

METHOD AND APPARATUS FOR UPDATING A
SHADER PROGRAM BASED ON CURRENT STATE

BACKGROUND

5 Field of the Invention

[0001] This invention relates generally to the field of computer processors. More particularly, the invention relates to an apparatus and method for updating a shader program based on current state.

Description of the Related Art

10 [0002] In graphic application programming interfaces (API's) such as OpenGL or DirectX there may be a distinguished state associated with fixed function units and shading programs (called "shaders"). In particular, shaders are executed on a graphics processing unit (GPU) in an environment defined by this state. As graphic API's describe abstract machine models, there is always some form of translation of this model state to the current hardware
15 architecture. Thus, states may describe the functionality of fixed function units which are typically emulated in existing hardware by the execution of the shader programs. In some implementations, for example, a plurality of execution units (EUs) may execute the shader programs in a particular state to emulate the graphics functions of the fixed function units.

[0003] In most cases such emulation is required because of hardware design, and is done
20 through instructions added or modified in shader programs provided by application. Such design leads to situations where the shader program can be dependent on the current state, and a change of such state requires shader code to be recompiled. Such state may be referred to as "Non-Orthogonal State" ("NOS").

[0004] Because the shader program is dependent on the NOS, in some implementations,
25 the shader may be recompiled during image frame rendering to reflect the current NOS. As the compilation process is complex and time consuming, this may lead to stalls in the rendering process and jaggling of the animation. To prevent such situations, shaders may be compiled into different versions for the most popular NOS settings, and then cached. This reduces the amount of shader recompilations but does not eliminate them completely as there are too many NOS
30 permutations. It also introduces cost associated with the additional memory required to store the additional copies of compiled shaders, and the additional power consumption required for compilation of these copies.

BRIEF DESCRIPTION OF THE DRAWINGS

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

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

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

5 [0008] FIG. 3 is a block diagram of one embodiment of a graphics processor which may be a discrete graphics processing unit, or may be graphics processor integrated with a plurality of processing cores;

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

10 [0010] FIG. 5 is a block diagram of another embodiment of a graphics processor;

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

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

15 [0013] FIG. 8 is a block diagram of another embodiment of a graphics processor which includes a graphics pipeline, a media pipeline, a display engine, thread execution logic, and a render output pipeline;

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

20 [0015] FIG. 9B is a block diagram illustrating a graphics processor command sequence according to an embodiment;

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

[0017] FIG. 11 illustrates one embodiment of an architecture for updating a shader
25 program based on a current state;

[0018] FIG. 12 illustrates additional details for an architecture for updating a shader program based on current state; and

[0019] FIG. 13 illustrates one embodiment of a method for updating a shader program based on a current state.

30

DETAILED DESCRIPTION

[0020] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the embodiments of the invention described below. It will be apparent, however, to one skilled in the art that the

35 embodiments of the invention may be practiced without some of these specific details. In other

instances, well-known structures and devices are shown in block diagram form to avoid obscuring the underlying principles of the embodiments of the invention.

EXEMPLARY GRAPHICS PROCESSOR ARCHITECTURES AND DATA TYPES

Overview – Figures 1-3

5 [0021] **Figure 1** is a block diagram of a data processing system 100, according to an embodiment. The data processing 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 data processing system 100 is a system on a chip integrated circuit (SOC) for use in mobile, handheld, or embedded devices.

[0022] An embodiment of the data processing 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 one embodiment, the data processing system 100 is a mobile phone, smart phone, tablet computing device or mobile Internet device. The 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 one embodiment, the 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.

20 [0023] 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 one embodiment, each of the one or more processor cores 107 is configured to process a specific instruction set 109. The 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. A processor core 107 may also include other processing devices, such as a digital signal processor (DSP).

[0024] In one embodiment, 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 one embodiment, the cache memory is shared among various components of the processor 102. In one embodiment, 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 the processor cores 107 using known cache coherency techniques. A register file 106 is additionally included in the 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.

[0025] The processor 102 is coupled to a processor bus 110 to transmit data signals between the processor 102 and other components in the system 100. The system 100 uses an exemplary 'hub' system architecture, including a memory controller hub 116 and an input output (I/O) controller hub 130. The memory controller hub 116 facilitates communication between a memory device and other components of the system 100, while the I/O controller hub (ICH) 130 provides connections to I/O devices via a local I/O bus.

[0026] The memory device 120, can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, or some other memory device having suitable performance to serve as process memory. The memory 120 can store data 122 and instructions 121 for use when the processor 102 executes a process. The memory controller hub 116 also couples with an optional external graphics processor 112, which may communicate with the one or more graphics processor 108 in the processor 102 to perform graphics and media operations.

