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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
28 December 2017 (28.12.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/222663 A1

(51) International Patent Classification:
G06T 15/06 (2011.01) *G06T 1/20* (2006.01)
G06T 15/08 (2011.01) *G06T 1/60* (2006.01)

(21) International Application Number:
PCT/US2017/031843

(22) International Filing Date:
10 May 2017 (10.05.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
15/187,148 20 June 2016 (20.06.2016) US

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

(72) Inventors: **WALD, Ingo**; 2321 S Lakeline Dr., Salt Lake City, Utah 84109 (US). **AMSTUTZ, Jefferson D.**; 10029 Deer Chase Trail, Austin, Texas 78747 (US).

(74) Agent: **TROP, Timothy N.** et al.; Trop, Pruner & Hu, P.C., 1616 S. Voss Rd., Ste. 750, Houston, Texas 77057-2631 (US).

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

(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).

(54) Title: PROGRESSIVELY REFINED VOLUME RAY TRACING

(57) Abstract: A first set of N samples is located along a ray through a volume in connection with a first frame. The first set of N samples is stored. A second set of N additional samples is located along the ray in connection with a second frame. Then the first and second sets of samples are accumulated.

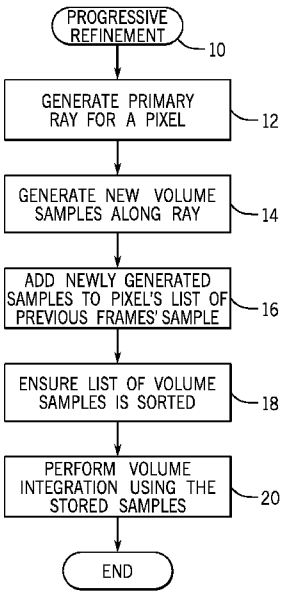


FIG. 1

WO 2017/222663 A1

[Continued on next page]

Declarations under Rule 4.17:

— *as to the identity of the inventor (Rule 4.17(i))*
— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*
— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

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

PROGRESSIVELY REFINED VOLUME RAY TRACING

Background

[0001] This relates to graphics processing and particularly to progressive refinement rendering for volume ray tracing.

[0002] Many rendering techniques require the taking of multiple "samples" for super-sampling, for Monte-Carlo integration of direct or indirect lighting effects in the context of photo-realistic rendering, or for volumetric integration of opacity and emittance along a ray in the context of volume rendering. Generally the more samples the better the accuracy of the result, but the higher the rendering time. Despite faster hardware and more efficient algorithms, it is not possible, in particular for interactive applications, to compute as many samples as desired in the time available for each frame.

[0003] The state of the art method to attack this problem is to perform "progressive refinement" rendering. Rather than computing all samples in every frame, progressive refinement takes only as many samples per frame as can be afforded to maintain interactivity. This approach trades image quality for interactivity and then "fixes" the reduced image quality by merging successive frames' partial results, typically by accumulating/averaging those frames' pixel values.

[0004] This form of progressive refinement generally works well for all sorts of Monte-Carlo integrations where individual samples are independent (and can thus be averaged in multiple batches). One important algorithm where it does not work well, however, is volume integration.

[0005] In volume rendering or volume integration, a volume is modelled as a medium that can emit, transmit and absorb light. A volume rendering integral may be solved by ray casting. Since the multiple samples along each ray are not independent in volume rendering, averaging multiple individually integrated rays' results (with few samples each) is not the same as integrating a single ray with many samples. Consequently, progressive refinement in the context of volume rendering will

generally not converge to the same image as a reference technique with a large number of samples all shot in a single frame.

Brief Description Of The Drawings

[0006] Some embodiments are described with respect to the following figures:

Figure 1 is a flow chart for one embodiment;

Figure 2 is a depiction of rays in a volume for a first (n) and second frame (n+1) according to one embodiment;

Figures 3A and 3B are a flow chart for another embodiment;

Figure 4 is a block diagram of a processing system according to one embodiment;

Figure 5 is a block diagram of a processor according to one embodiment;

Figure 6 is a block diagram of a graphics processor according to one embodiment;

Figure 7 is a block diagram of a graphics processing engine according to one embodiment;

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

Figure 9 is a depiction of thread execution logic according to one embodiment;

Figure 10 is a block diagram of a graphics processor instruction format according to some embodiments;

Figure 11 is a block diagram of another embodiment of a graphics processor;

Figure 12A is a block diagram of a graphics processor command format according to some embodiments;

Figure 12B is a block diagram illustrating a graphics processor command sequence according to some embodiments;

Figure 13 is a depiction of an exemplary graphics software architecture according to some embodiments;

Figure 14 is a block diagram illustrating an IP core development system according to some embodiments;

Figure 15 is a block diagram showing an exemplary system on chip integrated circuit according to some embodiments;

Figure 16 is a block diagram of a graphics processor in a system on a chip according to one embodiment; and

Figure 17 is a block diagram of another graphics processor according to one embodiment.

Detailed Description

[0007] In the context of volume integration it is not the integration of the multiple samples that is the performance bottleneck, but rather the taking of these samples, which typically occurs during many, and often incoherent, memory accesses. Progressive refinement is performed, not by accumulating previous frames' readily-integrated pixel values, but instead by accumulating all individual samples from previous frames. Additional samples per frame can be added progressively. When computing a given frame's pixel color, all samples, even those from previous frames, are completely reintegrated.

[0008] This technique accumulates samples, and eventually produces the same output as the non-progressive technique with all samples taken in a single frame. This technique is progressive in the sense that it takes only a few new samples every frame. Thus, it achieves roughly the same performance as traditional progressive refinement based on image accumulation but still converges to the same result as a non-progressive technique in some embodiments.

[0009] In volume rendering, "accumulation" is used in the sense that every sample along the ray has both a color and a transparency/opacity value (both coming from the transfer function). Thus, the opacity of the very "first" (as in, "closest" or "nearest to the camera") sample will determine how much further-away samples can contribute. If they are completely opaque the further-behind samples do not matter at all. If they are 50% opaque all further samples will only contribute to 50%, etc. Similarly, the second sample, itself weighted by the transparency of the first sample, influences how much the third, fourth, etc. samples are weighted, etc. In general, the weight of the Nth sample depends on the "accumulated" transparencies of all samples before it. When implementing a volume renderer all samples front to back are collected, and the "accumulated opacity" that the respective next sample is

weighted with is tracked. Usually the samples are processed front to back. If the samples are taken front to back, the color and weight are accumulated in the right order. Here, samples come from different frames, and though both of those two frames' sample sets are each sorted, there is a read to "merge" them into a single sorted set before accumulating opacity etc. Also it does not matter if those samples come from different nodes or different frames. The samples are processed front-to-back in one embodiment.

[0010] Rather than using an accumulation buffer in which only final color values are accumulated, a list of all volumes that have previously been taken for that pixel's ray(s) is stored. Resetting the accumulation buffer means clearing the list of samples. The sequence 10, shown in **Figure 1**, proceeds as follows:

1) For each pixel,
   a. generate a primary ray for this pixel as indicated in block 12;
   b. generate N new volume samples along this ray (using different random numbers to generate a sample starting position offset from the ray/volume entry point used in previous frames) as indicated in block 14, where N is a subset of the desired number of samples per ray;
   c. add the newly N generated volume samples to pixel's list of previous frames' samples as indicated in block 16;
   d. ensure list of volume samples is sorted as indicated in block 18; and
   e. perform volume integration (including volumetric shading, if required) using the stored samples as indicated in block 20.

[0011] Figure 2 shows two successive frames, a frame n and a frame n+1. A ray shown in connection with frame n enters a volume as indicated at one point and exits at a second point to the right of the first point as shown in **Figure 2**. Then for the first frame samples are taken starting at the volume/ray entry point at regular intervals. The number of samples N taken for each successive frame may be variable. In this case N is 6 for illustration purposes only.

[0012] In some embodiments, more and more samples are taken progressively from frame to successive frame. However all of the samples taken in the prior frames are stored and accumulated.

[0013] Then for the frame n+1, the samples indicated by open circles in Figure 2 are added to the samples taken in frame n. The samples taken in frame n+1 do not start at the volume ray entry point but rather are offset by an offset as indicated in Figure 2. In some embodiments this offset is determined randomly. Note that the samples taken for frame n are preserved in connection with frame n+1 but the samples taken in frame n+1 are now augmented by the samples taken in frame n.

[0014] There are many ways to schedule work of rendering an image. In some cases a single node or work station is used. In other cases, multiple nodes may be used, for example in connection with a distributed computer or supercomputer. Where multiple nodes are used, the rendering may be done by distributing the work across a plurality of nodes. In some cases, an image may be subdivided so that each node only has to render a piece of the image and then the pieces rendered by different nodes are composited together. Thus a subset of the volume is handled by each of a plurality of nodes in a cluster of nodes and then the results from the various nodes are combined.

[0015] In parallel rendering, multiple nodes are used to do the rendering. It is possible to put a copy of the entire data set on every node and then let each node do a part of the work.

[0016] In data parallel rendering, the data is spread over more than a single node. Both the data and the work may be distributed in parts across multiple nodes. Then each node only does a part of the work and only has a part of the data. In data parallel rendering, every node renders a portion of image for its part of the data. Then the pieces are composited to handle the entire image.

[0017] Since each node sends its results to another node for accumulation, the only thing that changes is that samples may come from multiple nodes. Then it is necessary to determine which sample portions map to data stored on a particular node. This is so that the node has the right data for the samples it must handle.

[0018] Thus referring again to Figure 2, the volume (only partially shown in Figure 2) may be a box or other shape and a ray is traced through that volume. There are a number of samples N (e.g. six as illustrated) along each ray in the volume. Suppose one node has the left half of the volume as indicated by the samples zero and the other node has the right half of the volume as indicated by samples one. So some samples are on a node that has the left half of the volume and other samples along the same ray are on a different node. Every node must determine what samples fall in the data that that node owns and then that node generates those samples that the node owns the data for and sends them on for final compositing.

[0019] A sequence 22 shown in **Figures 3A** and **3B** may be used for data parallel rendering, as follows:

1) Statically create an assignment of pixels to nodes, such that every pixel has a fixed and well-defined node owner, as indicated in block 24.

2) On each node:

a. for each pixel (even those that are not owned on this node)

i. generate a primary ray for this pixel, as indicated in block 26;

ii. generate N new volume sample positions along this ray (using different random numbers than in previous frames) as indicated in block 28;

iii. determine which sample positions map to data stored on this node as indicated in diamond 30. Discard those that do not map as indicated in block 32. Evaluate those that do map as indicated in block 34;

iv. check whether this is the last sample (diamond 35) and return to diamond 30 if not the last sample;

v. if that was the last sample, send newly generated to sample positions whichever node “owns” the current pixel (if node itself is owner, send to “self”) as indicated in block 36;

b. for each pixel owned by this node i (block 38), for each node n:

1. receive new samples from node n as indicated in block 40;

2. add newly received volume samples to pixel's list of previous frames' samples as indicated in block 42;

3. ensure list of volume samples is sorted as indicated in block 44;

4. perform volume integration as indicated in block 46; and

5. if this is last pixel (diamond 18) end, otherwise iterate back to block 40.

[0020] The first frame's samples may be sorted by default (because they are sorted from front to back), and sorting is only required whenever new frames' samples get added to exiting frame's samples (because both old and new samples are sorted front-to-back, but to have a single front-to-back sorted list those two sorted lists have to be merged). In data parallel rendering even the first frames' sample lists have to be put into sorted order because a given pixel's samples come from different nodes. Sorting is done after the first frame because a starting point for samples along a ray is shifted or offset for each successive frame. The reason sorting is important in data parallel rendering is that, because of the offset in the place where samples begin to be taken between successive frames as shown in Figure 2, the samples will not be in the same order from front to back in successive frames. This is because the samples added in frame n+1 are out of order relative to the samples taken in frame n because they are in different places in the memory with respect to the front to back order. Generally the extent of the offset may be randomized so it is unknown from frame to frame and different between successive frames.

[0021] The other problem is the sorting of the samples to the right nodes. In other words, you must map the data to the corresponding node that actually holds the data for the samples being processed. Each node only has a portion of the data set in data parallel rendering and therefore each node can only operate on those samples for which it has the data. Each node determines which samples fall in the data set

that the node owns and then generates those samples for the data the node owns. The results from each node are then accumulated for all the nodes.

[0022] The techniques shown in Figures 1 and 3 may be used in different embodiments. In some cases, the technique shown in Figure 1 is tried and if the data set is too large then the technique shown in Figure 3 may be used. In other cases, only the technique shown in Figure 3 may be used. In some embodiments only the technique shown in Figure 1 may be used.

[0023] The techniques described herein decouple the generation of samples from the final storage and accumulation of the samples. Thus, a variety of modes are made possible because of these techniques.

[0024] These methods not only work in a data-parallel context, but actually work better the more nodes get added in some embodiments. More nodes can mean less pixels owned per node, which means lower (per-node) storage for pixel samples, lower (per-node) bandwidth to receive new samples, and/or lower integration cost.

[0025] There are multiple possible variants. The “classification” of the samples (i.e., using a transfer function to translate a raw volume sample to an RGBA color and opacity required for shading) can be done either as part of 1b (i.e., by storing a list of readily classified RGBA samples), or as part of 1e (i.e., by storing raw volume samples and performing classification on-the-fly during integration). The latter needs less memory for storing samples, and allows for changing the transfer function during accumulation while the former is faster.

[0026] In a data-parallel context, each node may pre-classify and pre-blend all samples on its node, requiring only a single classified value (i.e. RGBA color) to be sent to the master node for final blending. Each node continues to aggregate samples over the course of multiple frames, but blending all of the samples along the ray, where all nodes send all samples to the master to be blended in front-to-back order, results in the same image as the master blending each nodes pre-blended colors front-to-back order. This may result in less communication overhead without affecting the final result in some embodiments.

[0027] Rather than each node sending all its new samples to whatever node owns that pixel (which requires sending lots of samples over the network) this node could also accumulate its samples locally, then do its part of the integration/accumulation node, and only send the final accumulated weight and color for all its local samples (so the node that owns the pixel would only accumulate the partially integrated results, not re-shade all the samples).

[0028] This technique may be completely agnostic to which method is used to generate the samples. In particular, it is most powerful for volume data that is expensive to sample (such as, for example, unstructured or Adaptive Mesh Refinement (AMR) data, radial basis function (RBF) based scalar fields, etc).

[0029] The final sample-integration step is a relatively simple, straightforward computation that, with the proper data layout, lends itself to a highly vectorized implementation in some embodiments.

[0030] The method is fully consistent with volumetric lighting as long as each sample also stores the volume gradient (possibly in compressed form).

[0031] Instead of storing all samples, at the end of each frame, a data reduction or resampling step may be performed to determine which samples are important, and which ones could be discarded.

[0032] There is a point that some sample sets can be culled. For example, if a sample is always completely transparent it need not be stored at all. If its weight, after some sample N, drops to too low a value, no samples beyond that need be stored. If the changing step size is adjusted properly, N successive “very similar” samples may be replaced by a single sample that gets properly weighted.

[0033] There is a point that “sorting” can be done faster if the two sample sets being merged (e.g., the so far generated ones and the new frame’s ones) are each already sorted. Then a full sort is unnecessary. Instead, two sorted sets may be merged, which is much faster in some embodiments.

