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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
20 December 2012 (20.12.2012)

WIPO | PCT

(10) International Publication Number
WO 2012/174231 A1

(51) International Patent Classification:
G06T 9/00 (2006.01) *G06T 15/00* (2011.01)

(21) International Application Number:
PCT/US2012/042442

(22) International Filing Date:
14 June 2012 (14.06.2012)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
13/163,071 17 June 2011 (17.06.2011) US

(71) Applicant *(for all designated States except US)*: **ADVANCED MICRO DEVICES, INC.** [US/US]; One AMD Place, P.O. Box 3453, Sunnyvale, California 94088 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **IOURCHA, Konstantine** [US/US]; 7186 Wooded Lake Dr., San Jose, California 95120 (US). **BROTHERS, John, W.** [US/US]; 1257 Lakeside Dr. #1226, Sunnyvale, California 94085 (US).

(74) Agent: **KIVLIN, B. Noel**; Meyertons, Hood, Kivlin, Kowert & Goetzel, P.C., P.O. Box 398, Austin, Texas 78767-0398 (US).

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

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ,

[Continued on next page]

(54) Title: REAL TIME ON-CHIP TEXTURE DECOMPRESSION USING SHADER PROCESSORS

(57) Abstract: A processing unit, method, and medium for decompressing or generating textures within a graphics processing unit (GPU). The textures are compressed with a variable-rate compression scheme such as JPEG. The compressed textures are retrieved from system memory and transferred to local cache memory on the GPU without first being decompressed. A table is utilized by the cache to locate individual blocks within the compressed texture. A decompressing shader processor receives compressed blocks and then performs on-the-fly decompression of the blocks. The decompressed blocks are then processed as usual by a texture consuming shader processor of the GPU.

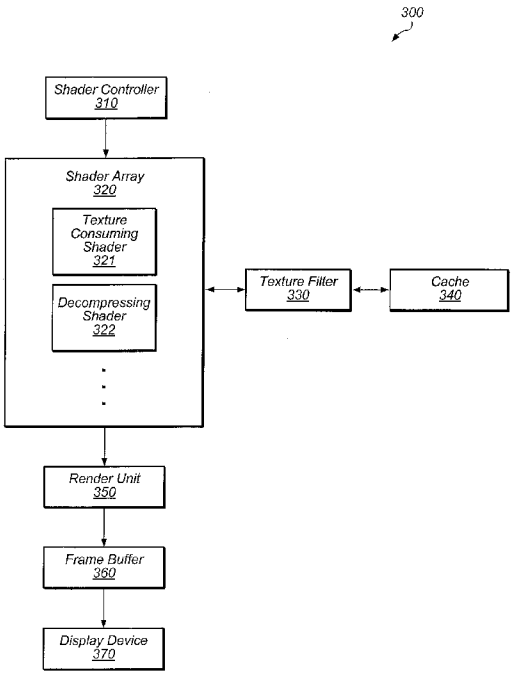

FIG. 3

WO 2012/174231 A1

TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

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

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

TITLE: REAL TIME ON-CHIP TEXTURE DECOMPRESSION USING SHADER PROCESSORS

BACKGROUND OF THE INVENTION

Field of the Invention

[0001] This disclosure relates generally to graphics processing, and in particular to a processing unit, method, and medium of texture decompression.

Description of the Related Art

[0002] Computer graphics processing systems process large amounts of data, typically with a graphics processing unit (GPU) performing a large percentage of the processing. A GPU is a complex integrated circuit that is configured to perform, inter alia, graphics-processing tasks. For example, a GPU may execute graphics-processing tasks required by an end-user application, such as a video-game application. The GPU may be a discrete device or may be included in the same device as another processor, such as a central processing unit (CPU).

[0003] A GPU produces the pixels that make up an image from a higher level description of its components in a process known as rendering. GPU's typically utilize a concept of continuous rendering by the use of computing elements to process pixel, texture, and geometric data. The computing elements may execute the functions of rasterizers, setup engines, color blenders, hidden surface removal, and texture mapping. These computing elements are often referred to as shaders, shader processors, shader arrays, shader pipes, shader pipe arrays, shader pipelines, or a shader engine, "shader" being a term in computer graphics referring to a set of software instructions or a program used by a graphics resource primarily to perform rendering effects. "Shader" may also refer to an actual hardware component or processor used to execute software instructions. A shader processor or program may read and render data and perform any type of processing of the data. GPU's equipped with a unified shader also simultaneously support many types of shader processing, from pixel, vertex, primitive, and generalized compute processing.

[0004] Much of the processing involved in generating complex graphics scenes involves texture data. Textures may be any of various types of data, such as color, transparency, lookup tables, or other data. In some embodiments, textures may be digitized images to be drawn onto geometric shapes to add visual detail. A large amount of detail, through the use of textures, may be mapped to the surface of a graphical model as the model is rendered to create a destination

image. The purpose of texture mapping is to provide a realistic appearance on the surface of objects. Textures may specify many properties, including colors, surface properties like specular reflection or fine surface details in the form of normal or bump maps. A texture could also be image data, color or transparency data, roughness/smoothness data, reflectivity data, etc. A 'texel' is a texture element in the same way a 'pixel' is a picture element. The terms 'texel' and 'pixel' may be used interchangeably within this specification.

[0005] In 3D computer graphics, surface detail on objects is commonly added through the use of textures. For example, a 2D bitmap image of a brick wall may be applied, using texture mapping, to a set of polygons representing a 3D model of a building to give the 3D rendering of that object the appearance that it is made of bricks. Providing realistic computer graphics typically requires many high-quality, detailed textures. The use of textures can consume large amounts of storage space and bandwidth, and consequently textures may be compressed to reduce storage space and bandwidth utilization.

[0006] Texture compression has thus become a widely accepted feature of graphics hardware in general and 3D graphics hardware in particular. The goal of texture compression is to reduce storage and bandwidth costs on the graphics system while retaining as much of the quality of the original texture as possible. The compression and decompression methods described herein may be used to compress various types of texture information including image data, picture data, transparency information, smoothness or roughness data, or any other similarly structured data. As such, the term texture is used broadly herein to refer to the data being compressed or decompressed as part of a GPU.

[0007] Fixed-rate compression schemes have traditionally been used to compress textures and may generally suffer from several shortcomings as compared to variable-rate schemes. Unlike fixed-rate compression, variable-rate compression is more flexible and may allow for adjustments to quality as desired. For example, variable-rate compression may be set to achieve lossless compression. In some cases, the use of variable-rate compression schemes may provide better compression than traditional fixed-rate compression schemes. A variable-rate compression scheme, such as Joint Photographic Experts Group (JPEG), is typically not used for texture compression when on-the-fly decompression is desired due to the high complexity and implementation cost. Therefore, there is a need in the art for methods and mechanisms to enable low-cost on-the-fly decompression of variable-rate compressed textures.

[0008] In view of the above, improved processing units, methods, and mediums for performing real time decompression of compressed textures are desired.

SUMMARY OF EMBODIMENTS OF THE INVENTION

[0009] Various embodiments of processing units, methods and mediums for decompressing texture data are contemplated. In one embodiment, a first shader of a plurality of shaders may require a block of a texture to produce data used by a display device or in further processing. The first shader may be configured to calculate a virtual address of the block within an uncompressed version of the texture and convey the virtual address with a request for the block to a cache memory device. In response to determining an uncompressed version of the block is not stored in the cache, a second shader of the plurality of shaders may be initiated as a decompressing shader and the virtual address of the uncompressed version of the block may be passed to the decompressing shader. Also, in response to determining the uncompressed version of the block is not in the cache, a cache line may be allocated for the requested block.

[0010] The second shader may be configured to receive the compressed version of the block from the cache. The cache may be configured to utilize a table which maps a virtual address space of an uncompressed version of the texture to an address space of a compressed version of the texture. The cache and/or the second shader may be configured to determine the location and size of the compressed version of the block from the table. The table may also contain additional information, such as the value of the DC coefficient of a compressed version of each block of the texture.

[0011] After receiving the compressed version of the block from the cache, the second shader may be configured to decompress the compressed version of the block and then write a decompressed version of the block to the cache. After the decompressed version of the block has been written to the cache, the first shader may be configured to receive the decompressed version of the block from the cache. The first shader may then be configured to process the decompressed version of the block such that it may be applied to a rendered surface for display.

