

PARALLEL ARRAY ARCHITECTURE FOR A GRAPHICS PROCESSOR

CROSS-REFERENCES TO RELATED APPLICATIONS

[0001] The present application claims the benefit of U.S. Provisional Application No.
5 60/752,265, filed December 19, 2005, which is incorporated herein by reference in its
entirety for all purposes.

[0002] The present application is related to the following commonly-assigned co-pending
U.S. Patent Applications: Application No. 11/290,303, filed November 29, 2005;
Application No. 11/289,828, filed November 29, 2005; and Application No. 11/311,993, filed
10 December 19, 2005, which are incorporated in their entirety, herein, by reference for all
purposes.

BACKGROUND OF THE INVENTION

[0003] The present invention relates in general to graphics processors, and in particular to a
15 parallel array architecture for a graphics processor.

[0004] Parallel processing techniques enhance throughput of a processor or multiprocessor
system when multiple independent computations need to be performed. A computation can
be divided into tasks that are defined by programs, with each task being performed as a
separate thread. (As used herein, a "thread" refers generally to an instance of execution of a
20 particular program using particular input data, and a "program" refers to a sequence of
executable instructions that produces result data from input data.) Parallel threads are
executed simultaneously using different processing engines inside the processor.

[0005] Numerous existing processor architectures support parallel processing. The earliest
such architectures used multiple discrete processors networked together. More recently,
25 multiple processing cores have been fabricated on a single chip. These cores are controlled in
various ways. In some instances, known as multiple-instruction, multiple data (MIMD)
machines, each core independently fetches and issues its own instructions to its own
processing engine (or engines). In other instances, known as single-instruction, multiple-data
(SIMD) machines, a core has a single instruction unit that issues the same instruction in
30 parallel to multiple processing engines, which execute the instruction on different input

operands. SIMD machines generally have advantages in chip area (since only one instruction unit is needed) and therefore cost; the downside is that parallelism is only available to the extent that multiple instances of the same instruction can be executed concurrently.

[0006] Conventional graphics processors use very wide SIMD architectures to achieve high throughput in image-rendering applications. Such applications generally entail executing the same programs (vertex shaders or pixel shaders) on large numbers of objects (vertices or pixels). Since each object is processed independently of all others but using the same sequence of operations, a SIMD architecture provides considerable performance enhancement at reasonable cost. Typically, a GPU includes one SIMD core that executes vertex shader programs, and another SIMD core of comparable size that executes pixel shader programs. In high-end GPUs, multiple sets of SIMD cores are sometimes provided to support an even higher degree of parallelism.

[0007] These designs have several shortcomings. First, the separate processing cores for vertex and shader programs are separately designed and tested, often leading to at least some duplication of effort. Second, the division of the graphics processing load between vertex operations and pixel operations varies greatly from one application to another. As is known in the art, detail can be added to an image by using many small primitives, which increases the load on the vertex shader core, and/or by using complex texture-mapping and pixel shading operations, which increases the load on the pixel shader core. In most cases, the loads are not perfectly balanced, and one core or the other is underused. For instance, in a pixel-intensive application, the pixel shader core may run at maximum throughput while the vertex core is idle, waiting for already-processed vertices to move into the pixel shader stage of the pipeline. Conversely, in a vertex-intensive application, the vertex shader core may run at maximum throughput while the pixel core is idle, waiting for new vertices to be supplied. In either case, some fraction of available processing cycles is effectively wasted.

[0008] It would therefore be desirable to provide a graphics processor that can adapt to varying loads on different shaders while maintaining a high degree of parallelism.

BRIEF SUMMARY OF THE INVENTION

[0009] Embodiments of the present invention provide graphics processors that use a scalable multithreaded core array to execute vertex shader programs, geometry shader programs, and/or pixel shader programs in any combination during rendering operations. The

core array includes a number of multithreaded processing cores arranged in one or more clusters, with cores in the same cluster being controlled by a shared core interface.

[0009a] In accordance with a first aspect of the present invention, there is provided a graphics processor comprising: a multithreaded core array including a plurality of processing clusters, each processing cluster including at least one processing core operable to execute a pixel shader program that generates pixel data from coverage data; a rasterizer configured to generate coverage data for each of a plurality of pixels; and pixel distribution logic configured to deliver the coverage data from the rasterizer to one of the processing clusters in the multithreaded core array, wherein the pixel distribution logic selects the one of the processing clusters to which the coverage data for a first pixel is delivered based at least in part on a location of the first pixel within an image area, and wherein each processing core is also operable to execute vertex shader programs and geometry shader programs.

