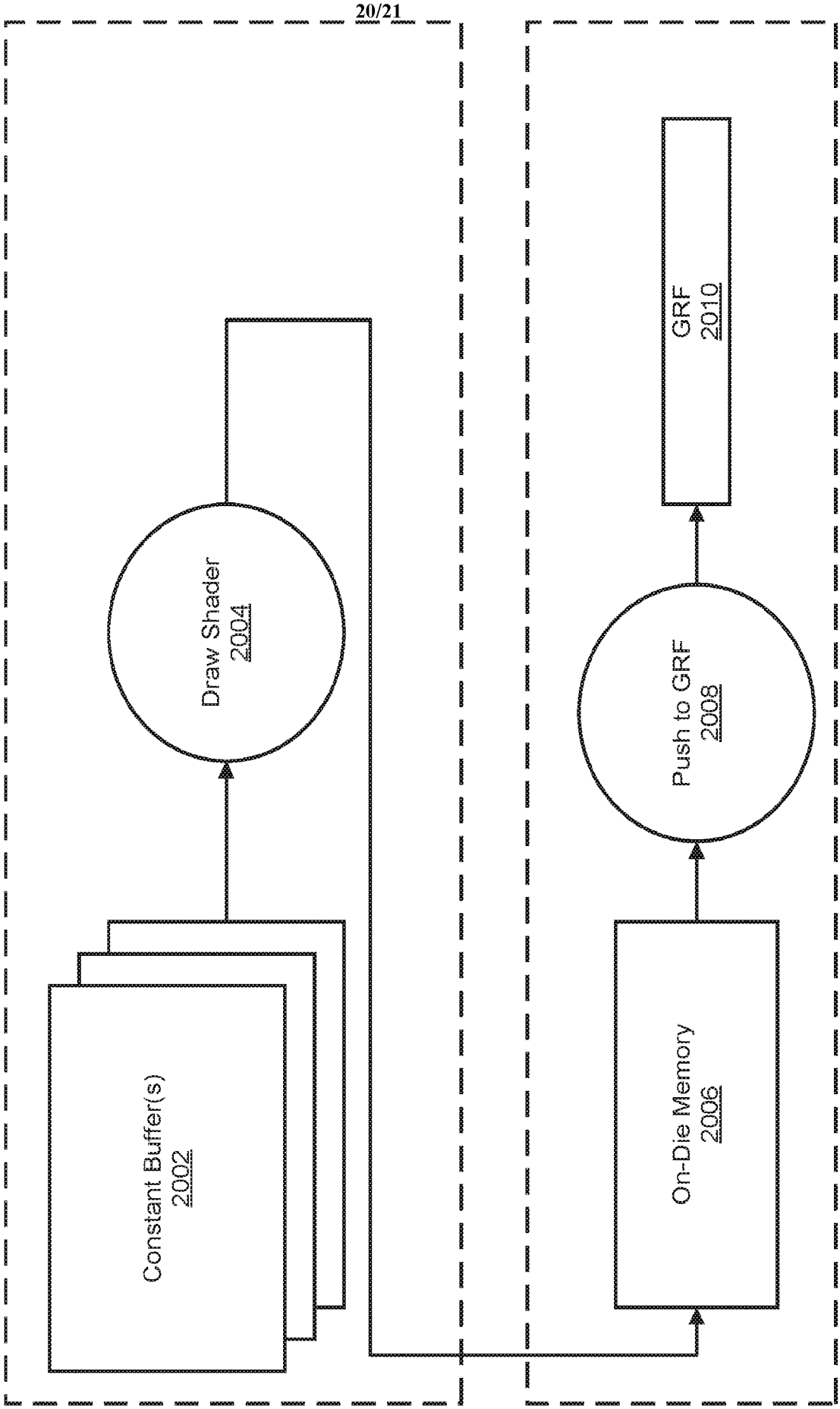

Constant Buffer(s)
2002
Draw Shader
2004
On-Die Memory
2006
Push to GRF
2008
GRF
2010

