



(51) International Patent Classification:

G06T 1/20 (2006.01) *G06T 5/00* (2006.01)
G06T 15/08 (2011.01)

(21) International Application Number:

PCT/US2016/028617

(22) International Filing Date:

21 April 2016 (21.04.2016)

(25) Filing Language:

English

(26) Publication Language:

English

(30) Priority Data:

14/729,915 3 June 2015 (03.06.2015) US

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

(72) Inventors; and

(71) Applicants : **TAYLOR, Robert B.** [US/US]; 760 NE 61st Court, Hillsboro, Oregon 97124 (US). **VENKATESH, Abhishek** [IN/US]; 7052 Ne Ronler Way, Apt 2917, Hillsboro, Oregon 97124 (US).

(74) Agent: **JORDAN, B. Delano**; Jordan IP Law LLC, c/o CPA Global, P.O. Box 52050, Minneapolis, Minnesota 55402 (US).

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

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

Published:

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

(54) Title: AUTOMATED CONVERSION OF GPGPU WORKLOADS TO 3D PIPELINE WORKLOADS

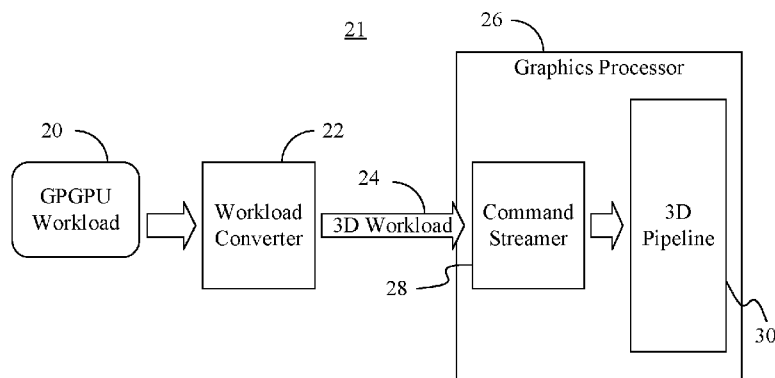


FIG. 1

(57) Abstract: Systems, apparatuses and methods may provide for receiving a general purpose graphics processing unit (GPGPU) workload and converting the GPGPU workload to a three-dimensional (3D) workload. Additionally, the 3D workload may be dispatched to a 3D pipeline. In one example, converting the GPGPU workload to the 3D workload includes identifying a plurality of thread groups in the GPGPU workload and mapping the plurality of thread groups to a 3D matrix of cubes.

AUTOMATED CONVERSION OF GPGPU WORKLOADS TO 3D PIPELINE WORKLOADS

CROSS-REFERENCE TO RELATED APPLICATIONS

5 The present application claims the benefit of priority to U.S. Non-Provisional Patent Application No. 14/729,915 filed on June 3, 2015.

BACKGROUND

10 Visually intensive computer graphics applications such as 3D (three-dimensional) computer games, flight simulators and other 3D imaging applications may involve user interaction, scene management and rendering, physics modeling, artificial intelligence and other relatively complex functions. While certain game applications may leverage the capabilities of a local GPU (graphics processing unit) by offloading graphical and non-graphical computation to the GPU in order to maintain interactive
15 frame rates, there remains considerable room for improvement.

 For example, conventional approaches may send 3D workloads to a 3D pipeline of the GPU, wherein the 3D workloads perform functions, such as, for example, rendering 3D images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). Other computational workloads,
20 however, may be sent as a general purpose GPU (GPGPU) workload (e.g., a compute kernel) to the GPU, wherein the compute kernel may facilitate native execution of the GPGPU workload on processing units of the GPU. Native execution of the GPGPU workload may result in relatively low efficiency from both a power and performance perspective.

25 More particularly, efficiency may be negatively impacted by overhead associated with switching between 3D and GPGPU pipelines, hardware restrictions such as memory hierarchies supporting the workload execution, cache flushes, execution lane (e.g., single instruction multiple data/SIMD) restrictions on resource accesses (e.g., typed/un-typed unordered access views/UAV, shader resource views,
30 raw access, etc.), and so forth.

BRIEF DESCRIPTION OF THE DRAWINGS

 The various advantages of the embodiments will become apparent to one skilled in the art by reading the following specification and appended claims, and by
35 referencing the following drawings, in which:

FIG. 1 is a block diagram of an example of a workload conversion according to an embodiment;

FIG. 2 is an illustration of an example of a 3D matrix of cubes according to an embodiment;

5 FIG. 3 is an illustration of an example of a semantic relationship between a set of thread groups and a dispatch interface according to an embodiment;

FIG. 4A is a flowchart of an example of a method of operating a computing architecture according to an embodiment;

10 FIG. 4B is a flowchart of an example of a method of converting a GPGPU workload to a 3D workload according to an embodiment;

FIG. 5 is a block diagram of an example of a computing device according to an embodiment;

FIGs. 6-8 are block diagrams of an example of an overview of a data processing system according to an embodiment;

15 FIG. 9 is a block diagram of an example of a graphics processing engine according to an embodiment;

FIGs. 10-12 are block diagrams of examples of execution units according to an embodiment;

20 FIG. 13 is a block diagram of an example of a graphics pipeline according to an embodiment;

FIGs. 14A-14B are block diagrams of examples of graphics pipeline programming according to an embodiment;

FIG. 15 is a block diagram of an example of a graphics software architecture according to an embodiment;

25 FIG. 16 is a block diagram of an example of an intellectual property (IP) core development system according to an embodiment; and

FIG. 17 is a block diagram of an example of a system on a chip integrated circuit according to an embodiment.

30

DETAILED DESCRIPTION

FIG. 1 shows a computing architecture 21 that may be used to execute visually intensive computer graphics applications such as 3D (three-dimensional) computer games, flight simulators and other 3D imaging applications involving user interaction, scene management and rendering, physics modeling, artificial intelligence

and other relatively complex functions. In the illustrated example, the graphics application generates a general purpose graphics processing unit (GPGPU) workload 20. The GPGPU workload 20 may generally perform non-3D functions such as, for example, image blurring, and other non-3D computational operations. A workload converter 22 may receive the GPGPU workload 20 and convert the GPGPU workload 20 to a 3D workload 24, which may be sent to a graphics processor 26 (e.g., graphics processing unit/GPU). A command streamer 28 on the graphics processor 26 may in turn dispatch the 3D workload 24 to a 3D pipeline 30 on the graphics processor 26. The 3D pipeline 30 may include programmable and fixed function elements that perform various tasks within the elements and/or spawn execution threads. More particularly, the 3D pipeline 30 may include fixed function units such as, for example, depth-stencil modules, output merger modules and additional support caches (e.g., render cache) not available to compute kernels.

The illustrated architecture 21 therefore enhances efficiency by eliminating overhead associated with switching between the 3D pipeline 30 and a GPGPU pipeline (not shown) and avoiding hardware restrictions such as memory hierarchies supporting execution of the GPGPU workload 20 outside of the 3D pipeline 30. Additionally, the illustrated architecture 21 minimizes the occurrence of cache flushes and eliminates execution lane (e.g., single instruction multiple data/SIMD) restrictions on resource accesses (e.g., typed/un-typed unordered access views/UAV, shader resource views, raw access, etc.). Indeed, enabling the GPGPU workload 20 to leverage the 3D-specific fixed function units of the 3D pipeline 30, may further enhance efficiency.

With continuing reference to FIGs. 1 and 2, the workload converter 22 may identify a plurality of thread groups in the GPGPU workload 20 and map the plurality of thread groups to a 3D matrix 36 of cubes. Each cube may represent an array of two-dimensional (2D) primitives, wherein each primitive is a quad (e.g., a four-sided shape). Thus, a particular cube 40 may have the dimensions NumThreads.Z*NumThreads.Y*NumThreads.Z and may include an array of quads 38. Additionally, a particular dispatch call may have the dimensions Dispatch.X*Dispatch.Y*Dispatch.Z.

Turning now to FIG. 3, a semantic relationship between a set of thread groups and a dispatch interface is shown. In the illustrated example, a dispatch summary 32 demonstrates that a dispatch of thread groups may generally have an X-dimension (“dim 0” containing five thread groups), a Y-dimension (“dim 1” containing

three thread groups) and a Z-dimension (“dim 2” containing two thread groups). Thus, a single dispatch call invokes $5*3*2=30$ thread groups, in the illustrated example. Moreover, a thread group summary 34 demonstrates a given thread group (e.g., group (2, 1, 0), in the illustrated example), may include a grid of $10*8*3=240$ threads.

5 In general, a vertex shader within the 3D pipeline may identify, for each of a plurality of two-dimensional (2D) primitives associated with the 3D workload, a plurality of vertices and generate system values (“SV”) for the plurality of vertices. Additionally, the system values may have a one-to-one relationship with a pixel shader interface of the 3D pipeline.

10 Dispatch

For example, a dispatch call may be modeled by the expression:

$\text{dispatch.x,y,z} = \text{Draw}(\text{primitive count} = \text{dispatch.x} * \text{dispatch.y} * \text{dispatch.z} * \text{numThreads.z})$.

Where the primitive topology is quad. Thus, $\text{dispatch.x,y,z}=5,3,2$ may
15 yield:

$\text{Draw}(\text{primitive count} = 5*3*2*3)$.

Number of threads

The number of threads may be modeled by the expression:

$\text{numThreads.xyz} = \text{Quad dimension}(\text{x,y}) = (\text{numThread.x}, \text{numThread.y})$.

20 Thus, $\text{numThreads.xyz} = 10,8,3$ may yield:

$\text{Quad dimension}(\text{x,y}) = (10,8)$.

Group Thread Identifier

A group thread identifier system value may be determined by the expression:

25 $\text{SV_GroupThreadID}(\text{x,y,z}) = \text{SV_Position}(\text{x}, \text{y}, \text{z})$

Where each value maps directly to a pixel position (e.g., has a one-to-one relationship with the pixel shader interface.

Thus, $\text{SV_GroupThreadID}(7,5,0)$ may yield:

$\text{SV_Position}(7,5,0)$.

30 Group Identifier

A group identifier system value may be determined by the expression:

$\text{SV_GroupID}(\text{x,y,z}) = (\text{SV_PrimitiveID}/\text{numThreads.Z} \% \text{dispatch.x}, \text{SV_PrimitiveID}/\text{numThreads.z}/\text{dispatch.x}, \text{SV_PrimitiveID}/\text{numThread.z}/(\text{dispatch.x} * \text{dispatch.y}))$

Which maps each groupID in the original GPGPU workload to a subset of unique primitiveIDs to effectively emulate the work done in the original GPGPU workload.

Where % represents the remainder.

5 Thus, SV_GroupID(2,1,0) may yield:

$$(21/3\%5, 21/3/5, 21/3/(5*3))=(2,1,0)$$

where $SV_PrimitiveID = (2 + 1*dispatch.x + 0*dispatch.x*dispatch.y) * numThreads.z = 21$

Group Index

10 A group index system value may be determined by the expression:

$$SV_GroupIndex(x) = SV_GroupThreadID.z * numThreads.x * numThreads.y + SV_GroupThreadID.y * numThreads.x + SV_GroupThreadID.x$$

Thus, SV_GroupIndex(57) may yield:

15 $0*10*8 + 5 * 10 + 7 = 57$

Dispatch Thread Identifier

A dispatch thread identifier system value may be determined by the expression:

20 $SV_DispatchThreadID(x,y,z) = SV_GroupThreadID + SV_GroupID * numThreads$

Thus, SV_DispatchThreadID(x,y,z) for the above example may yield:

$$(7,5,0) + (2,1,0) * (10,8,3) = (27,13,0)$$

FIG. 4A shows a method 42 of operating a computing architecture such as, for example, the computing architecture 21 (FIG. 1), already discussed. The method 42 may be implemented in one or more modules as a set of logic instructions stored in a machine- or computer-readable storage medium such as random access memory (RAM), read only memory (ROM), programmable ROM (PROM), flash memory, etc., as configurable logic such as, for example, programmable logic arrays (PLAs), field programmable gate arrays (FPGAs), complex programmable logic devices (CPLDs), as fixed-functionality logic hardware using circuit technology such as, for example, application specific integrated circuit (ASIC), complementary metal oxide semiconductor (CMOS) or transistor-transistor logic (TTL) technology, or any combination thereof. For example, computer program code to carry out operations shown in the method 42 may be written in any combination of one or more

programming languages, including an object oriented programming language such as C++ or the like and conventional procedural programming languages, such as the "C" programming language or similar programming languages. Moreover, the method 42 may be implemented using any of the herein mentioned circuit technologies.