[0027] The ICH 130 enables peripherals to connect to the memory 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include 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 for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 142 connect input devices, such as keyboard and mouse 144 combinations. A network controller 134 may also couple to the ICH 130. In one embodiment, a high-performance network controller (not shown) couples to the processor bus 110.

[0028] **Figure 2** is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-N, an integrated memory controller 214, and an integrated graphics processor 208. The processor 200 can include additional cores up to and including additional core 202N represented by the dashed lined boxes. Each of the cores 202A-N includes one or more internal cache units 204A-N. In one embodiment each core also has access to one or more shared cached units 206.

[0029] The internal cache units 204A-N 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 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 last level cache (LLC). In one embodiment, cache coherency logic maintains coherency between the various cache units 206

and 204A-N.

[0030] The processor 200 may also include a set of one or more bus controller units 216 and a system agent 210. The one or more bus controller units manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). The system agent 210 provides management functionality for the various processor components. In one embodiment, the system agent 210 includes one or more integrated memory controllers 214 to manage access to various external memory devices (not shown).

[0031] In one embodiment, one or more of the cores 202A-N include support for simultaneous multi-threading. In such embodiment, the system agent 210 includes components for coordinating and operating cores 202A-N during multi-threaded processing. The system agent 210 may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of the cores 202A-N and the graphics processor 208.

[0032] The processor 200 additionally includes a graphics processor 208 to execute graphics processing operations. In one embodiment, the graphics processor 208 couples with the set of shared cache units 206, and the system agent unit 210, including the one or more integrated memory controllers 214. In one embodiment, a display controller 211 is coupled with the graphics processor 208 to drive graphics processor output to one or more coupled displays. The display controller 211 may be separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor 208 or system agent 210.

[0033] In one embodiment 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 one embodiment, the graphics processor 208 couples with the ring interconnect 212 via an I/O link 213.

[0034] 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 one embodiment each of the cores 202A-N and the graphics processor 208 use the embedded memory modules 218 as shared last level cache.

[0035] In one embodiment cores 202A-N are homogenous cores executing the same instruction set architecture. In another embodiment, the cores 202A-N are heterogeneous in terms of instruction set architecture (ISA), where one or more of the cores 202A-N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set.

[0036] The processor 200 can be a part of or implemented on one or more substrates using

any of a number of process technologies, for example, Complementary metal-oxide-semiconductor (CMOS), Bipolar Junction/Complementary metal-oxide-semiconductor (BiCMOS) or N-type metal-oxide-semiconductor logic (NMOS). Additionally, the processor 200 can be implemented on one or more chips or as a system on a chip (SOC) integrated circuit having the illustrated components, in addition to other components.

[0037] Figure 3 is a block diagram of one embodiment of a graphics processor 300 which may be a discrete graphics processing unit, or may be graphics processor integrated with a plurality of processing cores. In one embodiment, the graphics processor is communicated with via a memory mapped I/O interface to registers on the graphics processor and via commands placed into the processor memory. The graphics processor 300 includes a memory interface 314 to access memory. The 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.

[0038] The graphics processor 300 also includes a display controller 302 to drive display output data to a display device 320. The 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 one embodiment the 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.

[0039] In one embodiment, the 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 the graphics-processing engine (GPE) 310. The graphics-processing engine 310 is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

[0040] The 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 the 3D pipeline 312 can be used to perform media operations, an embodiment of the GPE 310 also includes a media pipeline 316 that is specifically used to perform media operations, such as video post processing and image enhancement.

[0041] In one embodiment, the 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 the video codec engine 306. In one embodiment, the media pipeline 316 additionally includes a thread spawning unit to spawn threads for execution on the 3D/Media sub-system 315. The spawned threads perform computations for the media operations on one or more graphics execution units included in the 3D/Media sub-system.

[0042] The 3D/Media subsystem 315 includes logic for executing threads spawned by the 3D pipeline 312 and media pipeline 316. In one embodiment, the pipelines send thread execution requests to the 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 one embodiment, the 3D/Media subsystem 315 includes one or more internal caches for thread instructions and data. In one embodiment, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

3D/Media Processing - Figure 4

[0043] Figure 4 is a block diagram of an embodiment of a graphics processing engine 410 for a graphics processor. In one embodiment, the graphics processing engine (GPE) 410 is a version of the GPE 310 shown in Figure 3. The GPE 410 includes a 3D pipeline 412 and a media pipeline 416, each of which can be either different from or similar to the implementations of the 3D pipeline 312 and the media pipeline 316 of Figure 3.

[0044] In one embodiment, the GPE 410 couples with a command streamer 403, which provides a command stream to the GPE 3D and media pipelines 412, 416. The command streamer 403 is coupled to memory, which can be system memory, or one or more of internal cache memory and shared cache memory. The command streamer 403 receives commands from the memory and sends the commands to the 3D pipeline 412 and/or media pipeline 416. The 3D and media pipelines process the commands by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to the execution unit array 414. In one embodiment, the execution unit array 414 is scalable, such that the array includes a variable number of execution units based on the target power and performance level of the GPE 410.