[0012] These and other features and advantages will become apparent to those of ordinary skill in the art in view of the following detailed descriptions of the approaches presented herein.

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] The above and further advantages of the systems, methods, and mechanisms may be better understood by referring to the following description in conjunction with the accompanying drawings, in which:

[0014] FIG. 1 illustrates one embodiment of a computer graphics system.

[0015] FIG. 2 is a block diagram of a GPU in accordance with one or more embodiments.

[0016] FIG. 3 illustrates a block diagram of one embodiment of a graphics processing system.

[0017] FIG. 4A illustrates a block diagram of one embodiment of a data cache.

[0018] FIG. 4B is a block mapping table in accordance with one or more embodiments.

[0019] FIG. 5 illustrates one embodiment of a virtual address space for an 8x8 block of texels.

[0020] FIG. 6 is a block diagram of one embodiment of a portion of data.

[0021] FIG. 7 is a generalized flow diagram illustrating one embodiment of a method to decompress a compressed block of a texture.

DETAILED DESCRIPTION

[0022] In the following description, numerous specific details are set forth to provide a thorough understanding of the methods and mechanisms presented herein. However, one having ordinary skill in the art should recognize that the various embodiments may be practiced without these specific details. In some instances, well-known structures, components, signals, computer program instructions, and techniques have not been shown in detail to avoid obscuring the approaches described herein. It will be appreciated that for simplicity and clarity of illustration, elements shown in the figures have not necessarily been drawn to scale. For example, the dimensions of some of the elements may be exaggerated relative to other elements.

[0023] This specification includes references to "one embodiment" or "an embodiment." The appearances of the phrases "in one embodiment" or "in an embodiment" do not necessarily refer to the same embodiment. Particular features, structures, or characteristics may be combined in any suitable manner consistent with this disclosure.

[0024] Terminology. The following paragraphs provide definitions and/or context for terms found in this disclosure (including the appended claims):

[0025] "Comprising." This term is open-ended. As used in the appended claims, this term does not foreclose additional structure or steps. Consider a claim that recites: "A GPU comprising a plurality of shaders ...." Such a claim does not foreclose the GPU from including additional components (e.g., a texture unit, input/output circuitry, etc.).

[0026] "Configured To." Various units, circuits, or other components may be described or claimed as "configured to" perform a task or tasks. In such contexts, "configured to" is used to connote structure by indicating that the units/circuits/components include structure (e.g.,

circuitry) that performs those task or tasks during operation. As such, the unit/circuit/component can be said to be configured to perform the task even when the specified unit/circuit/component is not currently operational (e.g., is not on). The units/circuits/components used with the "configured to" language include hardware—for example, circuits, memory storing program instructions executable to implement the operation, etc. Reciting that a unit/circuit/component is "configured to" perform one or more tasks is expressly intended not to invoke 35 U.S.C. § 114, sixth paragraph, for that unit/circuit/component. Additionally, "configured to" can include generic structure (e.g., generic circuitry) that is manipulated by software and/or firmware (e.g., an FPGA or a general-purpose processor executing software) to operate in manner that is capable of performing the task(s) at issue. "Configure to" may also include adapting a manufacturing process (e.g., a semiconductor fabrication facility) to fabricate devices (e.g., integrated circuits) that are adapted to implement or perform one or more tasks.

[0027] "First," "Second," etc. As used herein, these terms are used as labels for nouns that they precede, and do not imply any type of ordering (e.g., spatial, temporal, logical, etc.). For example, in a processor having eight processing elements or cores, the terms "first" and "second" processing elements can be used to refer to any two of the eight processing elements. In other words, the "first" and "second" processing elements are not limited to logical processing elements 0 and 1.

[0028] Referring to FIG. 1, a block diagram of one embodiment of a computer graphics system is shown. Computer graphics system 100 includes computing system 102 and display device 114. Computing system 102 includes a graphics processing unit (GPU) 104 for processing graphics data. In some embodiments, GPU 104 may reside on a graphics card within computing system 102. GPU 104 may process graphics data to generate color and luminance values for each pixel of a frame for display on display device 114. GPU 104 may include one or more processing cores and/or an array of shaders to perform pixel manipulations.

[0029] Computing system 102 may include a software program application 108, an application programming interface (API) 110, and a driver 112, which may run on a CPU (not shown). API 110 may adhere to an industry-standard specification, such as OpenGL or DirectX. API 110 may communicate with driver 112. Driver 112 may translate standard code received from API 110 into a native format of instructions understood by GPU 104. GPU 104 may then execute the instructions received from driver 112.

[0030] Textures may be transferred to GPU 104 from system memory (not shown) or another storage device of computing system 102. In one embodiment, textures may be compressed using JPEG compression. In other embodiments, other types of variable-rate compression may be used to compress the textures. For the remainder of this specification, examples of JPEG type encoding will be used to describe the various embodiments. However, this is for illustrative purposes only, and other types of variable-rate compression may also be used with the methods and mechanisms described herein.

[0031] Driver 112 may reformat compressed textures as part of a tiling process. This reformatting may entail transcoding a JPEG-compressed texture into a hardware internal JPEG format. In other embodiments, the JPEG-compressed texture may be transcoded into other formats. The hardware internal JPEG format may contain additional information to facilitate the decompression process. For example, the hardware internal JPEG format may include a table with information on the location and sizes of the various blocks of the JPEG-compressed texture. The table may also include information on the DC coefficients of each 8x8 block of the JPEG-compressed texture. The table may further include Huffman codes, quantization tables, and other information to facilitate the decompression of the compressed texture. Driver 112 may also allocate a virtual address space for each of the compressed textures utilized by computing system 102. The size of each virtual address space may correspond to the size of the uncompressed texture.

[0032] Computing system 102 will typically have various other devices/components not shown in FIG. 1, such as a CPU, buses, memory, peripheral devices, etc. For example, computing system 102 may include an I/O interface which may be coupled to other devices, such as a keyboard, printer, and mouse, in addition to display device 114. In some embodiments, computing system 102 may include a plurality of GPU's.

[0033] In another embodiment, a processor, such as GPU 104, may be defined in software. The software instructions may be stored in a computer readable storage medium and when executed on a computing device, may define the processor. In a further embodiment, processors may comprise GPU's, CPU's, video processing units (VPU's), coprocessors, and/or other types of processors that are configured to process texture data. In various embodiments, the GPU and CPU may be separate integrated circuit devices/packages. In various embodiments, the GPU and CPU may be included in a single integrated circuit or package.

[0034] Referring to FIG. 2, a block diagram of one embodiment of a GPU 200 is shown. GPU 200 may be utilized to perform graphics-processing related tasks (e.g., using vertex shaders,

geometry shaders, pixel shaders, etc.) and general-computing tasks (e.g., mathematical algorithms, physics simulations, etc.). In the example shown, GPU 200 includes shader processor array 210, command processor 212, texture memory 220, and memory controller 222 which may be configured to support direct-memory access (DMA). It is noted that the embodiment of GPU 200 depicted in FIG. 2 is for illustrative purposes only, and those skilled in the art will appreciate numerous alternative embodiments are possible. All such alternative embodiments are contemplated. Note also that GPU 200 may include many other components not shown in FIG. 2.

[0035] In the embodiment shown, shader processor array 210 comprises multiple processing units which may perform in parallel. Command processor 212 may issue commands and assign processing tasks to individual shader processors of shader processor array 210. In some embodiments, command processor 212 may include a dispatch processor (not shown) configured to divide a received workload into threads and distribute the threads among processing units of the shader processor array. Shader processor array 210 may be configured to perform various types of functions, including processing texture data and performing rendering algorithms to transform 3-dimensional texture objects into a 2-dimensional image. As noted above, shader processor array 210 may include a plurality of shader processors, and the plurality of shader processors may implement algorithms using a wide range of mathematical and logical operations on vertices and other texture data.