5 Illustrated processing block 44 provides for receiving a GPGPU workload, wherein the GPGPU workload may be converted to a 3D workload at block 46. Block 48 may dispatch the 3D workload to a 3D pipeline. In one example, block 48 includes calling a plurality of non-GPGPU shader threads. Block 50 may identify, for each of a plurality of 2D primitives associated with the 3D workload, a plurality of vertices.
10 Illustrated block 52 generates system values for the plurality of vertices, wherein the system values have a one-to-one relationship with a pixel shader interface associated with the 3D pipeline.

Turning now to FIG. 4B, a method 54 of converting a GPGPU workload to a 3D workload is shown. The method 54 may therefore be implemented in a workload
15 converter such as, for example, the workload converter 22 (FIG. 1), already discussed and/or readily substituted for processing block 46 (FIG. 4A), already discussed. The method 54 may be implemented in one or more modules as a set of logic instructions stored in a machine- or computer-readable storage medium such as RAM, ROM, PROM, flash memory, etc., as configurable logic such as, for example, PLAs, FPGAs,
20 CPLDs, as fixed-functionality logic hardware using circuit technology such as, for example, ASIC, CMOS or TTL technology, or any combination thereof. Illustrated processing block 56 may identify a plurality of thread groups in the GPGPU workload, wherein the plurality of thread groups may be mapped to a 3D matrix of cubes at block 58. Each cube in the 3D matrix of cubes may represent an array of 2D primitives.
25 Additionally, each primitive may be a quad.

FIG. 5 shows a computing device 60 (e.g., server, desktop computer, tablet computer, convertible tablet, smart phone, mobile Internet device/MID, game console, media player, wearable computer, etc.). In the illustrated example, a semiconductor die 62 (e.g., chip, substrate) includes a host processor 64 and a graphics processor 66. The
30 host processor 64 may include system memory 68 (e.g., RAM) and a central processing unit (CPU) 70 having a driver 72 with a workload converter 74. The workload converter 74 may have similar functionality to that of the workload converter 22 (FIG. 1), already discussed. Thus, the workload converter 74 may convert a GPGPU

workload to a 3D workload and send the 3D workload to the graphics processor 66 for execution.

The illustrated graphics processor 66 includes a plurality of processing units 90 and a command streamer 76 to dispatch the 3D workload to a 3D pipeline 78 on the graphics processor 66. The command streamer 76 may call a plurality of non-GPGPU
 5 shader threads to dispatch the 3D workload. The 3D pipeline may include a pixel shader interface 80 and a vertex shader 82 that identifies, for each of a plurality of 2D primitives associated with the 3D workload, a plurality of vertices. The vertex shader 82 may also generate system values for the plurality of vertices, wherein the system
 10 values have a one-to-one relationship with the pixel shader interface. The illustrated computing device 60 also includes mass storage 84 (e.g., hard disk drive/HDD, optical disk, flash memory, solid state disk/SSD), a display 86 and dedicated graphics memory 88. The graphics processor 66 may therefore include a display interface 92 (e.g., frame
 15 buffer) to receive data associated with the execution of the 3D workload, wherein the illustrated display device 86 generates a display output based on the data in the display interface 92.

Thus, the following pseudo code might reflect the operation of a conventional GPGPU Compute Shader:

```

    Numthreads(x,y,z)    CS(    SV_GroupThreadID,    SV_GroupID,
20  SV_DispatchThreadID, SV_GroupIndex)
    {
        // code for compute shader
        ..
    }
  
```

25 By contrast, the following pseudo code may reflect the operation of an automatically generated 3D pixel shader:

```

    void PSmain( PS_INPUT input )
    {
        uint3 SV_GroupThreadID = Pos;
        uint3 SV_GroupID = uint3(primitiveId /numThreads.z % dispatch.x,
30  primitiveId /numThreads.z/dispatch.x,  primitiveId/numThreads.z/(dispatch.x *
        dispatch.y));
        uint3 SV_DispatchThreadID = SV_GroupThreadID+ SV_GroupID *
        numThreads;
  
```

```

uint   SV_GroupIndex   =   SV_GroupThreadID.z*numThreads.x*
numThreads.y +
        SV_GroupThreadID.y* numThreads.x +
        SV_GroupThreadID.x;
5      //code for Compute Shader
      ...
    }
    Where PS_INPUT is defined as
    struct PS_INPUT
10    {
        sample float4 Pos : SV_POSITION;
        uint primitiveId : SV_PrimitiveID;
    };

```

System Overview

15 FIG. 6 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
20 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
25 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
30 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
5 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 as 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
10 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
15 which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor 102.

In some embodiments, processor 102 is coupled to a processor bus 110 to
20 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
25 100, 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-
30 change memory device, or some other memory device having suitable performance to serve as process memory. In one embodiment the memory device 120 can operate as system memory for the system 100, to store data 122 and instructions 121 for use when the one or more processors 102 executes an application or process. Memory controller hub 116 also couples with an optional external graphics processor 112, which may

communicate with the one or more graphics processors 108 in processors 102 to perform graphics and media operations.

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

FIG. 7 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. 7 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. 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
5 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
10 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
15 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
20 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,
25 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,
30 each of the processor cores 202-202N and graphics processor 208 use embedded memory modules 218 as a shared Last Level Cache.

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

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

FIG. 8 is a block diagram of a graphics processor 300, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores. In some embodiments, the graphics processor communicates via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor memory. In some embodiments, graphics processor 300 includes a memory interface 314 to access memory. Memory interface 314 can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

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

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

In some embodiments, GPE 310 includes a 3D pipeline 312 for performing 3D operations, such as rendering three-dimensional images and scenes using processing

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

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

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

3D/Media Processing

FIG. 9 is a block diagram of a graphics processing engine 410 of a graphics processor in accordance with some embodiments. In one embodiment, the GPE 410 is a version of the GPE 310 shown in FIG. 8. Elements of FIG. 9 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

In some embodiments, GPE 410 couples with a command streamer 403, which provides a command stream to the GPE 3D and media pipelines 412, 416. In some embodiments, command streamer 403 is coupled to memory, which can be

system memory, or one or more of internal cache memory and shared cache memory. In some embodiments, command streamer 403 receives commands from the memory and sends the commands to 3D pipeline 412 and/or media pipeline 416. The commands are directives fetched from a ring buffer, which stores commands for the 3D and media pipelines 412, 416. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The 3D and media pipelines 412, 416 process the commands by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to an execution unit array 414. In some embodiments, execution unit array 414 is scalable, such that the array includes a variable number of execution units based on the target power and performance level of GPE 410.

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

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

In some embodiments, motion estimation engine 434 provides hardware acceleration for video operations by performing video acceleration functions such as motion vector estimation and prediction on video data. The motion estimation engine determines motion vectors that describe the transformation of image data between successive video frames. In some embodiments, a graphics processor media codec uses video motion estimation engine 434 to perform operations on video at the macro-block

level that may otherwise be too computationally intensive to perform with a general-purpose processor. In some embodiments, motion estimation engine 434 is generally available to graphics processor components to assist with video decode and processing functions that are sensitive or adaptive to the direction or magnitude of the motion within video data.

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

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

Execution Units

FIG. 10 is a block diagram of another embodiment of a graphics processor 500. Elements of FIG. 10 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

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

In some embodiments, graphics processor 500 receives batches of commands via ring interconnect 502. The incoming commands are interpreted by a

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

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

FIG. 11 illustrates thread execution logic 600 including an array of processing elements employed in some embodiments of a GPE. Elements of FIG. 11 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

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

In some embodiments, execution unit array 608A-608N is primarily used to execute “shader” programs. In some embodiments, the execution units in array 608A-608N execute an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders).

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

The execution unit instruction set includes single instruction multiple data (SIMD) instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (Quad-Word (QW) size data elements),

eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

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

 During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 600 via thread spawning and dispatch logic. In some
15 embodiments, thread execution logic 600 includes a local thread dispatcher 604 that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units 608A-608N. For example, the geometry pipeline (e.g., 536 of FIG. 10) dispatches vertex processing, tessellation, or geometry processing threads to thread execution logic 600 (FIG. 11). In
20 some embodiments, thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

 Once a group of geometric objects has been processed and rasterized into pixel data, pixel shader 602 is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers,
25 etc.). In some embodiments, pixel shader 602 calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. In some embodiments, pixel shader 602 then executes an application programming interface (API)-supplied pixel shader program. To execute the pixel shader program, pixel shader 602 dispatches threads to an execution unit (e.g., 608A) via thread dispatcher 604. In some
30 embodiments, pixel shader 602 uses texture sampling logic in sampler 610 to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

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

FIG. 12 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
10 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
15 support instructions in a 128-bit format 710. A 64-bit compacted instruction format 730 is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit format 710 provides access to all instruction options, while some options and operations are restricted in the 64-bit format
20 730. The native instructions available in the 64-bit format 730 vary by embodiment. In some embodiments, the instruction is compacted in part using a set of index values in an index field 713. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit format 710.

For each format, instruction opcode 712 defines the operation that the
25 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
30 unit performs each instruction across all data channels of the operands. In some embodiments, instruction control field 714 enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For 128-bit instructions 710 an exec-size field 716 limits the number of data

channels that will be executed in parallel. In some embodiments, exec-size field 716 is not available for use in the 64-bit compact instruction format 730.

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

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

In some embodiments, the 128-bit instruction format 710 includes an access/address mode field 726, which specifies an address mode and/or an access mode for the instruction. In one embodiment the access mode to define a data access alignment for the instruction. Some embodiments support access modes including a 16-byte aligned access mode and a 1-byte aligned access mode, where the byte alignment of the access mode determines the access alignment of the instruction operands. For example, when in a first mode, the instruction 710 may use byte-aligned addressing for source and destination operands and when in a second mode, the instruction 710 may use 16-byte-aligned addressing for all source and destination operands.

In one embodiment, the address mode portion of the access/address mode field 726 determines whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction 710 directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed based on an address register value and an address immediate field in the instruction.

In some embodiments instructions are grouped based on opcode 712 bit-fields to simplify Opcode decode 740. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group 742

includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group 742 shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group 744 (e.g.,
5 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
10 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

15 FIG. 13 is a block diagram of another embodiment of a graphics processor 800. Elements of FIG. 13 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
20 820, 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
25 800 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.

30 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.

5 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.

10 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
15 special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to graphics pipeline 820. In some embodiments, if tessellation is not used, tessellation components 811, 813, 817 can be bypassed.

In some embodiments, complete geometric objects can be processed by a
20 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
25 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
30 component 873 in the render output pipeline 870 dispatches pixel shaders to convert the geometric objects into their per pixel representations. In some embodiments, pixel shader logic is included in thread execution logic 850. In some embodiments, an application can bypass the rasterizer 873 and access un-rasterized vertex data via a stream out unit 823.

The graphics processor 800 has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units 852A, 852B and associated cache(s) 851, texture and media sampler 854, and texture/sampler cache
5 858 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
10 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
15 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
20 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
25 command to the media engine 837. In some embodiments, media engine 337 includes thread spawning functionality to spawn threads for dispatch to thread execution logic 850 via thread dispatcher 831.

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

may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

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

Graphics Pipeline Programming

FIG. 14A is a block diagram illustrating a graphics processor command format 900 according to some embodiments. FIG. 14B is a block diagram illustrating a graphics processor command sequence 910 according to an embodiment. The solid lined boxes in FIG. 14A 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. 14A 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. 14B shows an exemplary graphics processor command sequence 910. In some embodiments, software or firmware of a data processing system that features an embodiment of a graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only as embodiments are not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.

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

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

In some embodiments, a pipeline control command 914 configures a graphics pipeline for operation and is used to program the 3D pipeline 922 and the

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

5 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
10 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
15 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. In one example, the pipeline determination 920 follows the 3D pipeline 922 in all instances.

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

25 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.
30 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
5 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.

10 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
15 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
20 explicitly related to the rendering of graphics primitives.

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

30 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
5 operations.

Graphics Software Architecture

FIG. 15 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
10 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
15 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
20 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. When the Direct3D API is in use, the operating system 1020 uses a front-end
25 shader compiler 1024 to compile any shader instructions 1012 in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. In some embodiments, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application 1010.

