Computing and Informatics, Vol. 38, 2019, 173–196, doi: 10.31577/cai_2019_1_173

CL-VIS: VISUALIZATION PLATFORM FOR UNDERSTANDING AND CHECKING THE OPENCL PROGRAMS

SeongKi Kim

Division of Computer Science Engineering
Keimyung University, 1095, Dalgubeol-daero, Dalseo-Gu
Daegu, Republic of Korea
e-mail: skkim9226@gmail.com

HyukSoo Han

Department of Computer Science
Sangmyung University, 20, Hongjimun 2-gil, Jongno-Gu
Seoul, Republic of Korea
e-mail: hshan@smu.ac.kr

Abstract. Due to GPU's improved hardware performance, many researchers have tried to utilize the GPU for computer vision, image processing, cryptography, and artificial intelligence. As results, the GPU could successfully speed up algorithms from tens to hundreds of times in many cases. However, GPU programming is still known to be difficult because of its different characteristics from the traditional CPU programming. Also, it is hard to find the root causes of software failures because the failures are irreproducible in many cases. Our goal is to simplify the process of verifying intended actions when debugging GPGPU programs. To achieve this goal, we use the visualization method of executed codes because it can increase the human's understanding through seeing and analyzing the real actions by each thread. We developed a platform that can visualize the running OpenCL codes and algorithms that can identify data race, barrier divergence, and infinite loop in the GPU. To the best of our knowledge, this is the first study on the visualizations of OpenCL operations and detection of infinite loops in the programs. We also suggest an algorithm for detecting data race with GPU-specific lock-step execution and barrier function.

Keywords: Visualization, GPGPU, debug, data race, barrier divergence, infinite loop

1 INTRODUCTION

To meet the user's increasing demands for realistic software and the increasing display resolution (e.g. Full HD (1 920 × 1 080), Ultra HD (3 840 × 2 160)), the sizes of visual data (e.g. Texture, Vertex, Color, Normal) have become larger, and the GPU has improved. Also with the improvements, GPGPU platforms, such as Compute Unified Device Architecture (CUDA) [13] and Open Computing Language (OpenCL) [20], enabled the programming of the GPU for general purposes. Many researchers have tried to use them for performance improvements in different fields. As a result, the GPU could successfully accelerate algorithms for computer vision [17], image processing [7], cryptography [29], and artificial intelligence [19] fields. Furthermore, many researchers have built GPU-based supercomputers that are ranked highly in top 500 websites [22] and used for complex calculations.

Although the GPGPU has been widespread, most of the previous studies concentrated on performance improvement and correctness of results, without much focus on debugging GPGPU programs. Recently, GPGPU has started to become widespread in safety-critical systems, and its software faults can become important issues, shortly. For example, computer vision algorithms, deep learning within a car [14] or medical imaging [21] can exploit the GPGPU, but the failure of these systems can lead to a severe accident.

Regardless of the importance of error-free GPGPU programs, GPGPU is hard to program, and the possibilities of mistakes are high because of the following reasons. First, GPU's characteristics are different from the CPU. Because of it, we cannot simply convert the codes for CPU to the ones for GPU. The simple conversion can lead to a minor performance benefit in many cases, or the conversion can be impossible in many cases because of different characteristics. Second, the results of GPU codes may be different if a different number of threads or thread groups is employed. The developers can alter the number of threads for optimization cases, and the change may cause unexpected results from the inter-thread intervention. Third, the number of GPU threads can be huge (e.g. 10 000 000), and some failures can be irreproducible because their causes are from the thread scheduling or status.

In addition to the programming/debugging difficulties attributed to different characteristics, and thread intervention, data race, barrier divergence, and infinite loop also make the GPGPU programming error-prone. The data race can be a significant problem particularly in the GPU because a vast number of threads may access the same memory at the same time. These simultaneous accesses can cause an unexpected problem depending on the access orders of many threads. The barrier divergence is a specific problem to the GPU and can cause unexpected results in many cases. It happens when all of the GPU threads within a group do not reach the same barrier point. Differently from the traditional CPU case, the infinite loop can happen by GPU-specific lock-step execution.

To minimize these difficulties in GPGPU programming and help in checking the GPGPU programs, some tools such as Nsight systems [24], Allinea DDT [25], Profiler [27] and mem-check [26] have been developed for the CUDA platform. However,

the Nsight does not support the visualization of GPU codes but the visualization of CPU codes only. In addition, Allinea DDT, profiler, and cuda-memcheck are developed only for the CUDA platform. To make up for these limitations, we developed the visualization platform for the OpenCL case. The contribution of this paper is as follows. First, we suggest a visualization method of running OpenCL codes. Second, we propose a heuristic for finding the GPU-specific infinite loop. Third, we suggest a method for finding data races with the GPU-specific lockstep execution and barrier function.

The rest of this paper is organized as follows. Section 2 describes the OpenCL, Oclgrind as well as Data Race/Barrier Divergence/Infinite Loops as backgrounds and related works. Section 3 describes the GPU-specific characteristics and issues in more detail. Section 4 describes our visualization methods and algorithms for finding data races, barrier divergences, and infinite loops at the GPU and their rationales. Section 5 describes the results through our implementations, and the conclusion is drawn in Section 6.

