(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 3 101 650 A1

(12) **EUROPEAN PATENT APPLICATION**

(43) Date of publication:
07.12.2016 Bulletin 2016/49

(51) Int Cl.:
G09G 5/36 *(2006.01)*

(21) Application number: **16172382.0**

(22) Date of filing: **01.06.2016**

(84) Designated Contracting States:
AL AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HR HU IE IS IT LI LT LU LV MC MK MT NL NO PL PT RO RS SE SI SK SM TR
Designated Extension States:
BA ME
Designated Validation States:
MA MD

(30) Priority: **04.06.2015 US 201562171071 P**
28.12.2015 US 201514981395
04.02.2016 KR 20160014084

(71) Applicant: **Samsung Electronics Co., Ltd.**
Gyeonggi-do 16677 (KR)

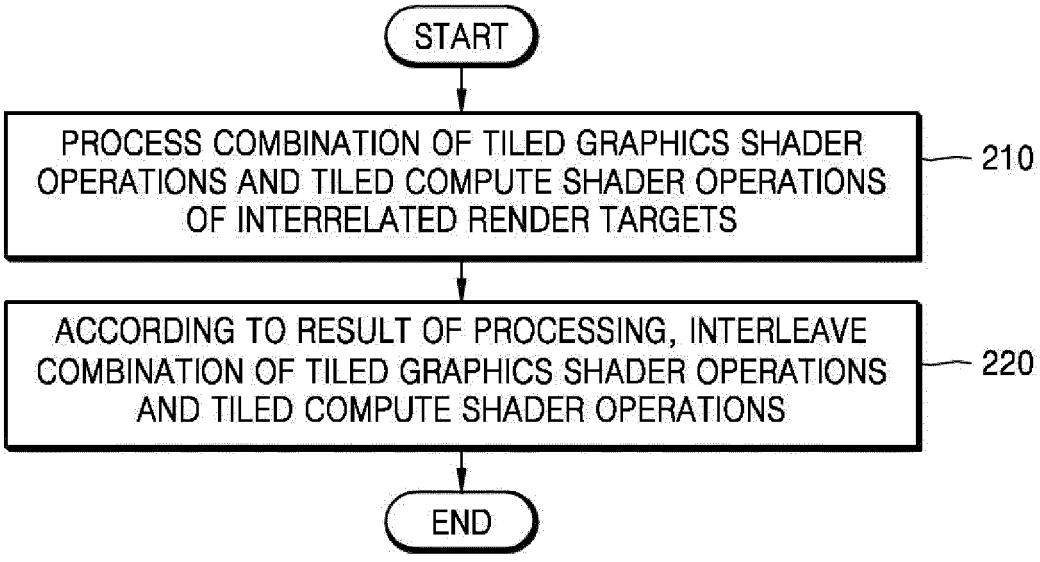
(72) Inventors:
- **BROTHERS, John**
  Mountain View, CA 94043 (US)
- **GOLAS, Abhinav**
  Mountain View, CA 94043 (US)
- **LEE, Joohoon**
  Mountain View, CA 94043 (US)

(74) Representative: **Portch, Daniel**
Elkington and Fife LLP
Prospect House
8 Pembroke Road
Sevenoaks, Kent TN13 1XR (GB)

(54) **METHOD AND APPARATUS FOR PERFORMING INTERLEAVING**

(57) A graphics system interleaves a combination of graphics shader operations and compute shader operations. A set of application programming interface (API) calls is analyzed so as to determine dependencies and to identify candidates for interleaving. A compute shader is adapted to have a tiled access pattern. The interleaving is scheduled so as to reduce requests to access an external memory so as to perform reads and writes of intermediate data.

FIG. 2



EP 3 101 650 A1

Description

FIELD OF THE INVENTION

[0001] The present disclosure relates to rendering of graphical images in which a graphics shader and a compute shader are utilized, and more particularly, to a method of interleaving graphics shader operations and compute shader operations.

BACKGROUND OF THE INVENTION

[0002] Graphical images are often generated through several steps. For example, an image may be generated and then may be read to generate another image through a sequence of render targets (RTs). An RT is an intermediate memory surface to which a three-dimensional (3D) image is rendered. A sequence of steps may be performed to generate an RT "A" and then to read the RT A so as to generate an RT "B". For example, in a first step, lighting parameters are written to a G-buffer so as to render an image and, in a second step, a lit image may be rendered by reading the G-buffer and doing light-related calculations. A sequence of operations (i.e., the sequence of steps) may be performed on different RTs before a final output image is generated.

[0003] However, these render target steps require a graphics processing unit (GPU) to access an external memory. In a case where a graphics application generates an intermediate image A and then an image A is read so as to generate an image B, an image size (e.g., 1920x3080 pixels) is commonly applied thereto, and each pixel is 4 bytes (RGBA8888 format), the intermediate image A has to be written to the external memory if a cache cannot store data of 8 megabytes (MBs).

[0004] Thus, a graphics processor renders all of a first RT (e.g., the RT A), and then reads a result of the rendering from the external memory so as to generate a second RT (e.g., the RT B).

SUMMARY OF THE INVENTION

[0005] Provided are methods and apparatuses for performing interleaving. Also provided are a non-transitory computer-readable recording medium having recorded thereon a program for executing the method, by using a computer.

[0006] Additional aspects will be set forth in part in the description which follows and, in part, will be apparent from the description, or may be learned by practice of the presented embodiments.

[0007] According to an aspect of an embodiment, a method of performing interleaving may include processing a combination of tiled graphics shader operations and tiled compute shader operations of interdependent (or interrelated) render targets; and interleaving the combination of the tiled graphics shader operations and the tiled compute shader operations, according to a result of the processing.

[0008] According to another aspect of an embodiment, a method performed in a graphics processing system may include recompiling a compute shader so as to have a tiled access pattern; and interleaving processing of a graphics shader and the recompiled compute shader for a set of interdependent images, wherein the interleaving is performed on a tile-by-tile basis for the interdependent images.

[0009] According to another aspect of an embodiment, a non-transitory computer-readable recording medium having recorded thereon a program which, when executed by using a processor, performs a method including determining dependencies of graphics shader operations and compute shader operations of a set of interdependent render target operations; and scheduling an interleaved order of tile processing on interleaved graphics shader operations and compute shader operations so as to reduce traffic to an external memory of a graphics system by maintaining at least one subset of intermediate tile processing computations of the interleaved graphics shader operations and compute shader operations in an on-chip memory of a graphics processing unit (GPU).

BRIEF DESCRIPTION OF THE EMBODIMENTS

[0010] These and/or other aspects will become apparent and more readily appreciated from the following description of the embodiments, taken in conjunction with the accompanying drawings in which:

FIG. 1 illustrates configuration of an apparatus for performing interleaving, according to an embodiment;

FIG. 2 is a flowchart for describing a method of performing interleaving, according to an embodiment;

FIG. 3 illustrates configuration of a graphics processing system, according to an embodiment;

FIGS. 4A through 4C illustrate examples of a directed acyclic graph (DAG);

FIG. 5 illustrates an example in which intermediate results are maintained in an on-chip memory;

FIG. 6 illustrates a flowchart for describing a method of interleaving graphics and compute operations, according to an embodiment;

FIG. 7 illustrates a flowchart for describing a method of interleaving graphics and compute operations, according to another embodiment;

FIG. 8 shows an example of analyzing a sequence of Application Programming Interface (API) calls (e.g., OpenGL™ API calls) for interleaving graphics

calls followed by computing calls for a graphics shader and a compute shader; and

FIG. 9 illustrates an example in which graphics and compute calls are interleaved by an interleaved scheduling module, according to an embodiment.

DETAILED DESCRIPTION OF THE DRAWINGS

[0011] All terms including descriptive or technical terms which are used herein should be construed as having meanings that are obvious to one of ordinary skill in the art. However, the terms may have different meanings according to an intention of one of ordinary skill in the art, precedent cases, or the appearance of new technologies. Also, some terms may be arbitrarily selected by the applicant, and in this case, the meaning of the selected terms will be described in detail in the detailed description of the invention. Thus, the terms used herein have to be defined based on the meaning of the terms together with the description throughout the specification.

