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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
22 August 2013 (22.08.2013)

WIPO | PCT

(10) International Publication Number
WO 2013/123295 A1

(51) International Patent Classification:
G06F 9/45 (2006.01) *G06F 9/30* (2006.01)

(21) International Application Number:
PCT/US2013/026292

(22) International Filing Date:
15 February 2013 (15.02.2013)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
13/398,798 16 February 2012 (16.02.2012) US

(71) Applicant: **MICROSOFT CORPORATION** [US/US]; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(72) Inventors: **GLAISTER, Andy**; c/o Microsoft Corporation, LCA - International Patents, One Microsoft Way, Redmond, Washington 98052-6399 (US). **TINE, Blaise Pascal**; c/o Microsoft Corporation, LCA - International Patents, One Microsoft Way, Redmond, Washington 98052-6399 (US). **SESSIONS, Derek**; c/o Microsoft Corporation, LCA - International Patents, One Microsoft Way, Redmond, Washington 98052-6399 (US). **LYAPUNOV, Mikhail**; c/o Microsoft Corporation, LCA - International Patents, One Microsoft Way, Redmond, Washington 98052-6399 (US). **DOTSENKO, Yuri**; c/o Microsoft Corporation, LCA - International Patents, One Microsoft Way, Redmond, Washington 98052-6399 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, 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, PA, 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.

[Continued on next page]

(54) Title: RASTERIZATION OF COMPUTE SHADERS

(57) Abstract: Described are compiler algorithms that partition a compute shader program into maximal-size regions, called thread-loops. The algorithms may remove original barrier-based synchronization yet the thus-transformed shader program remains semantically equivalent to the original shader program (i.e., the transformed shader program is correct). Moreover, the transformed shader program is amenable to optimization via existing compiler technology, and can be executed efficiently by CPU thread(s). A Dispatch call can be load-balanced on a CPU by assigning single or multiple CPU threads to execute thread blocks. In addition, the number of concurrently executing thread blocks do not overload the CPU.

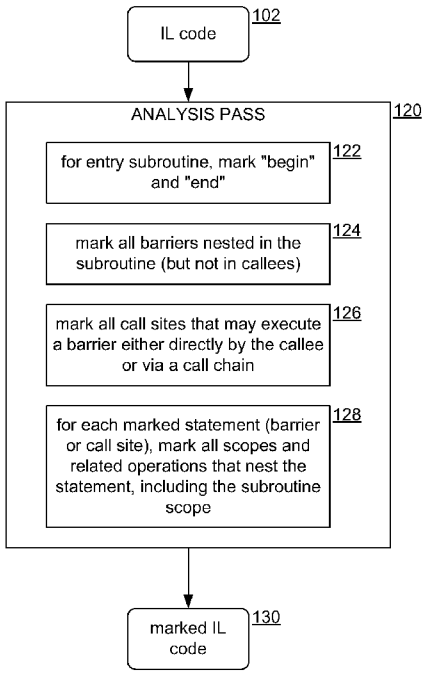

FIG. 2

WO 2013/123295 A1

(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, 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).

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

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

RASTERIZATION OF COMPUTE SHADERS

BACKGROUND

[0001] Recent trends indicate significant increase in the use of GPUs (graphics processing units) for general-purpose computing (GPGPU). That is, GPUs are tending to be used for computing not necessarily related to computer graphics, such as physics simulation, video transcoding, and other data-parallel computing. Furthermore, the introduction of on-chip shared memory in GPUs has led to marked performance improvements for widely-used compute-intensive algorithms such as all-prefix sum (scan), histogram computation, convolution, Fast Fourier Transform (FFT), physics simulations, and more. Microsoft Corporation offers the Direct X (TM) HLSL (High Level Shading Language) (TM) Compute Shader as a software API (application programming interface) to access and utilize shared memory capabilities. Note that Direct X, the HLSL, and Compute Shader will be referred to as examples, with the understanding that comments and discussion directed thereto are equally applicable to other shading languages such as CUDA (Compute Unified Device Architecture), OpenCL (Open Compute Language), etc. These will be referred to generically as "compute shaders".

[0002] A complete software platform should provide efficient software rasterization of a compute shader (or the like) on CPUs to provide a fallback when GPU hardware is not an option, or when the software platform is used in a headless VM (Virtual Machine) scenario, without the need to implement both GPU and CPU hardware solutions. That is, it is sometimes desirable to execute shader language code on a CPU rather than a GPU. However, mapping GPU-centric compute shaders onto CPUs efficiently is non-trivial primarily due to thread synchronization, which is enforced by thread barriers (or syncs).

[0003] While the efficiency of scalar shader code is important, discussion herein relates to efficiently mapping onto CPUs (as opposed to GPUs) the *parallelism* found in compute shaders. Compute shaders may expose parallelism in different ways. For example, the Direct Compute (TM) Dispatch call defines a grid of thread blocks to expose parallelism on a coarse level, which is trivial to map onto CPU threads. Each thread block is an instance of a compute shader program that is executed by multiple shader threads (a shader is analogous to a kernel in CUDA, for example). The shader threads of a block may share data via a shared memory that is common to threads in the block but private to the thread block. The threads of each thread block may be synchronized via barriers to

enable accesses to shared memory without concern for data-race conditions arising. GPUs typically execute compute shaders via hardware thread-contexts, in groups of threads (warps or wave-fronts), and each context may legally execute the program until it encounters a barrier, at which point the context must wait for all other contexts to reach the same barrier. Hardware context switching in GPUs is fast and heavily pipelined. However, CPUs do not have such hardware support, which makes it difficult to efficiently execute compute shaders on CPUs.