[0045] A sampling engine 430 couples with memory (e.g., cache memory or system memory) and the execution unit array 414. In one embodiment, the sampling engine 430 provides a memory access mechanism for the scalable execution unit array 414 that allows the execution array 414 to read graphics and media data from memory. In one embodiment, the

sampling engine 430 includes logic to perform specialized image sampling operations for media.

[0046] The specialized media sampling logic in the sampling engine 430 includes a de-noise / de-interlace module 432, a motion estimation module 434, and an image scaling and filtering module 436. The de-noise / de-interlace module 432 includes logic to perform one or more of a de-noise or a de-interlace algorithm on decoded video data. The de-interlace logic combines alternating fields of interlaced video content into a single frame of video. The de-noise logic reduces or remove data noise from video and image data. In one embodiment, the de-noise logic and de-interlace logic are motion adaptive and use spatial or temporal filtering based on the amount of motion detected in the video data. In one embodiment, the de-noise / de-interlace module 432 includes dedicated motion detection logic (e.g., within the motion estimation engine 434).

[0047] The motion estimation engine 434 provides hardware acceleration for video operations by performing video acceleration functions such as motion vector estimation and prediction on video data. The motion estimation engine determines motion vectors that describe the transformation of image data between successive video frames. In one embodiment, a graphics processor media codec uses the video motion estimation engine 434 to perform operations on video at the macro-block level that may otherwise be computationally intensive to perform using a general-purpose processor. In one embodiment, the motion estimation engine 434 is generally available to graphics processor components to assist with video decode and processing functions that are sensitive or adaptive to the direction or magnitude of the motion within video data.

[0048] The image scaling and filtering module 436 performs image-processing operations to enhance the visual quality of generated images and video. In one embodiment, the scaling and filtering module 436 processes image and video data during the sampling operation before providing the data to the execution unit array 414.

[0049] In one embodiment, the graphics processing engine 410 includes a data port 444, which provides an additional mechanism for graphics subsystems to access memory. The data port 444 facilitates memory access for operations including render target writes, constant buffer reads, scratch memory space reads/writes, and media surface accesses. In one embodiment, the data port 444 includes cache memory space to cache accesses to memory. The cache memory can be a single data cache or separated into multiple caches for the multiple subsystems that access memory via the data port (e.g., a render buffer cache, a constant buffer cache, etc.). In one embodiment, threads executing on an execution unit in the execution unit array 414 communicate with the data port by exchanging messages via a data distribution interconnect that couples each of the sub-systems of the graphics processing engine 410.

Execution Units – Figures 5-7

[0050] **Figure 5** is a block diagram of another embodiment of a graphics processor. In one embodiment, the graphics processor includes a ring interconnect 502, a pipeline front-end 504, a media engine 537, and graphics cores 580A-N. The ring interconnect 502 couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In one embodiment, the graphics processor is one of many processors integrated within a multi-core processing system.

[0051] The graphics processor receives batches of commands via the ring interconnect 502. The incoming commands are interpreted by a command streamer 503 in the pipeline front-end 504. The graphics processor includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) 580A-N. For 3D geometry processing commands, the command streamer 503 supplies the commands to the geometry pipeline 536. For at least some media processing commands, the command streamer 503 supplies the commands to a video front end 534, which couples with a media engine 537. The 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. The geometry pipeline 536 and media engine 537 each generate execution threads for the thread execution resources provided by at least one graphics core 580A.

[0052] The graphics processor includes scalable thread execution resources featuring modular cores 580A-N (sometime referred to as core slices), each having multiple sub-cores 550A-N, 560A-N (sometimes referred to as core sub-slices). The graphics processor can have any number of graphics cores 580A through 580N. In one embodiment, the graphics processor includes a graphics core 580A having at least a first sub-core 550A and a second core sub-core 560A. In another embodiment, the graphics processor is a low power processor with a single sub-core (e.g., 550A). In one embodiment, the graphics processor includes multiple graphics cores 580A-N, each including a set of first sub-cores 550A-N and a set of second sub-cores 560A-N. Each sub-core in the set of first sub-cores 550A-N includes at least a first set of execution units 552A-N and media/texture samplers 554A-N. Each sub-core in the set of second sub-cores 560A-N includes at least a second set of execution units 562A-N and samplers 564A-N. In one embodiment, each sub-core 550A-N, 560A-N shares a set of shared resources 570A-N. In one embodiment, 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.

[0053] **Figure 6** illustrates thread execution logic 600 including an array of processing elements employed in one embodiment of a graphics processing engine. In one embodiment, the

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

[0054] In one embodiment, the execution unit array 608A-N is primarily used to execute “shader” programs. In one embodiment, the execution units in the array 608A-N execute an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed
15 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).