[0009b] In accordance with a second aspect of the present invention, there is provided a graphics processor comprising: a multithreaded core array including a plurality of processing clusters, each processing cluster including at least one processing core operable to execute a pixel shader program that generates pixel data from coverage data; a rasterizer configured to generate coverage data for each of a plurality of pixels; a pixel distribution logic configured to deliver the coverage data from the rasterizer to one of the processing clusters in the multithreaded core array; and a crossbar coupled to each of the processing clusters and configured to deliver pixel data from the processing clusters to a frame buffer comprising multiple partitions; wherein the pixel distribution logic selects the one of the processing clusters to which the coverage data for a first pixel is delivered based at least in part on a location of the first pixel within an image area.

[0010] In embodiments where a pixel shader program is to be executed, the cluster or core in which the program is to be executed is advantageously selected based on the location of the pixel within the image area. In one embodiment, the screen is tiled, with each tile being assigned to one or another of the processing clusters (or to a specific core within a processing cluster). The tiles assigned to a given processing cluster or core are advantageously scattered across the screen to provide approximate load balancing.

[0011] In some embodiments, the processing core or cluster includes a raster operations unit that integrates newly generated pixel data with existing data in a frame buffer. The

frame buffer can be partitioned to match the number of processing clusters, with each cluster writing all of its data to one partition. In other embodiments, the number of partitions of the frame buffer need not match the number of processing clusters in use. A crossbar or similar circuit structure may provide a configurable coupling between the processing clusters and the frame buffer partitions, so that any processing cluster can be coupled to any frame buffer partitions; in some embodiments, the crossbar is omitted, improving memory locality.

[0012] The following detailed description together with the accompanying drawings will provide a better understanding of the nature and advantages of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] FIG. 1 is a block diagram of a computer system according to an embodiment of the present invention;

[0014] FIG. 2 is a block diagram of a rendering pipeline that can be implemented in a graphics processor according to an embodiment of the present invention;

[0015] FIG. 3 is a block diagram of a multithreaded core array for a graphics processor according to an embodiment of the present invention;

[0016] FIG. 4 illustrates one possible tiling of an image area into a number of tiles according to an embodiment of the present invention;

[0017] FIG. 5 is a simplified block diagram illustrating a coupling between processing clusters and a frame buffer according to an embodiment of the present invention; and

[0018] FIG. 6 is a simplified block diagram illustrating a coupling between processing clusters and a frame buffer according to another embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

System Overview

[0019] FIG. 1 is a block diagram of a computer system 100 according to an embodiment of the present invention. Computer system 100 includes a central processing unit (CPU) 102 and a system memory 104 communicating via a bus path that includes a memory bridge 105. Memory bridge 105 is connected via a bus path 106 to an I/O (input/output) bridge 107. I/O bridge 107 receives user input from one or more user input devices 108 (e.g., keyboard, mouse) and forwards the input to CPU 102 via bus 106 and memory bridge 105. Visual output is provided on a pixel based display device 110 (e.g., a conventional CRT or LCD based monitor) operating under control of a graphics subsystem 112 coupled to memory bridge 105 via a bus 113. A system disk 114 is also connected to I/O bridge 107. A switch 116 provides connections between I/O bridge 107 and other components such as a network adapter 118 and various add-in cards 120, 121. Other components (not explicitly shown), including USB or other port connections, CD drives, DVD drives, and the like, may also be connected to I/O bridge 107. Bus connections among the various components may be implemented using bus protocols such as PCI (Peripheral Component Interconnect), PCI Express (PCI-E), AGP (Accelerated Graphics Port), HyperTransport, or any other bus protocol(s), and connections between different devices may use different protocols as is known in the art.

[0020] Graphics processing subsystem 112 includes a graphics processing unit (GPU) 122 and a graphics memory 124, which may be implemented, e.g., using one or more integrated circuit devices such as programmable processors, application specific integrated circuits (ASICs), and memory devices. GPU 122 may be configured to perform various tasks related to generating pixel data from graphics data supplied by CPU 102 and/or system memory 104 via memory bridge 105 and bus 113, interacting with graphics memory 124 to store and update pixel data, and the like. For example, GPU 122 may generate pixel data from 2-D or 3-D scene data provided by various programs executing on CPU 102. GPU 122 may also store pixel data received via memory bridge 105 to graphics memory 124 with or without further processing. GPU 122 also includes a scanout module configured to deliver pixel data from graphics memory 124 to display device 110.

[0021] CPU 102 operates as the master processor of system 100, controlling and coordinating operations of other system components. In particular, CPU 102 issues commands that control the operation of GPU 122. In some embodiments, CPU 102 writes a stream of commands for GPU 122 to a command buffer, which may be in system memory 104, graphics memory 124, or another storage location accessible to both CPU 102 and GPU 122. GPU 122 reads the command stream from the command buffer and executes commands asynchronously with operation of CPU 102. The commands may include conventional rendering commands for generating images as well as general-purpose computation commands that enable applications executing on CPU 102 to leverage the computational power of GPU 122 for data processing that may be unrelated to image generation.