[0004] Techniques discussed below relate to transforming a compute shader program into an equivalent CPU program that delivers acceptable performance on CPUs.

SUMMARY

[0005] The following summary is included only to introduce some concepts discussed in the Detailed Description below. This summary is not comprehensive and is not intended to delineate the scope of the claimed subject matter, which is set forth by the claims presented at the end.

[0006] Described herein are compiler algorithms that partition a compute shader program into maximal-size regions, called thread-loops. The algorithms may remove original barrier-based synchronization yet the thus-transformed shader program remains semantically equivalent to the original shader program (i.e., the transformed shader program is correct). Moreover, the transformed shader program is amenable to optimization via existing compiler technology, and can be executed efficiently by CPU thread(s). A Dispatch call can be load-balanced on a CPU by assigning single or multiple CPU threads to execute thread blocks. In addition, the number of concurrently executing thread blocks do not overload the CPU. Note that CPU thread(s) execute multiple thread blocks (instances of the compute shader program). The Dispatch call may specify the execution of millions of instances. If all of these thread blocks are launched together there may be problems such as overconsumption of memory, poor cache utilization, and frequent context switching.

[0007] Many of the attendant features will be explained below with reference to the following detailed description considered in connection with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] The present description will be better understood from the following detailed description read in light of the accompanying drawings, wherein like reference numerals are used to designate like parts in the accompanying description.

[0009] Figure 1 shows a context for a compute shader partitioning algorithm.

[0010] Figure 2 shows an analysis pass of the partitioning algorithm.

[0011] Figure 3 shows a step for forming maximal size thread loops from marked IL code.

[0012] Figure 4 shows an example input where a loop conditional break is nested in a thread loop.

[0013] Figure 5 shows an example of a return causing divergent control flow (DCF).

[0014] Figure 6 shows a computing device.

DETAILED DESCRIPTION

[0015] There are several ways that a compute shader can be mapped onto a CPU. A naïve approach is to mimic the GPU model; i.e., interpret the original program in groups of threads and yield the execution upon encountering a barrier. However, with this approach performance can be poor due to the high overhead of context switching for a CPU. Alternatively, the shader can be partitioned at barriers, e.g.:

```
B1      // a code block that does not execute a barrier
barrier // all threads of a thread block must
                    arrive here and only after that can
                    proceed execution
B2              // another code block
```

is transformed into

```
for all threads do B1[t]        // where B1[t] is the code
                                        // block instance as
                    // executed by thread t
for all threads do B2[t].
```

[0016] This technique is both correct and efficient. A thread loop (t-loop) is denoted as "for all threads do". As the name implies, a t-loop executes one iteration for each thread t of a thread block. To maintain the semantics of the original shader program, the original variables must be privatized; i.e., each thread uses its own set of variables, called t-loop variables. Among other ways, this may be achieved, for example, by making each variable v an array of size T, the number of threads in a thread block, so that thread t uses v[t] as its copy. Thus, the notation B[t], which indicates that the original code block B uses the variable set private to thread t. Note that not all variables should be replicated

and some optimizations are possible. Moreover, while efficient scalar code generation is useful, discussion herein concerns mapping compute shader parallelism onto CPUs efficiently. Thus, a t-loop may also be referred to as a parallel for-all loop. As will be seen, iterations of a t-loop are independent and can legally be executed in any order by one or multiple *CPU* threads.

[0017] The approach above is straightforward for simple partitioning. However, if a barrier is nested within a control flow construct (e.g., an if-statement, switch statement, etc.), care must be taken to not break the scoping structure of the program. Since HLSL and other shading languages have well-defined scopes (Sub, If, Switch, and Loop), these may be readily optimized. To preserve scoping properties, each t-loop should be properly nested. Alternatively, an arbitrary goto might be needed, which would complicate the optimization significantly and might not be acceptable for just-in-time (JIT) compiling. Consider the following example:

```
B1
if(c1)        // c1 is a variable
        B2
        barrier
        B3
endif
B4
```

which is transformed into

```
for all threads do B1[t]
        if(c1[0])
                for all threads do B2[t]
                for all threads do B3[t]
        endif
for all threads do B4[t]
```

[0018] Notice that any barrier must execute in uniform control flow (UCF) (all threads execute the statement). In other words, all threads of a thread block must reach the barrier in a correct program. Therefore, "if(c1)" in the example above must be a uniform transfer, and it is sufficient to check only one instance, e.g., c1 instance of thread 0 -- c1[0].

ALGORITHM FOR INSERTION OF T-LOOPS

[0019] Figure 1 shows a context for a compute shader partitioning algorithm. Initially, shading language source code 100 such as a HLSL compute shader is compiled to produce intermediate language (IL) 102 code (i.e., intermediate representation (IR) code, bytecode, etc.) which may be intended for the parallelism of a GPU. Alternatively, the algorithm can be applied to source code directly. Per compiler options or other environment or configuration settings, compiler 108 invokes a partitioning algorithm 110 that transforms the IL code 102 producing partitioned IL code 112 with maximal-size regions (thread loops) and with barrier-based synchronization removed. The partitioned IL code 112 is then compiled and possibly optimized, as with any other IL, into CPU executable machine instructions 114 for execution in a CPU, which is not a GPU. The compiler 108 may be an online JIT compiler or it may be an offline compiler that produces a stored machine executable program.