[0055] Each execution unit in the execution unit array 608A-N operates on arrays of data elements. The number of data elements is the “execution size,” or the number of channels for
20 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 ALUs or FPUs for a particular graphics processor. The execution units 608A-N support integer and floating-point data types.

[0056] The execution unit instruction set includes single instruction multiple data (SIMD)
25 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
30 (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.

[0057] 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 one embodiment, one
35 or more data caches (e.g., 612) are included to cache thread data during thread execution. A

sampler 610 is included to provide texture sampling for 3D operations and media sampling for media operations. In one embodiment, the 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.

[0058] During execution, the graphics and media pipelines send thread initiation requests to the thread execution logic 600 via thread spawning and dispatch logic. The thread execution logic 600 includes a local thread dispatcher 604 that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units 608A-N. For example, the geometry pipeline (e.g., 536 of Figure 5) dispatches vertex processing, tessellation, or geometry processing threads to the thread execution logic 600. The thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

[0059] Once a group of geometric objects have been processed and rasterized into pixel data, the pixel shader 602 is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In one embodiment, the pixel shader 602 calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. The pixel shader 602 then executes an API-supplied pixel shader program. To execute the pixel shader program, the pixel shader 602 dispatches threads to an execution unit (e.g., 608A) via the thread dispatcher 604. The 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.

[0060] In one embodiment, 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 one embodiment, 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.

[0061] Figure 7 is a block diagram illustrating a graphics processor execution unit instruction format according to an embodiment. In one 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. The instruction format 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.

[0062] In one embodiment, the graphics processor execution units natively support

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

[0063] For each format, an 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. An instruction control field 712 enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For 128-bit instructions 710 an exec-size field 716 limits the number of data channels that will be executed in parallel. The exec-size field 716 is not available for use in the 64-bit compact instruction format 730.

[0064] Some execution unit instructions have up to three operands including two source operands, src0 722, src1 722, and one destination 718. In one embodiment, 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 JJ12 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.

[0065] In one embodiment instructions are grouped based on opcode 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 exemplary. In one embodiment, a move and logic opcode group 742 includes data movement and logic instructions (e.g., mov, cmp). The move and logic group 742 shares the five most significant bits (MSB), where move instructions are in the form of 0000xxxxb (e.g., 0x0x) and logic instructions are in the form of 0001xxxxb (e.g., 0x01). A flow control instruction group 744 (e.g., call, 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, mul) in the form of 0100xxxxb (e.g., 0x40). The parallel math

group 748 performs the arithmetic operations in parallel across data channels. The vector math group 750 includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

Graphics Pipeline – Figure 8

5 [0066] **Figure 8** is a block diagram of another embodiment of a graphics processor which includes a graphics pipeline 820, a media pipeline 830, a display engine 840, thread execution logic 850, and a render output pipeline 870. In one embodiment, the graphics processor 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 the graphics processor via a ring interconnect 802. The ring interconnect 802 couples the graphics processor to other processing components, such as other graphics processors or general-purpose processors. Commands from the ring interconnect are interpreted by a command streamer 803 which supplies instructions to individual components of the graphics pipeline 820 or media pipeline 830.

15 [0067] The command streamer 803 directs the operation of a vertex fetcher 805 component that reads vertex data from memory and executes vertex-processing commands provided by the command streamer 803. The vertex fetcher 805 provides vertex data to a vertex shader 807, which performs coordinate space transformation and lighting operations to each vertex. The vertex fetcher 805 and vertex shader 807 execute vertex-processing instructions by dispatching execution threads to the execution units 852A, 852B via a thread dispatcher 831.

20 [0068] In one embodiment, the execution units 852A, 852B are an array of vector processors having an instruction set for performing graphics and media operations. The 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.

25 [0069] In one embodiment, the graphics pipeline 820 includes tessellation components to perform hardware-accelerated tessellation of 3D objects. 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 the 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 the graphics pipeline 820. If tessellation is not used, the tessellation components 811, 813, 817 can be bypassed.

30 [0070] The complete geometric objects can be processed by a geometry shader 819 via one or more threads dispatched to the execution units 852A, 852B, or can proceed directly to the clipper 829. The geometry shader operates on entire geometric objects, rather than vertices or

35

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. The geometry shader 819 is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

5 [0071] Prior to rasterization, vertex data is processed by a clipper 829, which is either a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In one embodiment, a rasterizer 873 in the render output pipeline 870 dispatches pixel shaders to convert the geometric objects into their per pixel representations. In one embodiment, pixel shader logic is included in the thread execution logic 850.