[0022] It will be appreciated that the system shown herein is illustrative and that variations and modifications are possible. The bus topology, including the number and arrangement of bridges, may be modified as desired. For instance, in some embodiments, system memory 104 is connected to CPU 102 directly rather than through a bridge, and other devices communicate with system memory 104 via memory bridge 105 and CPU 102. In other alternative topologies, graphics subsystem 112 is connected to I/O bridge 107 rather than to memory bridge 105. In still other embodiments, I/O bridge 107 and memory bridge 105 might be integrated into a single chip. The particular components shown herein are optional; for instance, any number of add-in cards or peripheral devices might be supported. In some embodiments, switch 116 is eliminated, and network adapter 118 and add-in cards 120, 121 connect directly to I/O bridge 107.

[0023] The connection of GPU 122 to the rest of system 100 may also be varied. In some embodiments, graphics system 112 is implemented as an add-in card that can be inserted into an expansion slot of system 100. In other embodiments, a GPU is integrated on a single chip with a bus bridge, such as memory bridge 105 or I/O bridge 107.

[0024] A GPU may be provided with any amount of local graphics memory, including no local memory, and may use local memory and system memory in any combination. For instance, in a unified memory architecture (UMA) embodiment, no dedicated graphics memory device is provided, and the GPU uses system memory exclusively or almost exclusively. In UMA embodiments, the GPU may be integrated into a bus bridge chip or provided as a discrete chip with a high-speed bus (e.g., PCI-E) connecting the GPU to the bridge chip and system memory.

[0025] It is also to be understood that any number of GPUs may be included in a system, e.g., by including multiple GPUs on a single graphics card or by connecting multiple graphics cards to bus 113. Multiple GPUs may be operated in parallel to generate images for the same display device or for different display devices.

- 5 [0026] In addition, GPUs embodying aspects of the present invention may be incorporated into a variety of devices, including general purpose computer systems, video game consoles and other special purpose computer systems, DVD players, handheld devices such as mobile phones or personal digital assistants, and so on.

Rendering Pipeline Overview

- 10 [0027] FIG. 2 is a block diagram of a rendering pipeline 200 that can be implemented in GPU 122 of FIG. 1 according to an embodiment of the present invention. In this embodiment, rendering pipeline 200 is implemented using an architecture in which any applicable vertex shader programs, geometry shader programs, and pixel shader programs are executed using the same parallel-processing hardware, referred to herein as a "multithreaded
15 core array" 202. Multithreaded core array 202 is described further below.

[0028] In addition to multithreaded core array 202, rendering pipeline 200 includes a front end 204 and data assembler 206, a setup module 208, a rasterizer 210, a color assembly module 212, and a raster operations module (ROP) 214, each of which can be implemented using conventional integrated circuit technologies or other technologies.

- 20 [0029] Front end 204 receives state information (STATE), rendering commands (CMD), and geometry data (GDATA), e.g., from CPU 102 of FIG. 1. In some embodiments, rather than providing geometry data directly, CPU 102 provides references to locations in system memory 104 at which geometry data is stored; data assembler 206 retrieves the data from system memory 104. The state information, rendering commands, and geometry data may be
25 of a generally conventional nature and may be used to define the desired rendered image or images, including geometry, lighting, shading, texture, motion, and/or camera parameters for a scene.

- [0030] In one embodiment, the geometry data includes a number of object definitions for objects (e.g., a table, a chair, a person or animal) that may be present in the scene. Objects
30 are advantageously modeled as groups of primitives (e.g., points, lines, triangles and/or other polygons) that are defined by reference to their vertices. For each vertex, a position is specified in an object coordinate system, representing the position of the vertex relative to the

object being modeled. In addition to a position, each vertex may have various other attributes associated with it. In general, attributes of a vertex may include any property that is specified on a per-vertex basis; for instance, in some embodiments, the vertex attributes include scalar or vector attributes used to determine qualities such as the color, texture, transparency,
5 lighting, shading, and animation of the vertex and its associated geometric primitives.

[0031] Primitives, as already noted, are generally defined by reference to their vertices, and a single vertex can be included in any number of primitives. In some embodiments, each vertex is assigned an index (which may be any unique identifier), and a primitive is defined by providing an ordered list of indices for the vertices making up that primitive. Other
10 techniques for defining primitives (including conventional techniques such as triangle strips or fans) may also be used.

[0032] The state information and rendering commands define processing parameters and actions for various stages of rendering pipeline 200. Front end 204 directs the state information and rendering commands via a control path (not explicitly shown) to other
15 components of rendering pipeline 200. As is known in the art, these components may respond to received state information by storing or updating values in various control registers that are accessed during processing and may respond to rendering commands by processing data received in the pipeline.