[0036] In some embodiments, GPU 200 may be configured to utilize one or more on-chip and/or off chip memories for temporarily storing data. While such memories may be referred to herein as "caches", it is noted that the use of such a term does not necessarily require any particular organization, structure or policies for such memories. For example, while such memories may utilize organizations and policies typically associated with central processing unit (CPU) caches – such as set associative organizations and replacement policies, any desired organization and/or storage policies may be utilized. In various embodiments, texture memory 220 is used for storing texture data. In such an embodiment, texture memory 220 may provide faster access to certain texture data, such as texture data that is frequently used, than would be possible if the texture data were only stored in system memory 226 or local memory 230. System memory 226 may represent memory accessible by both GPU 200 and a central processing unit (CPU, not shown), while local memory may represent memory which is directly accessible by only GPU 200. In various embodiments, texture memory 220 may include multiple levels in a hierarchical arrangement as is commonly known in the cache arts. The

number of such cache levels included in texture cache system 220 may vary from one embodiment to the next. Texture memory 220 may be implemented using a variety of memory technologies, such as static memory (e.g., SRAM), stacked-memory using dynamic memory (e.g., DRAM), or otherwise. Texture memory 220 may also include caching logic. The caching logic may be configured to cache data into texture memory 220 and to implement cache management policies that consider the relative latency and/or bandwidth of cache system 220 versus system memory 226.

[0037] GPU 200 may also include memory controller 222. Memory controller 222 may be coupled to system memory 226 and local memory 230. Memory controller 222 may access data, such as compressed textures 228, in system memory 226. Compressed textures 228 may include a plurality of textures which may be compressed with any of a variety of variable-rate compression techniques, such as JPEG. Compressed textures 228, or portions of individual textures within compressed textures 228, may be transferred to texture memory 220 and shader processor array 210 of GPU 200 (via memory controller 222) without first being decompressed. Host driver 240 may transfer commands and data to GPU 200 via system memory 226. Local memory 230 may be utilized for storing vertex data and other data used by GPU 200, and GPU 200 may write frame data to local memory 230.

[0038] Referring now to FIG. 3, a block diagram of one embodiment of a graphics processing system is shown. Graphics processing system 300 may include shader controller 310, and shader controller 310 may assign specific graphics processing tasks to individual shader computing units within shader array 320. Shader controller 310 may perform pre-processing on graphics-processing tasks and general-computing tasks, and issue these tasks to shader array 320. Shader controller 310 may identify which processing elements of the shader array are available to process new workloads, and shader controller 310 may send the new workloads to the available processing elements of shader array 320. Shader controller 310 may keep track of which workloads are being processed by the different processing elements of the shader array, enabling a plurality of threads to execute in parallel.

[0039] Shader array 320 may include texture consuming shader 321 and decompressing shader 322, which are representative of any number and type of shader processors which may be included in shader array 320. In various embodiments, shader array 320 may include an additional shader processor which may be configured to generate texture data procedurally. Generally speaking, procedural texture generation refers to the process of generating a texture algorithmically. In various embodiments this procedural generation of texture is performed

dynamically rather than in advance. Shader array 320 may be used for texture mapping and producing image data for a display device, among other tasks. As part of performing these operations, texture consuming shader 321 may issue a texture request to texture filter 330. The texture request may be for one or more portions (e.g., blocks, texels) of the texture. Texture filter 330 may generate a virtual address for the requested texture, and convey the virtual address with the request to cache 340. Cache 340 may store textures in the form of texel data associated with pixels. Some of the textures may be compressed, and some of the textures may be uncompressed.

[0040] After receiving the virtual address from texture filter 330, cache 340 may perform an address check against all known virtual address ranges to determine if the requested texture is stored in cache 340. If an uncompressed version of the requested texture is stored in cache 340, cache 340 may return the uncompressed version of the texture to texture filter 330. If the uncompressed version of the texture is not stored in cache 340, the attempted request may result in a cache miss. In response to a cache miss, decompressing shader 322 may be initiated for the purpose of decompressing a compressed version of the texture. In various embodiments, shader array 320 may receive a request from cache 340, or otherwise, to initiate a decompressing shader. Also in response to a cache miss, texture consuming shader 321 may pass the virtual address of the texture to decompressing shader 322. Resources for the decompressing shader program may be pre-allocated on decompressing shader 322 to decrease the shader start latency and simplify resource management. The request may be routed to a particular shader processor of shader array 320 based on the virtual address of the block being requested.

[0041] Cache 340 may be queried for a compressed version of the texture, and if the compressed version of the texture is stored in cache 340, the compressed version of the texture may be returned to decompressing shader 322. If the compressed version of the texture is not stored in cache 340, the compressed version of the texture may be retrieved from system memory or another location. Decompressing shader 322 may also receive additional tables, textures, and/or constants to facilitate the decompression operation. Decompressing shader 322 may decompress some additional compressed data necessary to decompress the requested texture. In the case of a JPEG-compressed texture, the texture may be transcoded from the original code to a new encoding scheme, and the new encoding scheme may be designed to make decompression more efficient. After decompressing shader 322 has received and decompressed the compressed version of the texture, texture consuming shader 321 may utilize the decompressed version of the texture for the appropriate rendering calculations. This process

may continue for a plurality of textures and/or portions of textures. In another embodiment, the functions described as being performed by texture filter 330 may be performed by shader array 320, and shader array 320 may be coupled directly to cache 340.

[0042] Cache 340 may utilize a table to determine the address to which a given virtual address maps for the compressed versions of textures stored in cache 340. In various embodiments, the table (or portions thereof) may be stored in cache 340 or elsewhere. In one embodiment, the table may map a virtual address to another address of the compressed version of a texture. The address to which the virtual address is mapped may or may not itself be a virtual address. Numerous options for the types of addressing schemes utilized are possible and are contemplated. The table may store an offset for each block of the compressed version of the texture, wherein the offset gives the location from the beginning of the compressed version of the texture to the block. In various embodiments, the table may facilitate random access to the blocks of one or more compressed textures. The cache logic of cache 340 may determine an address of a given block in response to a request for the compressed version of the block. The cache logic may use the table to determine an offset at which the desired block is stored within a page or fetch unit of the cache. The plurality of shaders of shader array 320 may also use the table to determine the offset of a requested block of a texture. In various embodiments, cache 340 may utilize a plurality of tables with mapping information on a plurality of textures.

[0043] After the texture data has been processed, shader array 320 may convey the image data to render unit 350. Render unit 350 may assign a specific number value that defines a unique color attribute for each pixel of an image frame. The number values may be passed to frame buffer 360 where they may be stored for use at the appropriate time, such as when they are rendered on display device 370.

[0044] On a subsequent operation, texture consuming shader 321 may be configured to perform the functions of a decompressing shader, and decompressing shader 322 may be configured to perform the functions of a texture consuming shader. Each shader processor of shader array 320 may be configured to perform a variety of functions depending on the requirements of the current operation.

[0045] In various embodiments, load balancing may be utilized to assign decompression tasks to underutilized shaders. Also, some space may be reserved in a number of compute units to allow decompression shaders to be launched on a number of compute units. Furthermore, multiple decompression requests may be packed into single instruction multiple data (SIMD) vectors. The SIMD vectors may facilitate the decompression of multiple blocks in one vector.

In one embodiment, 16 blocks may be decompressed in one vector, with one block per four lanes.

[0046] In various embodiments, graphics processing system 300 may enable on-the-fly procedural generation of texture data. One shader may generate on-the-fly texture data, and a second shader may utilize the generated texture data for rendering operations. A decompressing shader may access compressed data and another shader may be utilized to decompress additional data, such as one or more tables. Some of the compressed data may be compressed using a variety of compression techniques. In various embodiments, the decompressing shader may request data from the cache, and in response to a cache miss, another shader may be initiated to procedurally generate texture data.

[0047] Turning now to FIG. 4A, a block diagram of one embodiment of a data cache is shown. Cache 410 may contain portions of textures 420 and 430, which are representative of any number of portions of textures which may be stored in cache 410. Textures 420 and 430 may be compressed textures, while the plurality of textures stored in cache 410 may be a mix of compressed and uncompressed textures. Texture 420 may include blocks 422 and 423, which are representative of any number of blocks of texture 420. Texture 420 may also include table 421, which may map a virtual address space of texture 420 to an address space of compressed texture 420. Texture 430 may be organized similarly to texture 420. In another embodiment, table 421 may be stored separately from texture 420.