In some embodiments, user mode graphics driver 1026 contains a back-end
30 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

5 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
10 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
15 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. 16 is a block diagram illustrating an IP core development system 1100 that may be used to manufacture an integrated circuit to perform operations according
20 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
25 design, test, and verify the behavior of the IP core using a simulation model 1112. The simulation model 1112 may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design can then be created or synthesized from the simulation model 11001112. The RTL design 1115 is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers,
30 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.

FIG. 17 is a block diagram illustrating an exemplary system on a chip integrated circuit 1200 that may be fabricated using one or more IP cores, according to an embodiment. The exemplary integrated circuit includes one or more application processors 1205 (e.g., CPUs), at least one graphics processor 1210, and may additionally include an image processor 1215 and/or a video processor 1220, any of which may be a modular IP core from the same or multiple different design facilities. The integrated circuit includes peripheral or bus logic including a USB controller 1225, UART controller 1230, an SPI/SDIO controller 1235, and an I2S/I2C controller 1240. Additionally, the integrated circuit can include a display device 1245 coupled to one or more of a high-definition multimedia interface (HDMI) controller 1250 and a mobile industry processor interface (MIPI) display interface 1255. Storage may be provided by a flash memory subsystem 1260 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller 1265 for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine 1270.

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

Additional Notes and Examples:

Example 1 may include a graphics application computing device comprising a host processor including a workload converter to convert a general purpose graphics processing unit (GPGPU) workload to a three-dimensional (3D) workload, and a 3D pipeline and a command streamer to dispatch the 3D workload to the 3D pipeline, a

display interface to receive data associated with an execution of the 3D workload, and a display device to generate a display output based on the data in the display interface.

Example 2 may include the computing device of Example 1, wherein the workload converter is to identify a plurality of thread groups in the GPGPU workload and map the plurality of thread groups to a 3D matrix of cubes.

Example 3 may include the computing device of Example 2, wherein each cube is to represent an array of two-dimensional (2D) primitives.

Example 4 may include the computing device of Example 3, wherein each primitive is to be a quad.

Example 5 may include the computing device of Example 1, wherein the 3D pipeline includes a pixel shader interface, and a vertex shader to identify, for each of a plurality of two-dimensional (2D) primitives associated with the 3D workload, a plurality of vertices and generate system values for the plurality of vertices, wherein the system values are to have a one-to-one relationship with the pixel shader interface.

Example 6 may include the computing device of any one of Examples 1 to 5, wherein the command streamer is to call a plurality of non-GPGPU shader threads to dispatch the 3D workload.

Example 7 may include an efficiency enhancement computing architecture comprising a workload converter to receive a general purpose graphics processing unit (GPGPU) workload and convert the GPGPU workload to a three-dimensional (3D) workload, a 3D pipeline, and a command streamer to dispatch the 3D workload to the 3D pipeline.

Example 8 may include the computing architecture of Example 7, wherein the workload converter is to identify a plurality of thread groups in the GPGPU workload and map the plurality of thread groups to a 3D matrix of cubes.

Example 9 may include the computing architecture of Example 8, wherein each cube is to represent an array of two-dimensional (2D) primitives.

Example 10 may include the computing architecture of Example 9, wherein each primitive is to be a quad.

Example 11 may include the computing architecture of Example 7, wherein the 3D pipeline includes a pixel shader interface, and a vertex shader to identify, for each of a plurality of two-dimensional (2D) primitives associated with the 3D workload, a plurality of vertices and generate system values for the plurality of vertices, wherein the system values are to have a one-to-one relationship with the pixel shader interface.

Example 12 may include the computing architecture of any one of Examples 7 to 11, wherein the command streamer is to call a plurality of non-GPGPU shader threads to dispatch the 3D workload.

5 Example 13 may include a method of operating an efficiency enhancement computing architecture, comprising receiving a general purpose graphics processing unit (GPGPU) workload, converting the GPGPU workload to a three-dimensional (3D) workload, and dispatching the 3D workload to a 3D pipeline.

Example 14 may include the method of Example 13, wherein converting the GPGPU workload to the 3D workload includes identifying a plurality of thread groups
10 in the GPGPU workload, and mapping the plurality of thread groups to a 3D matrix of cubes. 15. The method of Example 14, wherein each cube represents an array of two-dimensional (2D) primitives.

Example 16 may include the method of Example 15, wherein each 2D primitive is a quad.

15 Example 17 may include the method of Example 13, further including identifying, for each of a plurality of two-dimensional (2D) primitives associated with the 3D workload, a plurality of vertices, and generating system values for the plurality of vertices, wherein the system values have a one-to-one relationship with a pixel shader interface associated with the 3D pipeline.

20 Example 18 may include the method of any one of Examples 13 to 17, wherein dispatching the 3D workload to the 3D pipeline includes calling a plurality of non-GPGPU shader threads.

Example 19 may include at least one computer readable storage medium comprising a set of instructions, which when executed by a computing device, cause
25 the computing device to receive a general purpose graphics processing unit (GPGPU) workload, convert the GPGPU workload to a three-dimensional (3D) workload, and dispatch the 3D workload to a 3D pipeline.

Example 20 may include the at least one computer readable storage medium of Example 19, wherein the instructions, when executed, cause the computing device
30 to identify a plurality of thread groups in the GPGPU workload, and map the plurality of thread groups to a 3D matrix of cubes to convert the GPGPU workload to the 3D workload.

Example 21 may include the at least one computer readable storage medium of Example 20, wherein each cube is to represent an array of two-dimensional (2D) primitives.

Example 22 may include the at least one computer readable storage medium
5 of Example 21, wherein each 2D primitive is to be a quad.

Example 23 may include the at least one computer readable storage medium of Example 19, wherein the instructions, when executed, cause the computing device to identify, for each of a plurality of two-dimensional (2D) primitives associated with the 3D workload, a plurality of vertices, and generate system values for the plurality of
10 vertices, wherein the system values are to have a one-to-one relationship with a pixel shader interface associated with the 3D pipeline.

Example 24 may include the at least one computer readable storage medium of any one of Examples 19 to 23, wherein the instructions, when executed, cause the computing device to call a plurality of non-GPGPU shader threads to dispatch the 3D
15 workload.

Example 25 may include an efficiency enhancement computing architecture comprising means for receiving a general purpose graphics processing unit (GPGPU) workload, means for converting the GPGPU workload to a three-dimensional (3D) workload, and means for dispatching the 3D workload to a 3D pipeline.

Example 26 may include the computing architecture of Example 25,
20 wherein the means for converting the GPGPU workload to the 3D workload includes means for identifying a plurality of thread groups in the GPGPU workload, and means for mapping the plurality of thread groups to a 3D matrix of cubes.

Example 27 may include the computing architecture of Example 26,
25 wherein each cube is to represent an array of two-dimensional (2D) primitives.

Example 28 may include the computing architecture of Example 27, wherein each 2D primitive is to be quad.

Example 29 may include the computing architecture of Example 25, further including means for identifying, for each of a plurality of two-dimensional (2D)
30 primitives associated with the 3D workload, a plurality of vertices, and means for generating system values for the plurality of vertices, wherein the system values have a one-to-one relationship with a pixel shader interface associated with the 3D pipeline.

Example 30 may include the computing architecture of any one of Examples 25 to 29, wherein the means for dispatching the 3D workload to the 3D pipeline includes means for calling a plurality of non-GPGPU shader threads.

Techniques described herein may therefore remove CPU and GPU overhead
5 associated with switching between 3D and GPGPU pipelines. Techniques may also simplify hardware design by removing the need for hardware to manage GPGPU threads, which may in turn save die space. Moreover, scheduling 3D threads on GPUs may be more lightweight than GPGPU threads. As a result, increased performance and reduced power consumption may be achieved.

10 The term “coupled” may be used herein to refer to any type of relationship, direct or indirect, between the components in question, and may apply to electrical, mechanical, fluid, optical, electromagnetic, electromechanical or other connections. In addition, the terms “first”, “second”, etc. may be used herein only to facilitate discussion, and carry no particular temporal or chronological significance unless
15 otherwise indicated. Additionally, it is understood that the indefinite articles “a” or “an” carries the meaning of “one or more” or “at least one”.

Those skilled in the art will appreciate from the foregoing description that the broad techniques of the embodiments can be implemented in a variety of forms. Therefore, while the embodiments have been described in connection with particular
20 examples thereof, the true scope of the embodiments should not be so limited since other modifications will become apparent to the skilled practitioner upon a study of the drawings, specification, and following claims.

CLAIMS

We claim:

- 5 1. A graphics application computing device comprising:
 a host processor including a workload converter to convert a general purpose
graphics processing unit (GPGPU) workload to a three-dimensional (3D) workload;
 a graphics processor including:
 a 3D pipeline, and
10 a command streamer to dispatch the 3D workload to the 3D pipeline.
 a display interface to receive data associated with an execution of the 3D
workload; and
 a display device to generate a display output based on the data in the display
interface.
- 15 2. The computing device of claim 1, wherein the workload converter is to
identify a plurality of thread groups in the GPGPU workload and map the plurality of
thread groups to a 3D matrix of cubes.
- 20 3. The computing device of claim 2, wherein each cube is to represent an
array of two-dimensional (2D) primitives.
4. The computing device of claim 3, wherein each primitive is to be a
quad.
- 25 5. The computing device of claim 1, wherein the 3D pipeline includes:
 a pixel shader interface; and
 a vertex shader to identify, for each of a plurality of two-dimensional (2D)
primitives associated with the 3D workload, a plurality of vertices and generate
30 system values for the plurality of vertices, wherein the system values are to have a
one-to-one relationship with the pixel shader interface.

6. The computing device of any one of claims 1 to 5, wherein the command streamer is to call a plurality of non-GPGPU shader threads to dispatch the 3D workload.

5 7. An efficiency enhancement computing architecture comprising:
a workload converter to receive a general purpose graphics processing unit (GPGPU) workload and convert the GPGPU workload to a three-dimensional (3D) workload;
a 3D pipeline; and
10 a command streamer to dispatch the 3D workload to the 3D pipeline.

8. The computing architecture of claim 7, wherein the workload converter is to identify a plurality of thread groups in the GPGPU workload and map the plurality of thread groups to a 3D matrix of cubes.

15 9. The computing architecture of claim 8, wherein each cube is to represent an array of two-dimensional (2D) primitives.

10 10. The computing architecture of claim 9, wherein each primitive is to be a quad.

11. The computing architecture of claim 7, wherein the 3D pipeline includes:
a pixel shader interface; and
25 a vertex shader to identify, for each of a plurality of two-dimensional (2D) primitives associated with the 3D workload, a plurality of vertices and generate system values for the plurality of vertices, wherein the system values are to have a one-to-one relationship with the pixel shader interface.

30 12. The computing architecture of any one of claims 7 to 11, wherein the command streamer is to call a plurality of non-GPGPU shader threads to dispatch the 3D workload.

13. A method of operating an efficiency enhancement computing architecture, comprising:

receiving a general purpose graphics processing unit (GPGPU) workload;
converting the GPGPU workload to a three-dimensional (3D) workload; and
5 dispatching the 3D workload to a 3D pipeline.

14. The method of claim 13, wherein converting the GPGPU workload to the 3D workload includes:

identifying a plurality of thread groups in the GPGPU workload; and
10 mapping the plurality of thread groups to a 3D matrix of cubes.

15. The method of claim 14, wherein each cube represents an array of two-dimensional (2D) primitives.

16. The method of claim 15, wherein each 2D primitive is a quad.

17. The method of claim 13, further including:

identifying, for each of a plurality of two-dimensional (2D) primitives associated with the 3D workload, a plurality of vertices; and
20 generating system values for the plurality of vertices, wherein the system values have a one-to-one relationship with a pixel shader interface associated with the 3D pipeline.

18. The method of any one of claims 13 to 17, wherein dispatching the 3D workload to the 3D pipeline includes calling a plurality of non-GPGPU shader threads.

19. At least one computer readable storage medium comprising a set of instructions, which when executed by a computing device, cause the computing
30 device to:

receive a general purpose graphics processing unit (GPGPU) workload;
convert the GPGPU workload to a three-dimensional (3D) workload; and
dispatch the 3D workload to a 3D pipeline.

20. The at least one computer readable storage medium of claim 19,
wherein the instructions, when executed, cause the computing device to:
identify a plurality of thread groups in the GPGPU workload; and
map the plurality of thread groups to a 3D matrix of cubes to convert the
5 GPGPU workload to the 3D workload.

21. The at least one computer readable storage medium of claim 20,
wherein each cube is to represent an array of two-dimensional (2D) primitives.