[0034] Having access to previous samples suggests using these samples to guide where the current frame’s samples should be placed.

[0035] While the method is fully compatible with "pre-integration," the same benefits may apply as without pre-integration.

[0036] The sequences shown in Figures 1 and 3 may be implemented in hardware, software and/or firmware. In software and firmware embodiments the sequences may be implemented using computer implemented instructions stored in one or more non-transitory computer readable media such as magnetic, optical or semiconductor storage. The storage may be part of or coupled to a graphics processor, for example.

[0037] **Figure 4** is a block diagram of a processing system 100, according to an embodiment. In various embodiments the system 100 includes one or more processors 102 and one or more graphics processors 108, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 102 or processor cores 107. In one embodiment, the system 100 is a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices.

[0038] An embodiment of system 100 can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In some embodiments system 100 is a mobile phone, smart phone, tablet computing device or mobile Internet device. Data processing system 100 can also include, couple with, or be integrated within a wearable device, such as a smart watch wearable device, smart eyewear device, augmented reality device, or virtual reality device. In some embodiments, data processing system 100 is a television or set top box device having one or more processors 102 and a graphical interface generated by one or more graphics processors 108.

[0039] In some embodiments, the one or more processors 102 each include one or more processor cores 107 to process instructions which, when executed, perform operations for system and user software. In some embodiments, each of the one or more processor cores 107 is configured to process a specific instruction set 109. In some embodiments, instruction set 109 may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a

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

[0040] In some embodiments, the processor 102 includes cache memory 104. Depending on the architecture, the processor 102 can have a single internal cache or multiple levels of internal cache. In some embodiments, the cache memory is shared among various components of the processor 102. In some embodiments, the processor 102 also uses an external cache (e.g., a Level-3 (L3) cache or Last Level Cache (LLC)) (not shown), which may be shared among processor cores 107 using known cache coherency techniques. A register file 106 is additionally included in processor 102 which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor 102.

[0041] In some embodiments, processor 102 is coupled with a processor bus 110 to transmit communication signals such as address, data, or control signals between processor 102 and other components in system 100. In one embodiment the system 100 uses an exemplary 'hub' system architecture, including a memory controller hub 116 and an Input Output (I/O) controller hub 130. A memory controller hub 116 facilitates communication between a memory device and other components of system 100, while an I/O Controller Hub (ICH) 130 provides connections to I/O devices via a local I/O bus. In one embodiment, the logic of the memory controller hub 116 is integrated within the processor.

[0042] Memory device 120 can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. In one embodiment the memory device 120 can operate as system memory for the system 100, to store data 122 and instructions 121 for use when the one or more processors 102 executes an application or process. Memory controller hub 116 also couples with an optional

external graphics processor 112, which may communicate with the one or more graphics processors 108 in processors 102 to perform graphics and media operations.

[0043] In some embodiments, ICH 130 enables peripherals to connect to memory device 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller 146, a firmware interface 128, a wireless transceiver 126 (e.g., Wi-Fi, Bluetooth), a data storage device 124 (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller 140 for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 142 connect input devices, such as keyboard and mouse 144 combinations. A network controller 134 may also couple with ICH 130. In some embodiments, a high-performance network controller (not shown) couples with processor bus 110. It will be appreciated that the system 100 shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, the I/O controller hub 130 may be integrated within the one or more processor 102, or the memory controller hub 116 and I/O controller hub 130 may be integrated into a discreet external graphics processor, such as the external graphics processor 112.

[0044] **Figure 5** is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-202N, an integrated memory controller 214, and an integrated graphics processor 208. Those elements of Figure 5 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. 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.

[0045] 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.

[0046] In some embodiments, processor 200 may also include a set of one or more bus controller units 216 and a system agent core 210. The one or more bus controller units 216 manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). System agent core 210 provides management functionality for the various processor components. In some embodiments, system agent core 210 includes one or more integrated memory controllers 214 to manage access to various external memory devices (not shown).

[0047] In some embodiments, one or more of the processor cores 202A-202N include support for simultaneous multi-threading. In such embodiment, the system agent core 210 includes components for coordinating and operating cores 202A-202N during multi-threaded processing. System agent core 210 may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of processor cores 202A-202N and graphics processor 208.

[0048] In some embodiments, processor 200 additionally includes graphics processor 208 to execute graphics processing operations. In some embodiments, the graphics processor 208 couples with the set of shared cache units 206, and the system agent core 210, including the one or more integrated memory controllers 214. In some embodiments, a display controller 211 is coupled with the graphics processor 208 to drive graphics processor output to one or more coupled displays. In some embodiments, display controller 211 may be a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor 208 or system agent core 210.

[0049] In some embodiments, a ring based interconnect unit 212 is used to couple the internal components of the processor 200. However, an alternative interconnect unit may be used, such as a point-to-point interconnect, a switched interconnect, or

other techniques, including techniques well known in the art. In some embodiments, graphics processor 208 couples with the ring interconnect 212 via an I/O link 213.

[0050] The exemplary I/O link 213 represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module 218, such as an eDRAM module. In some embodiments, each of the processor cores 202A-202N and graphics processor 208 use embedded memory modules 218 as a shared Last Level Cache.

[0051] In some embodiments, processor cores 202A-202N are homogenous cores executing the same instruction set architecture. In another embodiment, processor cores 202A-202N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores 202A-202N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set. In one embodiment processor cores 202A-202N are heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. Additionally, processor 200 can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.

[0052] **Figure 6** 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.

[0053] 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.

[0054] In some embodiments, graphics processor 300 includes a block image transfer (BLIT) engine 304 to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations are performed using one or more components of graphics processing engine (GPE) 310. In some embodiments, GPE 310 is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

[0055] 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.

[0056] 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.

[0057] 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.

[0058] **Figure 7** is a block diagram of a graphics processing engine 410 of a graphics processor in accordance with some embodiments. In one embodiment, the graphics processing engine (GPE) 410 is a version of the GPE 310 shown in Figure 6. Elements of Figure 8 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. For example, the 3D pipeline 312 and media pipeline 316 of Figure 6 are illustrated. The media pipeline 316 is optional in some embodiments of the GPE 410 and may not be explicitly included within the GPE 410. For example and in at least one embodiment, a separate media and/or image processor is coupled to the GPE 410.

[0059] In some embodiments, GPE 410 couples with or includes a command streamer 403, which provides a command stream to the 3D pipeline 312 and/or media pipelines 316. In some embodiments, command streamer 403 is coupled with memory, which can be system memory, or one or more of internal cache memory and shared cache memory. In some embodiments, command streamer 403 receives commands from the memory and sends the commands to 3D pipeline 312 and/or media pipeline 316. The commands are directives fetched from a ring buffer, which stores commands for the 3D pipeline 312 and media pipeline 316. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The commands for the 3D pipeline 312 can also include references to data stored in memory, such as but not limited to vertex and geometry data for the 3D pipeline 312 and/or image data and memory objects for the

media pipeline 316. The 3D pipeline 312 and media pipeline 316 process the commands and data by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to a graphics core array 414.

[0060] In various embodiments the 3D pipeline 312 can execute one or more shader programs, such as vertex shaders, geometry shaders, pixel shaders, fragment shaders, compute shaders, or other shader programs, by processing the instructions and dispatching execution threads to the graphics core array 414. The graphics core array 414 provides a unified block of execution resources. Multi-purpose execution logic (e.g., execution units) within the graphic core array 414 includes support for various 3D API shader languages and can execute multiple simultaneous execution threads associated with multiple shaders.

[0061] In some embodiments the graphics core array 414 also includes execution logic to perform media functions, such as video and/or image processing. In one embodiment, the execution units additionally include general-purpose logic that is programmable to perform parallel general purpose computational operations, in addition to graphics processing operations. The general purpose logic can perform processing operations in parallel or in conjunction with general purpose logic within the processor core(s) 107 of Figure 5 or core 202A-202N as in Figure 5.

[0062] Output data generated by threads executing on the graphics core array 414 can output data to memory in a unified return buffer (URB) 418. The URB 418 can store data for multiple threads. In some embodiments the URB 418 may be used to send data between different threads executing on the graphics core array 414. In some embodiments the URB 418 may additionally be used for synchronization between threads on the graphics core array and fixed function logic within the shared function logic 420.

[0063] In some embodiments, graphics core array 414 is scalable, such that the array includes a variable number of graphics cores, each having a variable number of execution units based on the target power and performance level of GPE 410. In one embodiment the execution resources are dynamically scalable, such that execution resources may be enabled or disabled as needed.

[0064] The graphics core array 414 couples with shared function logic 420 that includes multiple resources that are shared between the graphics cores in the graphics core array. The shared functions within the shared function logic 420 are hardware logic units that provide specialized supplemental functionality to the graphics core array 414. In various embodiments, shared function logic 420 includes but is not limited to sampler 421, math 422, and inter-thread communication (ITC) 423 logic. Additionally, some embodiments implement one or more cache(s) 425 within the shared function logic 420. A shared function is implemented where the demand for a given specialized function is insufficient for inclusion within the graphics core array 414. Instead a single instantiation of that specialized function is implemented as a stand-alone entity in the shared function logic 420 and shared among the execution resources within the graphics core array 414. The precise set of functions that are shared between the graphics core array 414 and included within the graphics core array 414 varies between embodiments.

[0065] **Figure 8** is a block diagram of another embodiment of a graphics processor 500. Elements of Figure 8 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0066] 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.

[0067] 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.

[0068] In some embodiments, graphics processor 500 includes scalable thread execution resources featuring modular cores 580A-580N (sometimes referred to as core slices), each having multiple sub-cores 550A-550N, 560A-560N (sometimes referred to as core sub-slices). In some embodiments, graphics processor 500 can have any number of graphics cores 580A through 580N. In some embodiments, graphics processor 500 includes a graphics core 580A having at least a first sub-core 550A and a second sub-core 560A. In other embodiments, the graphics processor is a low power processor with a single sub-core (e.g., 550A). In some embodiments, graphics processor 500 includes multiple graphics cores 580A-580N, each including a set of first sub-cores 550A-550N and a set of second sub-cores 560A-560N. Each sub-core in the set of first sub-cores 550A-550N includes at least a first set of execution units 552A-552N and media/texture samplers 554A-554N. Each sub-core in the set of second sub-cores 560A-560N includes at least a second set of execution units 562A-562N and samplers 564A-564N. In some embodiments, each sub-core 550A-550N, 560A-560N shares a set of shared resources 570A-570N. In some embodiments, the shared resources include shared cache memory and pixel operation logic. Other shared resources may also be included in the various embodiments of the graphics processor.

[0069] **Figure 9** illustrates thread execution logic 600 including an array of processing elements employed in some embodiments of a GPE. Elements of Figure 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.

[0070] In some embodiments, thread execution logic 600 includes a shader processor 602, a thread dispatcher 604, instruction cache 606, a scalable execution unit array including a plurality of execution units 608A-608N, a sampler 610, a data cache 612, and a data port 614. In one embodiment the scalable execution unit array can dynamically scale by enabling or disabling one or more execution units (e.g., any of execution unit 608A, 608B, 608C, 608D, through 608N-1 and 608N) based on the computational requirements of a workload. In one embodiment the included components are interconnected via an interconnect fabric that links to each of the components. In some embodiments, thread execution logic 600 includes one or more connections to memory, such as system memory or cache memory, through one or more of instruction cache 606, data port 614, sampler 610, and execution units 608A-608N. In some embodiments, each execution unit (e.g. 608A) is a stand-alone programmable general purpose computational unit that is capable of executing multiple simultaneous hardware threads while processing multiple data elements in parallel for each thread. In various embodiments, the array of execution units 608A-608N is scalable to include any number individual execution units.

[0071] In some embodiments, the execution units 608A-608N are primarily used to execute shader programs. A shader processor 602 can process the various shader programs and dispatch execution threads associated with the shader programs via a thread dispatcher 604. In one embodiment the thread dispatcher includes logic to arbitrate thread initiation requests from the graphics and media pipelines and instantiate the requested threads on one or more execution unit in the execution units 608A-608N. For example, the geometry pipeline (e.g., 536 of Figure 9) can dispatch vertex, tessellation, or geometry shaders to the thread execution logic 600 (Figure 10) for processing. In some embodiments, thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

[0072] In some embodiments, the execution units 608A-608N support an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose

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

[0073] Each execution unit in execution units 608A-608N operates on arrays of data elements. The number of data elements is the "execution size," or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical Arithmetic Logic Units (ALUs) or Floating Point Units (FPUs) for a particular graphics processor. In some embodiments, execution units 608A-608N support integer and floating-point data types.

[0074] The execution unit instruction set includes SIMD instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (Quad-Word (QW) size data elements), eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two

separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

[0075] One or more internal instruction caches (e.g., 606) are included in the thread execution logic 600 to cache thread instructions for the execution units. In some embodiments, one or more data caches (e.g., 612) are included to cache thread data during thread execution. In some embodiments, a sampler 610 is included to provide texture sampling for 3D operations and media sampling for media operations. In some embodiments, sampler 610 includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

[0076] During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 600 via thread spawning and dispatch logic. Once a group of geometric objects has been processed and rasterized into pixel data, pixel processor logic (e.g., pixel shader logic, fragment shader logic, etc.) within the shader processor 602 is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In some embodiments, a pixel shader or fragment shader calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. In some embodiments, pixel processor logic within the shader processor 602 then executes an application programming interface (API)-supplied pixel or fragment shader program. To execute the shader program, the shader processor 602 dispatches threads to an execution unit (e.g., 608A) via thread dispatcher 604. In some embodiments, pixel shader 602 uses texture sampling logic in the sampler 610 to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

[0077] In some embodiments, the data port 614 provides a memory access mechanism for the thread execution logic 600 output processed data to memory for processing on a graphics processor output pipeline. In some embodiments, the data

port 614 includes or couples to one or more cache memories (e.g., data cache 612) to cache data for memory access via the data port.

[0078] **Figure 10** is a block diagram illustrating a graphics processor instruction formats 700 according to some embodiments. In one or more embodiment, the graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. In some embodiments, instruction format 700 described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

[0079] In some embodiments, the graphics processor execution units natively support instructions in a 128-bit instruction format 710. A 64-bit compacted instruction format 730 is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit instruction format 710 provides access to all instruction options, while some options and operations are restricted in the 64-bit format 730. The native instructions available in the 64-bit format 730 vary by embodiment. In some embodiments, the instruction is compacted in part using a set of index values in an index field 713. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit instruction format 710.

[0080] For each format, instruction opcode 712 defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. In some embodiments, instruction control field 714 enables control over certain execution options, such as channels selection (e.g., predication) and data

channel order (e.g., swizzle). For instructions in the 128-bit instruction format 710 an exec-size field 716 limits the number of data channels that will be executed in parallel. In some embodiments, exec-size field 716 is not available for use in the 64-bit compact instruction format 730.

[0081] Some execution unit instructions have up to three operands including two source operands, src0 720, src1 722, and one destination 718. In some embodiments, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 724), where the instruction opcode 712 determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

[0082] In some embodiments, the 128-bit instruction format 710 includes an access/address mode field 726 specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction.