[0048] When a texture consuming shader requests a block of a texture from cache 410, and the request results in a cache miss, cache 410 may allocate cache line 440 for the requested block. Cache 410 may convey the address of the allocated cache line to a decompressing shader. After the decompressing shader has decompressed the compressed block corresponding to the requested block, the decompressing shader may be configured to write the decompressed block to cache line 440. Alternatively, the decompressing shader may write the decompressed block to various locations within cache 410. In response to the decompressing shader writing the decompressed block to cache line 440, the texture consuming shader may be configured to fetch the decompressed block from cache 410. The corresponding latency compensation queues may need to be extended to accommodate the larger latency resulting from the on-the-fly decompression of the compressed block.

[0049] After the decompressed version of the block has been written to cache line 440, cache 410 may store the compressed version of the block and the decompressed version of the block. In various embodiments, cache 410 may execute a retention policy that discards one of the

versions of the block in response to determining both versions are stored in cache 410. In one embodiment, the decompressed version of the block may be discarded after it has been fetched by the texture consuming shader. In another embodiment, the compressed version of the block may be discarded after the decompressed version of the block has been written to cache 410. In a further embodiment, both the compressed and decompressed version of the block may be maintained in cache 410 for an extended period of time.

[0050] In response to a request for an uncompressed version of a block of a texture, cache 410 may determine that the uncompressed version is not stored in cache 410. In various embodiments, in response to such a determination, cache 410 may automatically search for the compressed version of the block. If the compressed version of the block is stored in cache 410, cache 410 may notify a shader or other processing unit and/or cache 410 may convey the compressed version of the block to the shader or other processing unit.

[0051] In some embodiments, in response to a cache miss on a request for an uncompressed block, a separate software thread may be started, and the thread may initiate a decompressing shader. The texture consuming shader may convey the virtual address of the block to the decompressing shader. In various embodiments, when the shader finishes the decompression task, the decompressing shader may convey the uncompressed block(s) to the cache. In other embodiments, when the decompressing shader finishes the decompression operation, the decompressing shader may convey the shader output to the texture consuming shader.

[0052] Referring now to FIG. 4B, a block diagram of one embodiment of a block mapping table is shown. Table 421 may store mapping information for the plurality of blocks of texture 420 (of FIG. 4A). In various embodiments, table 421 may be organized in a variety of ways with other types of information in addition to what is illustrated in FIG. 4B. For example, in one embodiment, table 421 may include a DC coefficient value for each block of texture 420.

[0053] Table 421 may map the virtual address space of texture 420 to the physical address space of compressed texture 420 (of FIG. 4A). A decompressing shader (not shown) may fetch or otherwise receive one or more blocks of texture 420 from cache 410, and the decompressing shader may determine the location and size of the compressed blocks from table 421. The size of a compressed block may be determined by calculating the difference between the starting physical addresses of two adjacent blocks. In other embodiments, additional data may be provided to indicate size and/or location information for blocks. Further, the decompression shader may obtain additional information from table 421, such as a DC coefficient value of each block.

[0054] In some embodiments, the texture may be organized according to superblocks. A superblock may be a set of 16 8x8 blocks, which is a tile of 32x32 pixels, for a total of 1024 pixels. The index table for the texture may include a table entry for each superblock, and each table entry may give the address of the start of each superblock. In one embodiment, this address may be the location of the superblock within the texture. In another embodiment, this address may be an offset from the start of the texture. Each entry may also include a 4-bit index of the first 8x8 block belonging to the superblock. In some embodiments, superblocks may not be aligned with 2 kilobit (Kb) boundaries of the cache. Each entry may also include a 16-bit mask. The 16-bit mask may include one bit per block indicating whether that block starts in the next 2 Kb word.

[0055] In some embodiments, the decompressing shader may transform the virtual address of the 8x8 block into the virtual address of a 32x32 superblock to calculate an entry number of the index table for lookup purposes. The decompressing shader may lookup the entry of the index table corresponding to the superblock. The index table may be processed by a shader in a similar manner as other textures. The entries of the index table may be cached and processed.

[0056] From each index table entry, the shader may obtain the base address, which may be a virtual address. The base address may be of the first fetch unit of the compressed superblock. The shader may also obtain the offset of the fetch unit containing the requested block which needs to be decompressed. The shader may also calculate if the block is compressed or not based on the address of the block. Certain address ranges may correspond to virtual addresses of uncompressed blocks, and other address ranges may correspond to physical addresses of compressed blocks. The shader may be able to distinguish between the different address ranges.

[0057] Referring now to FIG. 5, a block diagram of one embodiment of a virtual address space for an 8x8 block of texels is shown. Each texel may be mapped to a unique address within virtual address space 570. Texel 1 may be mapped to address 501, texel 2 may be mapped to address 502, and so on, for all 64 texels of 8x8 block 500. Block 500 may be a block within a compressed texture, and virtual address space 570 may be allocated for block 500 of the compressed texture. The texture may include a plurality of blocks in addition to block 500. Virtual address space 570 may also include a unique address for each texel of the plurality of blocks in the texture.

[0058] For purposes of illustration, it will be assumed that an uncompressed texel is a 32-bit value (4 sets of 8-bit values). Other sizes of uncompressed texels may also be utilized with the methods and mechanisms described herein. For example, an uncompressed texel with a 24-bit

value may be handled in a similar way. In various embodiments, a texture consuming shader may generate requests for individual texels. First, the shader may compute the virtual address of a texel. Then, the cache may be queried for the virtual address corresponding to the texel.

[0059] Turning now to FIG. 6, a block diagram of one embodiment of compressed data is shown. Data portion 605 may be a unit of fetch of the compressed data, and the size of data portion 605 may be based on the size of an uncompressed block. In one embodiment, a fetch unit may be of size 2 Kb. In other embodiments, a fetch unit may be any of various sizes. A plurality of compressed blocks may be packed into a fetch unit. In one embodiment, the maximum number of blocks that may be packed into a fetch unit may be assumed to be 16. In other embodiments, other numbers of blocks may be packed into a fetch unit. For one type of cache access scheme, it may be assumed that the data of the blocks do not cross boundaries of fetch units.

[0060] A block may be the smallest decodable unit of a compression format, such as JPEG. For JPEG, the block is an 8x8 pixel tile (with 64 pixels). When a texture is compressed, and a block of the texture requested by a shader needs to be decompressed, a cache line may be allocated in the cache for the block. In one embodiment, the cache line size may be 2 Kb to store an entire uncompressed block (32 bits * 64 = 2 Kb). In other embodiments, the cache line size may be any of various sizes.

[0061] If a fetch unit contains an uncompressed block, then only one block may fit in the fetch unit. For a fetch unit containing compressed blocks, the fetch unit may also include a 176-bit header. The fetch unit may be assumed to have a capacity of 16 blocks. The header may include 16 11-bit offset values to indicate the locations of the compressed blocks within the fetch unit. The offsets reference the starting bit positions of the blocks. In other embodiments, there may be a variable number of offset indicators in the header.

[0062] As shown in FIG. 6, data portion 605 may include header 610 and blocks 611-626. Blocks 611-626 may be sixteen different blocks of a compressed texture. Header 610 may include offsets 631-646. Each offset may be an 11-bit offset value corresponding to the location of the corresponding block within data portion 605. In other embodiments, other bit-sizes of offset values may be utilized. Offset 631 may represent the starting address of block 611, offset 632 may represent the starting address of block 612, and so on. In some embodiments, there may be an additional offset indicating the last bit of the last block, to reduce unnecessary fetch from the cache.

[0063] In some embodiments, compressed 8x8 blocks of the texture may be packed and cross fetch unit boundaries. The corresponding information, showing that the block uses two fetch units, may be stored in an index table, and a decompressing shader may generate two fetches instead of one for blocks that cross fetch unit boundaries.

[0064] Turning now to FIG. 7, one embodiment of a method for decompressing a compressed block of a texture is shown. For purposes of discussion, the steps in this embodiment are shown in sequential order. It should be noted that in various embodiments of the method described below, one or more of the elements described may be performed concurrently, in a different order than shown, or may be omitted entirely. Other additional elements may also be performed as desired.