10 22. The at least one computer readable storage medium of claim 21,
wherein each 2D primitive is to be a quad.

23. The at least one computer readable storage medium of claim 19,
wherein the instructions, when executed, cause the computing device to:
15 identify, for each of a plurality of two-dimensional (2D) primitives associated
with the 3D workload, a plurality of vertices; and
generate system values for the plurality of vertices, wherein the system values
are to have a one-to-one relationship with a pixel shader interface associated with the
3D pipeline.

20 24. The at least one computer readable storage medium of any one of
claims 19 to 23, wherein the instructions, when executed, cause the computing device
to call a plurality of non-GPGPU shader threads to dispatch the 3D workload.

25 25. An efficiency enhancement computing architecture comprising means
for performing the method of any one of claims 13 to 17.

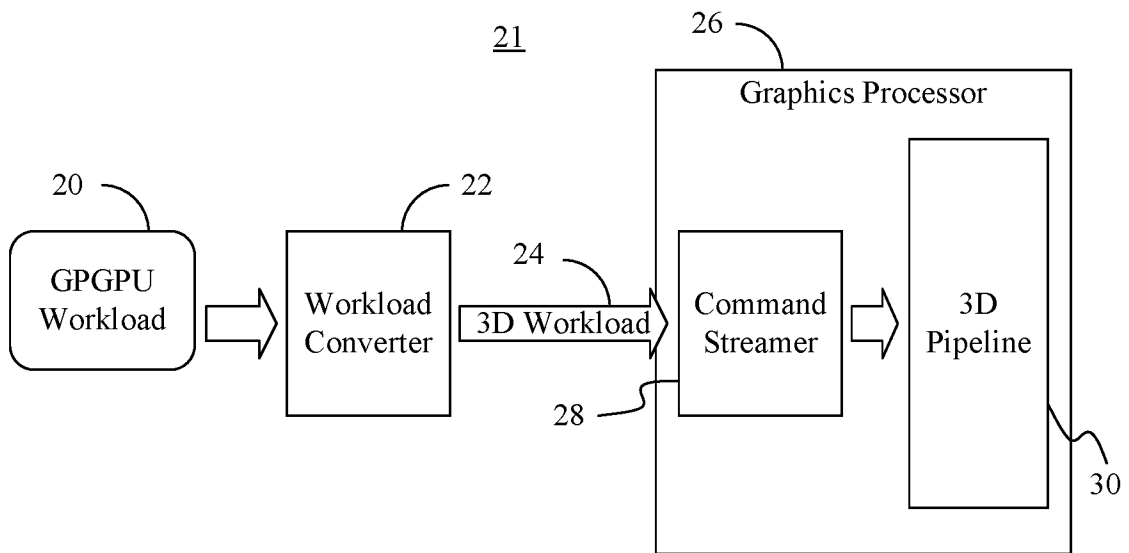
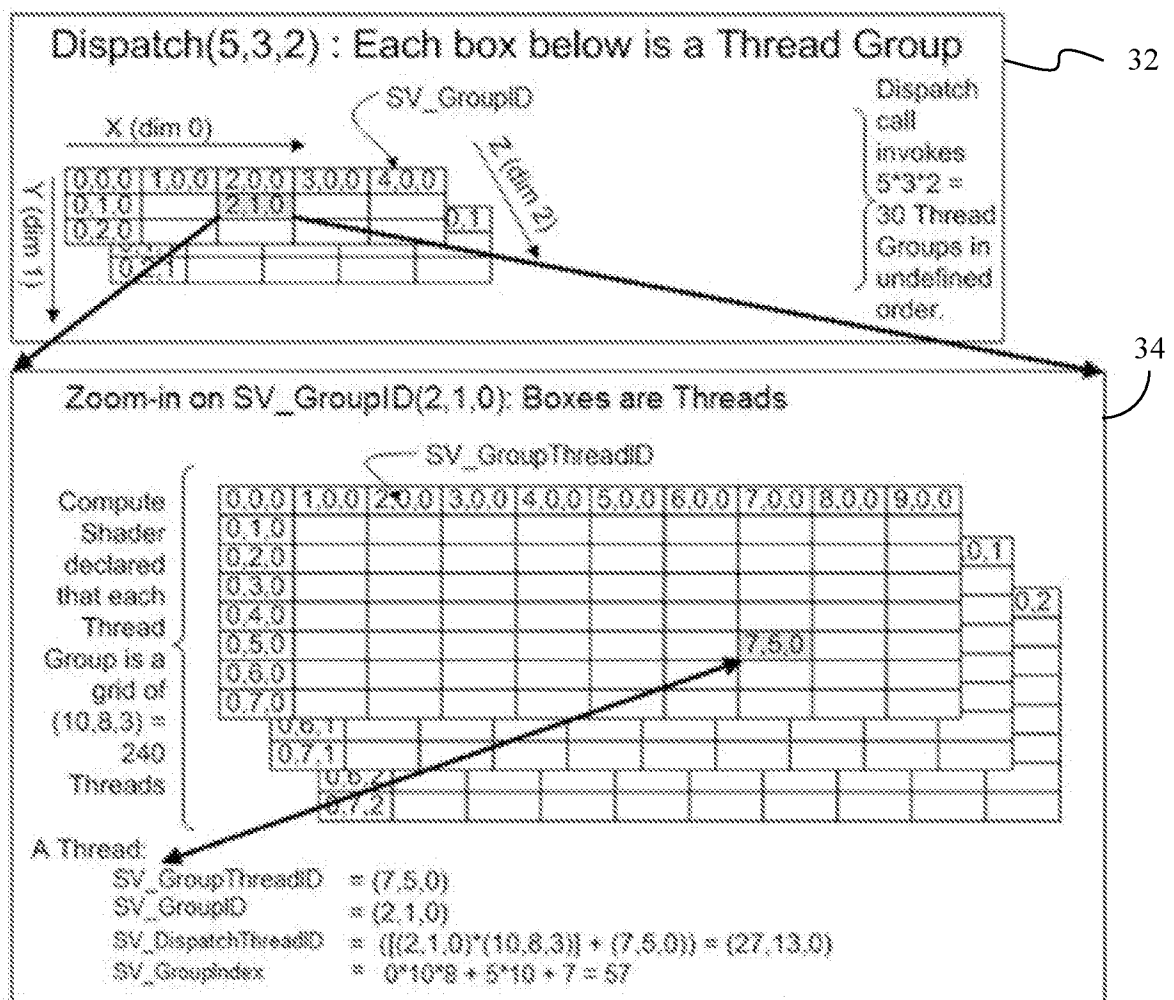
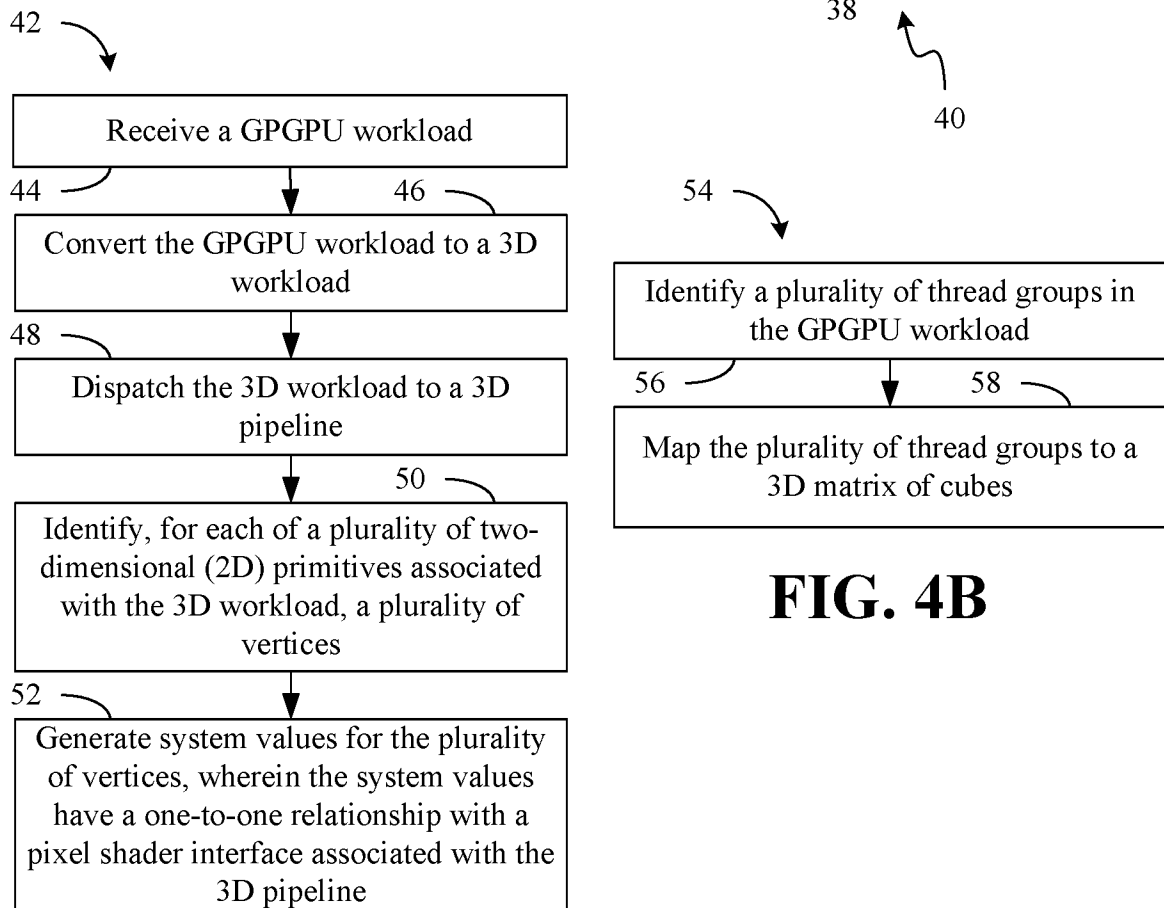
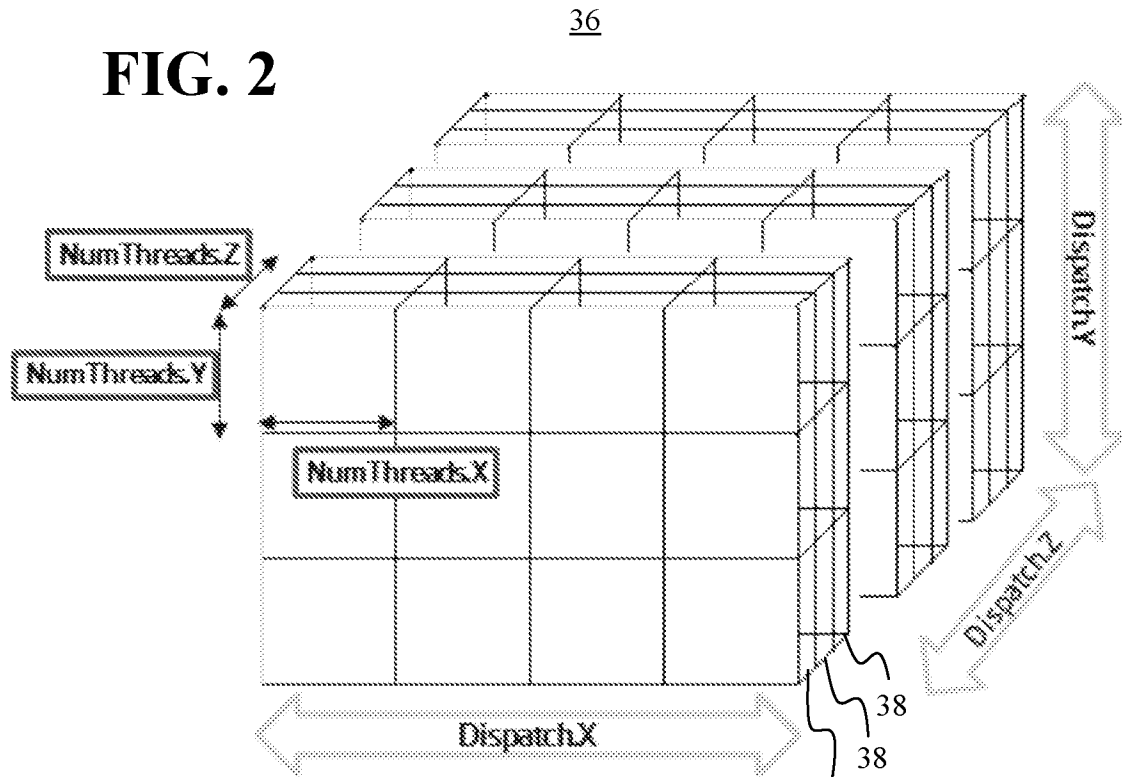
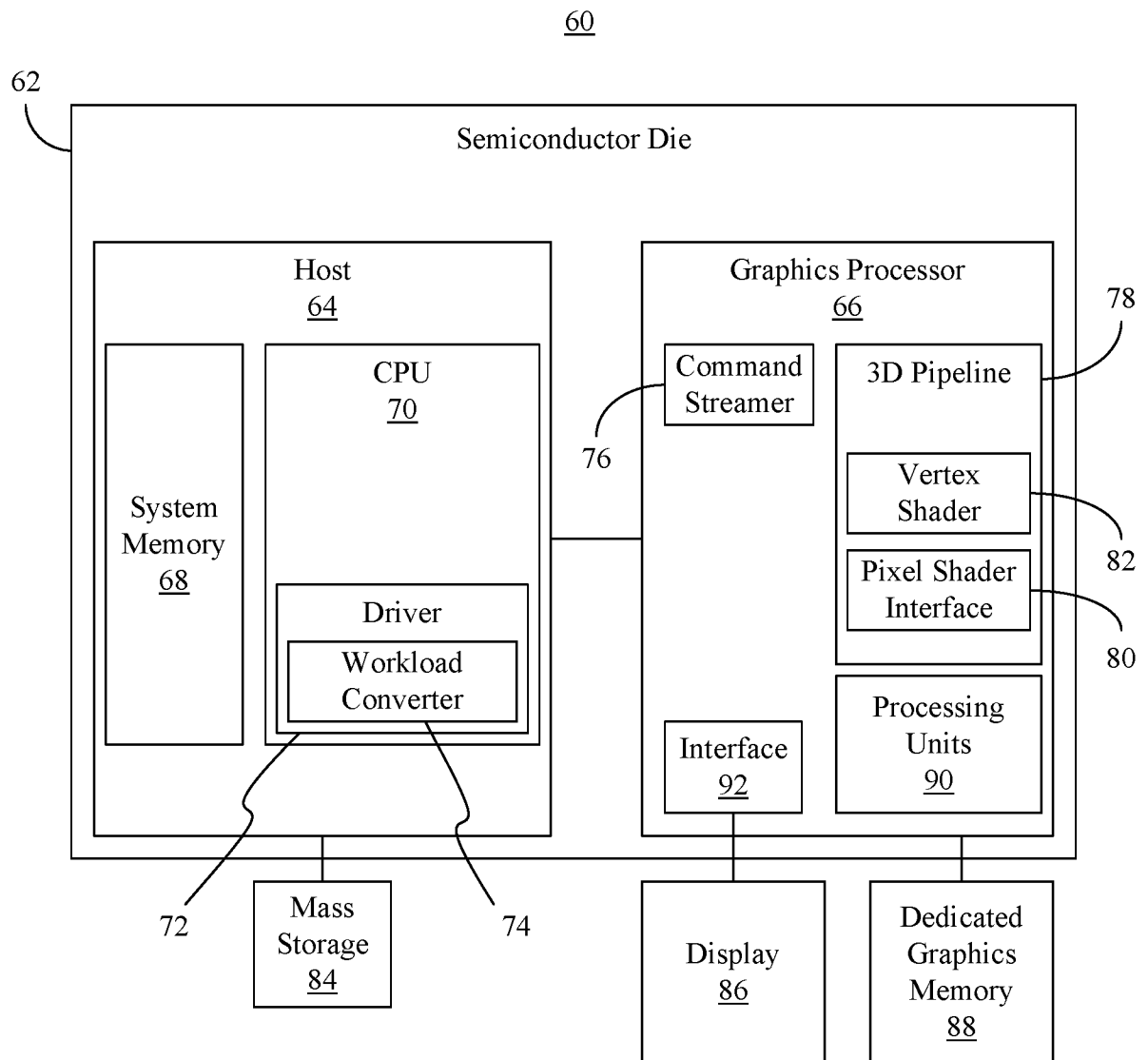
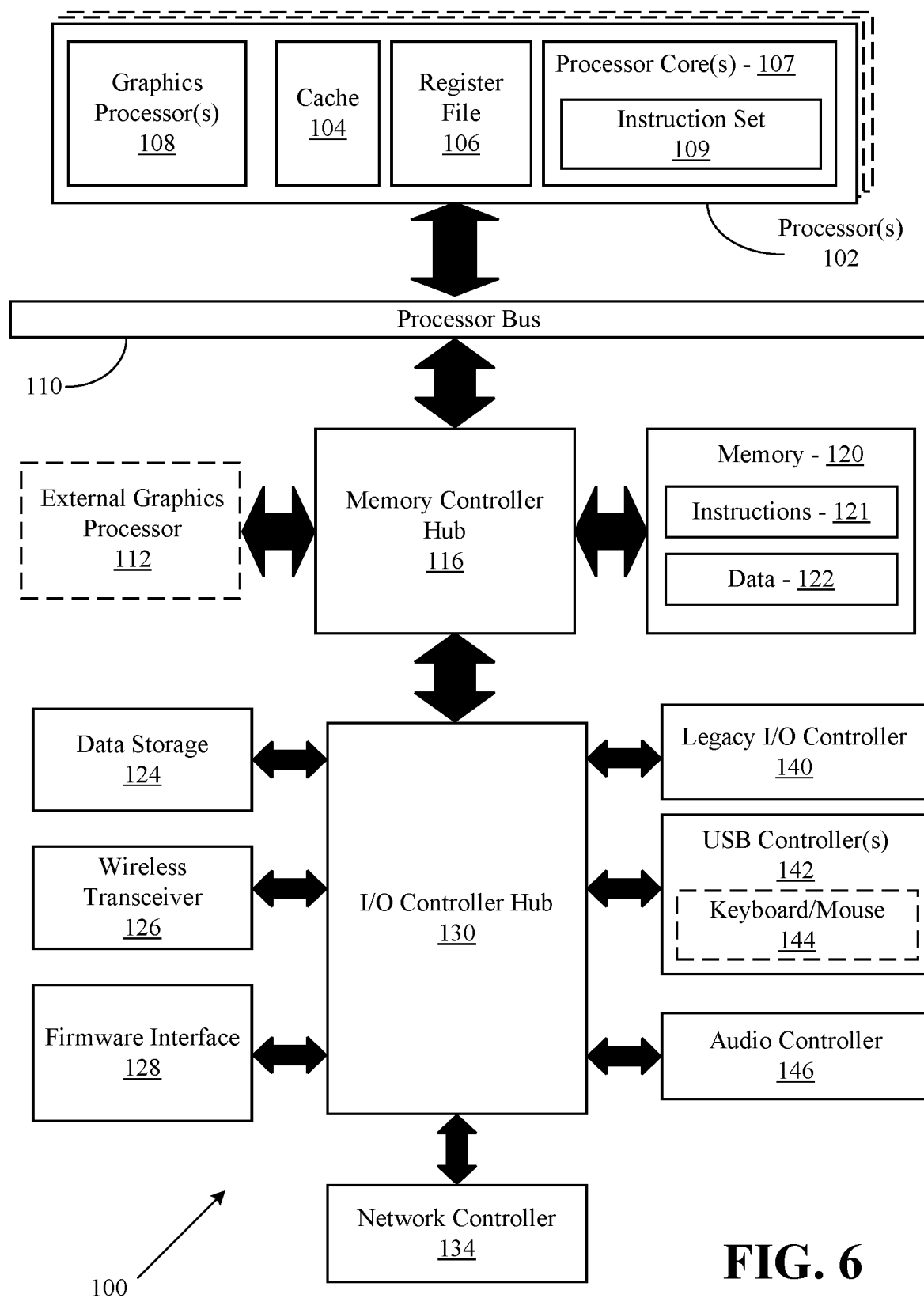
**FIG. 1****FIG. 3**