10 [0072] The graphics engine has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the graphics engine. In one embodiment the 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
15 graphics engine. In one embodiment, the sampler 854, caches 851, 858 and execution units 852A, 852B each have separate memory access paths.

[0073] In one embodiment, the render output pipeline 870 contains a rasterizer and depth test component 873 that converts vertex-based objects into their associated pixel-based representation. In one embodiment, the rasterizer logic includes a windower/masker unit to
20 perform fixed function triangle and line rasterization. An associated render and depth buffer caches 878, 879 are also available in one embodiment. 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
25 planes. In one embodiment a shared L3 cache 875 is available to all graphics components, allowing the sharing of data without the use of main system memory.

[0074] The graphics processor media pipeline 830 includes a media engine 337 and a video front end 834. In one embodiment, the video front end 834 receives pipeline commands from the command streamer 803. However, in one embodiment the media pipeline 830 includes a
30 separate command streamer. The video front-end 834 processes media commands before sending the command to the media engine 837. In one embodiment, the media engine includes thread spawning functionality to spawn threads for dispatch to the thread execution logic 850 via the thread dispatcher 831.

[0075] In one embodiment, the graphics engine includes a display engine 840. In one
35 embodiment, the display engine 840 is external to the graphics processor and couples with the

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

[0076] The 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 one embodiment, driver software for the
10 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 various embodiments, support is provided for the Open Graphics Library (OpenGL) and Open Computing Language (OpenCL) supported by the Khronos Group, the Direct3D library from the Microsoft Corporation, or, in one embodiment, both OpenGL and D3D. Support may also be provided for
15 the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

Graphics Pipeline Programming – Figure 9A-B

[0077] **Figure 9A** is a block diagram illustrating a graphics processor command format according to an embodiment and **Figure 9B** is a block diagram illustrating a graphics processor command sequence according to an embodiment. The solid lined boxes in Figure 9A illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format 900 of Figure 9A includes data
25 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.

[0078] The client 902 specifies the client unit of the graphics device that processes the command data. In one embodiment, a graphics processor command parser examines the client
30 field of each command to condition the further processing of the command and route the command data to the appropriate client unit. In one embodiment, 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-
35 opcode 905 to determine the operation to perform. The client unit performs the command using

information in the data 906 field of the command. For some commands an explicit command size 908 is expected to specify the size of the command. In one embodiment, the command parser automatically determines the size of at least some of the commands based on the command opcode. In one embodiment commands are aligned via multiples of a double word.

5 [0079] The flow chart in Figure 9B shows a sample command sequence 910. In one embodiment, software or firmware of a data processing system that features an embodiment of the 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 exemplary purposes, however embodiments are not limited to these commands or to this
10 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 an at least partially concurrent manner.

[0080] The sample 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.
15 In one embodiment, 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
20 'dirty' can be flushed to memory. A pipeline flush command 912 can be used for pipeline synchronization or before placing the graphics processor into a low power state.

[0081] A pipeline select command 913 is used when a command sequence requires the graphics processor to explicitly switch between pipelines. A pipeline select command 913 is required only once within an execution context before issuing pipeline commands unless the
25 context is to issue commands for both pipelines. In one embodiment, a pipeline flush command is 912 is required immediately before a pipeline switch via the pipeline select command 913.

[0082] 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. The pipeline control command 914 configures the pipeline state for the active pipeline. In one embodiment, the pipeline control
30 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.

[0083] 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
35 during processing. The graphics processor also uses one or more return buffers to store output

data and to perform cross thread communication. The return buffer state 916 includes selecting the size and number of return buffers to use for a set of pipeline operations.

[0084] 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.

[0085] The commands for the 3D pipeline state 930 include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based the particular 3D API in use. 3D pipeline state 930 commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

[0086] The 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. The 3D primitive 932 command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, the 3D pipeline 922 dispatches shader execution threads to graphics processor execution units.

[0087] The 3D pipeline 922 is triggered via an execute 934 command or event. In one embodiment a register write triggers command execution. In one embodiment 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.

[0088] The sample 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. 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.

[0089] The media pipeline 924 is configured in a similar manner as the 3D pipeline 922. A set of media pipeline state commands 940 are dispatched or placed into in a command queue before the media object commands 942. The 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. The media pipeline state commands 940 also support the use one or more pointers to “indirect” state elements that contain a batch of state settings.

[0090] 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 one embodiment, all media pipeline state 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 934 command or an equivalent execute event (e.g., register write). Output from the media pipeline 924 may then be post processed by operations provided by the 3D pipeline 922 or the media pipeline 924. In one embodiment, GPGPU operations are configured and executed in a similar manner as media operations.

Graphics Software Architecture – Figure 10

[0091] Figure 10 illustrates exemplary graphics software architecture for a data processing system according to an embodiment. The software architecture includes a 3D graphics application 1010, an operating system 1020, and at least one processor 1030. The 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.