[0083] In some embodiments, the 128-bit instruction format 710 includes an access/address mode field 726, which specifies an address mode and/or an access mode for the instruction. In one embodiment the access mode is used to define a data access alignment for the instruction. Some embodiments support access modes including a 16-byte aligned access mode and a 1-byte aligned access mode, where the byte alignment of the access mode determines the access alignment of the instruction operands. For example, when in a first mode, the instruction may use byte-aligned addressing for source and destination operands and when in a second mode, the instruction may use 16-byte-aligned addressing for all source and destination operands.

[0084] In one embodiment, the address mode portion of the access/address mode field 726 determines whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed

based on an address register value and an address immediate field in the instruction.

[0085] In some embodiments instructions are grouped based on opcode 712 bit-fields to simplify Opcode decode 740. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group 742 includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group 742 shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group 744 (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group 746 includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group 748 includes component-wise arithmetic instructions (e.g., add, multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group 748 performs the arithmetic operations in parallel across data channels. The vector math group 750 includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

[0086] **Figure 11** is a block diagram of another embodiment of a graphics processor 800. Elements of Figure 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.

[0087] In some embodiments, graphics processor 800 includes a graphics pipeline 820, a media pipeline 830, a display engine 840, thread execution logic 850, and a render output pipeline 870. In some embodiments, graphics processor 800 is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor 800 via a ring interconnect 802. In some embodiments, ring interconnect 802 couples graphics processor 800 to other processing components,

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

[0088] 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.

[0089] 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.

[0090] In some embodiments, graphics pipeline 820 includes tessellation components to perform hardware-accelerated tessellation of 3D objects. In some embodiments, a programmable hull shader 811 configures the tessellation operations. A programmable domain shader 817 provides back-end evaluation of tessellation output. A tessellator 813 operates at the direction of hull shader 811 and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to graphics pipeline 820. In some embodiments, if tessellation is not used, tessellation components (e.g., hull shader 811, tessellator 813, and domain shader 817) can be bypassed.

[0091] In some embodiments, complete geometric objects can be processed by a geometry shader 819 via one or more threads dispatched to execution units 852A-852B, or can proceed directly to the clipper 829. In some embodiments, the geometry shader operates on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation

is disabled the geometry shader 819 receives input from the vertex shader 807. In some embodiments, geometry shader 819 is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

[0092] Before rasterization, a clipper 829 processes vertex data. The clipper 829 may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In some embodiments, a rasterizer and depth test component 873 in the render output pipeline 870 dispatches pixel shaders to convert the geometric objects into their per pixel representations. In some embodiments, pixel shader logic is included in thread execution logic 850. In some embodiments, an application can bypass the rasterizer and depth test component 873 and access un-rasterized vertex data via a stream out unit 823.

[0093] The graphics processor 800 has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units 852A-852B and associated cache(s) 851, texture and media sampler 854, and texture/sampler cache 858 interconnect via a data port 856 to perform memory access and communicate with render output pipeline components of the processor. In some embodiments, sampler 854, caches 851, 858 and execution units 852A-852B each have separate memory access paths.

[0094] In some embodiments, render output pipeline 870 contains a rasterizer and depth test component 873 that converts vertex-based objects into an associated pixel-based representation. In some embodiments, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache 878 and depth cache 879 are also available in some embodiments. A pixel operations component 877 performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine 841, or substituted at display time by the display controller 843 using overlay display planes. In some embodiments, a shared L3 cache 875 is available to all graphics components, allowing the sharing of data without the use of main system memory.

[0095] In some embodiments, graphics processor media pipeline 830 includes a media engine 837 and a video front end 834. In some embodiments, video front end 834 receives pipeline commands from the command streamer 803. In some embodiments, media pipeline 830 includes a separate command streamer. In some embodiments, video front-end 834 processes media commands before sending the command to the media engine 837. In some embodiments, media engine 837 includes thread spawning functionality to spawn threads for dispatch to thread execution logic 850 via thread dispatcher 831.

[0096] In some embodiments, graphics processor 800 includes a display engine 840. In some embodiments, display engine 840 is external to processor 800 and couples with the graphics processor via the ring interconnect 802, or some other interconnect bus or fabric. In some embodiments, display engine 840 includes a 2D engine 841 and a display controller 843. In some embodiments, display engine 840 contains special purpose logic capable of operating independently of the 3D pipeline. In some embodiments, display controller 843 couples with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

[0097] In some embodiments, graphics pipeline 820 and media pipeline 830 are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL), Open Computing Language (OpenCL), and/or Vulkan graphics and compute API, all from the Khronos Group. In some embodiments, support may also be provided for the Direct3D library from the Microsoft Corporation. In some embodiments, a combination of these libraries may be supported. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

[0098] **Figure 12A** is a block diagram illustrating a graphics processor command format 900 according to some embodiments. **Figure 12B** is a block diagram illustrating a graphics processor command sequence 910 according to an embodiment. The solid lined boxes in Figure 12A illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format 900 of Figure 12A 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.

[0099] 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.

[0100] The flow diagram in Figure 12B 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.

[0101] 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.

[0102] In some embodiments, a pipeline select command 913 is used when a command sequence requires the graphics processor to explicitly switch between pipelines. In some embodiments, a pipeline select command 913 is required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. In some embodiments, a pipeline flush command 912 is required immediately before a pipeline switch via the pipeline select command 913.

[0103] 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.

[0104] In some embodiments, return buffer state commands 916 are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more

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

[0105] The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination 920, the command sequence is tailored to the 3D pipeline 922 beginning with the 3D pipeline state 930 or the media pipeline 924 beginning at the media pipeline state 940.

[0106] The commands to configure the 3D pipeline state 930 include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based on the particular 3D API in use. In some embodiments, 3D pipeline state 930 commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

[0107] In some embodiments, 3D primitive 932 command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive 932 command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive 932 command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. In some embodiments, 3D primitive 932 command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, 3D pipeline 922 dispatches shader execution threads to graphics processor execution units.

[0108] In some embodiments, 3D pipeline 922 is triggered via an execute 934 command or event. In some embodiments, a register write triggers command execution. In some embodiments execution is triggered via a ‘go’ or ‘kick’ command in the command sequence. In one embodiment, command execution is triggered using a pipeline synchronization command to flush the command sequence through

the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

[0109] In some embodiments, the graphics processor command sequence 910 follows the media pipeline 924 path when performing media operations. In general, the specific use and manner of programming for the media pipeline 924 depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. In some embodiments, the media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general purpose processing cores. In one embodiment, the media pipeline also includes elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

[0110] In some embodiments, media pipeline 924 is configured in a similar manner as the 3D pipeline 922. A set of commands to configure the media pipeline state 940 are dispatched or placed into a command queue before the media object commands 942. In some embodiments, media pipeline state commands 940 include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. In some embodiments, media pipeline state commands 940 also support the use of one or more pointers to "indirect" state elements that contain a batch of state settings.

[0111] In some embodiments, media object commands 942 supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. In some embodiments, all media pipeline states must be valid before issuing a media object command 942. Once the pipeline state is configured and media object commands 942 are queued, the media

pipeline 924 is triggered via an execute command 944 or an equivalent execute event (e.g., register write). Output from media pipeline 924 may then be post processed by operations provided by the 3D pipeline 922 or the media pipeline 924. In some embodiments, GPGPU operations are configured and executed in a similar manner as media operations.

[0112] **Figure 13** illustrates exemplary graphics software architecture for a data processing system 1000 according to some embodiments. In some embodiments, software architecture includes a 3D graphics application 1010, an operating system 1020, and at least one processor 1030. In some embodiments, processor 1030 includes a graphics processor 1032 and one or more general-purpose processor core(s) 1034. The graphics application 1010 and operating system 1020 each execute in the system memory 1050 of the data processing system.

[0113] In some embodiments, 3D graphics application 1010 contains one or more shader programs including shader instructions 1012. The shader language instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL Shader Language (GLSL). The application also includes executable instructions 1014 in a machine language suitable for execution by the general-purpose processor core 1034. The application also includes graphics objects 1016 defined by vertex data.

[0114] In some embodiments, operating system 1020 is a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. The operating system 1020 can support a graphics API 1022 such as the Direct3D API, the OpenGL API, or the Vulkan API. When the Direct3D API is in use, the operating system 1020 uses a front-end shader compiler 1024 to compile any shader instructions 1012 in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. In some embodiments, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application 1010. In some embodiments, the shader instructions 1012 are provided in an intermediate

form, such as a version of the Standard Portable Intermediate Representation (SPIR) used by the Vulkan API.

[0115] In some embodiments, user mode graphics driver 1026 contains a back-end shader compiler 1027 to convert the shader instructions 1012 into a hardware specific representation. When the OpenGL API is in use, shader instructions 1012 in the GLSL high-level language are passed to a user mode graphics driver 1026 for compilation. In some embodiments, user mode graphics driver 1026 uses operating system kernel mode functions 1028 to communicate with a kernel mode graphics driver 1029. In some embodiments, kernel mode graphics driver 1029 communicates with graphics processor 1032 to dispatch commands and instructions.

[0116] One or more aspects of at least one embodiment may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as "IP cores," are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model may be supplied to various customers or manufacturing facilities, which load the hardware model on fabrication machines that manufacture the integrated circuit. The integrated circuit may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

[0117] **Figure 14** is a block diagram illustrating an IP core development system 1100 that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system 1100 may be used to generate modular, re-usable designs that can be incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility 1130 can generate a software simulation 1110 of an IP core design in a high level programming language (e.g., C/C++). The software simulation 1110 can be used to design, test, and verify the behavior of the IP core using a simulation

model 1112. The simulation model 1112 may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design 1115 can then be created or synthesized from the simulation model 1112. The RTL design 1115 is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design 1115, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

[0118] 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.

[0119] **Figures 15-17** illustrated exemplary integrated circuits and associated graphics processors that may be fabricated using one or more IP cores, according to various embodiments described herein. In addition to what is illustrated, other logic and circuits may be included, including additional graphics processors/cores, peripheral interface controllers, or general purpose processor cores.

[0120] **Figure 15** is a block diagram illustrating an exemplary system on a chip integrated circuit 1200 that may be fabricated using one or more IP cores, according to an embodiment. Exemplary integrated circuit 1200 includes one or more application processor(s) 1205 (e.g., CPUs), at least one graphics processor 1210, and may additionally include an image processor 1215 and/or a video processor 1220, any of which may be a modular IP core from the same or multiple different design facilities. Integrated circuit 1200 includes peripheral or bus logic including a

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

[0121] **Figure 16** is a block diagram illustrating an exemplary graphics processor 1310 of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor 1310 can be a variant of the graphics processor 1210 of Figure 15. Graphics processor 1310 includes a vertex processor 1305 and one or more fragment processor(s) 1315A1315N (e.g., 1315A, 1315B, 1315C, 1315D, through 1315N-1, and 1315N). Graphics processor 1310 can execute different shader programs via separate logic, such that the vertex processor 1305 is optimized to execute operations for vertex shader programs, while the one or more fragment processor(s) 1315A-1315N execute fragment (e.g., pixel) shading operations for fragment or pixel shader programs. The vertex processor 1305 performs the vertex processing stage of the 3D graphics pipeline and generates primitives and vertex data. The fragment processor(s) 1315A-1315N use the primitive and vertex data generated by the vertex processor 1305 to produce a framebuffer that is displayed on a display device. In one embodiment, the fragment processor(s) 1315A-1315N are optimized to execute fragment shader programs as provided for in the OpenGL API, which may be used to perform similar operations as a pixel shader program as provided for in the Direct 3D API.

[0122] Graphics processor 1310 additionally includes one or more memory management units (MMUs) 1320A-1320B, cache(s) 1325A-1325B, and circuit interconnect(s) 1330A-1330B. The one or more MMU(s) 1320A-1320B provide for virtual to physical address mapping for integrated circuit 1310, including for the vertex processor 1305 and/or fragment processor(s) 1315A-1315N, which may reference vertex or image/texture data stored in memory, in addition to vertex or

image/texture data stored in the one or more cache(s) 1325A-1325B. In one embodiment the one or more MMU(s) 1325A-1325B may be synchronized with other MMUs within the system, including one or more MMUs associated with the one or more application processor(s) 1205, image processor 1215, and/or video processor 1220 of **Figure 16**, such that each processor 1205-1220 can participate in a shared or unified virtual memory system. The one or more circuit interconnect(s) 1330A-1330B enable graphics processor 1310 to interface with other IP cores within the SoC, either via an internal bus of the SoC or via a direct connection, according to embodiments.

[0123] **Figure 17** is a block diagram illustrating an additional exemplary graphics processor 1410 of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor 1410 can be a variant of the graphics processor 1210 of **Figure 15**. Graphics processor 1410 includes the one or more MMU(s) 1320A-1320B, caches 1325A-1325B, and circuit interconnects 1330A-1330B of the integrated circuit 1300 of **Figure 16**.

[0124] Graphics processor 1410 includes one or more shader core(s) 1415A-1415N (e.g., 1415A, 1415B, 1415C, 1415D, 1415E, 1415F, through 1315N-1, and 1315N), which provides for a unified shader core architecture in which a single core or type or core can execute all types of programmable shader code, including shader program code to implement vertex shaders, fragment shaders, and/or compute shaders. The exact number of shader cores present can vary among embodiments and implementations. Additionally, graphics processor 1410 includes an inter-core task manager 1405, which acts as a thread dispatcher to dispatch execution threads to one or more shader cores 1415A-1415N and a tiling unit 1418 to accelerate tiling operations for tile-based rendering, in which rendering operations for a scene are subdivided in image space, for example to exploit local spatial coherence within a scene or to optimize use of internal caches.

[0125] The following clauses and/or examples pertain to further embodiments:

One example embodiment maybe a method comprising locating a first set of N samples along a ray through a volume in connection with a first frame, storing the first set of N samples, locating a second set of N additional samples along the ray in

connection with a second frame, and combining said first and second sets of samples. The method may include changing the point along the ray where a first sample of said second set is taken in the second frame. The method may include randomly selecting the point to begin sampling along the ray in connection with the second frame. The method may include performing data parallel rendering using a plurality of nodes. The method may include sorting the samples to a node that stores data for a sample being processed on the node. The method may include sorting a list of samples from front to back. The method may include processing samples along the same ray in different nodes. The method may include where each node sends only the result of the node's blended samples instead of all samples taken.

[0126] Another example embodiment may be one or more non-transitory computer readable media storing instructions to perform a sequence comprising locating a first set of N samples along a ray through a volume in connection with a first frame, storing the first set of N samples, locating a second set of N additional samples along the ray in connection with a second frame, and combining said first and second sets of samples. The media may further storing instructions to perform a sequence including changing the point along the ray where a first sample of said second set is taken in the second frame. The media may further store instructions to perform a sequence including randomly selecting the point to begin sampling along the ray in connection with the second frame. The media may further store instructions to perform a sequence including performing data parallel rendering using a plurality of nodes. The media may further store instructions to perform a sequence including sorting the samples to a node that stores data for a sample being processed on the node. The media may further store instructions to perform a sequence including sorting a list of samples from front to back. The media may further store instructions to perform a sequence including processing samples along the same ray in different nodes. The media may further store instructions to perform a sequence where each node sends only the result of the node's blended samples instead of all samples taken.