[0020] Figure 2 shows an analysis pass 120 of the partitioning algorithm 110. As will be explained, the analysis pass 120 marks code for partitioning and is applied to each IL operation in IL code 102. The entry subroutine (entry point of the shader program) and each subroutine that may execute a barrier itself or via a call chain can be partitioned as follows. At step 122, for the entry subroutine, the begin and end of subroutine operations are marked, which accommodates the case where the main subroutine does not have nested barriers. At step 124, barriers nested within the subroutine are marked, but not in callees (called subroutines). At step 126, callees that may execute a barrier are marked. At step 128, each barrier or the like is processed by marking all scopes that nest the barrier including the subroutine scope. Such marked operations may include both begin and end or scope operations as well as related IR operations such as OpElse for an if-statement, and OpSwitchCase and OpSwitchDefault for a switch statement. One embodiment may also verify that all nesting scopes execute in UCF and that their conditional expressions, if any, are uniform values (the same for each thread). Note that determining whether a subroutine may execute a barrier either directly or via a call site nested in the subroutine may be trivially computed by taking a transitive closure on a call graph The steps 124, 126, and 128 should be performed in the order shown in Figure 2. The analysis pass outputs marked IL code 130.

[0021] Figure 3 shows a step 132 for forming maximal size thread loops from the marked IL code 130. Step 132 involves traversing subroutine operations in order and for each marked operation (Op) performing any of the relevant steps 134, 136, 138, 140. At

step 134, unless the operation currently being processed is the beginning of a subroutine, an OpEndThreadLoop operation is inserted right before the current operation. At step 136, it is determined if the operation preceding an OpEndThreadLoop is an OpBeginThreadLoop, and if so, both the OpEndThreadLoop and the OpBeginThreadLoop are removed as they represent an empty t-loop. To elaborate, it may happen that two marked operations follow each other. Then, the code would have been something like:

```
OpBeginThreadLoop
…
OpEndThreadLoop
Op1
OpBeginThreadLoop     // empty t-loop to
OpEndThreadLoop       //    be removed
Op2
OpBeginThreadLoop
…
OpEndThreadLoop
```

Therefore, there is an empty t-loop that may still be in the code, but it does nothing and can be removed.

[0022] At step 138, unless the current operation is the end of the subroutine, an OpBeginThreadLoop operation is inserted right after the current operation. Finally, at step 140, if the current operation is a barrier or analogous type of synchronization operation, the operation is removed because the original data dependencies are now enforced by the order of the execution of the thread loops.

[0023] Note that the partitioning algorithm 110 creates maximal size t-loops without breaking the scoping structure and inserted t-loops are properly nested. There are, however, control-flow transfer operations such as break, continue and return that may transfer control outside of a t-loop. These may need to be specially treated to preserve correctness of the shader program. If these operations do not transfer control outside of a t-loop (i.e., they are nested within their corresponding scopes), they are handled in a standard way.

[0024] Regarding the term "maximal", this term means that it is not possible to increase the amount of code encompassed by a t-loop without breaking the nested structure of the program. A minimal size t-loop (or region) would encompass a single operation – also a possible partitioning, but the program will be slow due to t-loop

overheads. Maximal size t-loops reduce the overall number of t-loops and thus reduce the associated overhead.

[0025] Figure 4 shows an example input 150 where a loop conditional break is nested in a t-loop. The input 150 is transformed as shown in output 152.

[0026] Four helper flag variables are used, one for each for four different types of control transfer (break, continue, switch break, and return). Each such variable is a scalar because the value of the transfer conditional is uniform. The variables are initialized to false before a t-loop if they are used in the t-loop. A loop break/continue, nested in a uniform t-loop, will set the bBreakFlag/bContinueFlag to true and transfer control to the next iteration of the enclosing t-loop and, after the t-loop is complete, break/continue the corresponding loop if the bBreakFlag/bContinueFlag is set. An executed uniform switch break, nested in a uniform t-loop, sets the bSwitchBreakFlag to true and transfers control to the next iteration of the enclosing t-loop and, after the t-loop is done, breaks the switch if bSwitchBreakFlag is set. Finally, uniform return, nested in a uniform t-loop, sets the bReturnFlag to true and transfers control to the next iteration of the enclosing t-loop and, after the t-loop is done, returns from the subroutine if the bReturnFlag is set.

DIVERGENT TRANSFER OF CONTROL OUTSIDE OF T-LOOP

[0027] It may happen that a transfer outside of a t-loop is divergent (non-uniform). Figure 5 shows an example of such a return causing divergent control flow (DCF). DCF input 160 is transformed as shown in output 162. The transformation introduces a t-loop mask TLoopRetMask, one instance per thread, to capture which threads executed a divergent return. If a subroutine contains such a return, the rules for the insertion of TLoopRetMask are as follows:

[0028] (1) Initialize TLoopRetMask to true on entry to the subroutine at the very beginning of the entry t-loop. If there is no entry t-loop, generate one.

[0029] (2) Set TLoopRetMask to false for every DCF return as well as UCF return that transfers control outside of a t-loop if there is at least one DCF return that transfers control outside of this t-loop.

[0030] (3) For every t-loop that starts in DCF, generate guarding code at the very beginning of a t-loop that would skip the iteration if TLoopRetMask is false.

[0031] (4) Re-initialize TLoopRetMask to true on exit from the subroutine at the very end of the exit t-loop, unless it is the main subroutine. If there is no exit t-loop, generate one.

[0032] It is sufficient to use only one TLoopRetMask per program, even though it may be required in several subroutines. While this reduces overhead, it is also the reason for re-initialization of the mask on exit from the subroutine. Using more mask variables, which are replicated across threads, increases memory footprint and is not desirable.

[0033] A loop's break and continue that transfer control outside of a t-loop cannot be in DCF. If the break or continue did so transfer control, the entire loop wouldn't have been in DCF (due to the back edge) and cannot execute any barrier. And, because the t-loops are induced by barriers such a loop must be nested inside a t-loop according to our partitioning algorithm.