[0012] Also, when a part "includes" or "comprises" an element, unless there is a particular description contrary thereto, the part can further include other elements, not excluding the other elements. In the following description, terms such as "unit" and "module" indicate a unit for processing at least one function or operation, wherein the unit and the block may be embodied as hardware or software or embodied by combining hardware and software.

[0013] The present disclosure will now be described more fully with reference to the accompanying drawings, in which one or more embodiments are shown. The present disclosure may, however, be embodied in many different forms and should not be construed as being limited to the embodiments set forth herein; rather, these embodiments are provided so that this present disclosure will be thorough and complete, and will fully convey the concept of the present disclosure to those of ordinary skill in the art.

[0014] Hereinafter, the present disclosure will be described in detail by explaining embodiments with reference to the attached drawings.

[0015] FIG. 1 illustrates configuration of an apparatus for performing interleaving, according to an embodiment.

[0016] Referring to FIG. 1, a memory 100 includes a driver 111, and a graphics processing unit (GPU) 120 includes a cache 121. The driver 111 may correspond to program instructions. Here, the program instructions provide a software interface with respect to the GPU 120, which allows software programs to access hardware configurations of the GPU 120 and to communicate data with the GPU 120. For example, the driver 111 may include program instructions and a GPU Application Programming Interface (GPU API) stored in the memory 110.

[0017] The GPU 120 may include not only the cache 121 but may also include programmable GPU hardware and at least one processor. In this regard, the cache 121 may be used as a tile buffer for buffering tiled data.

[0018] FIG. 2 is a flowchart for describing a method of performing interleaving, according to an embodiment.

[0019] In operation 210, the driver 111 processes a combination of tiled graphics shader operations and tiled compute shader operations of interrelated render targets (RTs).

[0020] In operation 220, according to a result of the processing, the driver 111 interleaves the combination of the tiled graphics shader operations and the tiled compute shader operations.

[0021] Hereinafter, with reference to FIGS. 3 through 9, examples in which the driver 111 and the GPU 320 perform interleaving are described in detail.

[0022] FIG. 3 illustrates configuration of a graphics processing system 300, according to an embodiment.

[0023] A memory 304 and a driver 308 shown in FIG. 3 may respectively correspond to the memory 110 and the driver 111 shown in FIG. 1. Also, a GPU 302 and an on-chip memory 360 shown in FIG. 3 may respectively correspond to the GPU 120 and the cache 121 shown in FIG. 1.

[0024] Referring to FIG. 3, the graphics processing system 300 includes a central processing unit (CPU) 301 and the memory 304. The driver 308 may be stored on the memory 308. For example, the driver 308 may correspond to program instructions. In this regard, the program instructions provide a software interface with respect to the GPU 302 that enables software programs to access hardware configurations of the GPU 302 and to communicate data with the GPU 302. For example, the software programs may include an operating system program, an application program, and/or the like software programs. The driver 308 may support physical and/or virtual hardware.

[0025] For example, the driver 308 includes program instructions stored in memory 304. The GPU 302 includes programmable GPU hardware 350, at least one processor 357, and an on-chip cache memory 360. Here, the on-chip cache memory 360 may be used as a tile buffer to buffer tiled data 362. The driver 308 includes a GPU API 306. Referring to FIG. 3, the GPU 302 is illustrated as a physical GPU component, however, the GPU 302 may correspond to a virtual GPU.

[0026] The GPU 302 may access an external memory 320. However, in some embodiments, in order to reduce memory traffic from and to the external memory 320, the external memory 320 may be an external dynamic random access memory (DRAM). For example, when a graphics processor renders a first RT, writes a result of the rendering to the external memory 320, and then reads the result of the rendering from the result of the rendering so to generate a second RT, these processes generate a lot of traffic from and to the external memory 320. In addition, the processes may include rendering with respect to unnecessary portions included in intermediate images.

[0027] For example, in order to maintain intermediate results or data on the on-chip cache memory 360 and to reduce the number of accessing the external memory 320, a support is provided to the driver 308 so as to interleave graphics shader operations and compute shader operations. Portions of the programmable GPU hardware 350 may be optionally modified with a hardware scheduling support 355 so as to optimize the execution of the interleaved scheduling in the GPU 302.

[0028] Graphics rendering operations are performed in a graphics pipeline including a fixed function logic, one or more graphics shaders, and compute shaders. The graphics shader is a program used to do shading. The graphics shader is a program that runs as a part of the graphics rendering operation. The graphics shader processes pixels, vertices, patches, or primitives. The compute shader is a program used for computing arbitrary information and provides more flexibilities. Here, the flexibility indicates a characteristic of software which may be easily changed according to different types of a machine and various demands from a user. With the addition of a compute shader that supports the OpenGL-ES™ 3.1 Application Programming Interface (API), benchmarks are moving image post-processing operations like motion blur, depth of field, scaling and filtering from graphics shaders to compute shaders. In the OpenGL™ standards, a compute shader is used in computing arbitrary information. In general, the compute shaders operate in an abstract space. However, a compute shader may have limitations on a work group size (a smallest amount of compute operations) and a local size. In this regard, the local size is defined by the number of invocations of the shader which may take place within each work group. The Open GL™ also defines rules for shared variables and memory barriers. The memory barriers ensure that all memory transactions before the barrier must complete before processing.

[0029] Compute shaders offer more flexibilities and features that may improve the efficiency of post-processing operations. For example, the compute shader gives more freedom in how data is accessed or written. However, the API does not guarantee memory consistency without the use of a global memory barrier (e.g., a g1MemoryBarrier).

[0030] A first analysis module 312 analyzes API graphics shader calls and compute shader calls. In order to construct a directed acyclic graph (DAG) that defines dependencies between images and tiles, a second analysis module 314 uses the first analysis module 312. Examples of the DAG are as illustrated in FIGS. 4A, 4B, and 4C. At run time, an API dependency graph (corresponding to the DAG) is built up in the driver 308 so as to detect cases where graphics calls and compute calls may be interleaved. As described in connection with FIG. 6, API calls are grouped to build a sequence of interleaved execution of graphics shader operations and compute shader operations,.

[0031] A scheduling module 330 determines a schedule of interleaved operations. For example, at the shader compilation time, the data access pattern of image load/store operations in the compute shader (or the compute shaders) are analyzed so as to determine whether the image load/store operations are candidates for interleaving. Certain types of the data access patterns have characteristics that are compatible with interleaving. For example, if the data access pattern is a statically known strided pattern that is static in a one-dimensional (1D) or two-dimensional (2D) space, the compute shader is a candidate for the interleaving. The strided pattern has a sequence of memory accesses with respect to addresses separated from each other by a stride length. A compute shader with a statically known strided pattern has an order of processing that has some general similarities in shape to a tile pattern, which facilitates converting a compute shader to have a tiled access. However, in principle, other types of data access patterns may be suitable candidates for interleaving. Operations of the scheduling module 330 are described in greater detail with reference to FIGS. 5, 6, and 9.

[0032] For example, the scheduling module 330 determines sets of interleaved tiled graphics shader operations and compute shader operations on interdependent RTs so as to reduce external memory traffic. The operation by the scheduling module 330 may be performed by maintaining at least some intermediate results of the interleaved operations in the on-chip memory 360. An example of the operation of scheduling the interleaved tiled graphics shader operations and the compute shader operation so as to reduce the external memory traffic is described in more detail with reference to FIG. 5.

[0033] For example, a support is provided in the driver 308 and a compiler (not shown) so as to automatically convert a compute shader (or compute shaders) to operate with a tiled access pattern by a conversion module 340. The compute shaders are converted to operate on tiles so as to generate a tiled equivalent version. Such conversion includes recompiling, by using the compiler, the compute shader to adapt the compute shader to perform a tiled memory access in a tile format compatible with that of the tiled memory access of a graphics shader. For example, such conversion includes redefining the workgroup dimension of the compute shader to be an integer divisor of a tile's width and height. Additionally, image load instructions may be replaced with tile buffer load instructions. If feasible, for the interleaved graphics shader computations and the compute shader operations, a removal module 342 may remove the memory barrier.

[0034] For example, the graphics processing system 300 automatically (e.g., without human intervention) interleaves the processing of graphics shader and compute shaders in order to reduce or eliminate writing the intermediate data to an external (off-chip) memory 320. The driver 308 re-orders the API calls and performs any necessary recompiling on the compute shaders so as to process, by using the driver 308 (and the support of a compiler

software stack), existing applications written to graphics APIs having the compute shaders.