[0092] In one embodiment, the 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.

[0093] The operating system 1020 may be a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. When the Direct3D API is in use, the

operating system 1020 uses a front-end shader compiler 1024 to compile any shader instructions 1012 in HLSL into a lower-level shader language. The compilation may be a just-in-time compilation or the application can perform share pre-compilation. In one embodiment, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application 1010.

[0094] The user mode graphics driver 1026 may contain 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. The user mode graphics driver uses operating system kernel mode functions 1028 to communicate with a kernel mode graphics driver 1029. The kernel mode graphics driver 1029 communicates with the graphics processor 1032 to dispatch commands and instructions.

[0095] To the extent various operations or functions are described herein, they can be described or defined as hardware circuitry, software code, instructions, configuration, and/or data. The content can be embodied in hardware logic, or as directly executable software ("object" or "executable" form), source code, high level shader code designed for execution on a graphics engine, or low level assembly language code in an instruction set for a specific processor or graphics core. The software content of the embodiments described herein can be provided via an article of manufacture with the content stored thereon, or via a method of operating a communication interface to send data via the communication interface.

[0096] A non-transitory machine readable storage medium can cause a machine to perform the functions or operations described, and includes any mechanism that stores information in a form accessible by a machine (e.g., computing device, electronic system, etc.), such as recordable/non-recordable media (e.g., read only memory (ROM), random access memory (RAM), magnetic disk storage media, optical storage media, flash memory devices, etc.). A communication interface includes any mechanism that interfaces to any of a hardwired, wireless, optical, etc., medium to communicate to another device, such as a memory bus interface, a processor bus interface, an Internet connection, a disk controller, etc. The communication interface is configured by providing configuration parameters or sending signals to prepare the communication interface to provide a data signal describing the software content. The communication interface can be accessed via one or more commands or signals sent to the communication interface.

[0097] Various components described can be a means for performing the operations or functions described. Each component described herein includes software, hardware, or a combination of these. The components can be implemented as software modules, hardware

modules, special-purpose hardware (e.g., application specific hardware, application specific integrated circuits (ASICs), digital signal processors (DSPs), etc.), embedded controllers, hardwired circuitry, etc. Besides what is described herein, various modifications can be made to the disclosed embodiments and implementations of the invention without departing from their scope. Therefore, the illustrations and examples herein should be construed in an illustrative, and not a restrictive sense. The scope of the invention should be measured solely by reference to the claims that follow.

APPARATUS AND METHOD FOR UPDATING A SHADER PROGRAM BASED ON CURRENT STATE

[0098] The compilation process is complex, and requires several intermediate language representations to describe shader program code at different steps of compilation. Before generating a final shader program kernel, the most time consuming operations are performed on the abstract representation of program instructions called the Low Level Intermediate Representation (LLIR).

[0099] To reduce the negative impact of these time-consuming re-compilation operations during runtime, one embodiment of the invention compiles the shader program up to the end phase but marks the final instructions if they interact with any non-orthogonal state (NOS). The compiled shader program is then saved as a patchable shader kernel. During execution, the marked instructions may then be dynamically patched for the current NOS to generate a patched shader kernel which is executed on the GPU hardware.

[00100] As illustrated generally in **Figure 11**, in one embodiment of the invention, shader source code 1124 is compiled by a compiler with instruction marking logic 1123 to generate marked shader instructions 1104 for instructions which are dependent on different NOSs and to generate unmarked shader instructions 1125 for instructions which are not dependent on NOSs. In one embodiment, the instructions are marked using a sequence of bits which would not otherwise be used for the given instruction set architecture. Each bit in the sequence may indicate whether the instruction is dependent on a particular NOS. For example, bit 0 may indicate whether the instruction is dependent on NOS # 0 (e.g., 1 = dependent on NOS, 0 = not dependent on NOS); bit 1 may indicate whether the instruction is dependent on NOS # 1; bit 2 may indicate whether the instruction is dependent on NOS # 2, etc. It should be noted, however, that the underlying principles of the invention are not limited to any particular encoding for identifying dependence on a NOS.

[00101] At execution time, the marked shader instructions 1104 are patched according to the current NOS. In **Figure 11**, for example, instruction patch logic 1103 dynamically patches

the marked shader instructions 1104 in accordance with the current NOS 1101 to generate patched shader instructions 1105. The patched shader instructions 1105 and unmarked shader instructions 1125 may be saved together in a patched shader kernel (which is specifically adapted for the current NOS because of the patched shader instructions).