FIG. 2**FIG. 4B****FIG. 4A**

**FIG. 5**

**FIG. 6**

Processor
200

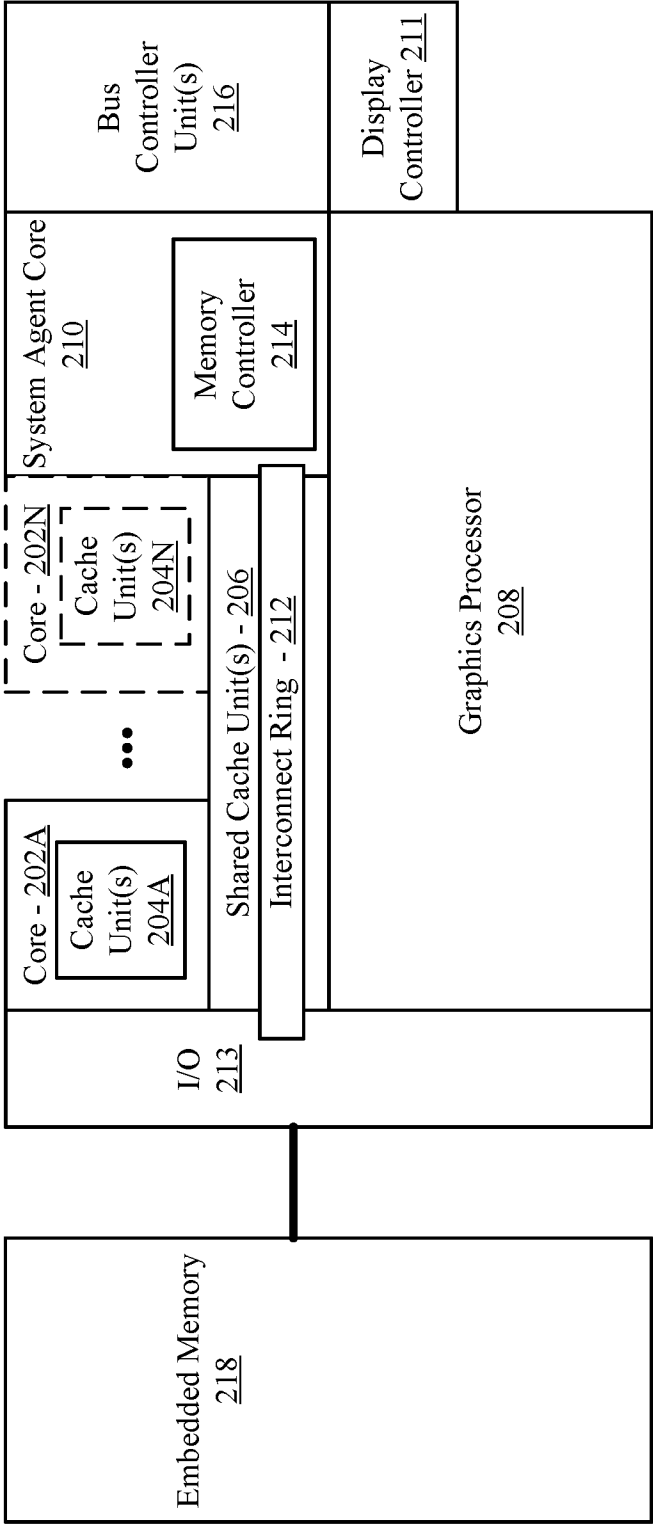


FIG. 7

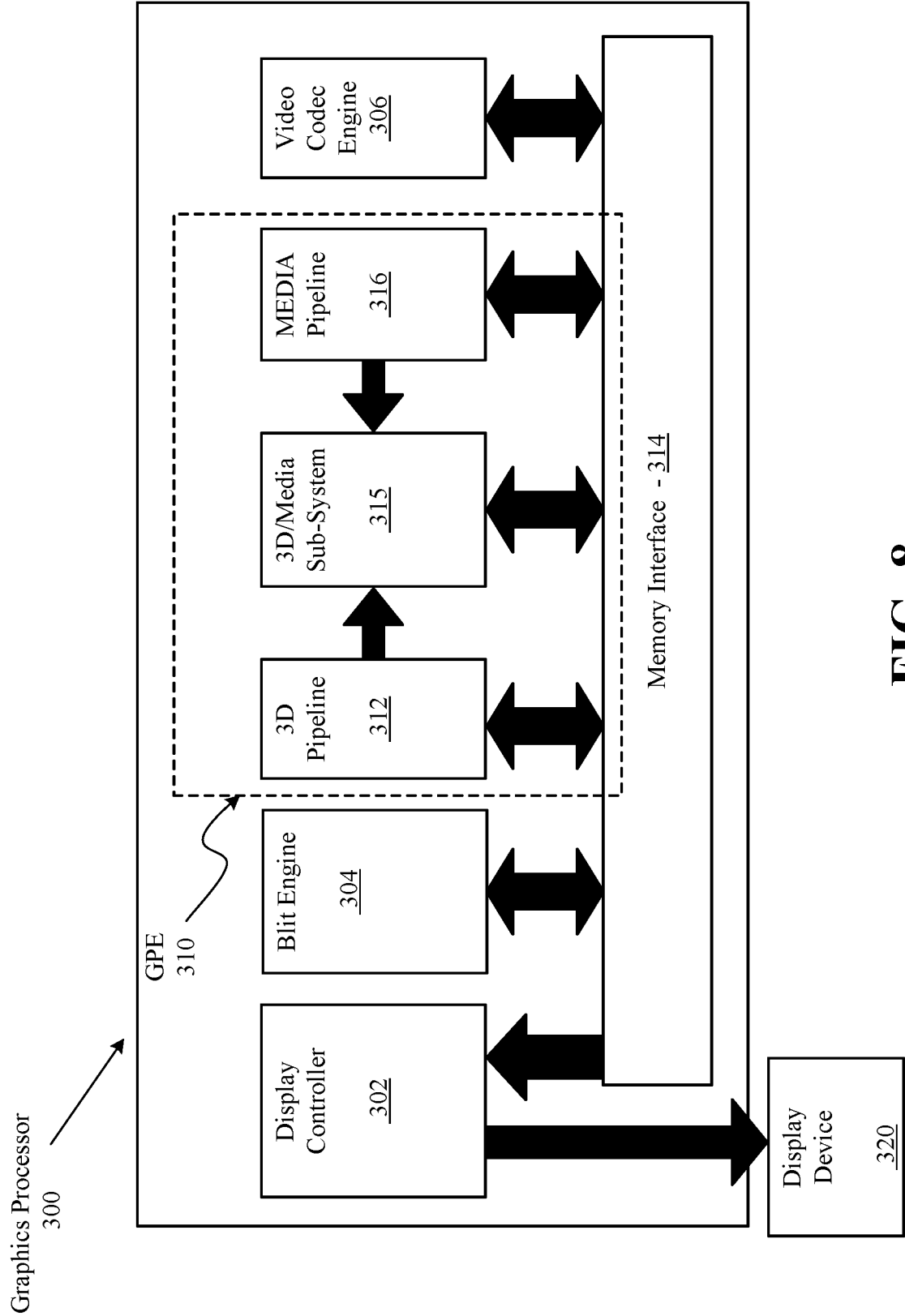


FIG. 8

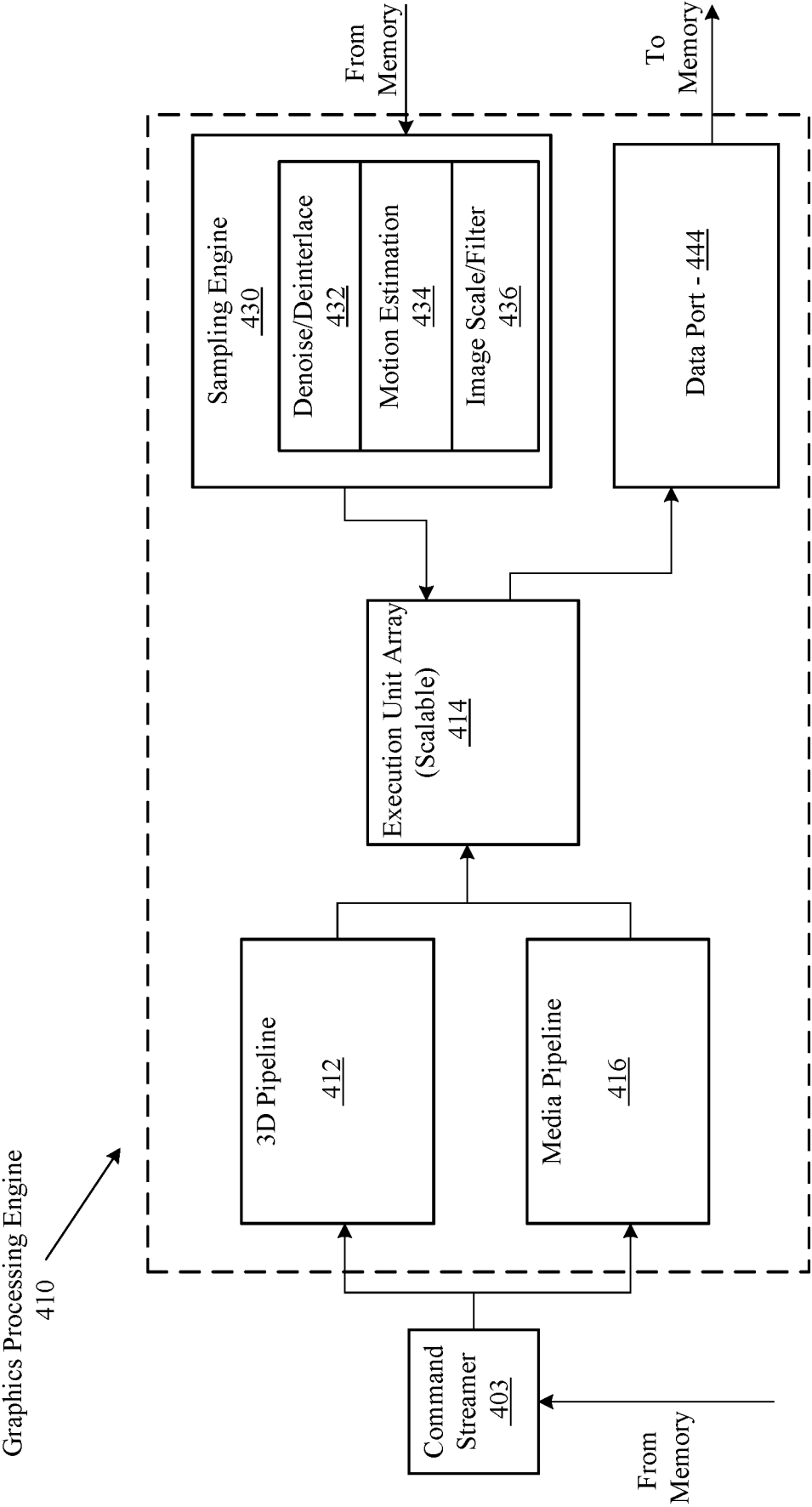


FIG. 9

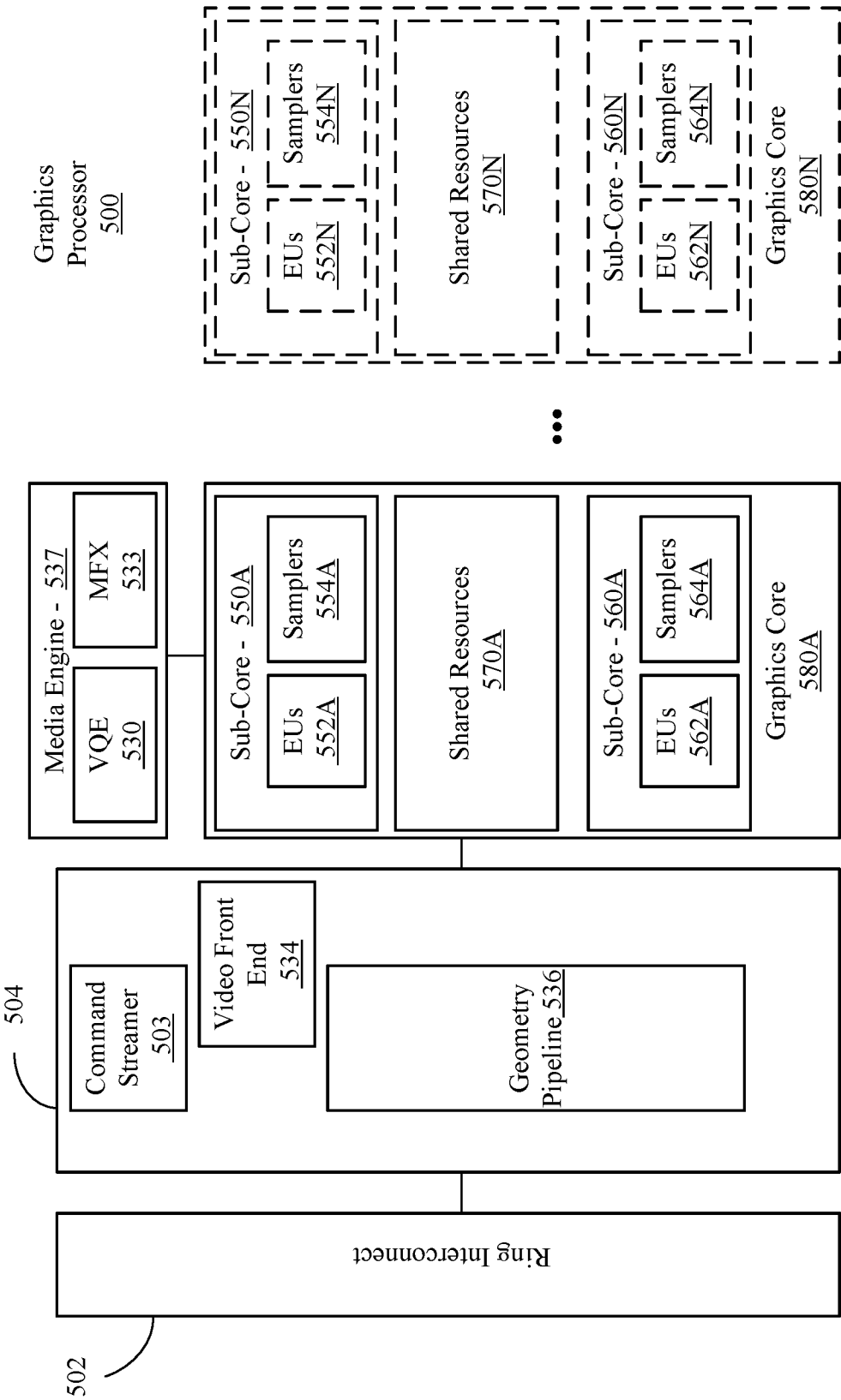


FIG. 10

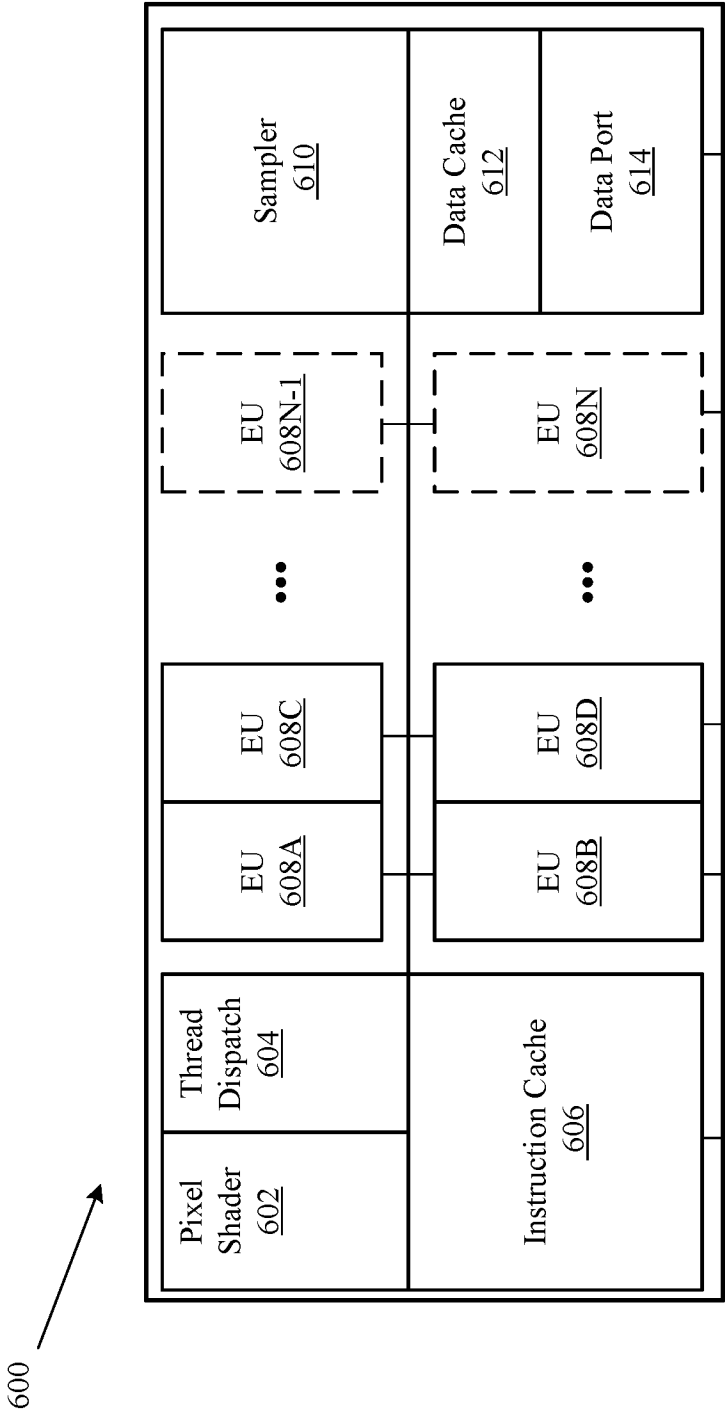


FIG. 11

Graphics Core Instruction Formats

700



Opcode 712	Control 714	Exec-Size 716	Dest 718	SRC0 720	SRC1 722	SRC2 724	Access/Address Mode 726
---------------	----------------	------------------	-------------	-------------	-------------	-------------	----------------------------