[0035] FIGS. 4A through 4C illustrate examples of a DAG.

[0036] Referring to FIG. 4A, one embodiment of a data flow graph (or a data structure equivalent thereto) that is generated by the second analysis module 314 is shown. The data flow graph of FIG. 4A shows the immediate dependency between a set of two or more images, such as images A and B, and may also include other subsequent images (e.g., C, D). FIG. 4A corresponds to a DAG defining dependencies between different RTs. The interdependence of different RT images may be simple (e.g., the image B directly flows from the image A) or it may have more complex interdependent relationships (e.g., an image may depend on more than one image). More generally, as the image A and the image B are used to generate the image C as illustrated in FIG. 4B, one or more images might depend on several others at the same time. Thus, while there may be a simple sequence of one RT feeding (e.g., used as input) into another RT, the interdependence therebetween may be generally more complex. For example, two graphics shader RTs may be required so as to generate a compute shader RT. In addition, dependency on a tile level may also be complex. For example, referring to FIG. 4C, a tile in a given image B may depend on more than one tile in a previous image (e.g., an image A). In addition, there may be individual tiles in a given RT which are not used by any tiles in a subsequent RT.

[0037] For example, the first analysis module 312 and the second analysis module 314 of the driver 308 examine a stream of commands and then determine if the image of the RT B directly or indirectly (e.g., downstream) depends on an RT A. When it is determined that the RT B depends on the RT A, the driver 308 determines how many levels are required to follow the dependency graph so as to reduce external memory accesses. Following every level of the dependency graph may not be required in all cases to achieve a reduction in the external memory accesses. Moreover, following every level of a complex dependency graph consumes processing resources. Thus, the second analysis module 314 limits the number of levels that follow the dependency graph is followed. If the dependency graph is complex, a rule by which the number of levels which follow the dependency graph may be applied thereto. Afterward, the scheduling module 330 generates interleaving schedule so as to attempt to keep intermediate results in an on-chip cache memory (e.g., a tile buffer).

[0038] In addition, the DAG maps dependencies at a tile level between tiles rendered by graphics shader operations and tiles operating on by compute shaders. The generation of the DAG may include back-projecting each output tile to the input tile (or input tiles) required to generate them. For example, as illustrated in FIG. 4C, the image A may be used in generating the image B, which in turn may be used in generating the image C. There is a dependency of images and a dependency of tiles from the images. For example, as illustrated in FIG. 4C, a single tile in a given image may depend on more than one tile from another image.

[0039] FIG. 5 illustrates an example in which intermediate results are maintained in the on-chip memory 360.

[0040] FIG. 5 illustrates the example in which the intermediate computations of a sequence of operations are maintained in the on-chip cache memory 360 used as a tile buffer. In this example, the second analysis module 314 determines dependencies between an RT A, an RT B, an RT C, and an RT D. The scheduling module 330 determines a schedule to write tiles of the RT A to the on-chip cache memory 360 and then read them to the RT B. The scheduling of the processing of writing and reading the tiles may be selected to generate an interleaved schedule such as after a tile of an image A is produced and in stored in the on-chip cache memory 360. The tile is "immediately read back" so as to produce a tile of the RT B, thus, a memory bandwidth may be saved. It will be appreciated that "immediately read back" (also referred to as "directly consumed" or "consumed immediately") may include processing intervening operations between the storing of the tile of the image A in the on-chip memory 360 and the reading of the stored tile from the on-chip memory 360 so as to produce a tile of the RT B. For example, "immediately read back" may correspond to reading the tile of the image A from the on-chip memory 360 rather than reading the tile from an external memory 320 so as to produce the tile of the RT B.

[0041] The interleaved schedule may proceed in an order matching with the dependency analysis performed by the second analysis module 314. The dependency analysis may be selected to optimize use of the on-chip cache memory 360, so that the need to use the external memory 320 for the intermediate computations may be minimized. In addition, the dependency analysis performed by the second analysis module 314 may be used in the scheduling scheme for eliminating that processing is performed on tiles that do not contribute to a tile of a final image. By doing so, an unnecessary work may be eliminated. After a sequence of the intermediate operations is performed, the final image may be rendered and then may be sent to the external memory 320 or may be output for display. While it is desirable to completely maintain all intermediate results in the on-chip cache memory 360, in general, even maintaining a subset of intermediate data results in the on-chip cache memory 360 may be beneficial in reducing the number of external memory accesses that are required to perform graphics processing. Further, it is possible to reduce tile sizes so as to minimize the amount of intermediate data that must be maintained, which in effect, allows for an even finer grain of interleaving.

[0042] For example, interleaving may be performed at the tile level for graphics rendering and compute shader operations on a set of producer/consumer render targets (e.g., from images 1, 2, 3). Therefore, tile data of a RT1

may be rendered just before it is required so as to generate a corresponding tile of a RT2 with the tiled compute shader, which is rendered just before that portion of the RT2 is required so as to generate a RT3 and so on. Alternatively, a compute shader may generate input data required to render a graphics tile. For example, a compute shader may generate a position of particles, and only a particle affecting a specific XY region may be of interest for the graphics shader.

[0043] FIG. 6 illustrates a flowchart for describing a method of interleaving graphics and compute operations, according to an embodiment.

[0044] In operation 610, the first analysis module 312 processes API calls, and finds interleaving candidates. API call processing is deferred and queued to build a DAG by the second analysis module 314 that defines dependencies. In a tile based deferred rendering (TBDR) processing for a graphics only workload, the driver 308 uses certain events, such as a render target change or a memory barrier, in order to determine when to start processing deferred API calls. In a case where graphics and compute calls are mixed, the removal module 342 determines whether one or more memory barriers can be safely removed by interleaving the graphics with compute calls in a tiled manner (e.g., the memory barrier can be safely removed if the graphics processing for the tile ends before the compute processing occurs). For example, a determination of when interleaving is allowed is based on an analysis of memory access patterns of image load/store operations in the compute shader during a compilation time. If the memory access pattern is a statically-known strided pattern in a 1D or 2D space, then the compute shader becomes a candidate for interleaving.

[0045] In operation 620, the conversion module 340 recompiles the compute shader for tiled access patterns. For example, a workgroup dimension of the compute shader is redefined to be an integer divisor of a tile's width and height. Alternately or in addition, the tile's width and height may be modified. These modifications ensure that the range of access fits with the defined tile dimensions, so all data may be stored in the on-chip memory 360. For example, image load instructions may be replaced with tile buffer load instructions. This may include replacing instructions used to compute the target address to index in a tile buffer.

[0046] In operation 630, the scheduling module 330 groups API calls so as to build a sequence of interleaved execution of graphics and compute operations. For example, the scheduling module 330 determines whether interleaving is feasible. The scheduling module 330 removes the memory barrier, based on a result of determining whether interleaving is feasible.

[0047] In operation 640, the scheduling module 330 schedules the interleaved execution of each tile. The output of a graphics shader for a given tile is stored in the on-chip memory 360 (or, another on-chip memory such as a data register or a buffer). For example, data is immediately read back by a subsequent compute shader, so that the unnecessary external memory accesses are eliminated.

[0048] FIG. 7 illustrates a flowchart for describing a method of interleaving graphics and compute operations, according to another embodiment.

[0049] In operation 710, the second analysis module 314 determines dependencies between images and tiles in order to build a DAG. Candidates are identified in operation 720 for interleaving of graphics calls and compute calls on a tiled basis. For example, the scheduling module 330 identifies the candidates for interleaving. In operation 730, an associated candidate compute shader (or compute shaders) is adapted to be a tiled equivalent having a tiled access pattern, by the conversion module 340. For example, the conversion module 340 recompiles the compute shader for tiled access patterns. In operation 740, interleaving of the combined graphics shader and compute shader is scheduled by the scheduling module 330 in a tiled manner so as to reduce traffic to an external memory by maintaining at least some of the intermediate data result in the on-chip memory 360 (e.g., a tile buffer).

[0050] FIG. 8 shows an example of analyzing a sequence of API calls (e.g., OpenGL™ API calls) for interleaving graphics calls followed by computing calls for a graphics shader 830 and a compute shader 850.