5 [00102] As illustrated in **Figure 11**, a scheduler/dispatch unit 1110 may then dispatch the patched shader instructions 1105 and unmarked shader instructions 1125 for execution on one or more execution units 1111-1118 to implement the shader. The underlying principles of the invention may be implemented for any type of shader including those discussed above. This may include, for example, the pixel shader 602 shown in **Figure 6**, and the vertex shader 807,
10 geometry shader 819, hull shader 811, and domain shader 817 shown in **Figure 8** (to name just a few). In addition, in one embodiment, the back-end shader compiler 1027 shown in **Figure 10** (implemented within user mode graphics driver 1026) may be used to generate marked shader instructions and patched shader instructions as described herein.

[00103] Additional details for one embodiment of the invention are illustrated in **Figure**
15 **12** which includes an application component 1120 which uses a graphics API (e.g., Direct3D, OpenGL, etc); a driver component 1130 for marking instructions and patching instructions according to NOSs as described herein; and a hardware component 1140 for executing the patched instructions. An API compile function 1201 called by the application 1220 compiles a shading language code such as a high level shader language (HLSL) code or OpenGL shading
20 language (GLSL) code to a lower-level shader language. In particular, in response to the compile function 1201, the HLSL or GLSL code is input to a compiler block 1203 within the driver component 1230 which performs a variety of compilation operations including converting the code to a low level intermediate representation (LLIR), performing LLIR optimizations, and generating register allocation information. In addition, the compiler block 1203 evaluates the
25 instructions to identify NOS dependent instructions, and then marks the NOS dependent instructions (identified in **Figure 12** as “X_NOS” instructions). The marked instructions and unmarked instructions are then combined into a patchable shader kernel 1204 to be used at execution time.

[00104] At execution time, in response to a set state function 1205 and set shader function
30 1209 called by the application 1220, a process state module 1206 of the driver 1130 determines the current NOS 1207 in which the patchable kernel implementing the shader 1204 will execute. In particular, the shader function 1209 points to the patchable shader kernel 1204 which will be used and, as indicated by the dotted arrow in **Figure 12**, the process state 1206 may be dependent on the shader kernel 1204.

[00105] An instruction patch module 1208 of the driver identifies those marked instructions within the patchable shader kernel 1204 which are dependent on the current NOS 1207. The instruction patch module 1208 then patches the instructions for execution according to the current NOS 1207 (e.g., in response to an execute function 1211, indicating that the kernel is to be executed). The patched instructions are used (along with un-patched instructions) to generate the patched shader kernel 1213 which is executed on the execution resources 1214 of the hardware component 1240 (e.g., which may be implemented as a plurality of execution units as previously described). The hardware unit state 1212 is also updated to reflect the current process state 1206 (e.g., the current NOS for which the instructions of the patched kernel 1213 have been patched).

[00106] A method in accordance with one embodiment of the invention is illustrated in **Figure 13**. The method may be implemented within the context of the system architectures described above (e.g., as shown in **Figures 11-12**), but is not limited to any particular system architecture.

[00107] At 1301, instructions which are dependent on NOS are identified and, at 1302, the incoming shader is compiled to a patchable binary kernel in which the NOS-dependent instructions are marked to indicate dependencies with different NOSs. For example, as mentioned above, a bit field of each instruction may be used to indicate those NOSs on which the instruction is dependent (e.g., with each bit position of the bit field indicating a different NOS). At 1303, the optimized patchable kernel comprising the NOS-dependent instructions is saved.

[00108] At 1304, the current NOS is determined during shader execution (e.g., in response to the execution of the shader). At 1305, each specially-marked instruction (marked in operation 1302) is modified to implement the current NOS specification (i.e., based on the detected current NOS). The resulting marked instructions are included in the patched shader kernel. At 1306, the patched shader kernel is executed in accordance with the current NOS.

[00109] In one embodiment, the techniques described above may be used only partially. For example, in one embodiment, NOSs that are in most cases configured in the same way can be directly compiled (rather than using the patching techniques described herein).

[00110] Using the techniques described herein removes the need for caching shader kernels with multiple instances of instructions (one for each NOS), as the kernels can be dynamically patched. Such approach reduces compilation time, performance and power usage, and thus improves the end user experience.

[00111] Embodiments of the invention may include various steps, which have been described above. The steps may be embodied in machine-executable instructions which may be

used to cause a general-purpose or special-purpose processor to perform the steps. Alternatively, these steps may be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

5 [00112] As described herein, instructions may refer to specific configurations of hardware such as application specific integrated circuits (ASICs) configured to perform certain operations or having a predetermined functionality or software instructions stored in memory embodied in a non-transitory computer readable medium. Thus, the techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices (e.g., an
10 end station, a network element, etc.). Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer machine-readable media, such as non-transitory computer machine-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer machine-readable communication media (e.g.,
15 electrical, optical, acoustical or other form of propagated signals – such as carrier waves, infrared signals, digital signals, etc.). In addition, such electronic devices typically include a set of one or more processors coupled to one or more other components, such as one or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors
20 and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage device of a given electronic device typically stores code and/or data for execution on the set of one or more processors of that electronic device. Of course, one or more
25 parts of an embodiment of the invention may be implemented using different combinations of software, firmware, and/or hardware. Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the invention may be practiced without some of these specific details. In certain instances, well
30 known structures and functions were not described in elaborate detail in order to avoid obscuring the subject matter of the present invention. Accordingly, the scope and spirit of the invention should be judged in terms of the claims which follow.