[0127] In another example embodiment may be an apparatus comprising a processor to locate a first set of N samples along a ray through a volume in connection with a first frame, store the first set of N samples, locate a second set of N additional samples along the ray in connection with a second frame, and combine said first and second sets of samples, and a memory coupled to said processor. The apparatus of said processor to change the point along the ray where a first sample of said second set is taken in the second frame. The apparatus of said processor to randomly select the point to begin sampling along the ray in connection with the second frame. The apparatus of said processor to perform data parallel rendering using a plurality of nodes. The apparatus of said processor to sort the samples to a node that stores data for a sample being processed on the node. The apparatus of said processor to sort a list of samples from front to back. The apparatus of said processor to process samples along the same ray in different nodes. The apparatus of said processor where each node sends only the result of the node's blended samples instead of all samples taken.

[0128] The graphics processing techniques described herein may be implemented in various hardware architectures. For example, graphics functionality may be integrated within a chipset. Alternatively, a discrete graphics processor may be used. As still another embodiment, the graphics functions may be implemented by a general purpose processor, including a multicore processor.

[0129] References throughout this specification to "one embodiment" or "an embodiment" mean that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one implementation encompassed within the present disclosure. Thus, appearances of the phrase "one embodiment" or "in an embodiment" are not necessarily referring to the same embodiment. Furthermore, the particular features, structures, or characteristics may be instituted in other suitable forms other than the particular embodiment illustrated and all such forms may be encompassed within the claims of the present application.

[0130] While a limited number of embodiments have been described, those skilled in the art will appreciate numerous modifications and variations therefrom. It is

intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this disclosure.

What is claimed is:

1. A method comprising:
locating a first set of N samples along a ray through a volume in connection with a first frame;
storing the first set of N samples;
locating a second set of N additional samples along the ray in connection with a second frame; and
combining said first and second sets of samples.

2. The method of claim 1 including changing the point along the ray where a first sample of said second set is taken in the second frame.

3. The method of claim 2 including randomly selecting the point to begin sampling along the ray in connection with the second frame.

4. The method of claim 3 including performing data parallel rendering using a plurality of nodes.

5. The method of claim 4 including sorting the samples to a node that stores data for a sample being processed on the node.

6. The method of claim 4 including sorting a list of samples from front to back.

7. The method of claim 4 including processing samples along the same ray in different nodes.

8. The method of claim 4 where each node sends only the result of the node's blended samples instead of all samples taken.

9. One or more non-transitory computer readable media storing instructions to perform a sequence comprising:

locating a first set of N samples along a ray through a volume in connection with a first frame;

storing the first set of N samples;

locating a second set of N additional samples along the ray in connection with a second frame; and

combining said first and second sets of samples.

10. The media of claim 9, further storing instructions to perform a sequence including changing the point along the ray where a first sample of said second set is taken in the second frame.

11. The media of claim 10, further storing instructions to perform a sequence including randomly selecting the point to begin sampling along the ray in connection with the second frame.

12. The media of claim 11, further storing instructions to perform a sequence including performing data parallel rendering using a plurality of nodes.

13. The media of claim 12, further storing instructions to perform a sequence including sorting the samples to a node that stores data for a sample being processed on the node.

14. The media of claim 12, further storing instructions to perform a sequence including sorting a list of samples from front to back.

15. The media of claim 12, further storing instructions to perform a sequence including processing samples along the same ray in different nodes.

16. The media of claim 12, further storing instructions to perform a sequence where each node sends only the result of the node's blended samples instead of all samples taken.

17. An apparatus comprising:

a processor to locate a first set of N samples along a ray through a volume in connection with a first frame, store the first set of N samples, locate a second set of N additional samples along the ray in connection with a second frame, and combine said first and second sets of samples; and

a memory coupled to said processor.

18. The apparatus of claim 17, said processor to change the point along the ray where a first sample of said second set is taken in the second frame.

19. The apparatus of claim 17, said processor to randomly select the point to begin sampling along the ray in connection with the second frame.

20. The apparatus of claim 19, said processor to perform data parallel rendering using a plurality of nodes.

21. The apparatus of claim 20, said processor to sort the samples to a node that stores data for a sample being processed on the node.

22. The apparatus of claim 20, said processor to sort a list of samples from front to back.

23. The apparatus of claim 20, said processor to process samples along the same ray in different nodes.

24. The apparatus of claim 20, said processor where each node sends only the result of the node's blended samples instead of all samples taken.

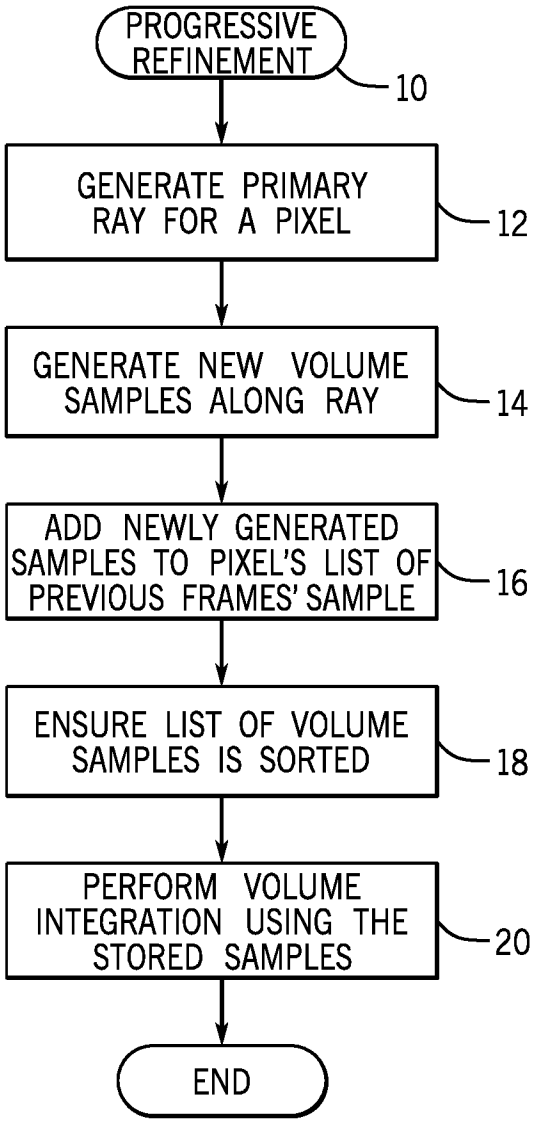

PROGRESSIVE REFINEMENT
10
GENERATE PRIMARY RAY FOR A PIXEL
12
GENERATE NEW VOLUME SAMPLES ALONG RAY
14
ADD NEWLY GENERATED SAMPLES TO PIXEL'S LIST OF PREVIOUS FRAMES' SAMPLE
16
ENSURE LIST OF VOLUME SAMPLES IS SORTED
18
PERFORM VOLUME INTEGRATION USING THE STORED SAMPLES
20
END

FIG. 1

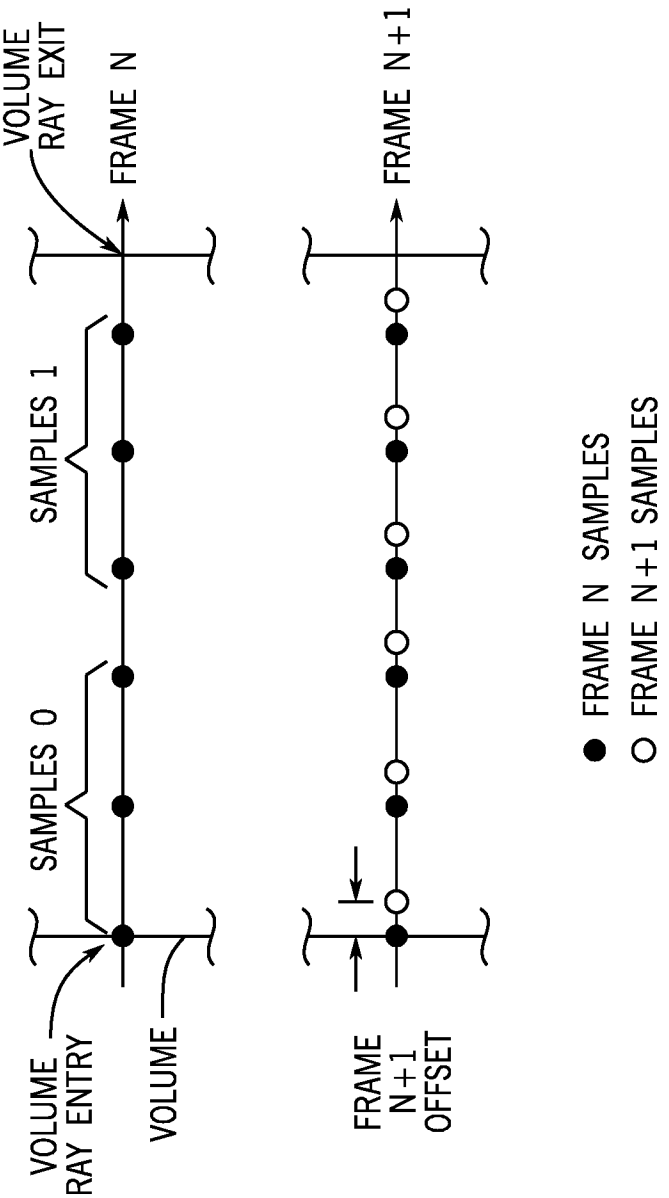

VOLUME RAY EXIT
FRAME N
FRAME N+1
SAMPLES 1
SAMPLES 0
VOLUME RAY ENTRY
VOLUME
FRAME N+1 OFFSET
FRAME N SAMPLES
FRAME N+1 SAMPLES

FIG. 2

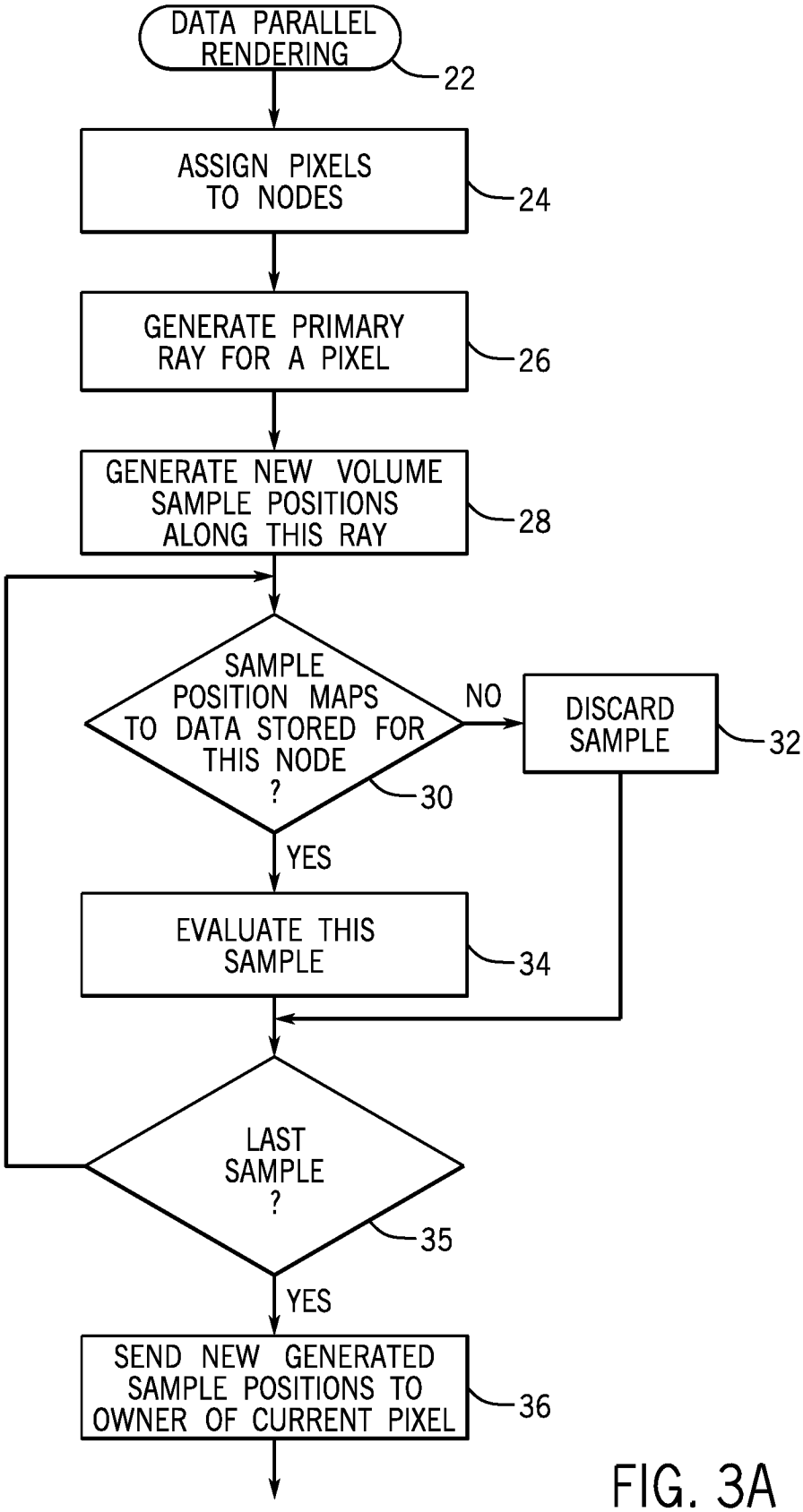

DATA PARALLEL RENDERING
22
ASSIGN PIXELS TO NODES
24
GENERATE PRIMARY RAY FOR A PIXEL
26
GENERATE NEW VOLUME SAMPLE POSITIONS ALONG THIS RAY
28
SAMPLE POSITION MAPS TO DATA STORED FOR THIS NODE ?
30
NO
DISCARD SAMPLE
32
YES
EVALUATE THIS SAMPLE
34
LAST SAMPLE ?
35
YES
SEND NEW GENERATED SAMPLE POSITIONS TO OWNER OF CURRENT PIXEL
36