[0051] The first analysis module 312 analyzes API graphics calls 810 and API compute calls 820 in order to determine whether or not graphics and compute calls can be interleaved. When the compute call 820 is encountered, the second analysis module 314 determines the graphics calls on which the compute call is dependent. A render target 840 is an intermediate memory surface to which a 3D image is rendered.

[0052] FIG. 9 illustrates an example in which graphics and compute calls are interleaved by an interleaved scheduling module, according to an embodiment.

[0053] A stride pattern that a compute shader 950 will need may be determined by monitoring memory access patterns of the load/store operations of the compute shader 950. A graphics shader 930 may generate tiles in an order by which the compute shader 950 consumes the tiles. For example, once a tile 910 has been produced, then the compute shader 950 processes the tile 910. While the compute shader 950 processes the tile 910, a graphics shader 930 produces a tile 920. Then, while the compute shader 950 processes the tile 920, the graphics shader 930 produces another tile. As a result, the processing of the tiles is interleaved between the graphics shader 930 and a compute shader 950.

[0054] For example, the rendering of a graphics frame may include a mixture of graphics and compute commands. A graphics rendering engine may mix graphics operations and compute operations for many stages of post-processing and graphics calculations. In particular, without interleaving some stages of processing, intermediate results that may be written to a memory (e.g., an external memory) may be generated and may be read

back later so as to generate a final resulting image or a next intermediate image. As an example scenario, the compute shader operations may include lighting calculations based on parameter data from a G-buffer (generated due to graphics rendering operations), a motion blur, a depth of field (DOF), compositing, and anti-aliasing. By tiling the compute shader operations, data production and consumption between graphics and compute operations may be interleaved at a tile level. By doing so, data may retain in the on-chip memory 360, and some graphics applications may eliminate some or even most traffic to an external memory. Therefore, power efficiency and performance may be significantly improved. In addition, an automatic (e.g., without human intervention) removal of explicit global memory barriers also improves performance.

[0055] In a number of graphics applications, a graphics shader is followed be a sequence of compute shader operations. However, in one embodiment, the compute shader may be followed by the graphics shader. In the embodiment, a similar analysis may be applied to a case where the compute shader writes an output to an external memory and the graphics shader loads the data from the external memory. The compute shader and graphics shader may be interleaved by recompiling the compute shader so as to output the data to the on-chip buffer. The graphics shader consumes the data from the on-chip buffer, and executes the data in a tiled manner.

[0056] In another embodiment, caches are used instead of a tile buffer for intermediate data. In certain architectures, a direct access to a low-level on-chip cache being used as the tile-buffer may not be possible due to a lack of appropriate data paths. In such a case, the nearest cache level may be used instead. However, the energy and performance benefits may be reduced depending on proximity of the cache to compute logic within the GPU. If a next-level cache is used, then several changes can be made. For example, image load instructions cannot be changed to tile buffer load instructions. However, hints or directives may be issued to the cache so as to ensure that the data remains resident in the cache and is not swapped out to an external DRAM memory until the dependent calculations are completed. The range of access is selected to fit within the cache capacity, so that all working data may stay in the cache. In addition, an output of a graphics shader for a given tile is stored in a nearest cache. The data is consumed immediately by a subsequent compute shader, and any unnecessary access to a DRAM or a lower cache level is eliminated.

[0057] As used herein, a module can correspond to a hardware component, a software component, or a combination thereof. For example, a module may include one or more processors (e.g., computer processors) and a data storage device including program instructions. The one or more processors may be configured by the instructions to function as a special purpose processor to perform one or more methods described herein. Software, hardware, and other modules may reside on servers, workstations, mobile devices, smart phones, wearable computers, personal computers, tablet computers, image data encoders, image data decoders, personal digital assistants (PDAs), video projectors, audio-visual receivers, displays (such as televisions (TVs)), digital cinema projectors, media reproducers, and other devices suitable for the purposes described herein. It is obvious to one of ordinary skill in the art that aspects of the system may be practiced as a standalone device or by a system of devices, such as, e.g., a client-server system.

[0058] A software (or program) component may include any non-transitory medium which carries a set of computer-readable instructions which, when executed by a processor, cause a processor to execute one or more methods described herein. The software component may be in any of a wide variety of forms. For example, the program component may include non-transitory media such as magnetic data storage media including floppy diskettes, hard disk drives, optical data storage media including CD ROMs, DVDs, electronic data storage media including ROMs, flash RAM, EPROMs, hardwired or preprogrammed chips (e.g., EEPROM semiconductor chips), nanotechnology memory, or the like. The computer-readable instructions on the program components may optionally be compressed or encrypted.

[0059] Example embodiments of software components may include (but are not limited to) firmware, middleware, operating system software, resident software, application software, microcode, and the like. Both hardware and software components may be centralized or distributed (or a combination thereof), in whole or in part, as known to those skilled in the art. For example, software components and other modules may be accessible via local memory, via a network, via a browser or other application in a distributed computing context or via other means suitable for the purposes described above.

[0060] While the present disclosure has been described in conjunction with specific embodiments, it will be understood that it is not intended to limit the present disclosure to the described embodiments. On the contrary, it is intended to cover alternatives, modifications, and equivalents as may be included within the scope of the present disclosure as defined by the appended claims. The present disclosure may be practiced without some or all of these specific details. In addition, well known features may not have been described in detail to avoid unnecessarily obscuring the present disclosure. In accordance with the present disclosure, the components, process steps, and/or data structures may be implemented using various types of operating systems, programming languages, computing platforms, computer programs, and/or computing devices. In addition, one of ordinary skill in the art will recognize that devices such as hardwired devices, field programmable gate arrays (FPGAs), application specific integrated circuits (ASICs), or the like, may also be used without departing from the scope of the present disclosure described herein. The present disclosure may also be tangibly embodied as a

set of computer instructions stored on a computer-readable medium such as a memory device.

Claims

1. A method of performing interleaving, the method comprising:

   processing a combination of tiled graphics shader operations and tiled compute shader operations of interdependent render targets; and
   interleaving the combination of the tiled graphics shader operations and the tiled compute shader operations, according to a result of the processing.

2. The method of claim 1, wherein the interleaving comprises determining a schedule of a sequence of interleaved and tiled operations, and
   wherein the interleaved and tiled operations are selected such that at least one intermediate data result of a first operation of the sequence is directly read from an on-chip memory by a second operation of the sequence.

3. The method of claim 1 or 2, wherein the interleaving comprises determining a schedule of a sequence of interleaved and tiled operations, and
   wherein the interleaved and tiled operations are selected such that traffic to an external memory associated with intermediate data results is reduced.

4. The method of any preceding claim, wherein a data access pattern of the tiled graphics shader operations and the tiled compute shader operations is analyzed so as to determine whether a global memory barrier is safely removed as a condition for the interleaving and for removing memory barriers.

5. The method of any preceding claim, wherein the interleaving is performed on a tile-by-tile basis and a compute shader operates with respect to a tiled access pattern.

6. The method of claim 5, wherein a workgroup dimension of the compute shader is redefined so as to be an integer divisor of a width and a height of a tile.

7. The method of claim 6, wherein the processing comprises replacing, for the compute shader, image load instructions with tile buffer load instructions.

8. The method of any preceding claim, wherein the processing of the combination comprises analyzing application programming interface (API) calls and grouping the API calls so as to build a sequence of interleaved execution of the tiled graphics shader operations and the tiled compute shader operations.

9. The method of any preceding claim, wherein the processing comprises generating a directed acyclic graph (DAG) of the interdependent render targets and interdependent tiles, and using the DAG so as to schedule interleaving of the tiled graphics shader operations and the tiled compute shader operations.

10. The method of any preceding claim, wherein the interleaving comprises grouping API calls so as to form a sequence of interleaved execution of the tiled graphics shader operations and the tiled compute shader operations.

11. The method of any preceding claim, wherein the processing comprises analyzing a data access pattern of load and store operations so as to determine candidates for the interleaving.

12. The method of claim 11, wherein the processing comprises identifying a data access pattern of a statically predetermined strided pattern, as a candidate for the interleaving.

13. A computer program comprising computer program codes means adapted to perform all of the steps of any preceding claim when said program is run on a computer.