[0034] A DCF switch-break that transfers control outside of a t-loop can be handled in exactly the same manner as the DCF return, via managing the state of TLoopSwitchMask. To avoid using several such variables and complicated code generation, TLoopSwitchMask may be initialized to True right before such a switch and TLoopSwitchMask may be re-initialized to True right after the switch. This approach correctly transforms nested switches. Suppose switch S2 is nested in a case of switch S1. If S2 requires TLoopSwitchMask, S2 contains a nested barrier (otherwise, it would have been nested in a t-loop), so both S1 and S2 must start in UCF. Furthermore, the control is uniform in S1's case at least up until the beginning of S2 (otherwise, S2 would have been in DCF), thus TLoopSwitchMask must be true before the start of S2. Since the effect of the DCF switch break in S2 propagates only to the end of S2 and has no effect on control-flow type in S1, it is safe to re-initialize TLoopSwitchMask right after S2 because TLoopSwitchMask has not yet captured any effect of S1's DCF break (although it was used inside S2). What may be done here is to insert extra re-initializations of TLoopRetMask and TLoopSwitchMask to essentially avoid complicated analysis of where exactly to insert mask initializations for the situation that would occur rarely in performant compute shaders. Finally, the guarding condition for t-loops that are nested in a switch and start in DCF must be set to "TLoopRetMask && TLoopSwitchMask" to account for effects of both DCF return and switch break.

EXECUTION OF T-LOOPS ON CPUs

[0035] To achieve high performance, it may be helpful to minimize overhead of synchronization by having fairly coarse units of work and by load-balancing the processors. A typical invocation of a compute shader dispatch call creates many instances of the compute shader program, each of which usually performs roughly similar amounts of work. Thus, it is natural to use a single CPU thread to execute an instance: units of

work are coarse, the concurrency overhead of executing t-loops does not exist due to serial execution, and the load-balance is reasonable.

[0036] Some compute shaders, though, are written to "stream" units of work rather than rely on the Dispatch call to do the streaming; i.e., the Dispatch call instantiates just a few instances of the compute shader program, and each instance has a streaming loop that processes several units of work, one after another. In this scenario, the load-balance may suffer due to under-utilization of some CPU threads. To avoid this, several CPU threads can be used to run each thread loop, which is legal because t-loop iterations are independent. These threads must synchronize before and after each t-loop and use atomic increment to obtain the thread index t of a t-loop iteration. The result is better load-balancing at the expense of small synchronization overhead. Note that Dispatch (and its equivalents in various shader languages) may be able to not only invoke shader language code for execution, but a Dispatch or Draw call may also reference various graphical objects in the shader language, such as pixels, vertices, etc. Additional details of Dispatch (and its equivalents) and thread blocks, are available elsewhere (see, e.g., "Practical Rendering and Computation with Direct3D 11", Zink et al., 2011, CRC Press).

OPTIMIZING DISPATCH

[0037] Usually, the Dispatch call (or its equivalent) creates hundreds or thousands of compute shader instances. When run on a CPU rather than a GPU, executing all of the instances concurrently will (1) create too many threads, (2) use too much memory, (3) cause many expensive context switches, and (4) pollute the cache due to frequent context switches. This results in poor performance due to overbooking of the system. Thus it may be helpful to limit the number of compute shader instances that execute concurrently, for example, the number of compute shader instances may be limited to two plus the number of CPU cores.

CONCLUSION

[0038] Figure 6 shows a computing device 180. The computing device 180 is an example of a type of device that can perform embodiments described above. The computing device 180 may have some or all of the following: a display 182, an input device 184 (e.g., keyboard, mouse touch sensitive area, etc.), a CPU 186, a GPU 188, and storage media 190. These components may cooperate in ways well known in the art of computing.

[0039] Embodiments and features discussed above can be realized in the form of information stored in volatile or non-volatile computer or device readable storage media.

This is deemed to include at least physical storage media such as optical storage (e.g., compact-disk read-only memory (CD-ROM)), magnetic media, flash read-only memory (ROM), or any means of physically storing digital information (excluding carrier waves, signals per se, and the like). The stored information can be in the form of machine executable instructions (e.g., compiled executable binary code), source code, bytecode, or any other information that can be used to enable or configure computing devices to perform the various embodiments discussed above. This is also deemed to include at least volatile memory such as random-access memory (RAM) and/or virtual memory storing information such as central processing unit (CPU) instructions during execution of a program carrying out an embodiment, as well as non-volatile media storing information that allows a program or executable to be loaded and executed. The term media as used herein refers to physical devices and material and does not refer to signals per se, carrier waves, or any other transient forms of energy per se. The embodiments and features can be performed on any type of computing device, including portable devices, workstations, servers, mobile wireless devices, and so on.

CLAIMS

1. A method of transforming intermediate representation (IR) code compiled from a compute shader comprised of source code in a shading language, comprising:

performing a marking and analysis pass that includes marking the beginnings and ends of subroutines in the IR code, marking memory barriers nested in the subroutines, for each such barrier marking nested scopes and related operations, and marking call sites of the subroutines that are determined to execute a barrier; and

using the marked IR code to form maximal-size thread loops by processing operations of the subroutine.

2. A method according to claim 1, further comprising determining that a current operation being processed is not the beginning of a subroutine, and in response inserting an end-thread-loop operation before the current operation being processed.

3. A method according to claim 2, removing the end-thread-loop operation and a corresponding begin-thread-loop operation when the begin-thread-loop operation immediately precedes the end-thread-loop-operation.

4. A method according to claim 1, further comprising determining that a current operation being processed is not the end of a subroutine, and in response inserting a begin-thread-loop operation directly after the current operation being processed.