[0065] The method 700 starts in block 705, and then in block 710, a first shader of a plurality of shaders may determine the need for a block of a texture as part of the rendering operations for an image. The first shader may be a texture consuming shader. Next, the first shader may calculate the virtual address of the block (block 715). The first shader may have an uncompressed view of the texture, corresponding to the uncompressed version of the texture, and the virtual address may correspond to the location of the requested block within the uncompressed view. After block 715, the first shader may request the block from the cache and convey the virtual address with the request (block 720). Next, the cache may determine if an uncompressed version of the block is stored in the cache (conditional block 725). If the uncompressed version of the block is stored in the cache, the first shader may receive the uncompressed version of the block from the cache and process the block (block 770).

[0066] If the uncompressed version of the block is not stored in the cache, a second shader of the plurality of shaders may be initiated as a decompressing shader (block 730). The resources for the decompressing shader may be pre-allocated on one or more shader processors to decrease the shader start latency and simplify resource management. Also, the virtual address of the requested block may be passed from the first shader to the second shader. Next, a cache line may be allocated for the requested block (block 735). Then, the cache may determine if a compressed version of the block is stored in the cache (conditional block 740). In various embodiments, the cache may make this determination in response to a request by the second shader for the compressed version of the block. In other embodiments, the cache may make this determination automatically in response to determining the uncompressed version of the block is not stored in the cache (conditional block 725).

[0067] If the compressed version of the block is stored in the cache (conditional block 740), then the cache and/or second shader may determine the location and size of the compressed version of the block from the table (block 750). If the compressed version of the block is not stored in the cache (conditional block 740), then the compressed version of the block may be fetched (e.g., from local or system memory) and stored in the cache (block 745). Fetching the compressed version of the block from system memory may entail fetching the entire compressed texture or some portion of the texture. The cache may be configured to utilize a table which maps the virtual address space of an uncompressed version of the texture to an address space of a compressed version of the texture. The cache and/or second shader may determine the location and size of the compressed version of the block from the table (block 750). The table may also contain additional information, such as the value of the DC coefficient of a compressed version of each block of the texture. After block 750, the compressed version of the block may be conveyed to the second shader from the cache (block 755).

[0068] In another embodiment, if the compressed version of the block is not in the cache (conditional block 740), steps 745, 750, and 755 may be replaced with alternate steps. In the alternate steps, the compressed version of the block may be fetched from system memory and provided directly to the second shader. These alternate steps may be more efficient than having the second shader receive the compressed version of the block from the cache. In a further embodiment, the compressed version of the block may be fetched from system memory and provided directly to the second shader while also being written to the cache.

[0069] After the second shader receives the compressed version of the block (block 755), the second shader may decompress the compressed version of the block (block 760). Next, the second shader may write the decompressed version of the block to the cache (block 765). Then, the first shader may receive the decompressed version of the block from the cache and process the block as part of the rendering operations for the current image (block 770). After block 770, the method may end in block 775. Method 700 may be repeated for a plurality of blocks from a plurality of textures.

[0070] Although the features and elements are described in the example embodiments in particular combinations, each feature or element can be used alone without the other features and elements of the example embodiments or in various combinations with or without other features and elements. The present invention may be implemented in a computer program or firmware tangibly embodied in a non-transitory computer-readable storage medium having machine readable instructions for execution by a machine, a processor, and/or any general

purpose computer for use with or by any non-volatile memory device. The computer-readable storage medium may contain program instructions which are operable to enable the functions, methods, and operations described in this specification. Suitable processors include, by way of example, both general and special purpose processors.

[0071] Typically, a processor will receive instructions and data from a read only memory (ROM), a RAM, and/or a storage device having stored software or firmware. Storage devices suitable for embodying computer program instructions and data include all forms of non-volatile memory, including by way of example semiconductor memory devices, read only memories (ROMs), magnetic media such as internal hard disks and removable disks, magneto-optical media, and optical media such as CD-ROM disks and digital versatile disks (DVDs).

[0072] The above described embodiments may be designed in software using a hardware description language (HDL) such as Verilog or VHDL. The HDL-design may model the behavior of an electronic system, and the design may be synthesized and ultimately fabricated into a hardware device. In addition, the HDL-design may be stored in a computer product and loaded into a computer system prior to hardware manufacture.

[0073] Types of hardware components, processors, or machines which may be used by or in conjunction with the present invention include Application Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs), microprocessors, or any integrated circuit. Such processors may be manufactured by configuring a manufacturing process using the results of processed hardware description language (HDL) instructions (such instructions capable of being stored on a computer readable media). The results of such processing may be maskworks that are then used in a semiconductor manufacturing process to manufacture a processor which implements aspects of the methods and mechanisms described herein.

[0074] Software instructions, such as those used to implement image rendering calculations and shader tasks, may be stored on a computer-readable storage medium. A computer-readable storage medium may include any mechanism for storing information in a form (e.g., software, processing application) readable by a machine (e.g., a computer). The computer-readable storage medium may include, but is not limited to, magnetic or optical media (e.g., disk (fixed or removable), tape, CD-ROM, DVD-ROM, CD-R, CD-RW, DVD-R, DVD-RW, or Blu-Ray), RAM (e.g., synchronous dynamic RAM (SDRAM), double data rate (DDR, DDR2, DDR3, etc.) SDRAM, low-power DDR (LPDDR2, etc.) SDRAM, Rambus DRAM (RDRAM), static RAM (SRAM)), ROM, non-volatile memory (e.g. Flash memory) accessible via a peripheral

interface such as the USB interface, micro-electro-mechanical systems (MEMS), and storage media accessible via a communication medium such as a network and/or a wireless link.

[0075] Although several embodiments of approaches have been shown and described, it will be apparent to those of ordinary skill in the art that a number of changes, modifications, or alterations to the approaches as described may be made. Changes, modifications, and alterations should therefore be seen as within the scope of the methods and mechanisms described herein. It should also be emphasized that the above-described embodiments are only non-limiting examples of implementations.

WHAT IS CLAIMED IS

1. An apparatus comprising:
   a first shader; and
   a second shader;
   wherein said second shader is configured to decompress a variable rate compressed texture block for use by the first shader.

2. The apparatus as recited in claim 1, wherein the second shader is configured to decompress the variable rate compressed texture block in response to a request by the first shader for a texture block that corresponds to the variable rate compressed texture block.

3. The apparatus as recited in claim 2, wherein the request by the first shader for the texture block that corresponds to the variable rate compressed texture block is a request to a memory for an uncompressed version of the variable rate compressed texture block.

4. The apparatus as recited in claim 3, wherein the second shader is configured to decompress the variable rate compressed texture block in further response to a determination that the uncompressed version of the variable rate compressed texture block is not in the memory.

5. The apparatus as recited in claim 3, wherein prior to decompressing the variable rate compressed texture block, the second shader is configured to receive the variable rate compressed texture block from the memory.

6. The apparatus of claim 1, wherein decompressing the variable rate compressed texture block is performed by the second shader executing a decompression program.

7. The apparatus of claim 2, wherein said memory comprises an on-chip memory configured to store data retrieved from an off-chip system memory.

8. The apparatus as recited in claim 2, further comprising a table which maps a virtual address space of an uncompressed version of the variable rate compressed texture block to an address space of the variable rate compressed texture block.

9. The apparatus as recited in claim 8, wherein the first shader is further configured to:

calculate a virtual address of the uncompressed version of the texture block within an uncompressed version of a corresponding texture, prior to requesting the texture block from the memory; and

convey the virtual address of the uncompressed version of the texture block to the memory with the request.

10. The apparatus as recited in claim 7, wherein in response to determining the uncompressed version of the texture block is not in the on-chip memory, storage is allocated in the on-chip memory for the uncompressed version of the texture block.

11. The apparatus as recited in claim 1, wherein the plurality of shaders comprises a shader configured to generate texture data procedurally.

12. A method for decompressing texture data, the method comprising:

a first shader requesting a texture block; and

a second shader decompressing a variable rate compressed texture block for use by the first shader.

13. The method as recited in claim 12, further comprising the second shader decompressing the variable rate compressed texture block in response to a request by the first shader for an uncompressed version of the variable rate texture block from a memory.

14. The method as recited in claim 13, further comprising the second shader decompressing the variable rate compressed texture block in further response to a determination that the uncompressed version of the variable rate compressed texture block is not in the memory.