14. Apparatus for performing interleaving, the apparatus comprising:

   a processor adapted to process a combination of tiled graphics shader operations and tiled compute shader operations of interdependent render targets; and
   an interleaving unit adapted to interleave the combination of the tiled graphics shader operations and the tiled compute shader operations, according to a result of the processing.

15. The apparatus of claim 14, wherein the interleaving unit is adapted to perform the interleaving on a tile-by-tile basis and wherein a compute shader operates with respect to a tiled access pattern.

FIG. 1

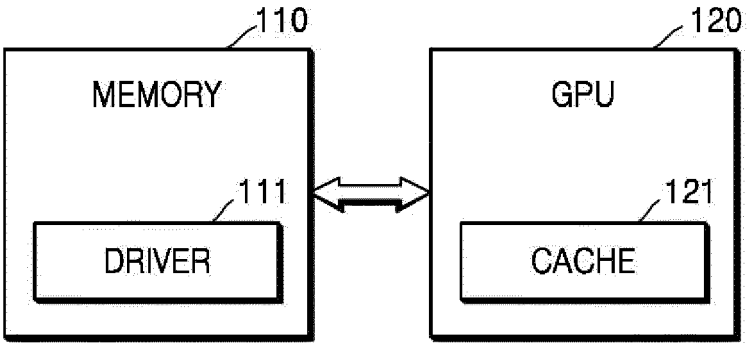

110
MEMORY
111
DRIVER
120
GPU
121
CACHE

FIG. 2

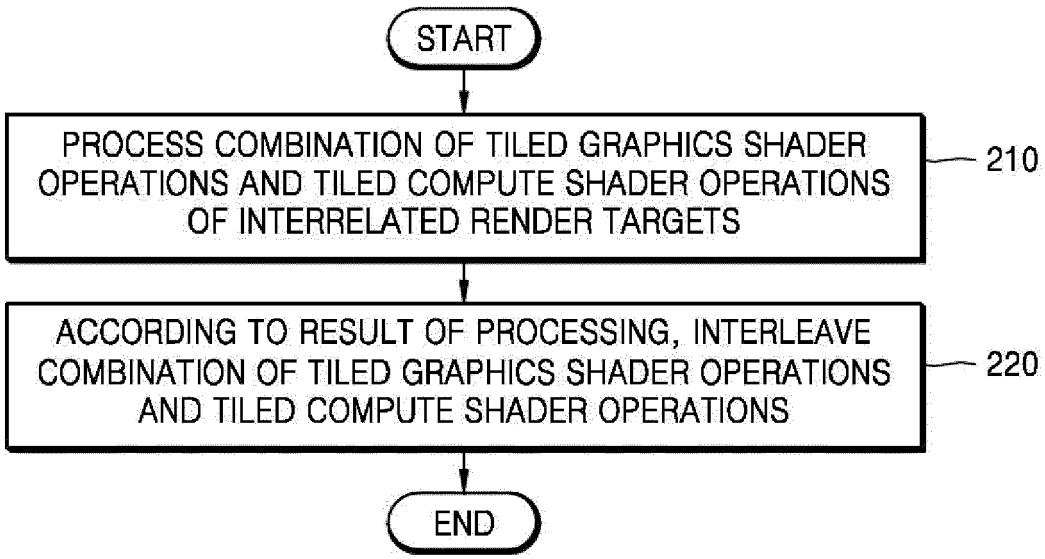

START
PROCESS COMBINATION OF TILED GRAPHICS SHADER OPERATIONS AND TILED COMPUTE SHADER OPERATIONS OF INTERRELATED RENDER TARGETS
210
ACCORDING TO RESULT OF PROCESSING, INTERLEAVE COMBINATION OF TILED GRAPHICS SHADER OPERATIONS AND TILED COMPUTE SHADER OPERATIONS
220
END

FIG. 3

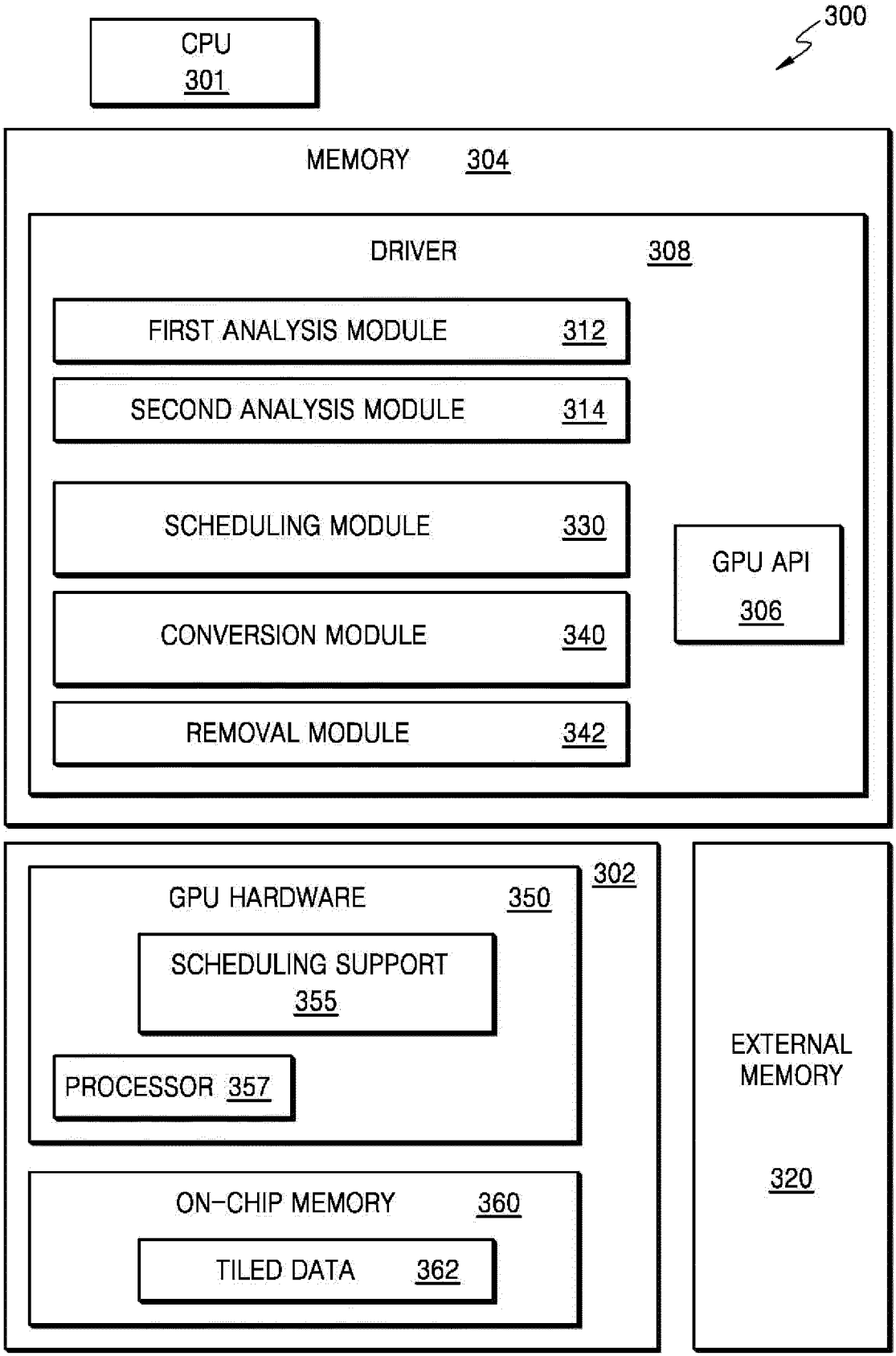

300
CPU
301
MEMORY
304
DRIVER
308
FIRST ANALYSIS MODULE
312
SECOND ANALYSIS MODULE
314
SCHEDULING MODULE
330
CONVERSION MODULE
340
REMOVAL MODULE
342
GPU API
306
302
GPU HARDWARE
350
SCHEDULING SUPPORT
355
PROCESSOR
357
ON-CHIP MEMORY
360
TILED DATA
362
EXTERNAL MEMORY
320