FIG. 3A

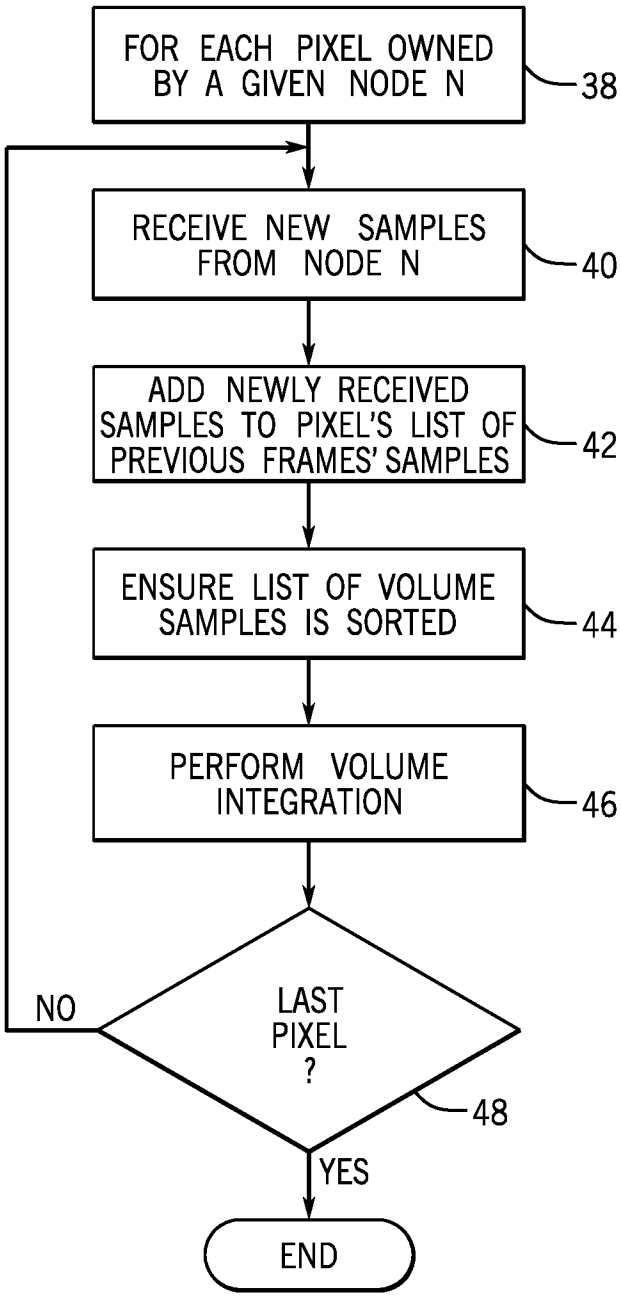

FOR EACH PIXEL OWNED BY A GIVEN NODE N
38
RECEIVE NEW SAMPLES FROM NODE N
40
ADD NEWLY RECEIVED SAMPLES TO PIXEL'S LIST OF PREVIOUS FRAMES' SAMPLES
42
ENSURE LIST OF VOLUME SAMPLES IS SORTED
44
PERFORM VOLUME INTEGRATION
46
LAST PIXEL ?
48
NO
YES
END

FIG. 3B

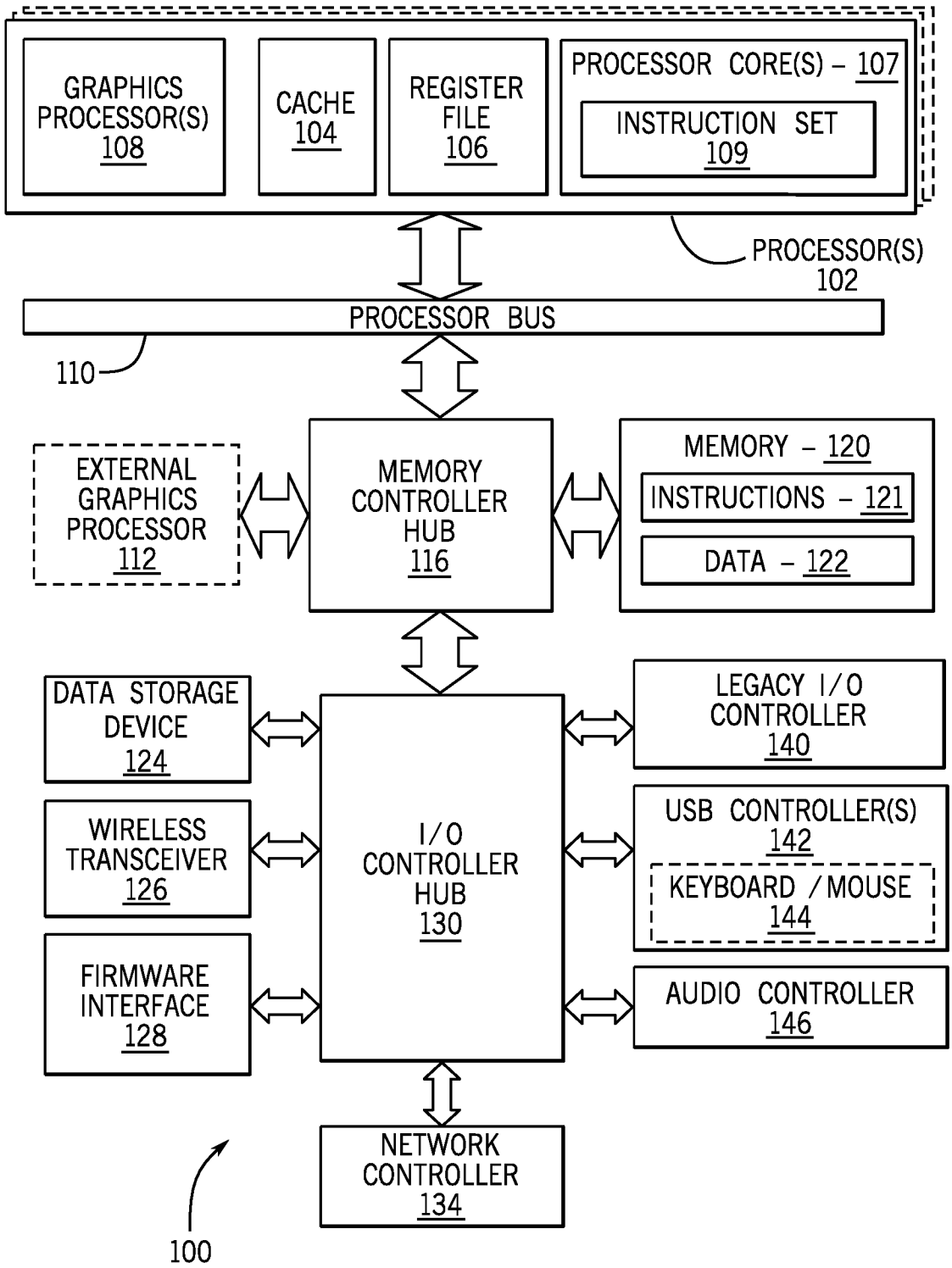

GRAPHICS PROCESSOR(S) 108
CACHE 104
REGISTER FILE 106
PROCESSOR CORE(S) – 107
INSTRUCTION SET 109
PROCESSOR(S) 102
PROCESSOR BUS
110
EXTERNAL GRAPHICS PROCESSOR 112
MEMORY CONTROLLER HUB 116
MEMORY – 120
INSTRUCTIONS – 121
DATA – 122
DATA STORAGE DEVICE 124
WIRELESS TRANSCEIVER 126
FIRMWARE INTERFACE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

FIG. 4

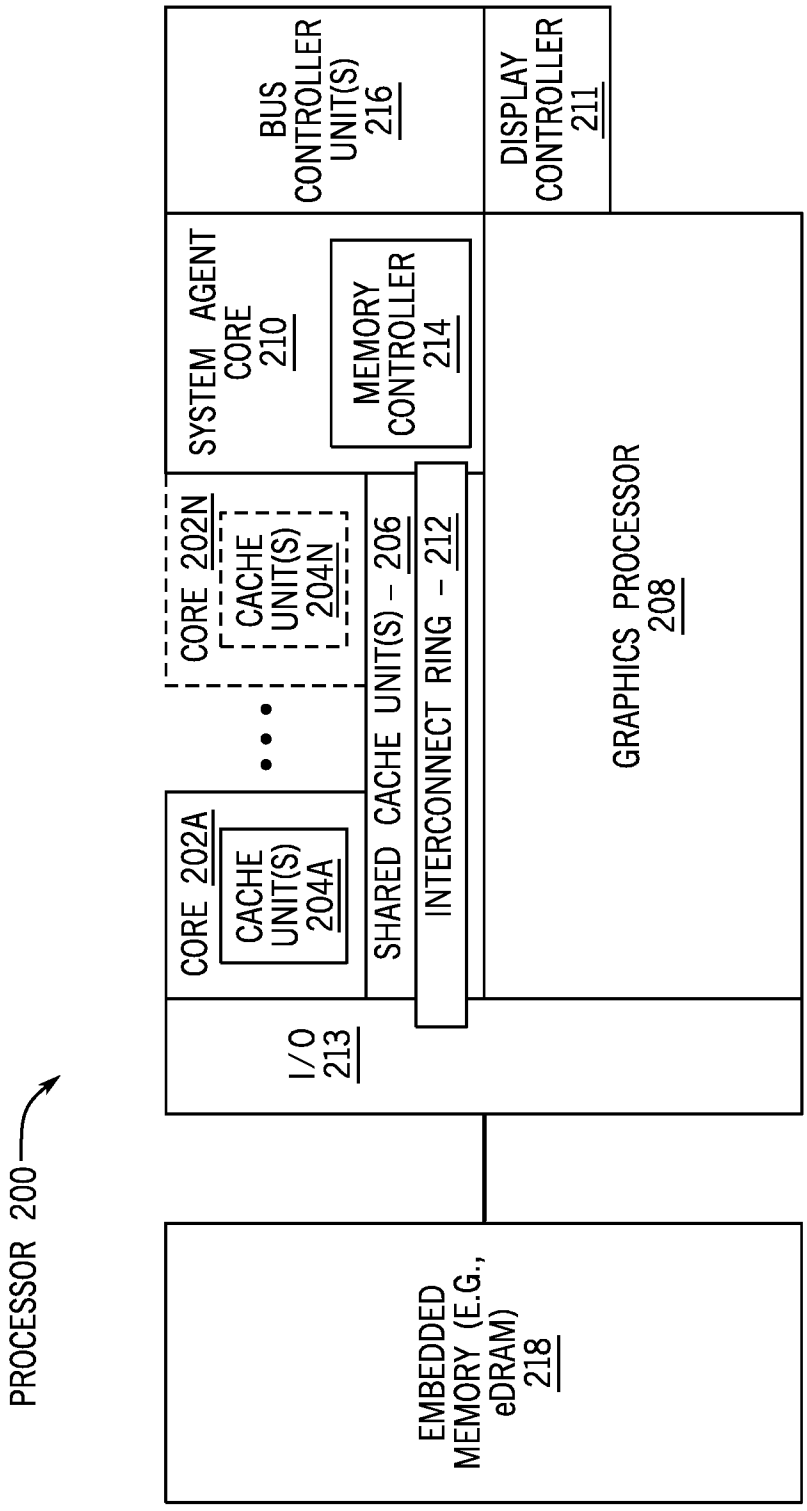

PROCESSOR 200
EMBEDDED MEMORY (E.G., eDRAM) 218
I/O 213
CORE 202A
CACHE UNIT(S) 204A
CORE 202N
CACHE UNIT(S) 204N
SHARED CACHE UNIT(S) - 206
INTERCONNECT RING - 212
GRAPHICS PROCESSOR 208
SYSTEM AGENT CORE 210
MEMORY CONTROLLER 214
BUS CONTROLLER UNIT(S) 216
DISPLAY CONTROLLER 211

FIG. 5

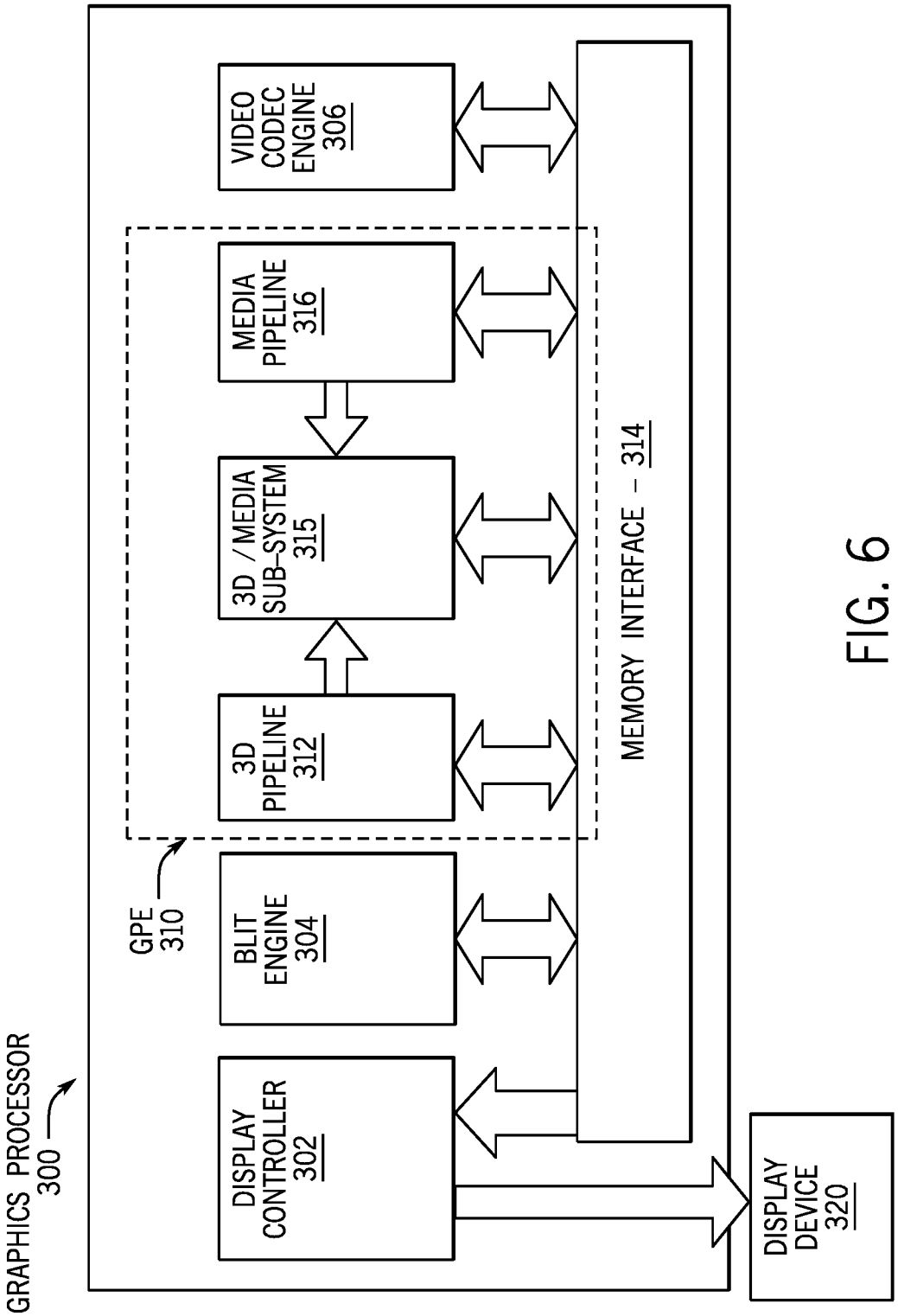

GRAPHICS PROCESSOR
300
DISPLAY CONTROLLER
302
BLIT ENGINE
304
GPE
310
3D PIPELINE
312
3D /MEDIA SUB-SYSTEM
315
MEDIA PIPELINE
316
VIDEO CODEC ENGINE
306
MEMORY INTERFACE – 314
DISPLAY DEVICE
320