CLAIMS

What is claimed is:

1. A method comprising:

identifying a first plurality of instructions which are dependent on a non-orthogonal state

5 (NOS);

marking the each of the first plurality of instructions which are dependent on the NOS;

detecting a current NOS; and

dynamically patching the marked instructions for the current NOS.

10 2. The method as in claim 1 wherein the first plurality of instructions comprise a plurality of shader instructions for implementing a graphics shader.

3. The method as in claim 2 wherein the operations of detecting the current NOS and dynamically patching the marked instructions for the current NOS are performed at shader
15 execution time.

4. The method as in claim 1 wherein marking each of the first plurality of instructions comprises setting one or more bits within each of the plurality of instructions to indicate NOSs on which the instructions depend.

20

5. The method as in claim 2 further comprising:

compiling shader program code to a patchable kernel including the first plurality of instructions and a second plurality of instructions which are not dependent on a NOS.

25 6. The method as in claim 5 wherein the patchable kernel is updated during runtime by performing the operation of dynamically patching the marked instructions for the current NOS.

7. The method as in claim 1 wherein the shader is selected from a group consisting
30 of a pixel shader, a vertex shader, a geometry shader, a hull shader, and a domain shader.

8. The method as in claim 1 further comprising:

executing the patched instructions to implement a shader dependent on the current NOS.

35 9. An apparatus comprising:

a shader compiler with instruction marking logic to identify a first plurality of instructions which are dependent on a non-orthogonal state (NOS) and to mark each of the first plurality of instructions which are dependent on the NOS; and

5 instruction patch logic to detect a current NOS during execution and dynamically patch the marked instructions for the current NOS.

10. The apparatus as in claim 9 wherein the first plurality of instructions comprise a plurality of shader instructions for implementing a graphics shader.

10 11. The apparatus as in claim 10 wherein the operations of detecting the current NOS and dynamically patching the marked instructions for the current NOS are performed at a time at or after a shader execution function is called by an application.

12. The apparatus as in claim 9 wherein marking each of the first plurality of
15 instructions comprises setting one or more bits within each of the plurality of instructions to indicate NOSs on which the instructions depend.

13. The apparatus as in claim 10 wherein the shader compiler compiles shader
20 program code to a patchable kernel including the first plurality of instructions and a second plurality of instructions which are not dependent on a NOS.

14. The apparatus as in claim 13 wherein the patchable kernel is updated during
runtime by performing the operation of dynamically patching the marked instructions for the
current NOS.

25 15. The apparatus as in claim 9 wherein the shader is selected from a group consisting of a pixel shader, a vertex shader, a geometry shader, a hull shader, and a domain shader.

16. The apparatus as in claim 9 further comprising:
30 a plurality of execution units (EUs) to execute the patched instructions to implement a shader dependent on the current NOS.

17. A machine-readable medium having program code stored thereon which, when
executed by a machine, causes the machine to perform the operations of:

identifying a first plurality of instructions which are dependent on a non-orthogonal state (NOS);

marking the each of the first plurality of instructions which are dependent on the NOS;

detecting a current NOS; and

5 dynamically patching the marked instructions for the current NOS.

18. The machine-readable medium as in claim 17 wherein the first plurality of instructions comprise a plurality of shader instructions for implementing a graphics shader.

10 19. The machine-readable medium as in claim 18 wherein the operations of detecting the current NOS and dynamically patching the marked instructions for the current NOS are performed at a time at or after a shader execution function is called by an application.

15 20. The machine-readable medium as in claim 17 wherein marking each of the first plurality of instructions comprises setting one or more bits within each of the plurality of instructions to indicate NOSs on which the instructions depend.

21. The machine-readable medium as in claim 18 comprising additional program code to cause the operation of:

20 compiling shader program code to a patchable kernel including the first plurality of instructions and a second plurality of instructions which are not dependent on a NOS.

22. The machine-readable medium as in claim 21 wherein the patchable kernel is updated during runtime by performing the operation of dynamically patching the marked
25 instructions for the current NOS.

23. The machine-readable medium as in claim 17 wherein the shader is selected from a group consisting of a pixel shader, a vertex shader, a geometry shader, a hull shader, and a domain shader.

30 24. The machine-readable medium as in claim 17 comprising additional program code to cause the operation of:

executing the patched instructions to implement a shader dependent on the current NOS.