FIG. 4A

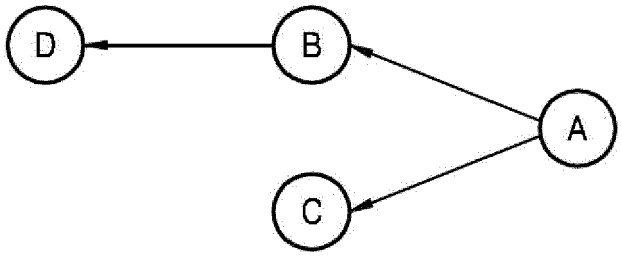


FIG. 4B

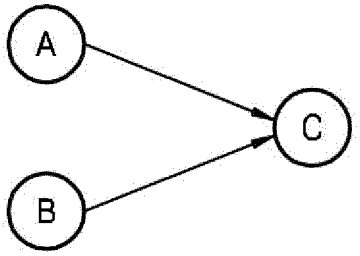


FIG. 4C

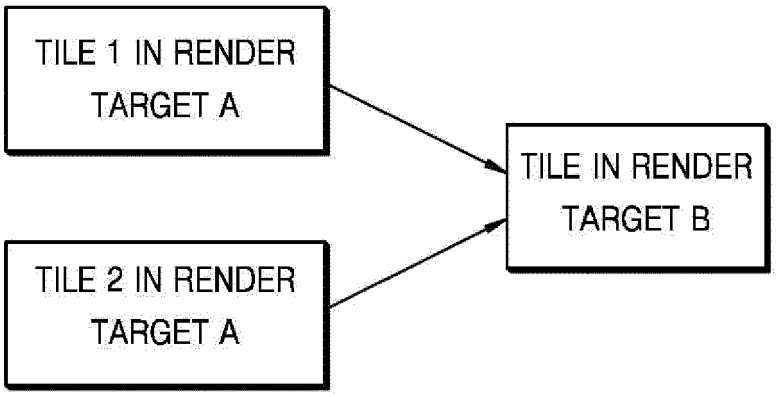


FIG. 5

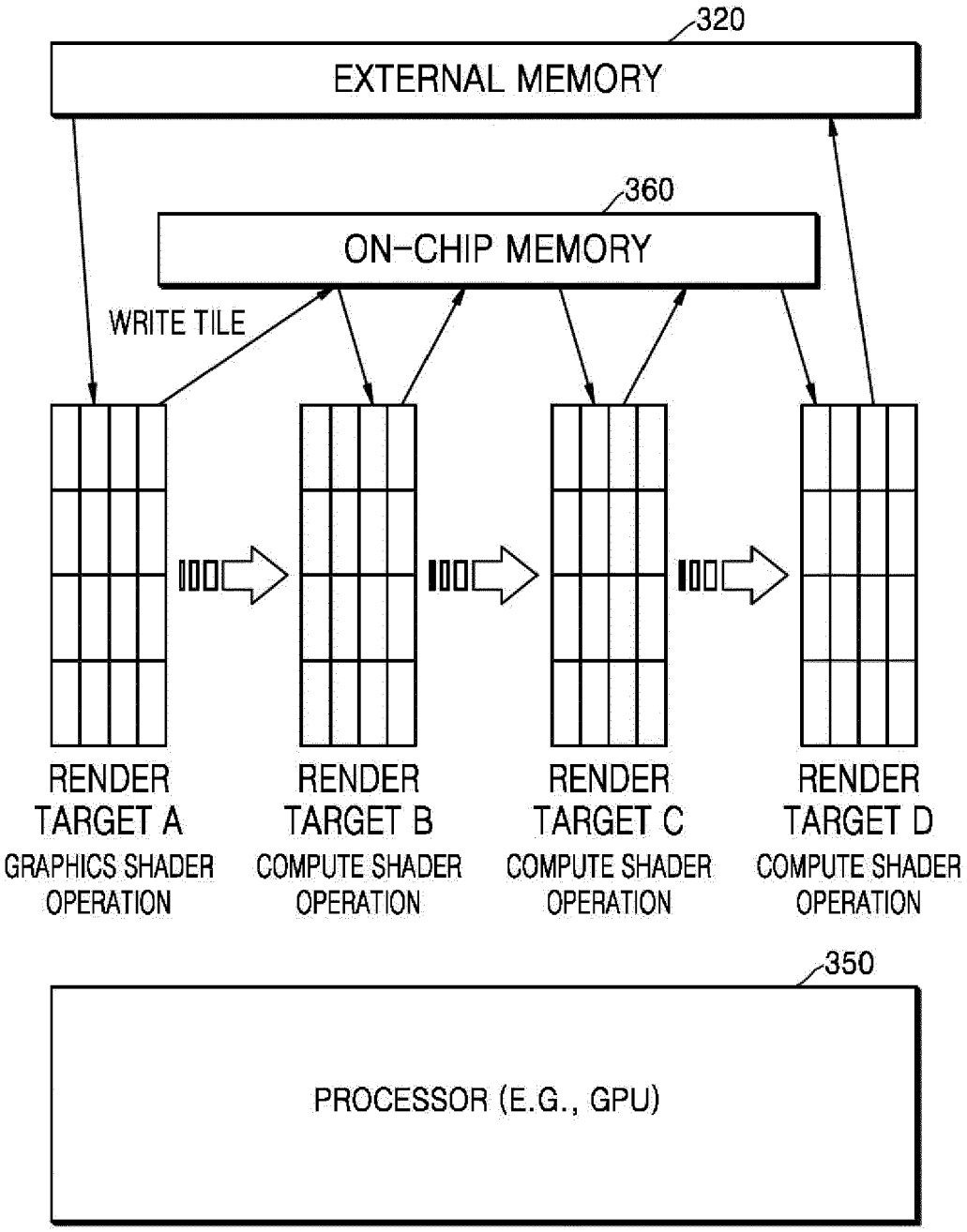

320
EXTERNAL MEMORY
360
ON-CHIP MEMORY
WRITE TILE
RENDER TARGET A
GRAPHICS SHADER OPERATION
RENDER TARGET B
COMPUTE SHADER OPERATION
RENDER TARGET C
COMPUTE SHADER OPERATION
RENDER TARGET D
COMPUTE SHADER OPERATION
350
PROCESSOR (E.G., GPU)

FIG. 6

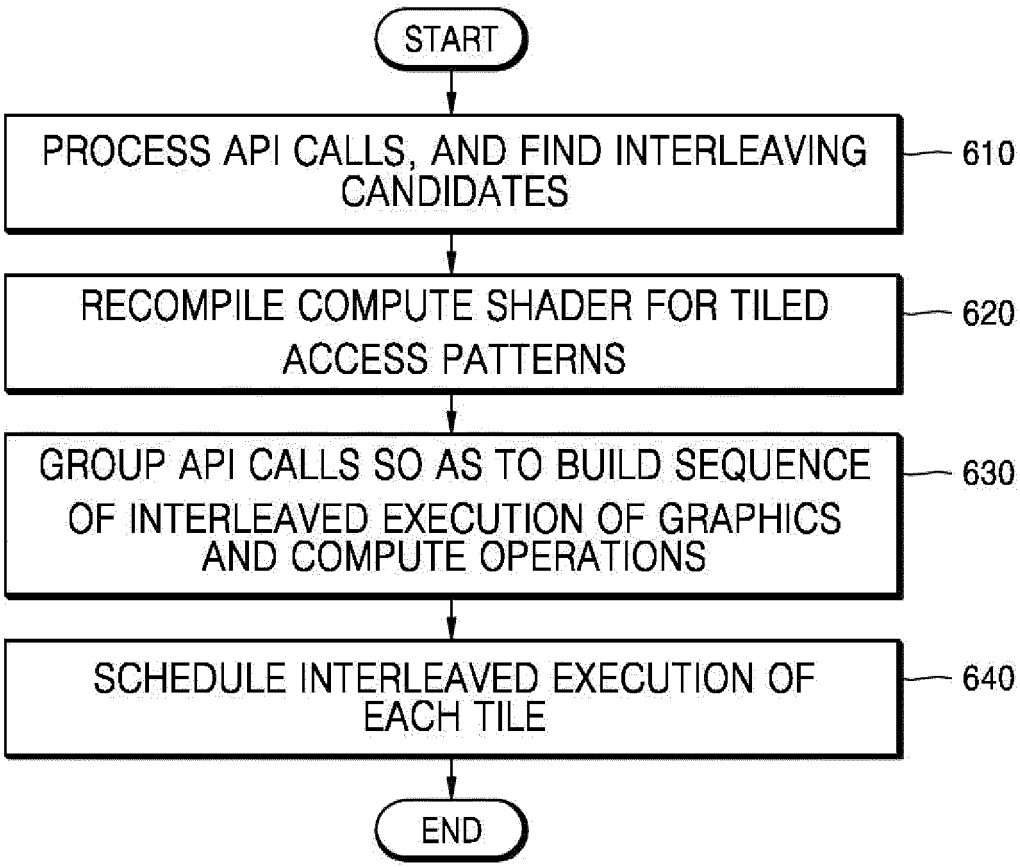

START
PROCESS API CALLS, AND FIND INTERLEAVING CANDIDATES
610
RECOMPILE COMPUTE SHADER FOR TILED ACCESS PATTERNS
620
GROUP API CALLS SO AS TO BUILD SEQUENCE OF INTERLEAVED EXECUTION OF GRAPHICS AND COMPUTE OPERATIONS
630
SCHEDULE INTERLEAVED EXECUTION OF EACH TILE
640
END