15. The method as recited in claim 14, wherein prior to decompressing the variable rate compressed texture block, the method comprises the second shader receiving the variable rate compressed texture block from the memory.

16. The method of claim 12, further comprising decompressing the variable rate compressed texture block by the second shader using a decompression program.

17. The method of claim 13, wherein said memory comprises an on-chip memory configured to store data retrieved from an off-chip system memory.

18. A computer readable storage medium comprising program instructions to decompress texture data, wherein when executed the program instructions are operable to:

enable a first shader to request a texture block; and

enable a second shader to decompress a variable rate compressed texture block for use by the first shader.

19. The computer readable storage medium as recited in claim 18, wherein the program instructions are further operable to enable the second shader to decompress the variable rate compressed texture block in response to a request by the first shader for an uncompressed version of the variable rate texture block from a memory.

20. The computer readable storage medium as recited in claim 19, wherein the program instructions are further operable to enable the second shader to decompress the variable rate compressed texture block in further response to a determination that the uncompressed version of the variable rate compressed texture block is not in the memory.

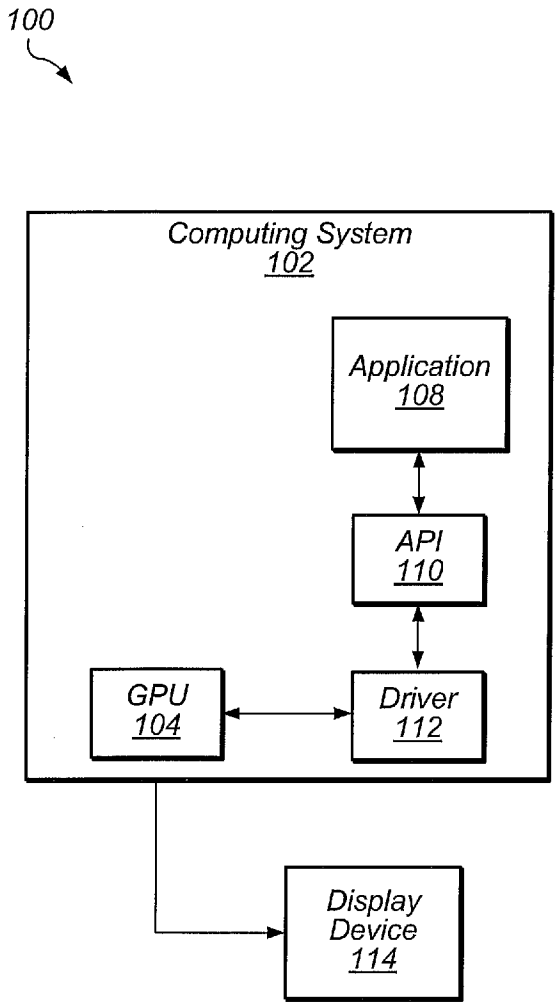
100
Computing System
102
Application
108
API
110
GPU
104
Driver
112
Display
Device
114

FIG. 1

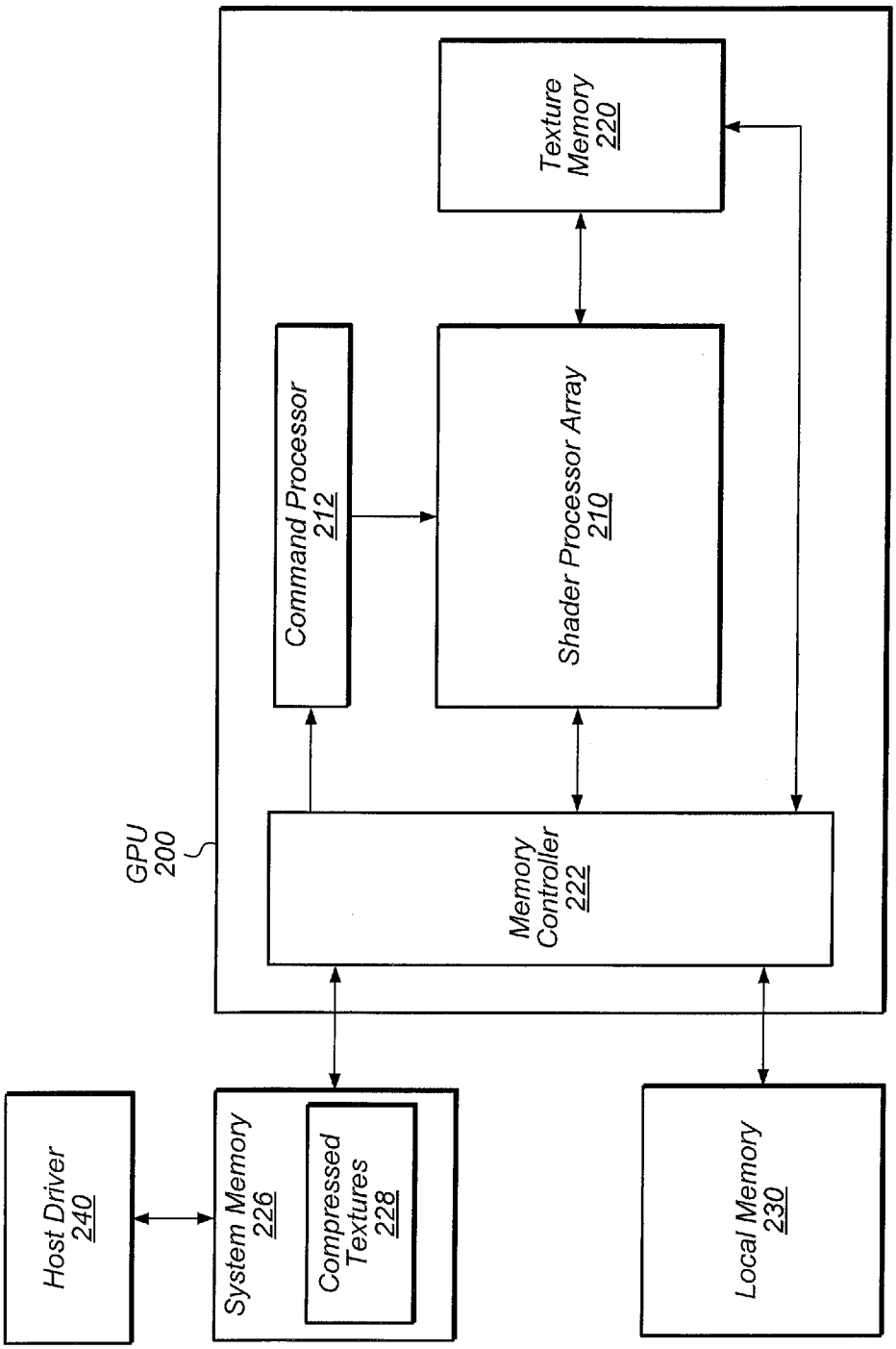
Texture Memory 220
Command Processor 212
Shader Processor Array 210
GPU 200
Memory Controller 222
Host Driver 240
System Memory 226
Compressed Textures 228
Local Memory 230