FIG. 20

```
1  cbuffer cbPerObject : register( b0 )
2  {
3      matrix      g_mWorldViewProjection  : packoffset( c0 );
4      matrix      g_mWorld                : packoffset( c4 );
5  };
6
7  struct DR_OUTPUT
8  {
9      matrix      mWorldViewProjection;
10     float3x3    mWorld;
11 };
12
13 struct VS_INPUT
14 {
15     float4 vPosition    : POSITION;
16     float3 vNormal      : NORMAL;
17     float2 vTexcoord    : TEXCOORD0;
18 };
19
20 struct VS_OUTPUT
21 {
22     float3 vNormal      : NORMAL;
23     float2 vTexcoord    : TEXCOORD0;
24     float4 vPosition    : SV_POSITION;
25 };
26
27 DR_OUTPUT DRMain( )
28 {
29     DR_OUTPUT Output;
30
31     Output.mWorldViewProjection = g_mWorldViewProjection;
32     Output.mWorld = (float3x3)g_mWorld;
33
34     return Output;
35 }
36
37 VS_OUTPUT VSMain( DR_OUTPUT perDraw, VS_INPUT perVertex )
38 {
39     VS_OUTPUT Output;
40
41     Output.vPosition = mul( perVertex.vPosition, perDraw.mWorldViewProjection );
42     Output.vNormal = mul( perVertex.vNormal, perDraw.mWorld );
43     Output.vTexcoord = perVertex.vTexcoord;
44
45     return Output;
46 }
```

FIG. 21

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/035069

A. CLASSIFICATION OF SUBJECT MATTER

G06T 1/60(2006.01)i, G06T 1/20(2006.01)i

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06T 1/60; G06T 7/40; G06F 9/30; G06F 9/48; G06F 9/06; G06F 15/76; G06F 9/50; G06F 12/08; G06T 1/20

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

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: uniform data, unified return buffer (URB), general register file (GRF), primitives, higher frequency operations

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2015-0379680 A1 (QUALCOMM INC.) 31 December 2015 See paragraphs [0024]-[0028], [0099]-[0104]; claim 1; and figure 3.	1-24
A	US 2009-0132793 A1 (SURESH K. VENKUMAHANTI et al.) 21 May 2009 See paragraphs [0003]-[0009], [0021]-[0026]; claim 1; and figure 1.	1-24
A	US 2009-0300621 A1 (MICHAEL J. MANTOR et al.) 03 December 2009 See paragraphs [0008]-[0010], [0026]-[0028]; claim 1; and figure 1.	1-24
A	KR 10-2014-0015631 A (QUALCOMM INC.) 06 February 2014 See paragraphs [0005]-[0008], [0043]-[0044]; claim 1; and figure 2.	1-24
A	US 2011-0252204 A1 (BRETT W. COON et al.) 13 October 2011 See paragraphs [0011]-[0013], [0119]-[0125]; claim 1; and figure 11.	1-24

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

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
28 August 2017 (28.08.2017)	**29 August 2017 (29.08.2017)**

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

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

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2017/035069

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2015-0379680 A1	31/12/2015	CN 106462941 A	22/02/2017
		EP 3161817 A1	03/05/2017
		US 9633411 B2	25/04/2017
		WO 2015-199997 A1	30/12/2015
US 2009-0132793 A1	21/05/2009	TW 200928967 A	01/07/2009
		US 7979681 B2	12/07/2011
		WO 2009-067590 A1	28/05/2009
US 2009-0300621 A1	03/12/2009	CN 102047241 A	04/05/2011
		CN 102047241 B	12/03/2014
		EP 2289001 A1	02/03/2011
		JP 2011-522325 A	28/07/2011
		JP 5461533 B2	02/04/2014
		KR 10-1474478 B1	19/12/2014
		KR 10-2011-0015439 A	15/02/2011
		US 9619428 B2	11/04/2017
		WO 2009-145917 A1	03/12/2009
KR 10-2014-0015631 A	06/02/2014	CN 103608777 A	26/02/2014
		CN 103608777 B	10/05/2017
		EP 2721488 A1	23/04/2014
		JP 2014-523021 A	08/09/2014
		JP 5752853 B2	22/07/2015
		KR 10-1590734 B1	01/02/2016
		US 2012-0320070 A1	20/12/2012
		US 9092267 B2	28/07/2015
		WO 2012-177343 A1	27/12/2012
US 2011-0252204 A1	13/10/2011	US 2012-0221808 A1	30/08/2012
		US 8176265 B2	08/05/2012
		US 8645638 B2	04/02/2014