FIG. 7

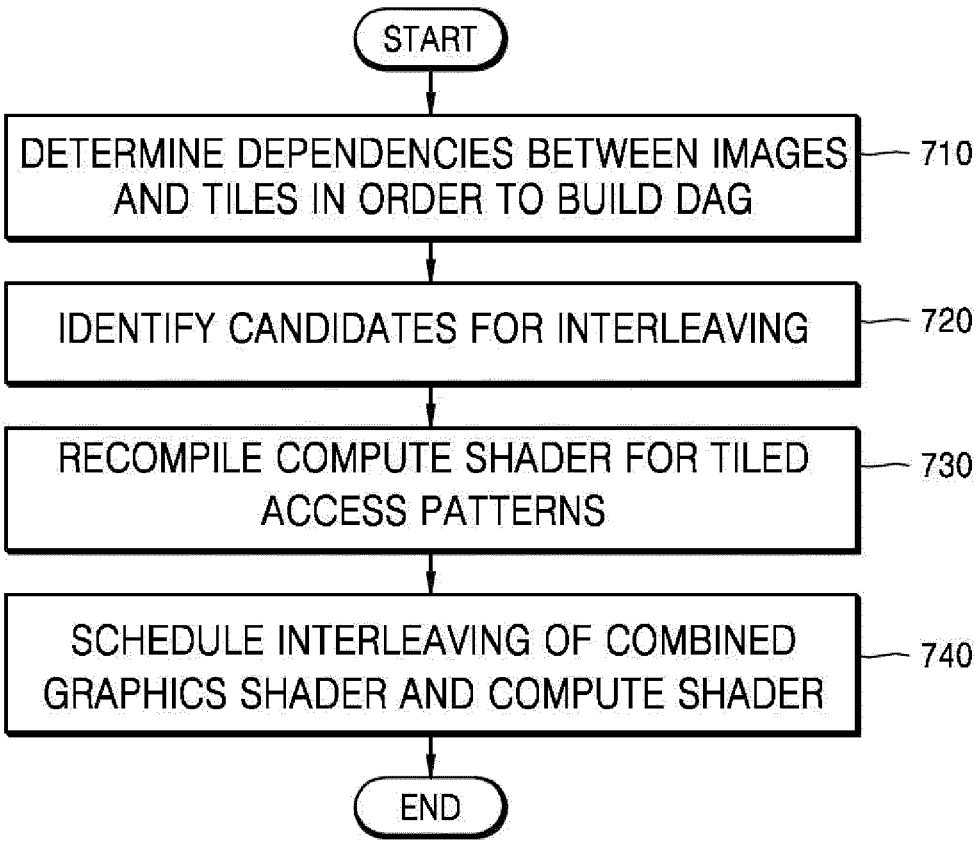

START
DETERMINE DEPENDENCIES BETWEEN IMAGES AND TILES IN ORDER TO BUILD DAG
710
IDENTIFY CANDIDATES FOR INTERLEAVING
720
RECOMPILE COMPUTE SHADER FOR TILED ACCESS PATTERNS
730
SCHEDULE INTERLEAVING OF COMBINED GRAPHICS SHADER AND COMPUTE SHADER
740
END

FIG. 8

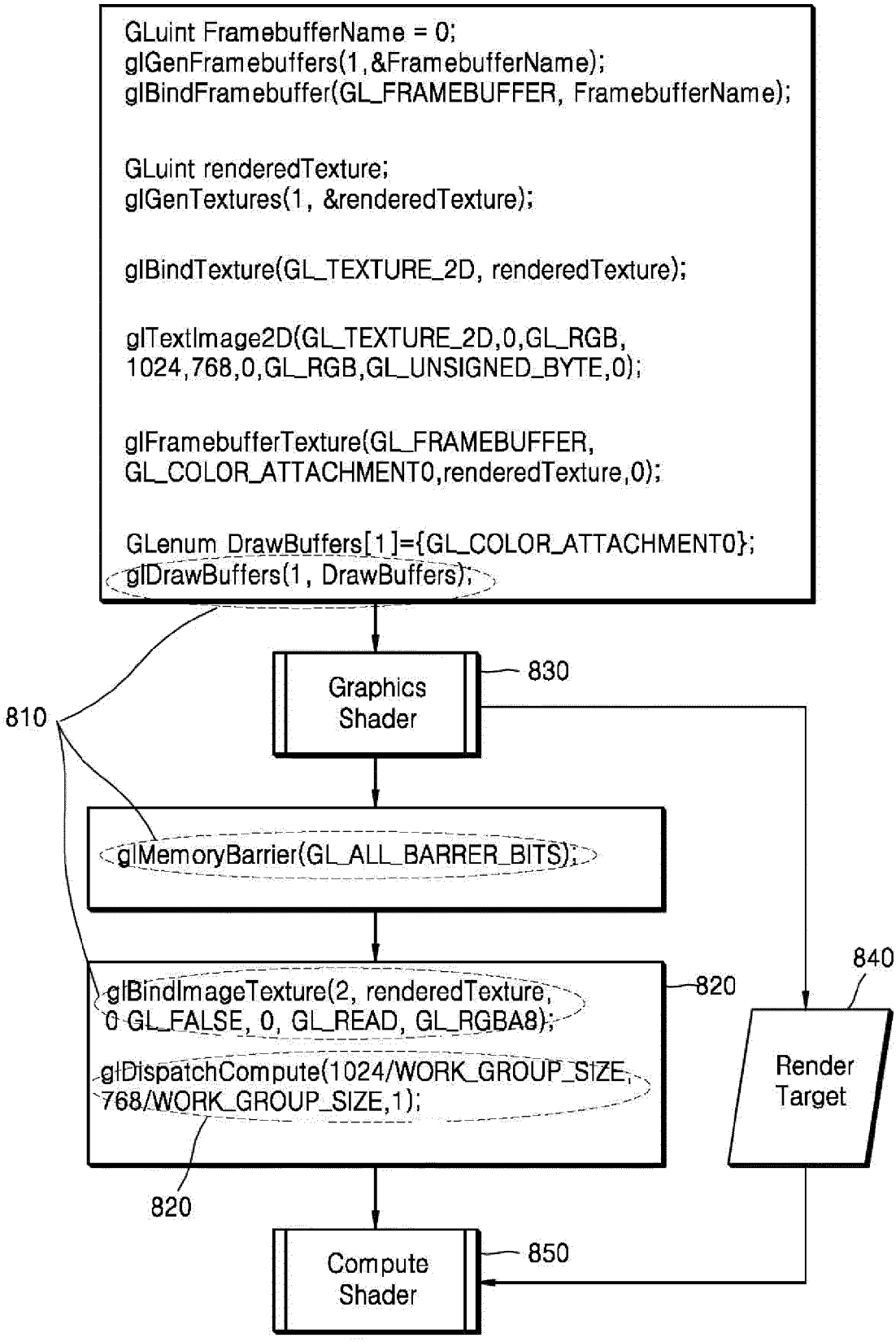
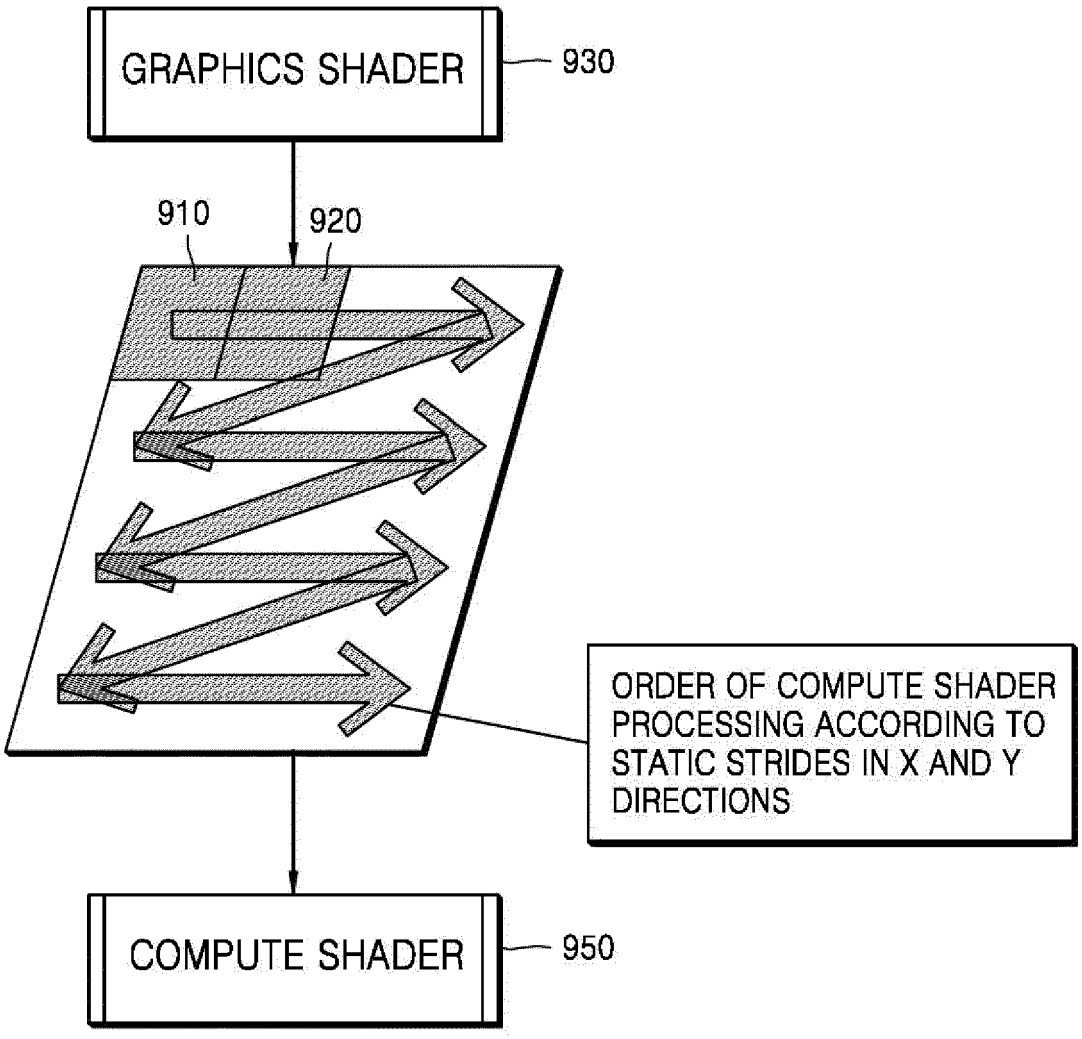