[0033] Front end 204 directs the geometry data to data assembler 206. Data assembler 206
20 formats the geometry data and prepares it for delivery to a geometry module 218 in multithreaded core array 202.

[0034] Geometry module 218 directs programmable processing engines (not explicitly shown) in multithreaded core array 202 to execute vertex and/or geometry shader programs on the vertex data, with the programs being selected in response to the state information
25 provided by front end 204. The vertex and/or geometry shader programs can be specified by the rendering application as is known in the art, and different shader programs can be applied to different vertices and/or primitives. The shader program(s) to be used can be stored in system memory or graphics memory and identified to multithreaded core array 202 via suitable rendering commands and state information as is known in the art. In some
30 embodiments, vertex shader and/or geometry shader programs can be executed in multiple passes, with different processing operations being performed during each pass. Each vertex and/or geometry shader program determines the number of passes and the operations to be

performed during each pass. Vertex and/or geometry shader programs can implement algorithms using a wide range of mathematical and logical operations on vertices and other data, and the programs can include conditional or branching execution paths and direct and indirect memory accesses.

5 [0035] Vertex shader programs and geometry shader programs can be used to implement a variety of visual effects, including lighting and shading effects. For instance, in a simple embodiment, a vertex program transforms a vertex from its 3D object coordinate system to a 3D clip space or world space coordinate system. This transformation defines the relative positions of different objects in the scene. In one embodiment, the transformation can be
10 programmed by including, in the rendering commands and/or data defining each object, a transformation matrix for converting from the object coordinate system of that object to clip space coordinates. The vertex shader program applies this transformation matrix to each vertex of the primitives making up an object. More complex vertex shader programs can be used to implement a variety of visual effects, including lighting and shading, procedural
15 geometry, and animation operations. Numerous examples of such per-vertex operations are known in the art, and a detailed description is omitted as not being critical to understanding the present invention.

[0036] Geometry shader programs differ from vertex shader programs in that geometry
shader programs operate on primitives (groups of vertices) rather than individual vertices.
20 Thus, in some instances, a geometry program may create new vertices and/or remove vertices or primitives from the set of objects being processed. In some embodiments, passes through a vertex shader program and a geometry shader program can be alternated to process the geometry data.

[0037] In some embodiments, vertex shader programs and geometry shader programs are
25 executed using the same programmable processing engines in multithreaded core array 202. Thus, at certain times, a given processing engine may operate as a vertex shader, receiving and executing vertex program instructions, and at other times the same processing engine may operate as a geometry shader, receiving and executing geometry program instructions. The processing engines can be multithreaded, and different threads executing different types
30 of shader programs may be in flight concurrently in multithreaded core array 202.

[0038] After the vertex and/or geometry shader programs have executed, geometry module 218 passes the processed geometry data (GDATA') to setup module 208. Setup module 208,

which may be of generally conventional design, generates edge equations from the clip space or screen space coordinates of each primitive; the edge equations are advantageously usable to determine whether a point in screen space is inside or outside the primitive.

[0039] Setup module 208 provides each primitive (PRIM) to rasterizer 210. Rasterizer 210, which may be of generally conventional design, determines which (if any) pixels are covered by the primitive, e.g., using conventional scan-conversion algorithms. As used herein, a "pixel" (or "fragment") refers generally to a region in 2-D screen space for which a single color value is to be determined; the number and arrangement of pixels can be a configurable parameter of rendering pipeline 200 and might or might not be correlated with the screen resolution of a particular display device. As is known in the art, pixel color may be sampled at multiple locations within the pixel (e.g., using conventional supersampling or multisampling techniques), and in some embodiments, supersampling or multisampling is handled within the pixel shader.

[0040] After determining which pixels are covered by a primitive, rasterizer 210 provides the primitive (PRIM), along with a list of screen coordinates (X,Y) of the pixels covered by the primitive, to a color assembly module 212. Color assembly module 212 associates the primitives and coverage information received from rasterizer 210 with attributes (e.g., color components, texture coordinates, surface normals) of the vertices of the primitive and generates plane equations (or other suitable equations) defining some or all of the attributes as a function of position in screen coordinate space.

[0041] These attribute equations are advantageously usable in a pixel shader program to interpolate a value for the attribute at any location within the primitive; conventional techniques can be used to generate the equations. For instance, in one embodiment, color assembly module 212 generates coefficients A , B , and C for a plane equation of the form $U = Ax + By + C$ for each attribute U .