Opcode 712	Index 713	Control 714	Dest 718	SRC0 720	SRC1 722
---------------	--------------	----------------	-------------	-------------	-------------

Opcode Decode 740

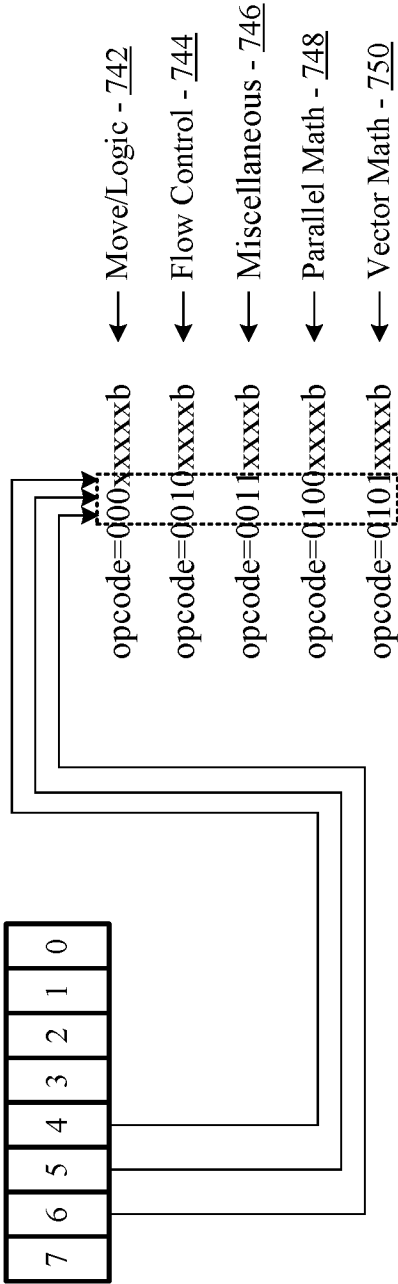


FIG. 12

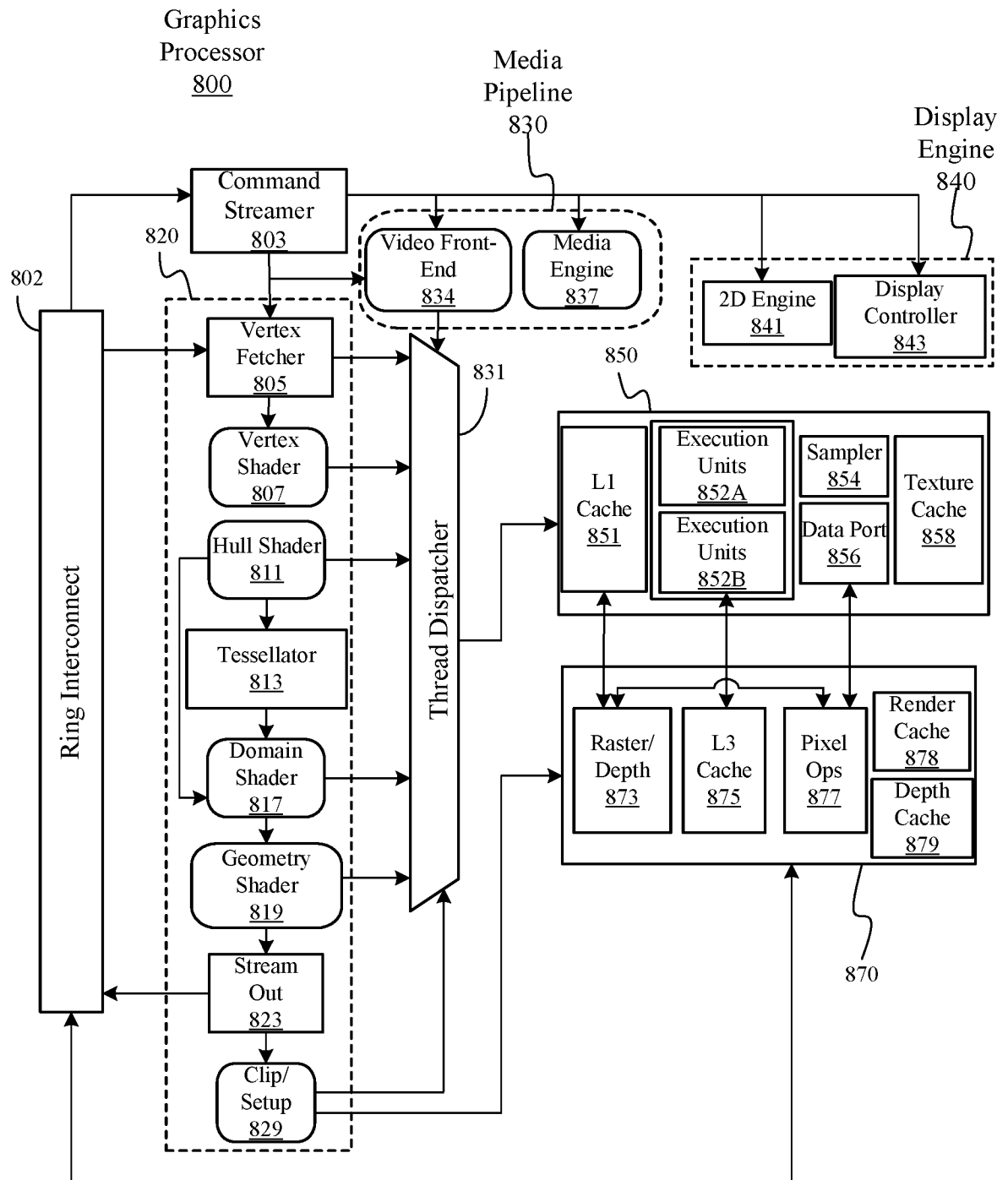
**FIG. 13**

FIG. 14A

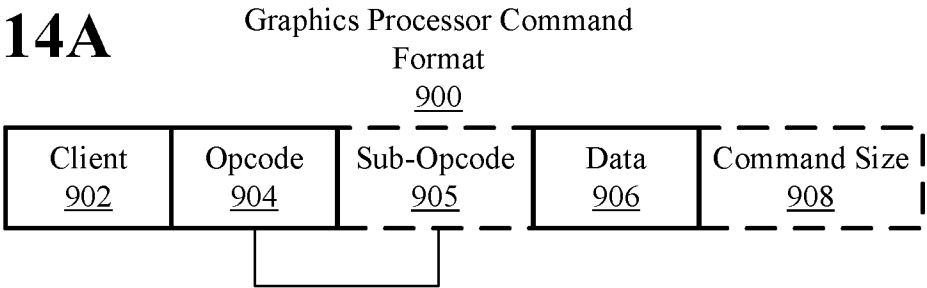
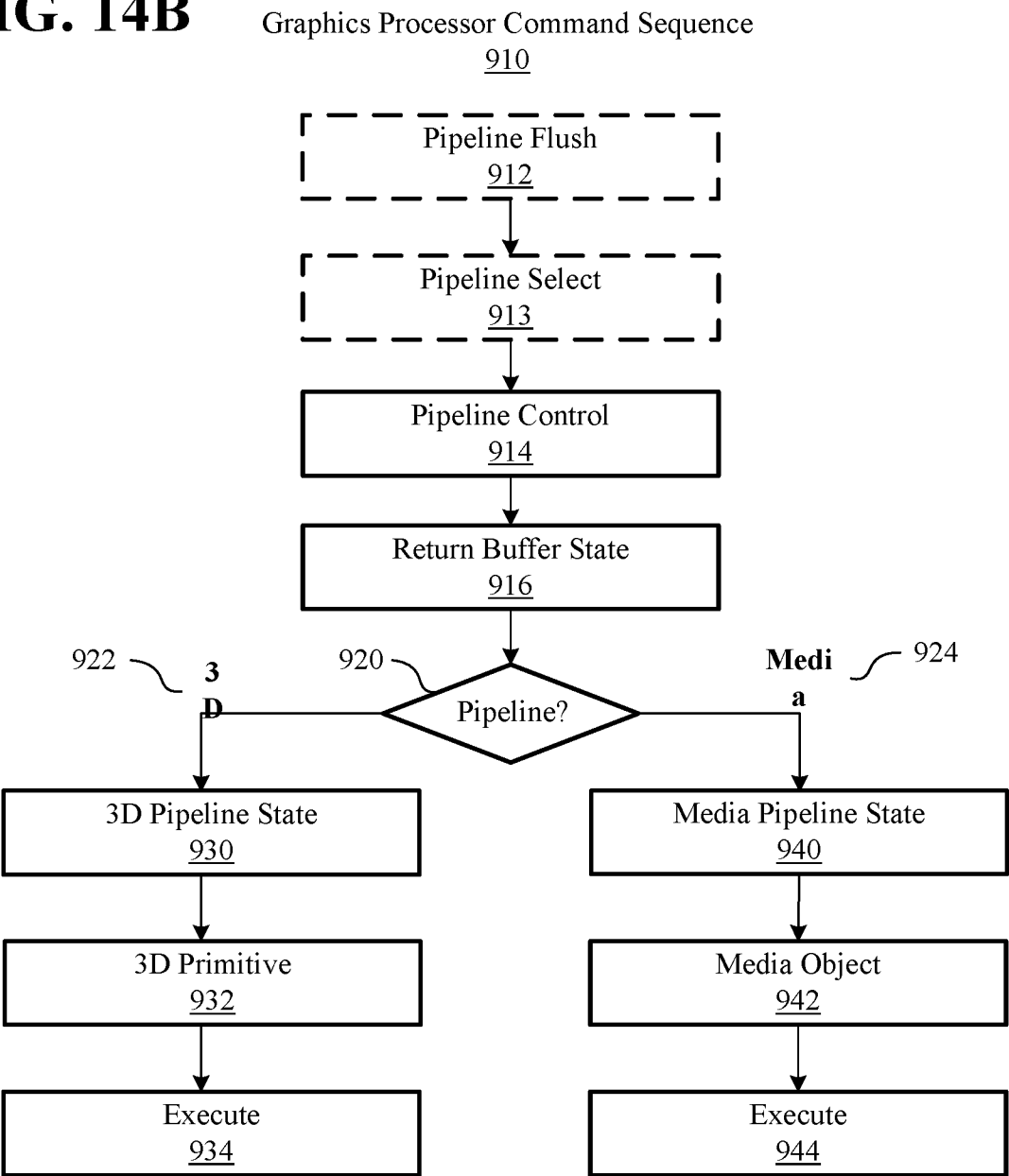
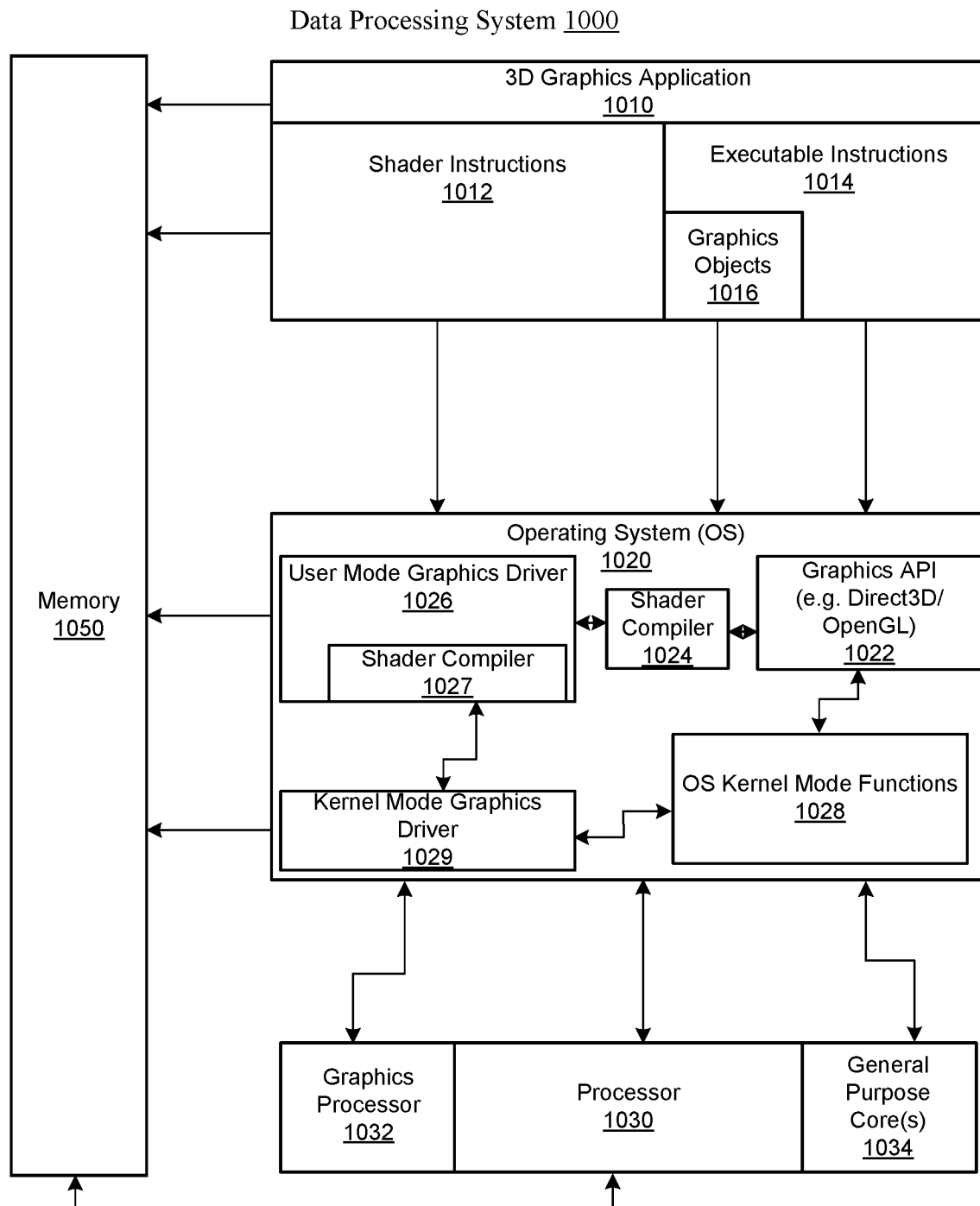