GLuint FramebufferName = 0;
glGenFramebuffers(1,&FramebufferName);
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
GLuint renderedTexture;
glGenTextures(1, &renderedTexture);
glBindTexture(GL_TEXTURE_2D, renderedTexture);
glTextImage2D(GL_TEXTURE_2D,0,GL_RGB,
1024,768,0,GL_RGB,GL_UNSIGNED_BYTE,0);
glFramebufferTexture(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,renderedTexture,0);
GLenum DrawBuffers[1]={GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, DrawBuffers);
Graphics Shader
830
810
glMemoryBarrier(GL_ALL_BARRER_BITS);
glBindImageTexture(2, renderedTexture,
0 GL_FALSE, 0, GL_READ, GL_RGBA8);
glDispatchCompute(1024/WORK_GROUP_SIZE,
768/WORK_GROUP_SIZE,1);
820
820
840
Render Target
Compute Shader
850

FIG. 9


GRAPHICS SHADER
930
910
920
ORDER OF COMPUTE SHADER PROCESSING ACCORDING TO STATIC STRIDES IN X AND Y DIRECTIONS
COMPUTE SHADER
950

Europäisches Patentamt
European Patent Office
Office européen des brevets

EUROPEAN SEARCH REPORT

Application Number
EP 16 17 2382

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (IPC)
Y	US 2011/148919 A1 (HEGGELUND FRODE [NO] ET AL) 23 June 2011 (2011-06-23) * figures 1-5 * * paragraphs [0014], [0015], [0040] - [0043], [0046], [0047], [0050], [0051], [0055] - [0062], [0068] - [0070], [0089], [0100], [0106] * * paragraphs [0171], [0176] - [0178], [0184], [0221], [0223], [0251] *	1-15	INV. G09G5/36
Y	US 2011/050716 A1 (MANTOR MICHAEL [US] ET AL) 3 March 2011 (2011-03-03) * figures 1-7 * * paragraphs [0008], [0014], [0015], [0018], [0031], [0032], [0047] - [0053] *	1-15	
A	US 2014/327671 A1 (NYSTAD JORN [NO] ET AL) 6 November 2014 (2014-11-06) * figures 1-3 * * paragraphs [0019] - [0021], [0043], [0044], [0052] *	1-15	TECHNICAL FIELDS SEARCHED (IPC) G09G G06T
A	US 2013/235057 A1 (LICEA-KANE WILLIAM W [US]) 12 September 2013 (2013-09-12) * figures 1-4 * * paragraphs [0007], [0013] *	1-15	
A	US 2013/069943 A1 (KALLIO KIIA KAAPPOO [FI] ET AL) 21 March 2013 (2013-03-21) * figures 1-6 * * paragraphs [0004], [0017], [0019], [0020], [0024], [0025], [0042], [0073], [0080], [0091] *	1-15	
A	US 2003/222870 A1 (DRIEMEYER THOMAS [DE] ET AL) 4 December 2003 (2003-12-04) * figure 3 * * paragraphs [0002], [0006], [0007] *	9	

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
The Hague	11 October 2016	Maciu, Emanoil

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 16 17 2382

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

11-10-2016

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2011148919 A1	23-06-2011	CN 102103739 A	22-06-2011
		GB 2477379 A	03-08-2011
		JP 2011129123 A	30-06-2011
		US 2011148919 A1	23-06-2011
US 2011050716 A1	03-03-2011	CN 102598061 A	18-07-2012
		EP 2473976 A1	11-07-2012
		JP 5711236 B2	30-04-2015
		JP 2013504129 A	04-02-2013
		KR 20120064093 A	18-06-2012
		US 2011050716 A1	03-03-2011
		WO 2011028981 A1	10-03-2011
US 2014327671 A1	06-11-2014	CN 104134183 A	05-11-2014
		GB 2516344 A	21-01-2015
		KR 20140131270 A	12-11-2014
		US 2014327671 A1	06-11-2014
US 2013235057 A1	12-09-2013	NONE	
US 2013069943 A1	21-03-2013	US 2013069943 A1	21-03-2013
		WO 2013043270 A1	28-03-2013
US 2003222870 A1	04-12-2003	AT 311637 T	15-12-2005
		AU 752048 B2	05-09-2002
		AU 8123798 A	25-01-1999
		CA 2294233 A1	14-01-1999
		DE 69832611 D1	05-01-2006
		DE 69832611 T2	10-08-2006
		DK 0993659 T3	27-03-2006
		EP 0993659 A1	19-04-2000
		ES 2255167 T3	16-06-2006
		JP 2001509620 A	24-07-2001
		US 6496190 B1	17-12-2002
		US 2003001844 A1	02-01-2003
		US 2003222870 A1	04-12-2003
		WO 9901846 A1	14-01-1999

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82