5. A method according to claim 1, further comprising outputting updated IR code wherein barrier operations in the IR code that correspond to barrier statements in the source code are removed from the IR code and thread loops are added to the IR code.

6. A method according to claim 5, wherein the transformed IR code with the barriers removed is compiled and executed on a CPU that is not a GPU, and wherein a thread-loop is executed by a plurality of CPU threads.

7. One or more computer readable media storing information to enable a computing device to perform a process, the process comprising:

executing a compiler algorithm invoked when the compiler receives code corresponding to a shader program, the shader program performing non-graphics computation and comprising shader language code; and

partitioning, by the algorithm, the code of the shader program into maximal-size thread loops and removing synchronization barriers from the code.

8. One or more computer-readable media according to claim 7, wherein the partitioning is performed such that data dependencies in the shader program that would

have been enforced by the synchronization barriers are preserved despite the removal of the synchronization barriers.

9. One or more computer-readable media according to claim 8, wherein the partitioning preserves nesting of at least subroutines or control flow constructs in the code.

10. A method of transforming intermediate representation (IR) code compiled from a compute shader comprised of source code in a shading language, comprising:

identifying uniform control flow (UCF) transfers and divergent control flow (DCF) transfers in the IR code;

when transforming the IR code to a thread loop, optimizing the UCF transfers outside of the thread loop and optimizing the DCF transfers outside of the thread loop; and

executing the transformed IR code on a CPU that is not a GPU.

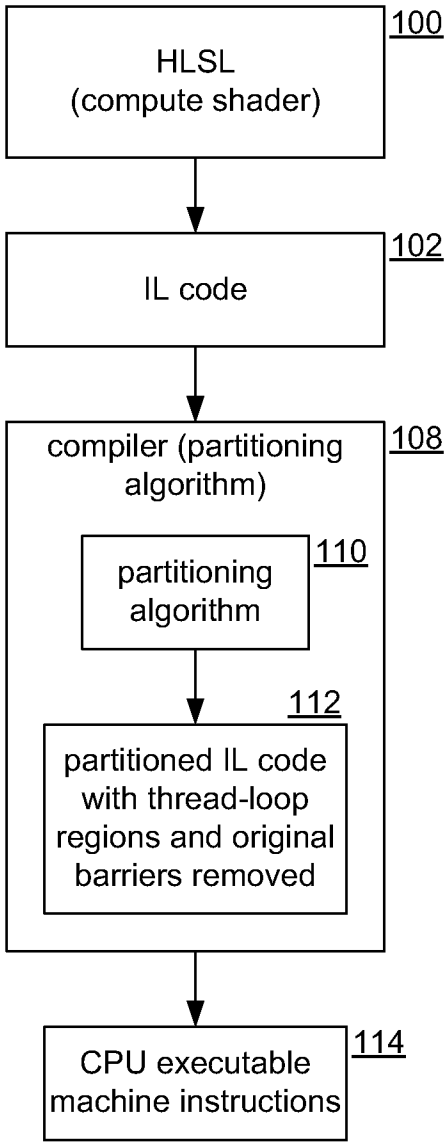
HLSL
(compute shader)
100
IL code
102
compiler (partitioning algorithm)
108
partitioning algorithm
110
112
partitioned IL code with thread-loop regions and original barriers removed
CPU executable machine instructions
114

FIG. 1

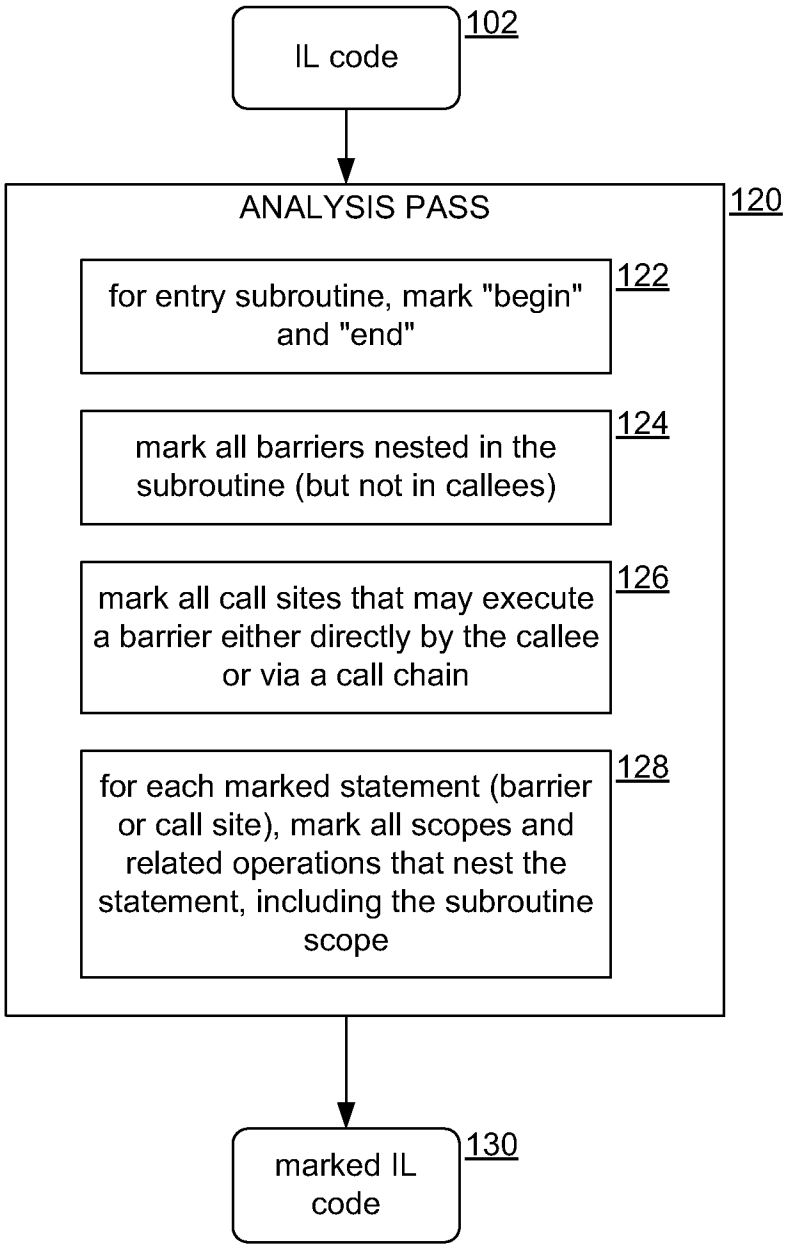
IL code
102
ANALYSIS PASS
120
for entry subroutine, mark "begin" and "end"
122
mark all barriers nested in the subroutine (but not in callees)
124
mark all call sites that may execute a barrier either directly by the callee or via a call chain
126
for each marked statement (barrier or call site), mark all scopes and related operations that nest the statement, including the subroutine scope
128
marked IL code
130