FIG. 2

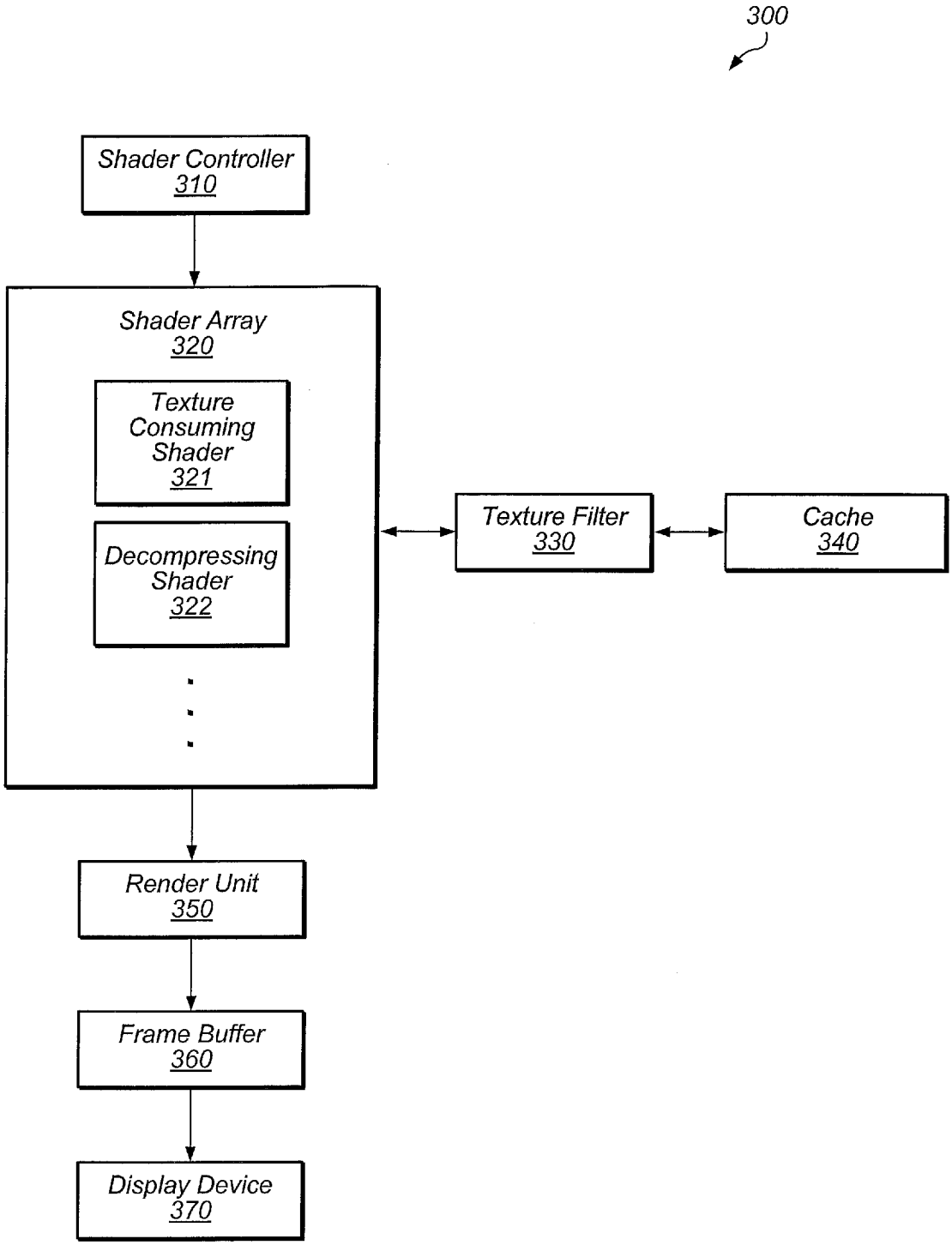
300
Shader Controller
310
Shader Array
320
Texture Consuming Shader
321
Decompressing Shader
322
Texture Filter
330
Cache
340
Render Unit
350
Frame Buffer
360
Display Device
370

FIG. 3

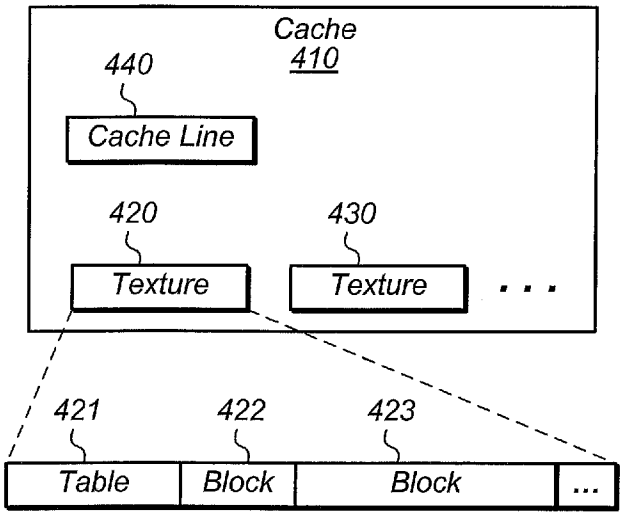

FIG. 4A

Table
421

Virtual Address	Physical Address	Block
0x000	0x100	422
0x800	0x200	423
0x2000	0x800	
⋮	⋮	⋮

FIG. 4B

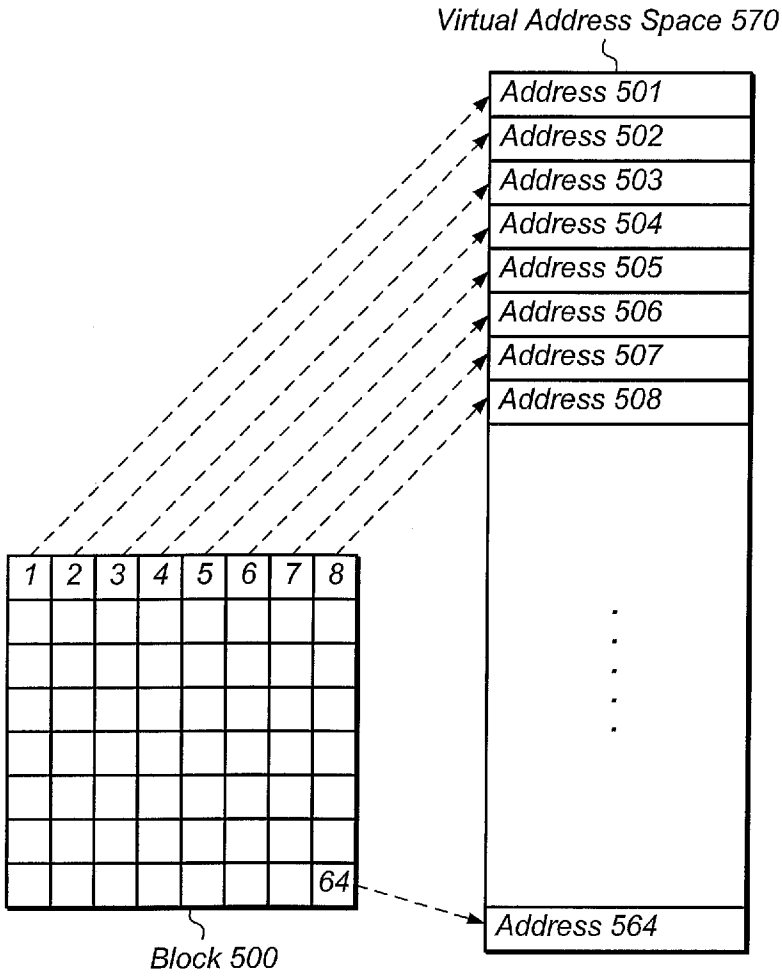
Virtual Address Space 570
Address 501
Address 502
Address 503
Address 504
Address 505
Address 506
Address 507
Address 508
Address 564
1
2
3
4
5
6
7
8
64
Block 500

FIG. 5

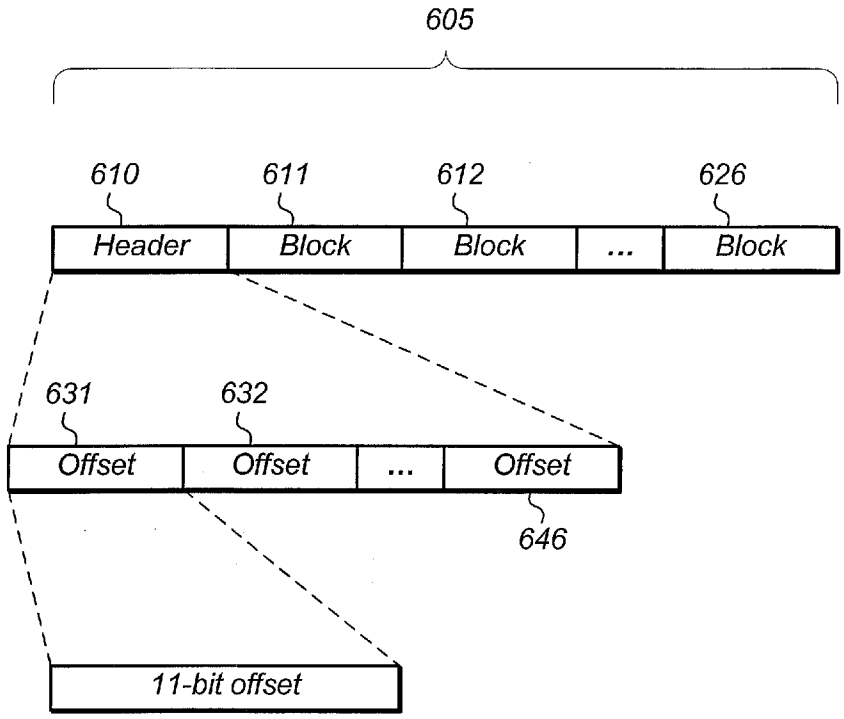
605
610
611
612
626
Header
Block
Block
...
Block
631
632
Offset
Offset
...
Offset
646
11-bit offset