FIG. 6

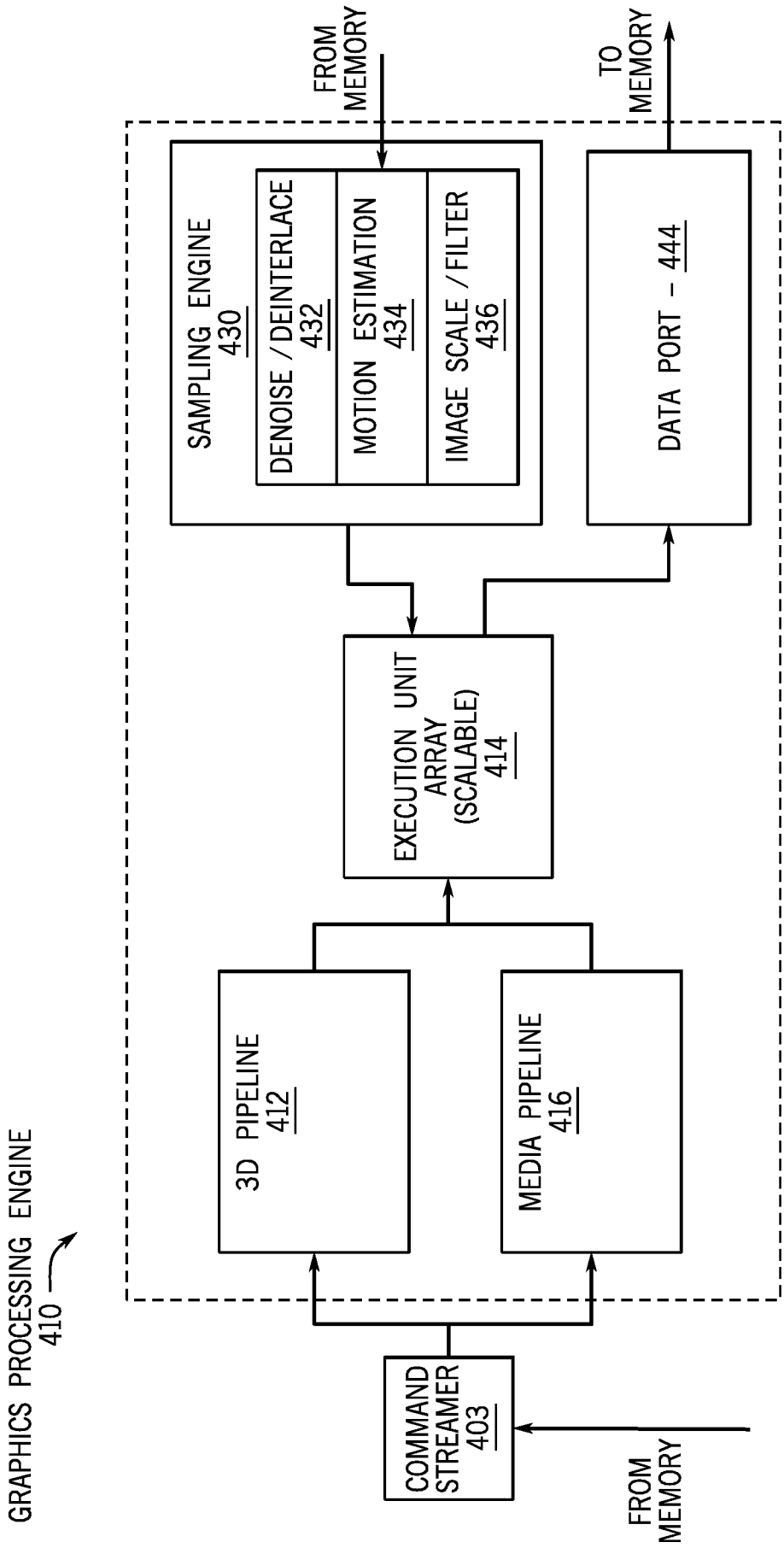

GRAPHICS PROCESSING ENGINE
410
FROM MEMORY
TO MEMORY
SAMPLING ENGINE
430
DENOISE / DEINTERLACE
432
MOTION ESTIMATION
434
IMAGE SCALE / FILTER
436
DATA PORT – 444
EXECUTION UNIT ARRAY (SCALABLE)
414
3D PIPELINE
412
MEDIA PIPELINE
416
COMMAND STREAMER
403
FROM MEMORY

FIG. 7

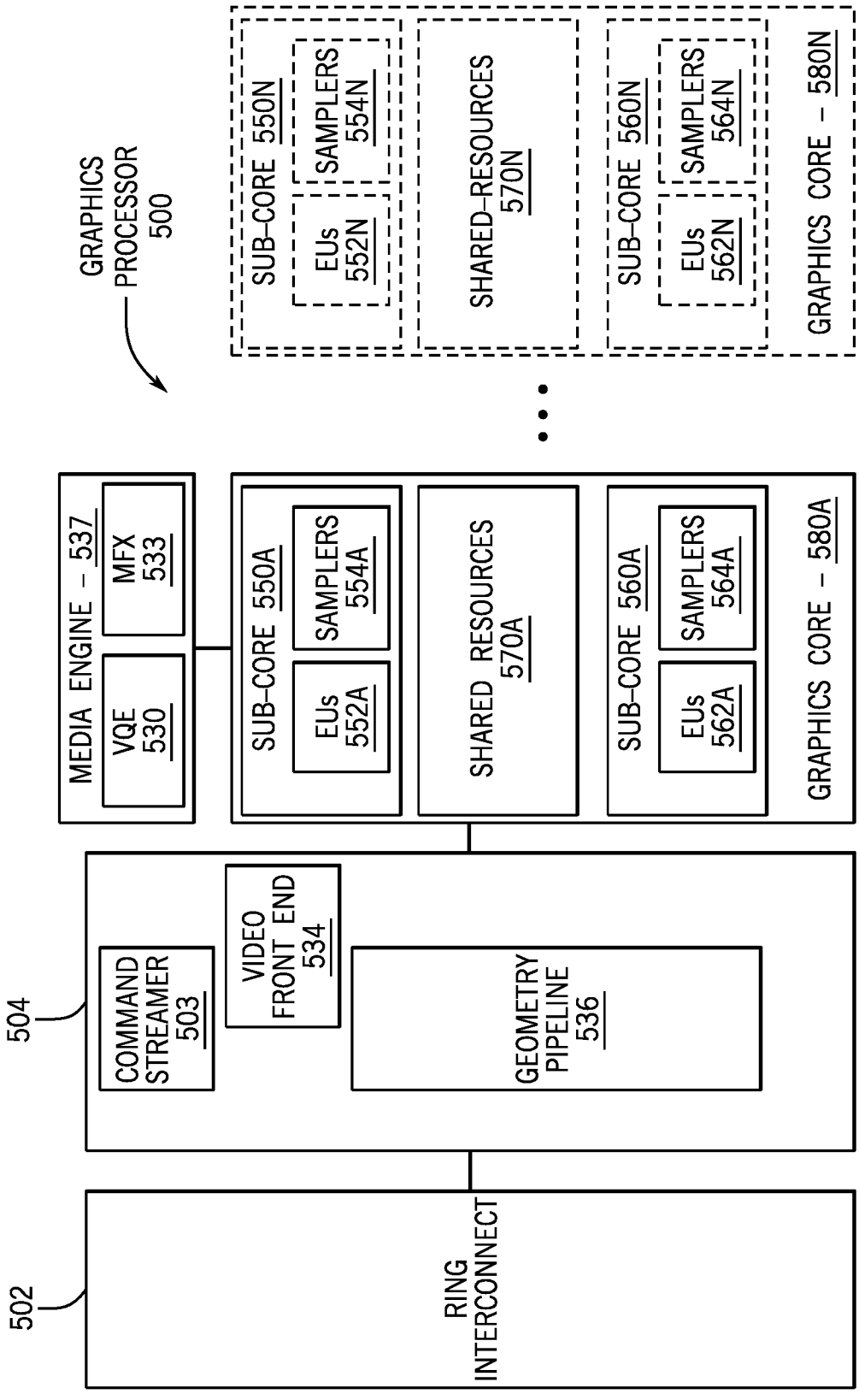

GRAPHICS PROCESSOR 500
SUB-CORE 550N
EUs 552N
SAMPLERS 554N
SHARED-RESOURCES 570N
SUB-CORE 560N
EUs 562N
SAMPLERS 564N
GRAPHICS CORE – 580N
MEDIA ENGINE – 537
VQE 530
MFX 533
SUB-CORE 550A
EUs 552A
SAMPLERS 554A
SHARED RESOURCES 570A
SUB-CORE 560A
EUs 562A
SAMPLERS 564A
GRAPHICS CORE – 580A
504
COMMAND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

FIG. 8

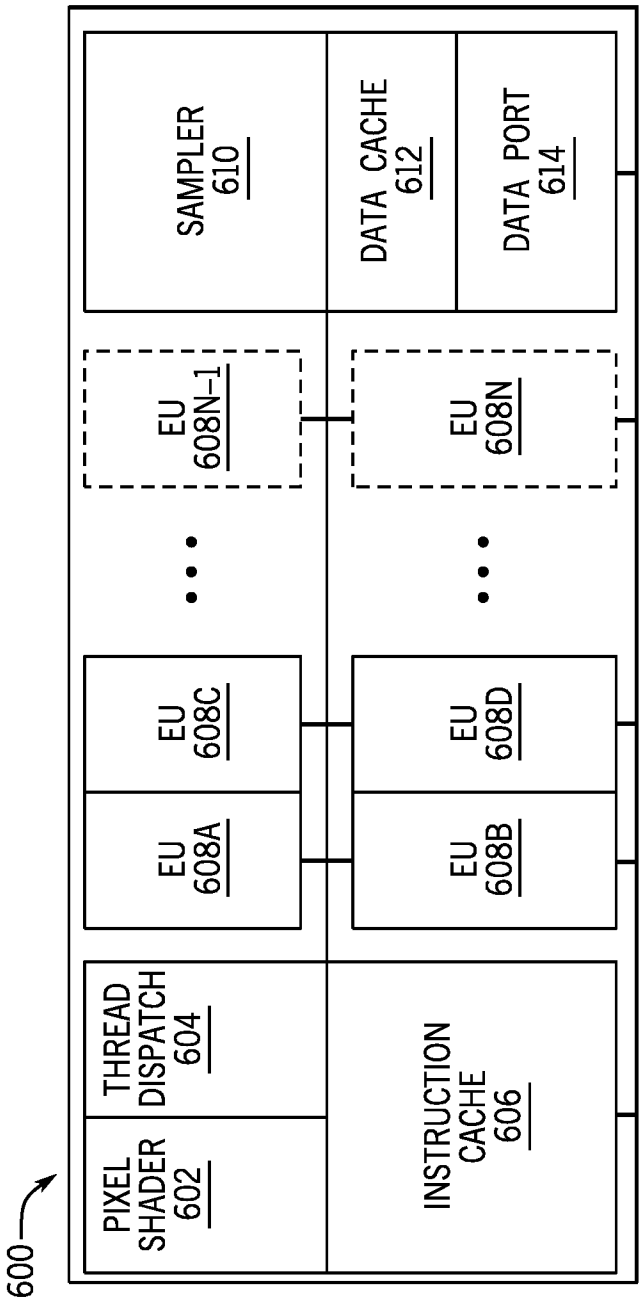

600
PIXEL SHADER 602
THREAD DISPATCH 604
INSTRUCTION CACHE 606
EU 608A
EU 608C
EU 608N–1
EU 608B
EU 608D
EU 608N
SAMPLER 610
DATA CACHE 612
DATA PORT 614

FIG. 9

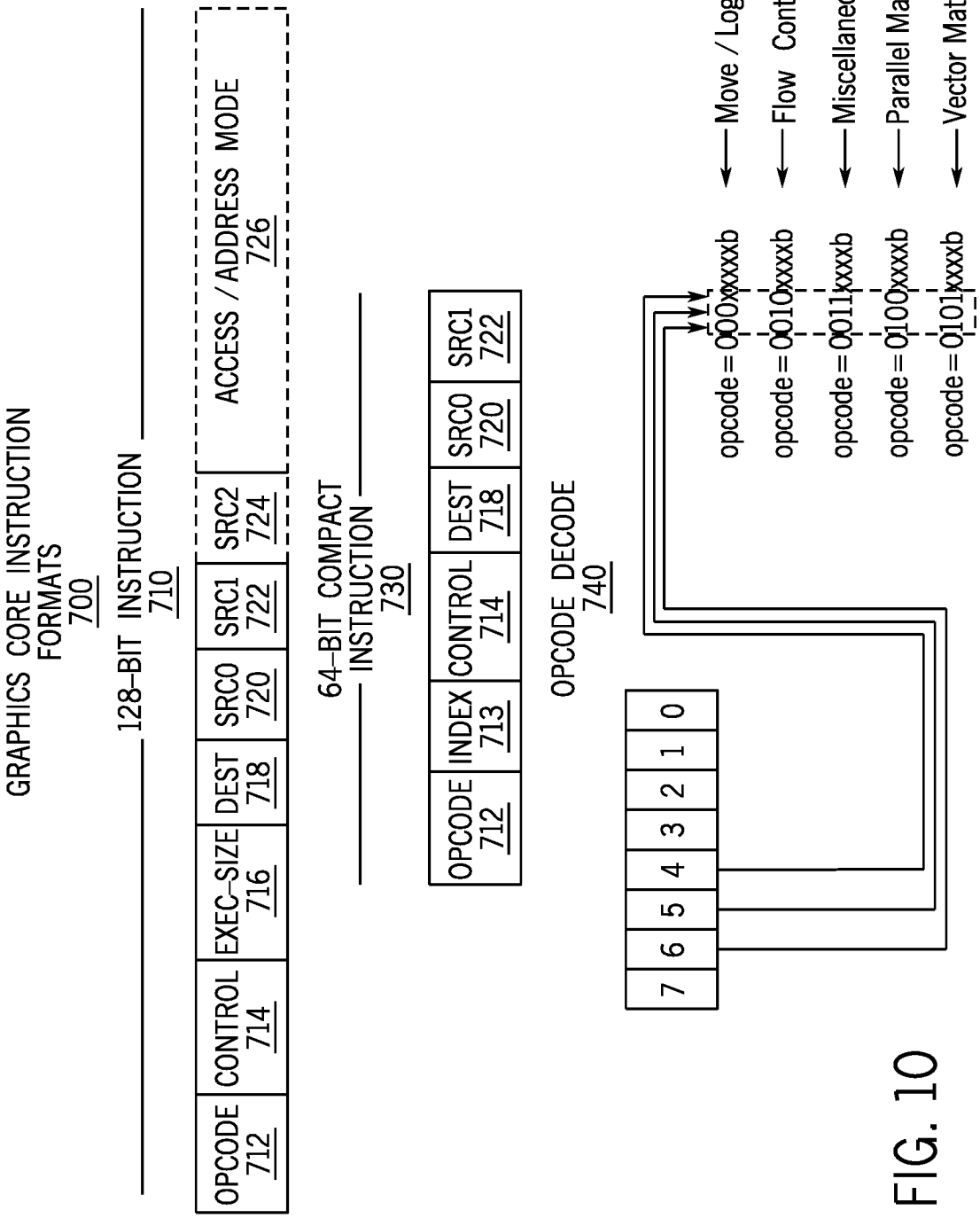

GRAPHICS CORE INSTRUCTION FORMATS 700
128-BIT INSTRUCTION 710
OPCODE 712
CONTROL 714
EXEC-SIZE 716
DEST 718
SRC0 720
SRC1 722
SRC2 724
ACCESS / ADDRESS MODE 726
64-BIT COMPACT INSTRUCTION 730
OPCODE 712
INDEX 713
CONTROL 714
DEST 718
SRC0 720
SRC1 722
OPCODE DECODE 740
7 6 5 4 3 2 1 0
opcode = 000xxxxb
opcode = 001xxxxb
opcode = 010xxxxb
opcode = 011xxxxb
opcode = 100xxxxb
opcode = 0101xxxxb
Move / Logic – 742
Flow Control – 744
Miscellaneous – 746
Parallel Math – 748
Vector Math – 750