FIG. 2

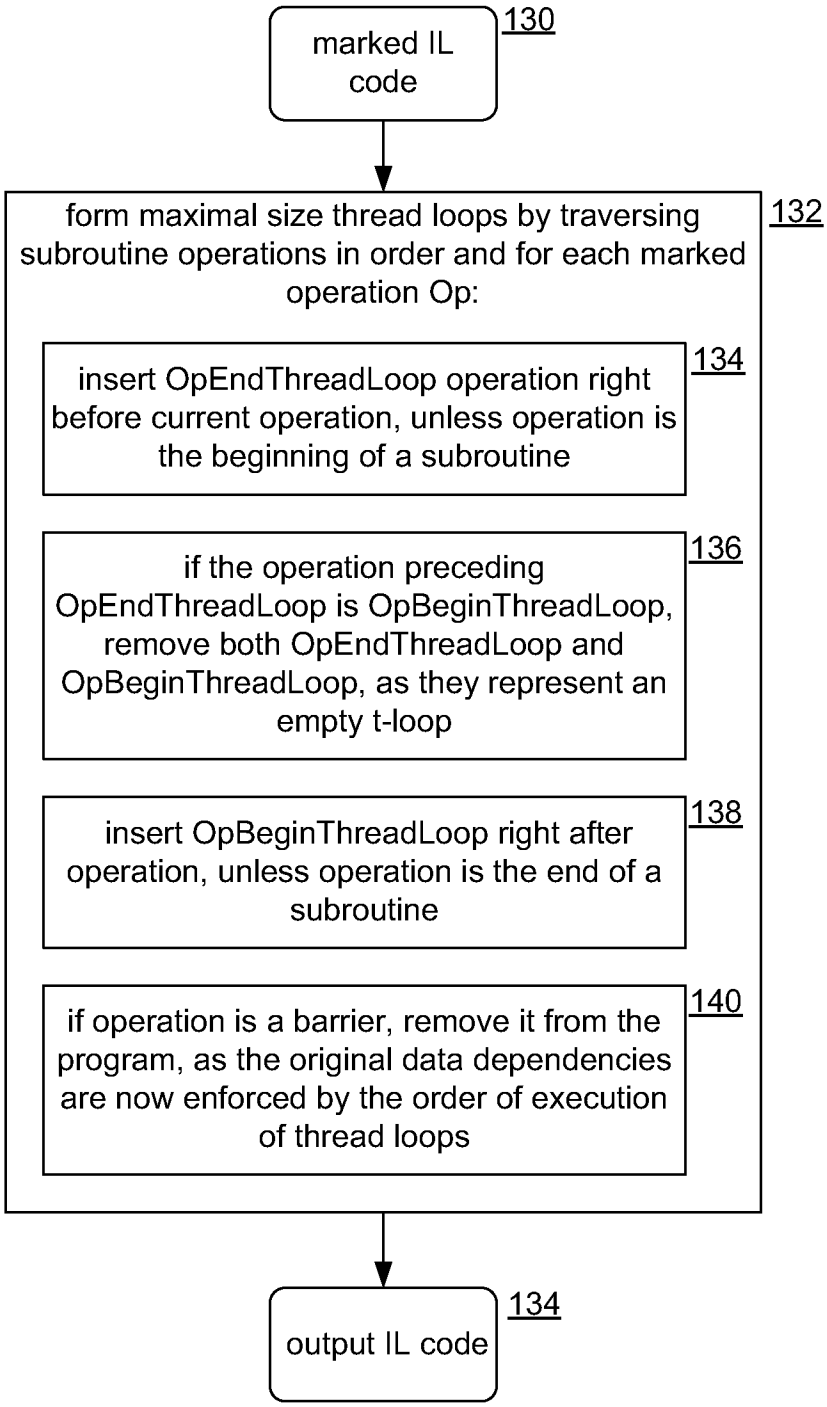
marked IL code
130
form maximal size thread loops by traversing subroutine operations in order and for each marked operation Op:
132
insert OpEndThreadLoop operation right before current operation, unless operation is the beginning of a subroutine
134
if the operation preceding OpEndThreadLoop is OpBeginThreadLoop, remove both OpEndThreadLoop and OpBeginThreadLoop, as they represent an empty t-loop
136
insert OpBeginThreadLoop right after operation, unless operation is the end of a subroutine
138
if operation is a barrier, remove it from the program, as the original data dependencies are now enforced by the order of execution of thread loops
140
output IL code
134