[0042] Color assembly module 212 provides the attribute equations (EQS, which may include e.g., the plane-equation coefficients A , B and C) for each primitive that covers at least one pixel and a list of screen coordinates (X,Y) of the covered pixels to a pixel module 224 in multithreaded core array 202. Pixel module 224 directs programmable processing engines (not explicitly shown) in multithreaded core array 202 to execute one or more pixel shader programs on each pixel covered by the primitive, with the program(s) being selected in response to the state information provided by front end 204. As with vertex shader programs

and geometry shader programs, rendering applications can specify the pixel shader program to be used for any given set of pixels. Pixel shader programs can be used to implement a variety of visual effects, including lighting and shading effects, reflections, texture blending, procedural texture generation, and so on. Numerous examples of such per-pixel operations are known in the art and a detailed description is omitted as not being critical to understanding the present invention. Pixel shader programs can implement algorithms using a wide range of mathematical and logical operations on pixels and other data, and the programs can include conditional or branching execution paths and direct and indirect memory accesses.

[0043] Pixel shader programs are advantageously executed in multithreaded core array 202 using the same programmable processing engines that also execute the vertex and/or geometry shader programs. Thus, at certain times, a given processing engine may operate as a vertex shader, receiving and executing vertex program instructions; at other times the same processing engine may operate as a geometry shader, receiving and executing geometry program instructions; and at still other times the same processing engine may operate as a pixel shader, receiving and executing pixel shader program instructions. It will be appreciated that the multithreaded core array can provide natural load-balancing between pixel and vertex processing: where the application is geometry intensive (e.g., many small primitives), a larger fraction of the processing cycles in multithreaded core array 202 will tend to be devoted to vertex and/or geometry shaders, and where the application is pixel intensive (e.g., fewer and larger primitives shaded using complex pixel shader programs with multiple textures and the like), a larger fraction of the processing cycles will tend to be devoted to pixel shaders.

[0044] Once processing for a pixel or group of pixels is complete, pixel module 224 provides the processed pixels (PDATA) to ROP 214. ROP 214, which may be of generally conventional design, integrates the pixel values received from pixel module 224 with pixels of the image under construction in frame buffer 226, which may be located, e.g., in graphics memory 124. In some embodiments, ROP 214 can mask pixels or blend new pixels with pixels previously written to the rendered image. Depth buffers, alpha buffers, and stencil buffers can also be used to determine the contribution (if any) of each incoming pixel to the rendered image. Pixel data PDATA' corresponding to the appropriate combination of each incoming pixel value and any previously stored pixel value is written back to frame buffer

226. Once the image is complete, frame buffer 226 can be scanned out to a display device and/or subjected to further processing.

[0045] It will be appreciated that the rendering pipeline described herein is illustrative and that variations and modifications are possible. The pipeline may include different units from those shown and the sequence of processing events may be varied from that described herein. For instance, in some embodiments, rasterization may be performed in stages, with a "coarse" rasterizer that processes the entire screen in blocks (e.g., 16x16 pixels) to determine which, if any, blocks the triangle covers (or partially covers), followed by a "fine" rasterizer that processes the individual pixels within any block that is determined to be at least partially covered. In one such embodiment, the fine rasterizer is contained within pixel module 224. In another embodiment, some operations conventionally performed by a ROP may be performed within pixel module 224 before the pixel data is forwarded to ROP 214.

[0046] Further, multiple instances of some or all of the modules described herein may be operated in parallel. In one such embodiment, multithreaded core array 202 includes two or more geometry modules 218 and an equal number of pixel modules 224 that operate in parallel. Each geometry module and pixel module jointly controls a different subset of the processing engines in multithreaded core array 202.

Multithreaded Core Array Configuration

[0047] In one embodiment, multithreaded core array 202 provides a highly parallel architecture that supports concurrent execution of a large number of instances of vertex, geometry, and/or pixel shader programs in various combinations. FIG. 3 is a block diagram of multithreaded core array 202 according to an embodiment of the present invention.

[0048] In this embodiment, multithreaded core array 202 includes some number (N) of processing clusters 302. Herein, multiple instances of like objects are denoted with reference numbers identifying the object and parenthetical numbers identifying the instance where needed. Any number N (e.g., 1, 4, 8, or any other number) of processing clusters may be provided. In FIG. 3, one processing cluster 302 is shown in detail; it is to be understood that other processing clusters 302 can be of similar or identical design.