FIG. 10

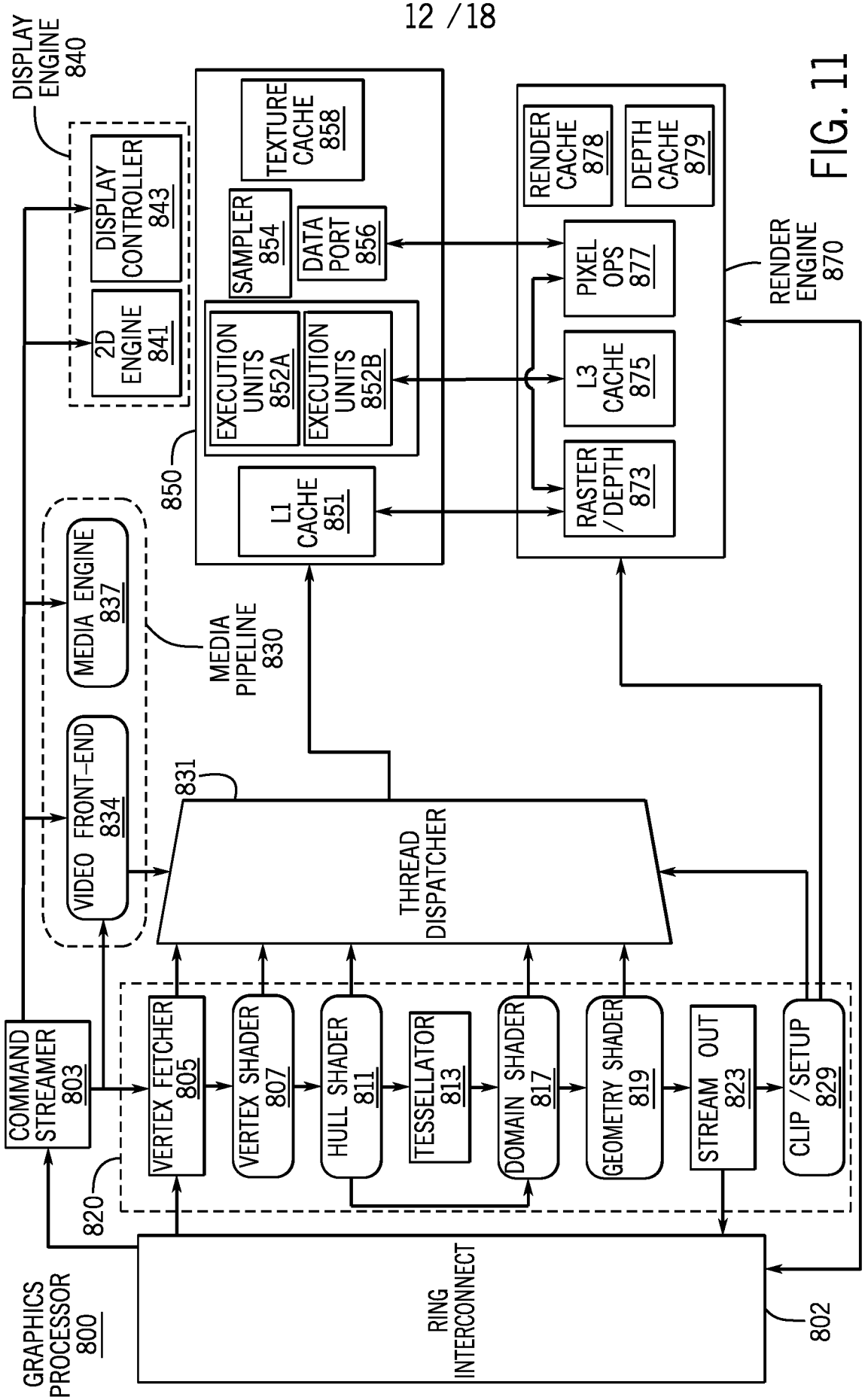

GRAPHICS PROCESSOR 800
RING INTERCONNECT
802
COMMAND STREAMER 803
820
VERTEX FETCHER 805
VERTEX SHADER 807
HULL SHADER 811
TESSELLATOR 813
DOMAIN SHADER 817
GEOMETRY SHADER 819
STREAM OUT 823
CLIP /SETUP 829
THREAD DISPATCHER
831
MEDIA PIPELINE 830
VIDEO FRONT-END 834
MEDIA ENGINE 837
DISPLAY ENGINE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

FIG. 11

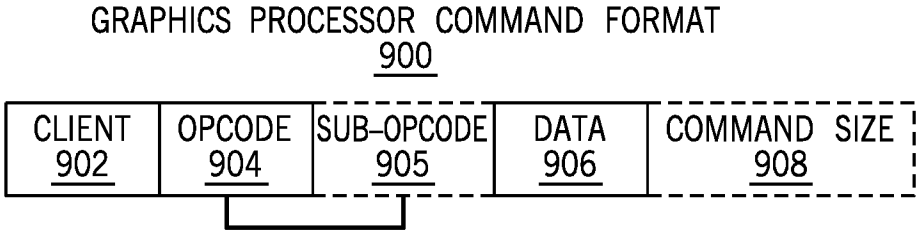


FIG. 12A

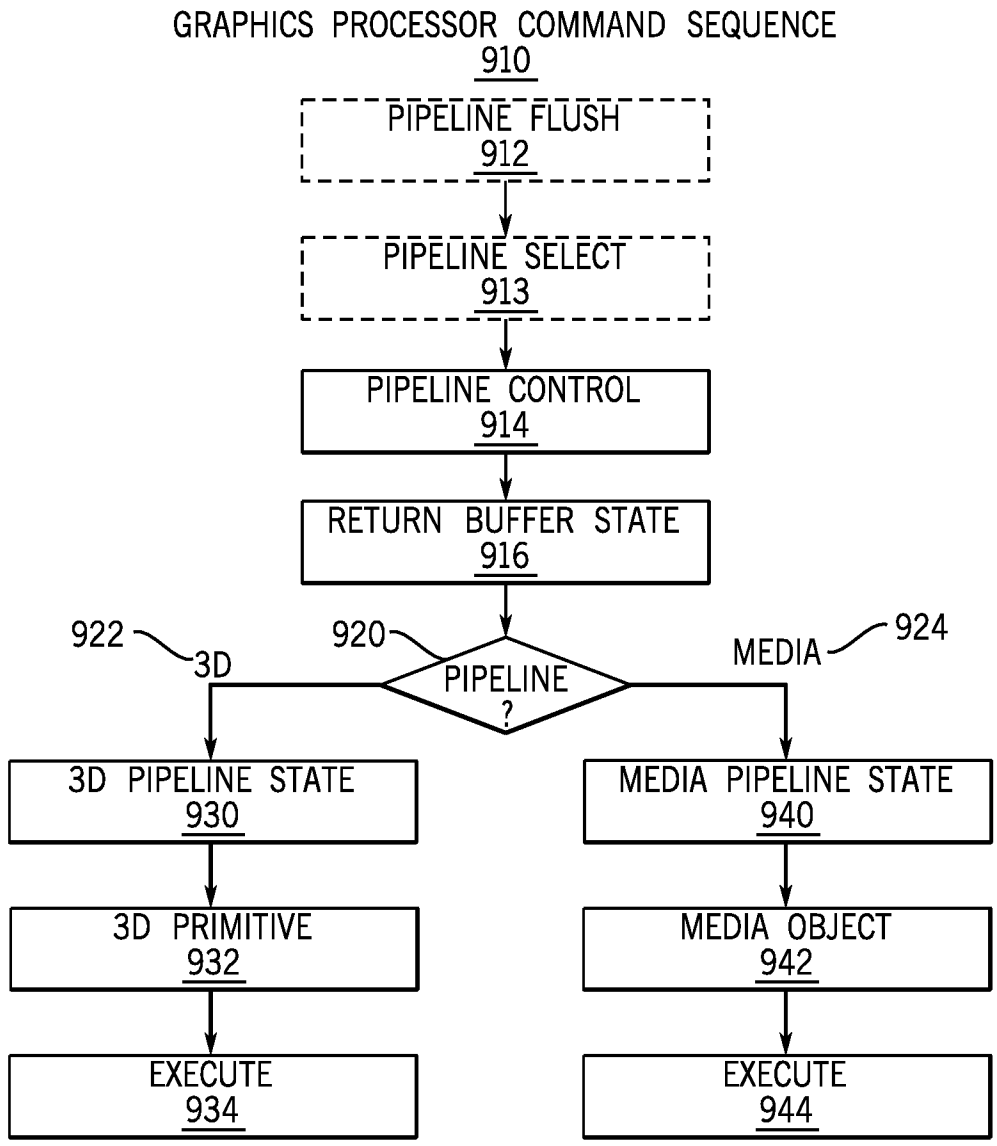


FIG. 12B

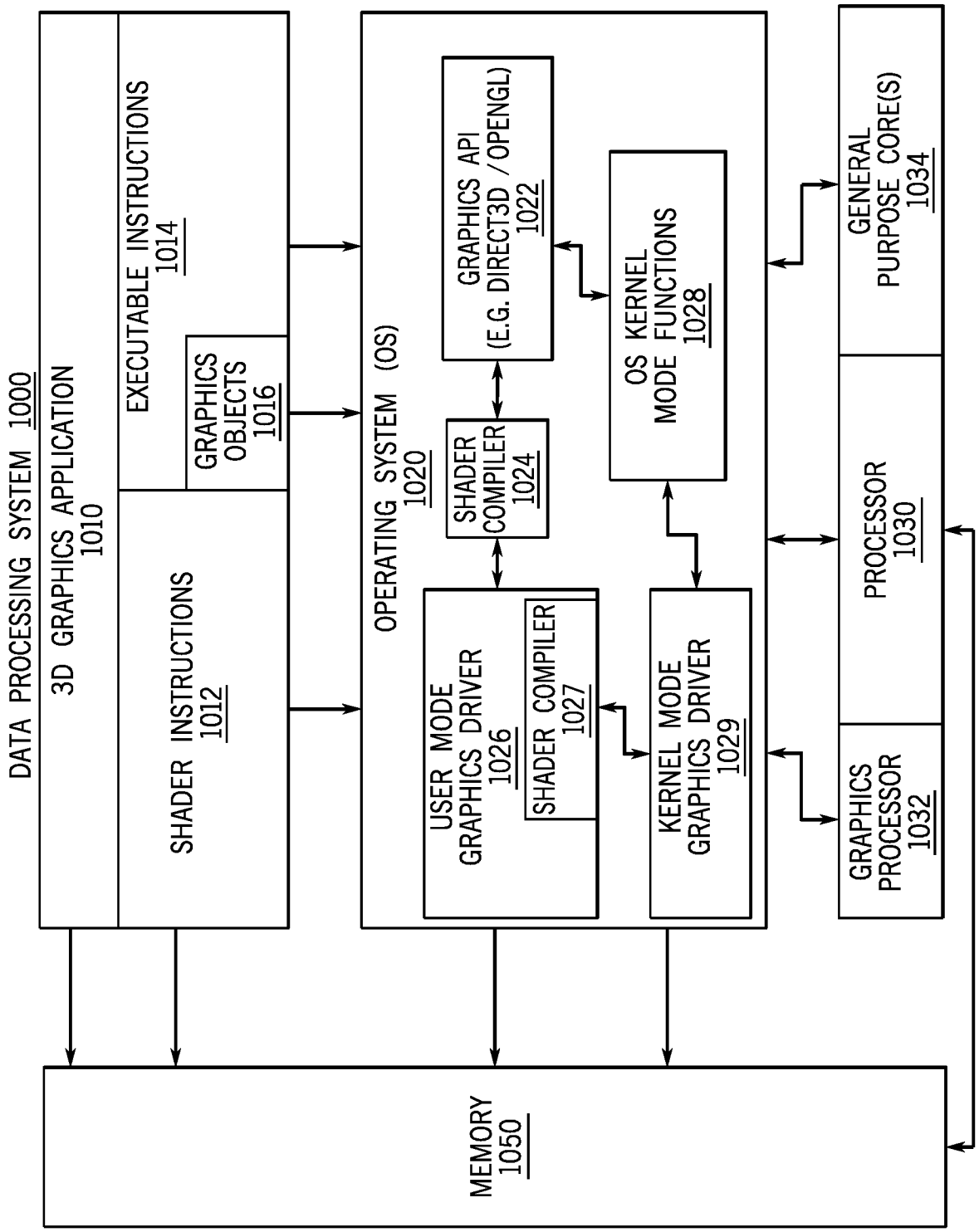

DATA PROCESSING SYSTEM 1000
3D GRAPHICS APPLICATION 1010
EXECUTABLE INSTRUCTIONS 1014
GRAPHICS OBJECTS 1016
SHADER INSTRUCTIONS 1012
OPERATING SYSTEM (OS) 1020
GRAPHICS API (E.G. DIRECT3D /OPENGL) 1022
SHADER COMPILER 1024
OS KERNEL MODE FUNCTIONS 1028
USER MODE GRAPHICS DRIVER 1026
SHADER COMPILER 1027
KERNEL MODE GRAPHICS DRIVER 1029
GENERAL PURPOSE CORE(S) 1034
PROCESSOR 1030
GRAPHICS PROCESSOR 1032
MEMORY 1050