FIG. 3

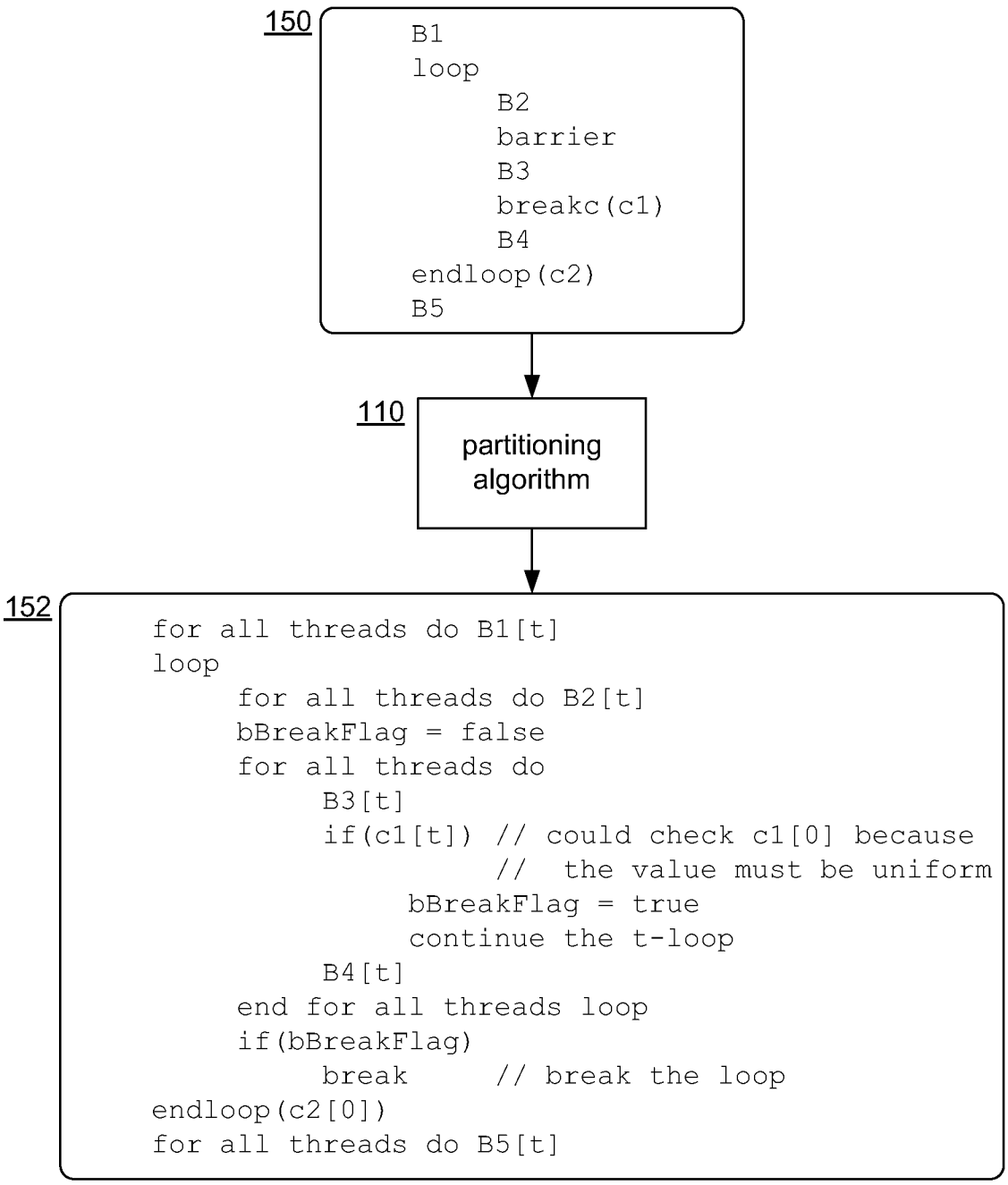
150
B1
loop
B2
barrier
B3
breakc(c1)
B4
endloop(c2)
B5
110
partitioning algorithm
152
for all threads do B1[t]
loop
for all threads do B2[t]
bBreakFlag = false
for all threads do
B3[t]
if(c1[t]) // could check c1[0] because
// the value must be uniform
bBreakFlag = true
continue the t-loop
B4[t]
end for all threads loop
if(bBreakFlag)
break // break the loop
endloop(c2[0])
for all threads do B5[t]

FIG. 4

160

```
B1                  // UCF
if(c1)              // UCF, c1 must be uniform because of
                    nested barrier
    B2              // UCF
    barrier // UCF
    B3              // UCF
    retc(c2)        // UCF, but c2 is a non-uniform value, so
                    only some threads execute return
    B4              // DCF because of non-uniform return
else                // UCF
    B5              // UCF
endif         // DCF
B6                  // DCF because of non-uniform return
```

110 partitioning algorithm

162

```
for all threads do              // UCF
      TLoopRetMask[t] = true    // UCF, initialize TLoopRetMask[t]
                                   on entry to subroutine
      B1[t]                     // UCF
end for all threads loop        // UCF
if(c1[0])                       // UCF
    for all threads do B2[t]    // UCF
    for all threads do          // UCF
        B3[t]                   // UCF
        if(c2[t])               // UCF, c2 is a non-uniform value
            TLoopRetMask[t] = false // DCF
            continue the t-loop     // DCF
        B4[t]                       // DCF
    end for all threads loop    // DCF
    // Do NOT generate a return here, because not all
      threads may have executed it.
    // Instead, the effect of executing the return is
      captured by TLoopRetMask.
else                            // UCF
    for all threads do B5[t]    // UCF, do not use
                                   TLoopRetMask (because of UCF)
endif                           // DCF
for all threads    do           // DCF
    if(TLoopRetMask[t])         // DCF, skip the iteration if
                                   TLoopRetMask is false; i.e.,
                                   thread returned
        B6[t]                      // DCF
    TLoopRetMask[t] = true      // DCF, re-initialize TLoopRetMask[t]
                                   on exit from subroutine
end for all threads loop        // DCF
```