FIG. 6

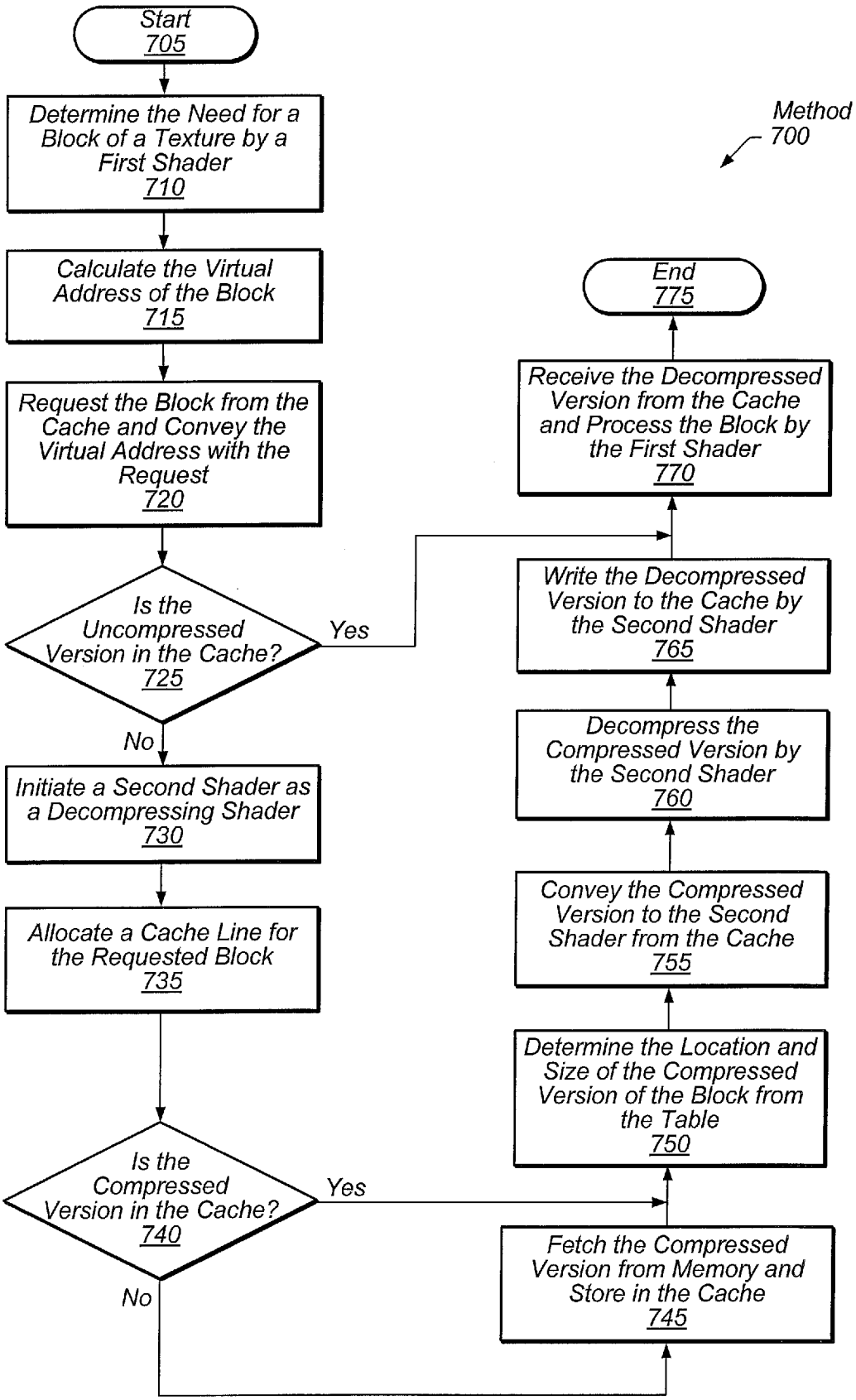
Start
705
Determine the Need for a Block of a Texture by a First Shader
710
Calculate the Virtual Address of the Block
715
Request the Block from the Cache and Convey the Virtual Address with the Request
720
Is the Uncompressed Version in the Cache?
725
Yes
No
Initiate a Second Shader as a Decompressing Shader
730
Allocate a Cache Line for the Requested Block
735
Is the Compressed Version in the Cache?
740
Yes
No
Fetch the Compressed Version from Memory and Store in the Cache
745
Determine the Location and Size of the Compressed Version of the Block from the Table
750
Convey the Compressed Version to the Second Shader from the Cache
755
Decompress the Compressed Version by the Second Shader
760
Write the Decompressed Version to the Cache by the Second Shader
765
Receive the Decompressed Version from the Cache and Process the Block by the First Shader
770
End
775
Method
700

FIG. 7

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2012/042442

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06T9/00 G06T15/00
ADD.

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06T

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal, INSPEC, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 6 959 110 B1 (DANSKIN JOHN M [US] ET AL) 25 October 2005 (2005-10-25) abstract figures 1,2,4 column 2, line 5 - line 32 column 4, line 22 - column 5, line 3	1-3,6, 11-13, 15-19
X	US 6 243 081 B1 (GORIS ANDREW C [US] ET AL) 5 June 2001 (2001-06-05) abstract figures 1-3, 7 column 1, line 59 - column 3, line 12 column 4, line 58 - column 6, line 7 column 7, line 48 - line 61	1-20

-/--

[X] Further documents are listed in the continuation of Box C.

[X] See patent family annex.

* Special categories of cited documents :

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

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

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

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

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

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

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

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

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
22 October 2012	31/10/2012

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Kontopodis, D

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2012/042442

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 6 452 602 B1 (MOREIN STEPHEN L [US]) 17 September 2002 (2002-09-17) abstract figures 1,2 column 1, line 11 - line 53 column 6, line 52 - column 7, line 40	1-7, 10-20
X	US 6 108 460 A (RICH HENRY H [US]) 22 August 2000 (2000-08-22) figures 2, 10 column 20, line 48 - column 23, line 18	1-7, 11-20
A	AKENINE-MÖLLER T ET AL: "Graphics Processing Units for Handhelds", PROCEEDINGS OF THE IEEE, IEEE. NEW YORK, US, vol. 96, no. 5, 1 May 2008 (2008-05-01), pages 779-789, XP011207044, ISSN: 0018-9219, DOI: 10.1109/JPROC.2008.917719 abstract page 780, right-hand column, last paragraph - page 781, right-hand column, line 2 section V.B figures 4,9	1-20
A	JEONG-HO WOO ET AL: "A 195 mW, 9.1 MVertices/s Fully Programmable 3-D Graphics Processor for Low-Power Mobile Devices", IEEE JOURNAL OF SOLID-STATE CIRCUITS, IEEE SERVICE CENTER, PISCATAWAY, NJ, USA, vol. 43, no. 11, 1 November 2008 (2008-11-01), pages 2370-2380, XP011238698, ISSN: 0018-9200, DOI: 10.1109/JSSC.2008.2004525 section III.A figure 5	1-20
A	"6.3 Procedural Texturing" In: Akenine-Möller T; Haines E; Hoffman N: "Real-Time Rendering, 3rd edition", 31 July 2008 (2008-07-31), A K Peters, XP002685684, ISBN: 1568814240 pages 178-180, the whole document	10

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2012/042442

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 6959110 B1	25-10-2005	NONE	
US 6243081 B1	05-06-2001	JP 3453088 B2	06-10-2003
		JP 2000105839 A	11-04-2000
		US 6243081 B1	05-06-2001
US 6452602 B1	17-09-2002	NONE	
US 6108460 A	22-08-2000	NONE	