FIG. 13

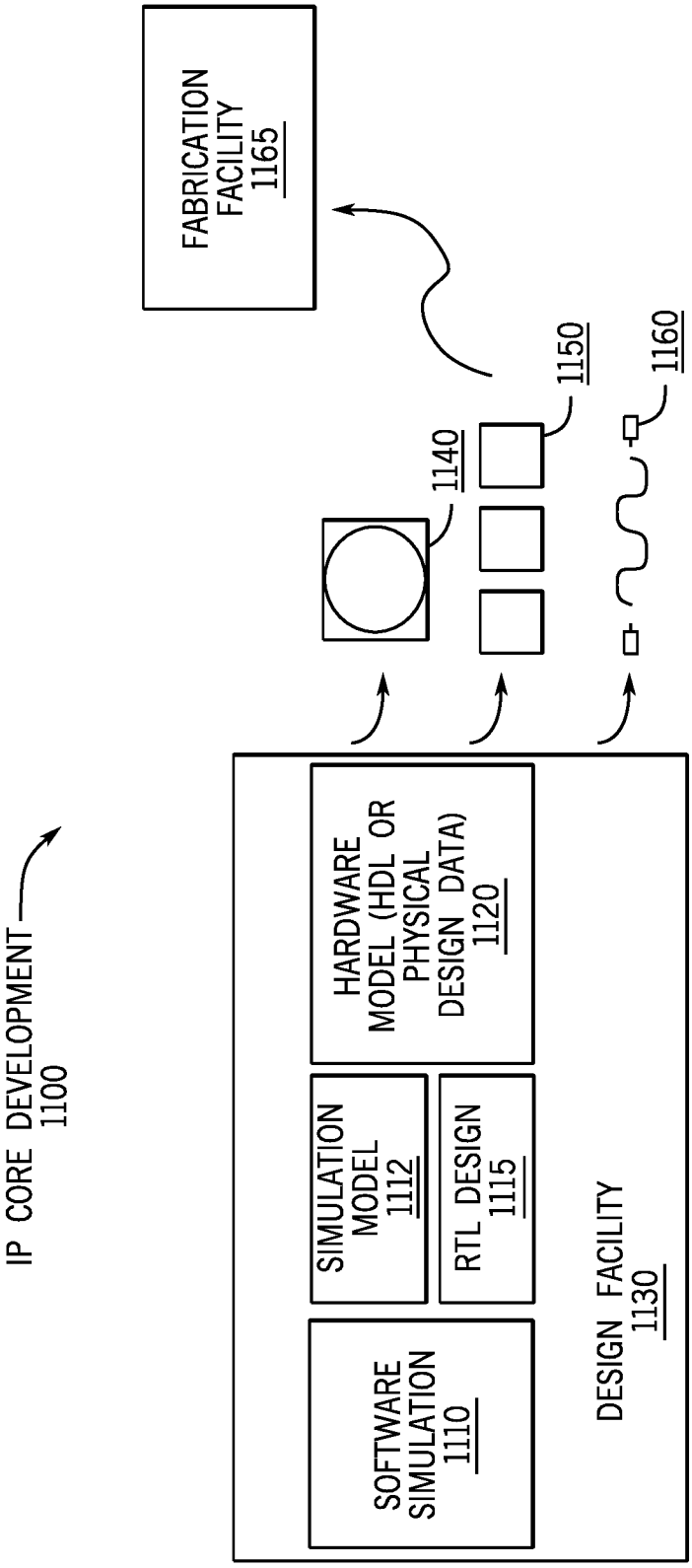

IP CORE DEVELOPMENT
1100
SOFTWARE SIMULATION
1110
SIMULATION MODEL
1112
RTL DESIGN
1115
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA)
1120
DESIGN FACILITY
1130
1140
1150
1160
FABRICATION FACILITY
1165

FIG. 14

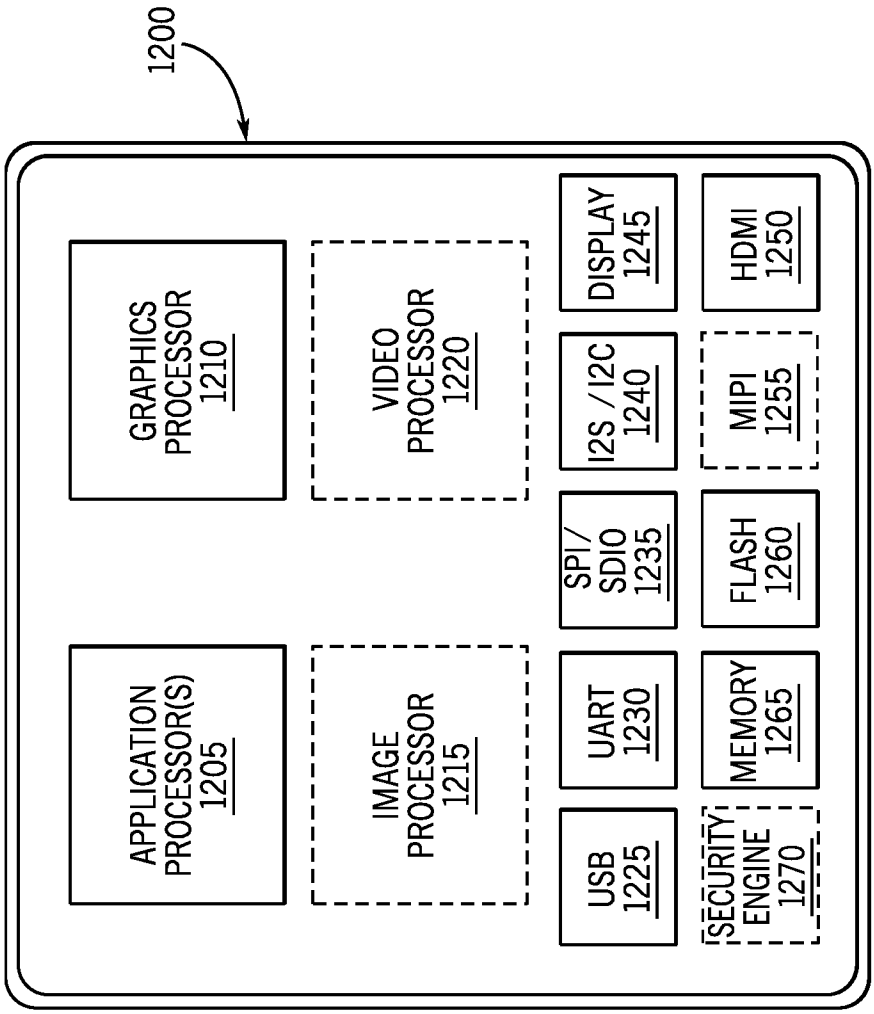

1200
APPLICATION PROCESSOR(S) 1205
GRAPHICS PROCESSOR 1210
IMAGE PROCESSOR 1215
VIDEO PROCESSOR 1220
USB 1225
UART 1230
SPI / SDIO 1235
I2S / I2C 1240
DISPLAY 1245
MEMORY 1265
FLASH 1260
MIPI 1255
HDMI 1250
SECURITY ENGINE 1270

FIG. 15

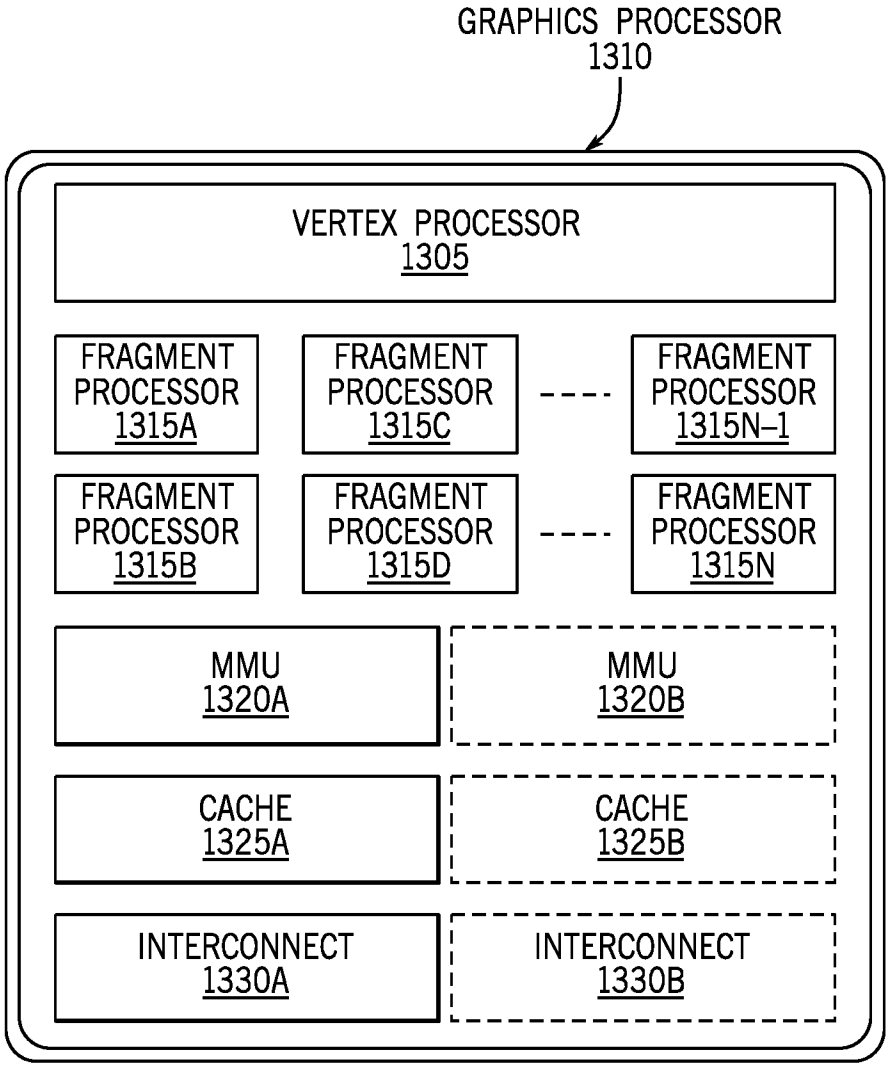

GRAPHICS PROCESSOR
1310
VERTEX PROCESSOR
1305
FRAGMENT PROCESSOR
1315A
FRAGMENT PROCESSOR
1315C
FRAGMENT PROCESSOR
1315N–1
FRAGMENT PROCESSOR
1315B
FRAGMENT PROCESSOR
1315D
FRAGMENT PROCESSOR
1315N
MMU
1320A
MMU
1320B
CACHE
1325A
CACHE
1325B
INTERCONNECT
1330A
INTERCONNECT
1330B

FIG. 16

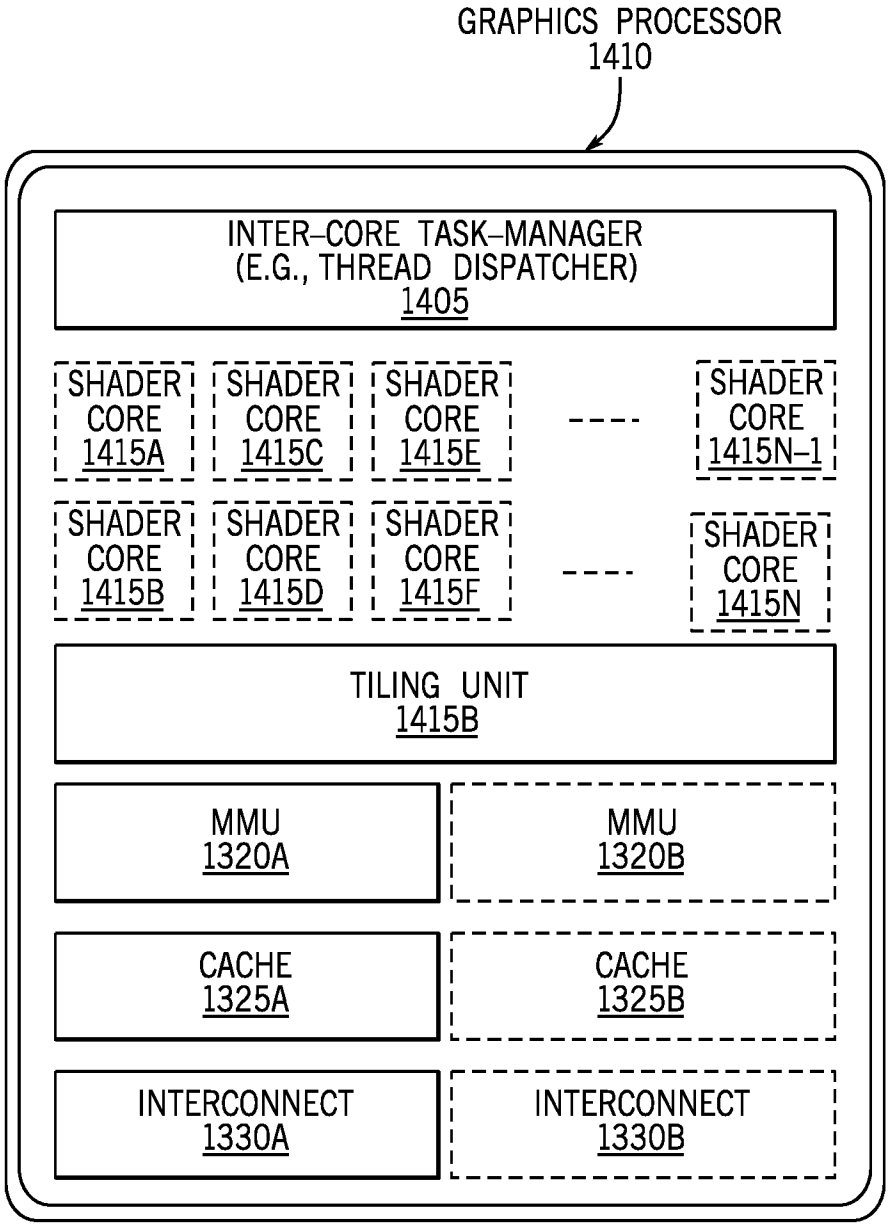

GRAPHICS PROCESSOR
1410
INTER-CORE TASK-MANAGER
(E.G., THREAD DISPATCHER)
1405
SHADER CORE 1415A
SHADER CORE 1415C
SHADER CORE 1415E
SHADER CORE 1415N-1
SHADER CORE 1415B
SHADER CORE 1415D
SHADER CORE 1415F
SHADER CORE 1415N
TILING UNIT
1415B
MMU
1320A
MMU
1320B
CACHE
1325A
CACHE
1325B
INTERCONNECT
1330A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

FIG. 17

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/031843

A. CLASSIFICATION OF SUBJECT MATTER

G06T 15/06(2011.01)i, G06T 15/08(2011.01)i, G06T 1/20(2006.01)i, G06T 1/60(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 15/06; G06T 17/00; G06K 9/00; G09G 5/00; G06T 15/08; G06T 1/20; G06T 1/60

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: ray, sample, volume rendering, combine

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2013-0249913 A1 (ANDREW MICHAEL CHRISTIAN SMOUT et al.) 26 September 2013 See paragraphs [0016]-[0019], [0061]-[0087]; and figures 1-2, 8-11.	1-24
Y	US 2014-0146049 A1 (CAUSTIC GRAPHICS, INC.) 29 May 2014 See paragraphs [0020]-[0041]; and figures 1-3.	1-24
A	US 2006-0147106 A1 (LINING YANG et al.) 06 July 2006 See paragraphs [0026]-[0027]; and figures 1, 4.	1-24
A	US 2013-0135306 A1 (KLAUS ENGEL) 30 May 2013 See paragraphs [0029]-[0031], [0046]; and figures 1, 6.	1-24
A	US 2013-0002673 A1 (DAQING XUE et al.) 03 January 2013 See paragraphs [0048]-[0055]; and figure 4.	1-24

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

☒ See patent family annex.

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

Date of the actual completion of the international search	Date of mailing of the international search report
16 August 2017 (16.08.2017)	**16 August 2017 (16.08.2017)**

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

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

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2017/031843

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2013-0249913 A1	26/09/2013	CN 103325139 A	25/09/2013
		JP 2013-192940 A	30/09/2013
		US 9384592 B2	05/07/2016
US 2014-0146049 A1	29/05/2014	US 8970591 B2	03/03/2015
US 2006-0147106 A1	06/07/2006	US 7317825 B2	08/01/2008
US 2013-0135306 A1	30/05/2013	CN 102855656 A	02/01/2013
		DE 102011076878 A1	06/12/2012
US 2013-0002673 A1	03/01/2013	US 2008-0031402 A1	07/02/2008
		US 8294706 B2	23/10/2012
		US 8928656 B2	06/01/2015