[0049] Each processing cluster 302 includes a geometry controller 304 (implementing geometry module 218 of FIG. 2) and a pixel controller 306 (implementing pixel module 224 of FIG. 2). Geometry controller 304 and pixel controller 306 each communicate with a core interface 308. Core interface 308 controls a number (M) of cores 310 that include the

processing engines of multithreaded core array 202. Any number M (e.g., 1, 2, 4 or any other number) of cores 310 may be connected to a single core interface. Each core 310 is advantageously implemented as a multithreaded execution core capable of supporting a large number (e.g., 100 or more) of concurrent execution threads (where the term "thread" refers to an instance of a particular program executing on a particular set of input data), including a combination of vertex threads, geometry threads, and pixel threads. In one embodiment, each core 310 implements a P-way SIMD architecture to execute P threads in parallel, where P is an arbitrary integer (e.g., 8, 16, 32) and is capable of managing a number G (e.g. 18, 24, etc.) of groups of P threads concurrently. A detailed description of core 310 can be found in U.S. Provisional Application No. 60/752,265, filed December 19, 2005, which is incorporated herein by reference in its entirety for all purposes.

[0050] Core interface 308 also controls a texture pipeline 314 that is shared among cores 310. Texture pipeline 314, which may be of generally conventional design, advantageously includes logic circuits configured to receive texture coordinates, to fetch texture data corresponding to the texture coordinates from memory, and to filter the texture data according to various algorithms. Conventional filtering algorithms including bilinear and trilinear filtering may be used. When a core 310 encounters a texture instruction in one of its threads, it provides the texture coordinates to texture pipeline 314 via core interface 308. Texture pipeline 314 processes the texture instruction and returns the result to the core 310 via core interface 308. Texture processing by pipeline 314 may consume a significant number of clock cycles, and while a thread is waiting for the texture result, core 310 advantageously continues to execute other threads.

[0051] In operation, data assembler 206 (FIG. 2) provides geometry data GDATA to processing clusters 302. In one embodiment, data assembler 206 divides the incoming stream of geometry data into portions and selects, e.g., based on availability of execution resources, which of processing clusters 302 is to receive the next portion of the geometry data. That portion is delivered to geometry controller 304 in the selected processing cluster 302.

[0052] Geometry controller 304 forwards the received data to core interface 308, which loads the vertex data into a core 310, then instructs core 310 to launch the appropriate vertex shader program. Upon completion of the vertex shader program, core interface 308 signals geometry controller 304. If a geometry shader program is to be executed, geometry controller 304 instructs core interface 308 to launch the geometry shader program. In some

embodiments, the processed vertex data is returned to geometry controller 304 upon completion of the vertex shader program, and geometry controller 304 instructs core interface 308 to reload the data before executing the geometry shader program. After completion of the vertex shader program and/or geometry shader program, geometry controller 304 provides the processed geometry data (GDATA') to setup module 208 of FIG. 2.

[0053] At the pixel stage, color assembly module 212 (FIG. 2) provides attribute equations EQS for a primitive and pixel coordinates (X,Y) of pixels covered by the primitive to processing clusters 302. In one embodiment, color assembly module 212 divides the incoming stream of coverage data into portions and selects, e.g., based on availability of execution resources or the location of the primitive in screen coordinates, which of processing clusters 302 is to receive the next portion of the data. That portion is delivered to pixel controller 306 in the selected processing cluster 302.

[0054] Pixel controller 306 delivers the data to core interface 308, which loads the pixel data into a core 310, then instructs the core 310 to launch the pixel shader program. Where core 310 is multithreaded, pixel shader programs, geometry shader programs, and vertex shader programs can all be executed concurrently in the same core 310. Upon completion of the pixel shader program, core interface 308 delivers the processed pixel data to pixel controller 306, which forwards the pixel data PDATA to ROP unit 214 (FIG. 2).

[0055] It will be appreciated that the multithreaded core array described herein is illustrative and that variations and modifications are possible. Any number of processing clusters may be provided, and each processing cluster may include any number of cores. In some embodiments, shaders of certain types may be restricted to executing in certain processing clusters or in certain cores; for instance, geometry shaders might be restricted to executing in core 310(0) of each processing cluster. Such design choices may be driven by considerations of hardware size and complexity versus performance, as is known in the art. A shared texture pipeline is also optional; in some embodiments, each core might have its own texture pipeline or might leverage general-purpose functional units to perform texture computations.

[0056] Data to be processed can be distributed to the processing clusters in various ways. In one embodiment, the data assembler (or other source of geometry data) and color assembly module (or other source of pixel-shader input data) receive information indicating the availability of processing clusters or individual cores to handle additional threads of various

types and select a destination processing cluster or core for each thread. In another embodiment, input data is forwarded from one processing cluster to the next until a processing cluster with capacity to process the data accepts it. In still another embodiment, processing clusters are selected based on properties of the input data, such as the screen coordinates of pixels to be processed.

[0057] The multithreaded core array can also be leveraged to perform general-purpose computations that might or might not be related to rendering images. In one embodiment, any computation that can be expressed in a data-parallel decomposition can be handled by the multithreaded core array as an array of threads executing in a single core. Results of such computations can be written to the frame buffer and read back into system memory.