FIG. 14B



**FIG. 15**

IP CORE DEVELOPMENT - 1100

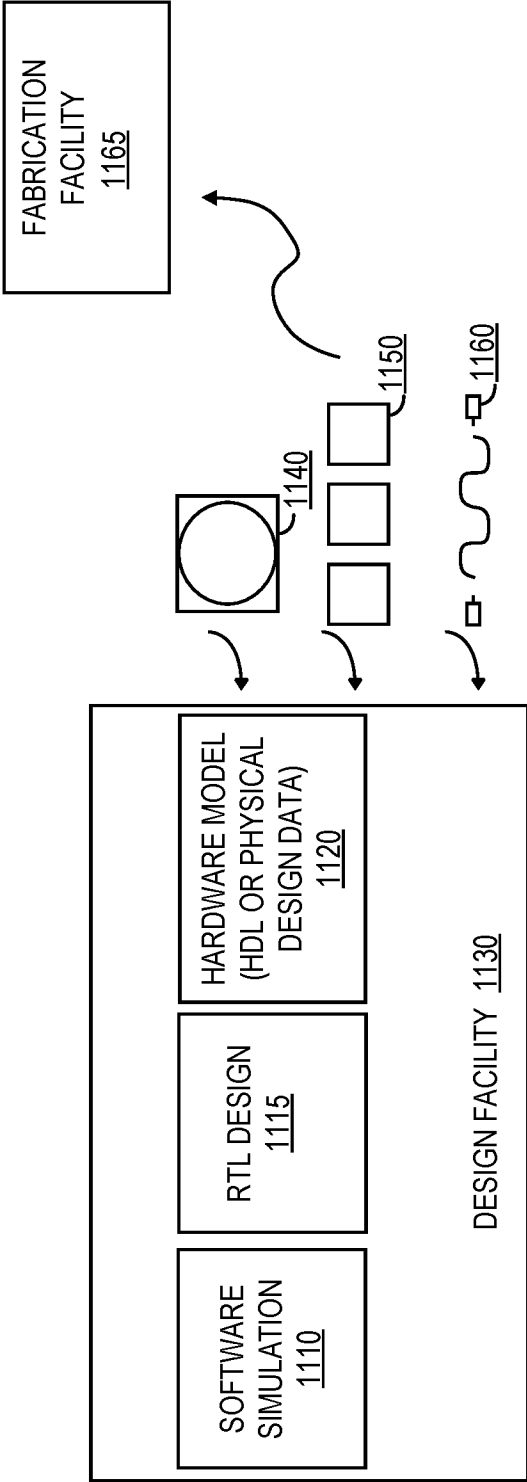
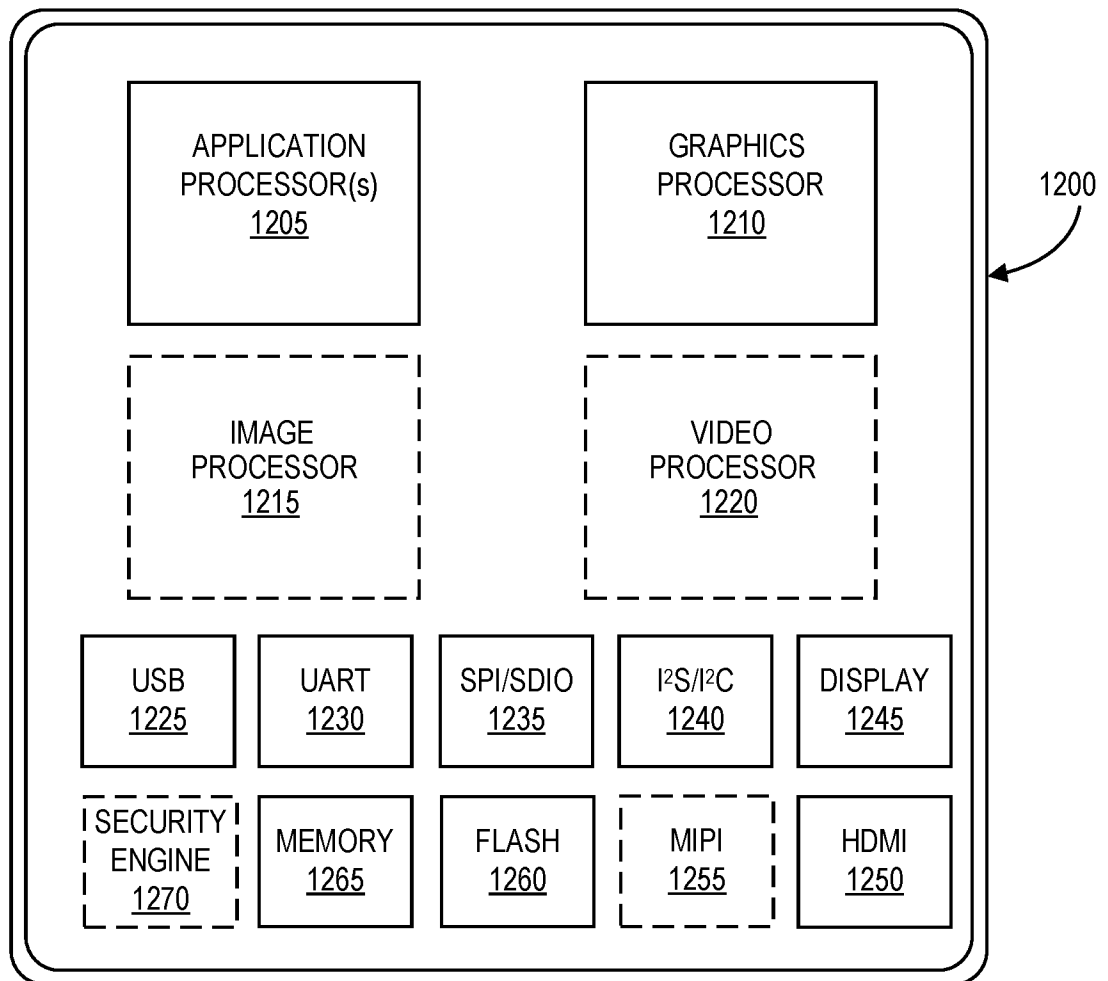


FIG. 16

**FIG. 17**

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/028617**A. CLASSIFICATION OF SUBJECT MATTER****G06T 1/20(2006.01)i, G06T 15/08(2011.01)i, G06T 5/00(2006.01)i**

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06T 1/20; H04N 13/00; G06F 13/14; G06T 15/08; G06T 5/00

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: workload converter, GPGPU workload, 3D workload, 3D pipeline

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2013-0002689 A1 (SELVAKUMAR PANNEER et al.) 03 January 2013 See paragraphs [0008]-[0012], [0014]-[0020]; claim 1; and figures 1, 2B-4.	1-25
Y	US 2013-0057647 A1 (JI-BUM MOON et al.) 07 March 2013 See paragraphs [0035]-[0043]; and figure 1.	1-25
A	US 2014-0035938 A1 (CHANGLIANG CHARLIE WANG et al.) 06 February 2014 See paragraphs [0044]-[0050]; and figures 4A-4B.	1-25
A	US 2014-0300614 A1 (INTEL CORPORATION) 09 October 2014 See paragraphs [0045]-[0046]; and figure 4.	1-25
A	US 6734862 B1 (JAMES S. CHAPPLE et al.) 11 May 2004 See column 4, line 20 - column 5, line 5; column 7, line 19 - column 8, line 16; and figures 4, 7.	1-25



Further documents are listed in the continuation of Box C.



See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search

26 July 2016 (26.07.2016)

Date of mailing of the international search report

26 July 2016 (26.07.2016)

Name and mailing address of the ISA/KR

International Application Division

Korean Intellectual Property Office

189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea

Facsimile No. +82-42-481-8578

Authorized officer

HAN, JOONG SUB

Telephone No. +82-42-481-3578



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2016/028617

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2013-0002689 A1	03/01/2013	CN 103620641 A TW 201305829 A WO 2013-003645 A2 WO 2013-003645 A3	05/03/2014 01/02/2013 03/01/2013 28/02/2013
US 2013-0057647 A1	07/03/2013	EP 2566168 A2 EP 2566168 A3 KR 10-2013-0026078 A	06/03/2013 09/10/2013 13/03/2013
US 2014-0035938 A1	06/02/2014	CN 103577269 A EP 2693761 A2 EP 2693761 A3 KR 10-1634500 B1 KR 10-2014-0018157 A KR 10-2015-0096639 A US 8928678 B2	12/02/2014 05/02/2014 16/04/2014 08/07/2016 12/02/2014 25/08/2015 06/01/2015
US 2014-0300614 A1	09/10/2014	TW 201342226 A TW I478054 B WO 2013-101560 A1	16/10/2013 21/03/2015 04/07/2013
US 6734862 B1	11/05/2004	AU 2001-65045 A1 CN 100443322 A CN 100443322 B EP 1295199 A1 JP 2004-503859 A KR 10-0501052 B1 KR 10-2003-0013441 A TW I508501 B WO 01-97006 A1	24/12/2001 17/09/2003 05/05/2010 26/03/2003 05/02/2004 18/07/2005 14/02/2003 01/11/2002 20/12/2001