FIG. 5

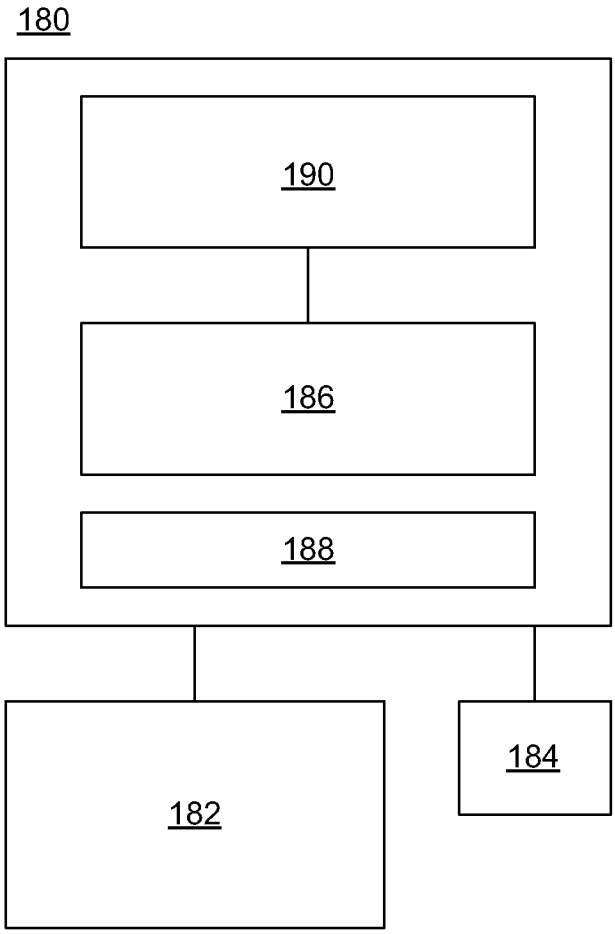

FIG. 6

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2013/026292

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/45(2006.01)i, G06F 9/30(2006.01)i

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F 9/45; G06F 9/30; G06F 11/28; G06F 9/06

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: compile, intermediate, shade

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2009-0113402 A1 (CHEN LINGJUN et al.) 30 April 2009 See abstract, paragraphs [26-28], claims 1-3 and figures 1-3.	1-10
A	US 2006-0005178 A1 (NVIDIA Corporation) 05 January 2006 See abstract, paragraphs [27-31], claims 1-4 10 and figures 1-2.	1-10
A	KR 10-2010-0081347 A (QUALCOMM INCORPORATED) 14 July 2010 See abstract, paragraphs [37-41], claims 6 8-9 and figures 7-9.	1-10
A	JP 2000-507373 A (ASYMETRIX CORPORATION) 13 June 2000 See abstract, claims 1-2 and figures 2b.	1-10

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

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of 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
08 May 2013 (08.05.2013)	**09 May 2013 (09.05.2013)**

Name and mailing address of the ISA/KR
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 302-701, Republic of Korea
Facsimile No. 82-42-472-7140

Authorized officer
SHIN, Hyoun Shang
Telephone No. 82-42-481-8514

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2013/026292

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2009-0113402 A1	30.04.2009	CN 101836188 A	15.09.2010
		EP 2053506 A1	29.04.2009
		JP 2011-501325 A	06.01.2011
		JP 2011-501325 T	06.01.2011
		KR 10-1237177 B1	25.02.2013
		KR20100081347A	14.07.2010
		TW 200925997 A	16.06.2009
		US 8365153 B2	29.01.2013
		WO 2009-055731 A1	30.04.2009
US 2006-0005178 A1	05.01.2006	CN 100507852 C	01.07.2009
		CN 1997964 A	11.07.2007
		CN 1997964 C0	11.07.2007
		JP 04-549392 B2	16.07.2010
		JP 2008-505422 A	21.02.2008
		JP 2008-505422 T	21.02.2008
		JP 4549392 B2	22.09.2010
		KR 10-0860427 B1	25.09.2008
		KR20070039936A	13.04.2007
		US 7426724 B2	16.09.2008
		US 7746347 B1	29.06.2010
		US 7777750 B1	17.08.2010
		US 7839410 B1	23.11.2010
		US 7852345 B1	14.12.2010
		US 7928989 B1	19.04.2011
		US 7958498 B1	07.06.2011
		WO 2006-014388 A2	09.02.2006
		WO 2006-014388 A3	15.03.2007
		WO 2006-014388 A3	09.02.2006
KR 10-2010-0081347 A	14.07.2010	CN 101836188 A	15.09.2010
		EP 2053506 A1	29.04.2009
		JP 2011-501325 A	06.01.2011
		TW 200925997 A	16.06.2009
		US 2009-0113402 A1	30.04.2009
		US 8365153 B2	29.01.2013
		WO 2009-055731 A1	30.04.2009
JP 2000-507373 A	13.06.2000	AU 1956897 A	16.09.1997
		CA 2248181 A1	04.09.1997
		EP 0883844 A1	16.12.1998
		EP 0883844 A1	23.02.2000
		JP 2000-507373 T	13.06.2000
		US 05764989A A	09.06.1998
		US 05848274A A	08.12.1998
		WO 97-32250 A1	04.09.1997
		WO 97-32250A1	04.09.1997
		WO 97-43711 A1	20.11.1997