Allocation of Pixel Shader Work

[0058] In accordance with an embodiment of the present invention, pixels to be processed by a pixel shader program are directed to a processing cluster 302 (FIG. 3) based on the position of the pixels within the image area. For example, the image area can be divided into a number of tiles. Each tile is associated with one of the processing clusters 302 in such a way that the tiles associated with one cluster are scattered across the image area (i.e., at least some of the tiles associated with one processing cluster are not contiguous with one another).

[0059] FIG. 4 illustrates one possible tiling of an image area 400 into a large number of tiles 402 according to an embodiment of the present invention. Each tile 402 might be, e.g., 16x16 pixels or any other convenient size. In this embodiment, multithreaded core array 202 of FIG. 3 includes eight processing clusters 302(0) through 302(7). Each tile 402 in FIG. 4 contains a number i (0 through 7) indicating which processing core 302(i) processes pixels in that tile. As can be seen in FIG. 4, each processing cluster 302 is assigned an equal (or nearly equal) number of tiles 402 within image area 400, and the tiles assigned to each cluster 302 are not contiguous with each other. It is expected that for many graphics applications, distributing work in this manner will provide approximate load balancing among the processing clusters 302. Suitable pixel distribution logic can be included in rendering pipeline 200, e.g., in color assembly module 212 of FIG. 2.

[0060] It will be appreciated that the tiling shown in FIG. 4 is illustrative and that variations and modifications are possible. The tiles may be of any size. In some embodiments, the size and number of tiles is configurable, with tile size being chosen based

on application characteristics such as whether a multisampling mode is in use. The arrangement of tiles assigned to each processing cluster can vary as desired.

[0061] Tiles may be assigned to any number of processing clusters, up to the total number that are present in a particular GPU. In some embodiments, tiles are assigned to fewer than all of the processing clusters. Thus, a GPU can render images using only some of its processing clusters to process pixel threads. As long as at least one processing cluster is available, the GPU will be able to render images, albeit at reduced throughput. Running with some clusters disabled from processing pixel threads may be useful, e.g., for operating in a reduced-power mode (some cores or clusters can be powered down while others operate), for determining minimum performance requirements (without requiring the availability of different GPUs), and/or for providing tolerance against defects in one or another of the cores.

[0062] In some alternative embodiments, tiles are assigned to particular cores 310 rather than to processing clusters 302.

Pixel Shader Coupling to Frame Buffer

[0063] In some embodiments, rather than the centralized ROP 214 suggested in FIG. 2, each pixel controller 306 in FIG. 3 includes its own ROP that communicates pixels to frame buffer 226 of FIG. 2. In such embodiments, a coupling from processing clusters 302 to the frame buffer is provided.

[0064] In one embodiment with N processing clusters, frame buffer 226 is partitioned into N partitions. Each cluster 302 is coupled to a different one of the N partitions.

[0065] FIG. 5 is a simplified block diagram illustrating a coupling between processing clusters 302 and a frame buffer 500 according to an embodiment of the present invention. It is to be understood that frame buffer 226 of FIG. 2 might include multiple frame buffers 500, where each frame buffer 500 stores a specified quantity on a per-pixel basis for the same image. For instance, in one embodiment, frame buffer 226 includes a Z buffer; color component buffers (e.g., for red, green and blue color components); and a transparency (alpha) buffer. It is to be understood that any number of frame buffers 500 may be provided and that "pixel" as used herein refers to sampling locations within the image, which might or might not correspond to the number of active pixels in a given display device. For simplicity, only one frame buffer 500 is shown; it is to be understood that the same partitioning can be applied to each frame buffer for an image.

[0066] Frame buffer 500 is partitioned (physically or logically) into N partitions 502, with each partition being large enough to store the data for at least $1/N$ of the pixels in the image. Each of the N processing clusters 302 is coupled to one of the N partitions 502. Thus, processing cluster 302(0) stores all of its output pixel data in partition 502(0), processing cluster 302(1) in partition 502(1) and so on. It should be noted that to the extent that tiles processed by a given processing cluster 302 are not contiguous, the data in a given frame buffer partition 502 will be from non-contiguous tiles. The display (scanout) logic advantageously observes the correct display ordering of the data when accessing frame buffer 500; in particular, it is not required that partitions be read out sequentially.

[0067] As noted above, in some embodiments, fewer than all of processing cores 302 might be used to generate pixels. In the embodiment shown in FIG. 5, only processing cluster 302(0) provides pixels to frame buffer partition 502(0). Accordingly, if processing cluster 302(0) is not operated to generate pixels, frame buffer partition 502(0) will not be used. The image area can be retiled or tiles can be reassigned among processing clusters 302(1) through 302(N-1) such that the area is divided among N-1 processing clusters. In some embodiments, the sampling resolution might need to be reduced, e.g., if frame buffer partitions 502(1) through 502(N-1) do not provide enough space to store all of the pixel data for the image at the original sampling resolution. In embodiments where the partitions are logical rather than physical, frame buffer 500 can be repartitioned to match the number of processing clusters 302 available.

[0068] In an alternative embodiment, each processing cluster can access multiple frame buffer partitions. FIG. 6 is a simplified block diagram illustrating a coupling between processing clusters 302 and a frame buffer 600 according to an embodiment of the present invention. It is to be understood that frame buffer 226 of FIG. 2 might include multiple frame buffers 600, where each frame buffer 600 stores a specified quantity on a per-pixel basis for the same image as described above with reference to frame buffer 500. It is to be understood that any number of frame buffers 600 may be provided. For simplicity, only one frame buffer 600 is shown; it is to be understood that the partitioning can be applied to each frame buffer for an image.

[0069] Frame buffer 600 is partitioned (physically or logically) into a number B of partitions 602, where B might be equal to or different from the number N of processing

clusters 302. Processing clusters 302 are coupled to partitions 602 via a crossbar 604. Each cluster 302 can write pixel data to any one (or more) of the B partitions 602.

[0070] In this embodiment, crossbar 604 is configurable, allowing the coupling of processing clusters 302 to frame buffer partitions 602 to be modified as desired. For

5 example, as noted above, in some embodiments, fewer than all of processing cores 302 might be used to generate pixels. In the embodiment shown in FIG. 6, if processing core 302(0) is disabled, crossbar 604 can be reconfigured so that all frame buffer partitions 602 are still accessible to one or another of processing cores 302. The display (scanout) logic is advantageously configurable such that the frame buffer data is correctly scanned out
10 regardless of the configuration of crossbar 604 or the assignment of tiles to processing cores 302.

[0071] Where multiple frame buffers are present (e.g., Z, color, alpha, etc.), each frame buffer may be partitioned into B partitions in the manner described. In some embodiments, the number of partitions is not the same for all frame buffers; for instance, a Z buffer might
15 have more or fewer partitions than a color buffer.

Further Embodiments

[0072] While the invention has been described with respect to specific embodiments, one skilled in the art will recognize that numerous modifications are possible. Thus, although the invention has been described with respect to specific embodiments, it will be appreciated that
20 the invention is intended to cover all modifications and equivalents within the scope of the following claims.

WHAT IS CLAIMED IS:

1. A graphics processor comprising:
 - a multithreaded core array including a plurality of processing clusters, each processing cluster including at least one processing core operable to execute a pixel shader program that generates pixel data from coverage data;
 - a rasterizer configured to generate coverage data for each of a plurality of pixels; and
 - pixel distribution logic configured to deliver the coverage data from the rasterizer to one of the processing clusters in the multithreaded core array,
 - wherein the pixel distribution logic selects the one of the processing clusters to which the coverage data for a first pixel is delivered based at least in part on a location of the first pixel within an image area, and
 - wherein each processing core is also operable to execute vertex shader programs and geometry shader programs.
2. The graphics processor of claim 1 wherein the image area is divided into a plurality of tiles, each tile being assigned to one of the processing clusters, and wherein the pixel distribution logic selects the one of the processing clusters based on a determination as to which of the plurality of tiles includes the first pixel.
3. The graphics processor of claim 2 wherein at least two of the plurality of tiles are assigned to each of the processing clusters, wherein for each processing cluster, the tiles assigned thereto are not contiguous with each other.
4. The graphics processor of claim 1 wherein each of the processing clusters is configured to deliver pixel data to a respective one of a plurality of partitions of a frame buffer.
5. The graphics processor of claim 1 further comprising a crossbar coupled to each of the processing clusters and configured to deliver pixel data from the processing clusters to a frame buffer having a plurality of partitions.
6. The graphics processor of claim 5 wherein the crossbar is configured such that pixel data generated by any one of the processing clusters is deliverable to any one of the frame buffer partitions.

7. A graphics processor comprising:

a multithreaded core array including a plurality of processing clusters, each processing cluster including at least one processing core operable to execute a pixel shader program that generates pixel data from coverage data;

a rasterizer configured to generate coverage data for each of a plurality of pixels;

a pixel distribution logic configured to deliver the coverage data from the rasterizer to one of the processing clusters in the multithreaded core array; and

a crossbar coupled to each of the processing clusters and configured to deliver pixel data from the processing clusters to a frame buffer comprising multiple partitions;

wherein the pixel distribution logic selects the one of the processing clusters to which the coverage data for a first pixel is delivered based at least in part on a location of the first pixel within an image area.