2 BACKGROUND AND RELATED WORKS

In this paper, we used the OpenCL as a GPGPU language because it is an open standard regardless of vendor and platforms. For background knowledge, this section describes OpenCL and Oclgrind. Also for related works, this section introduces the previous works for data race, barrier divergence and infinite loop at the GPU.

2.1 OpenCL

OpenCL is an open standard for heterogeneous computing and allows the programming of the CPU or GPU. Apple Inc. [1] originally developed the OpenCL, and the Khronos Group [15] maintains it presently. The Khronos Group defines the specification [10] of OpenCL so that all vendors should support for compatibility. The specification defines four different models in terms of platform, execution, memory, and programming aspects.

Platform Model: The platform model describes a *host* and a *device*. The host is a central unit that executes the main program, divides works to each device, and collects the results from each device. A device is a target unit that runs the parallel parts of a program. A device can have many compute units, and each compute unit can have many processing elements. Figure 1 illustrates these relations.

In Figure 1, a host creates a context to compute device 1 and enqueues commands to the device through the established context. The device schedules the commands, delivers them to the compute units and the processing elements according to its policy, and each processing element runs the commands. The CPU or GPU can be one of the devices.

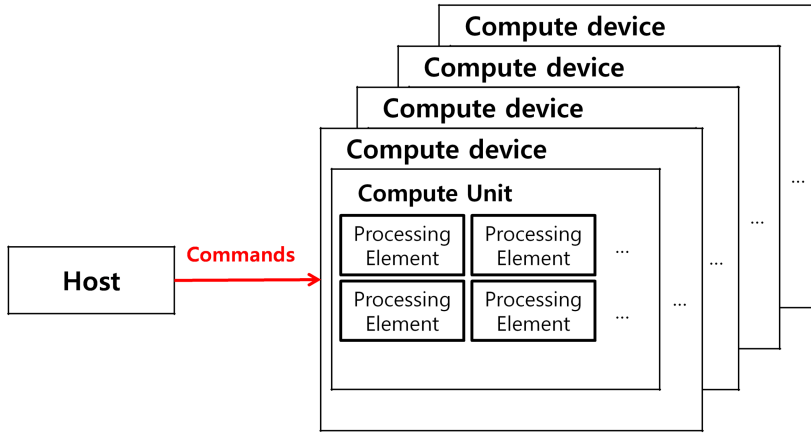

Figure 1. OpenCL platform model

Execution Model: The execution model defines a *host program* and a *kernel*. The host program is a program that runs on the host. A kernel is a function executed on the compute device. When the host program sends the commands (kernel execution, memory read, memory write), it also sends the number of work-groups and work-items. Work-group is a group of threads that run the same operation and runs on a compute unit. Work-item is a thread and executes on a processing element. Figure 2 illustrates the decompositions of work-groups and work-items handled by OpenCL.

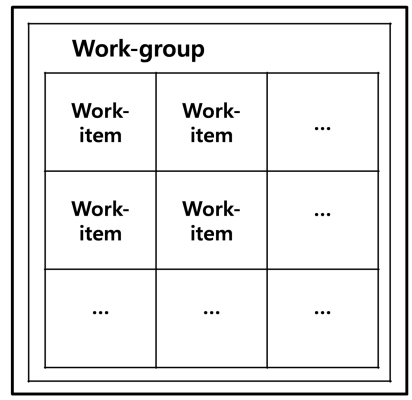

Figure 2. The decomposition of work-groups and work-items

Figure 2 shows the relationship among work-groups and work-items in two dimensions. Work-groups are independent of each other and simultaneously run, and work-items within the same work-group execute the same operation in the

lock-step method. A compute unit runs a work-group, and a processing element runs a work-item. If the number of work-groups and work-items are more than those of the physical compute units or processing elements, the mapping of work-group or work-item is scheduled according to the vendor's policy. Work-items within a single work-group can communicate through a shared buffer, and any accesses to the buffer should be synchronized.

2.2 Oclgrind

Oclgrind [18] is an open-source simulator based on the OpenCL's Standard Portable Intermediate Representation (SPIR) [16]. It enables to create a development tool for the OpenCL program and implements the OpenCL specification on the CPU. As inputs, Oclgrind receives an OpenCL application or a kernel and runs it on the CPU. Oclgrind can check barrier divergences and data races with additional options. It also enables to debug the OpenCL application or kernel interactively and exports some plugin interfaces so that any other applications can be developed.

Oclgrind can check the data race, but it can have some false positives because it only checks whether the other work-items access the same memory address or not and does not check the lock-step executions, which is described in detail in Section 3.

2.3 Data Race, Barrier Divergence, Infinite Loop

The data race happens when two or more threads access the same memory location and one of the accesses is a write operation. If many threads access the same memory, then the results are dependent on the threads' read/write orders and are unpredictable in some cases. This scenario has been a large problem to the CPU-based multithreaded system; therefore, many researchers have tried to detect it automatically with minimum false positives. FastTrack gave some idea to improve the slow but precise vector-clock race detector [8]. Relay was a static race detector based on the locksets that could be scaled to millions of lines of C code [28], and Pacer was low-overhead sampling-based data race detector based on the FastTrack [6]. However, most of these approaches are inapplicable to the GPU because the GPU programs only support the barrier function for synchronization [32].

The problem of a data race can be worse in the GPU case because thousands of work-items (threads) can run simultaneously the same instruction, and simultaneously make accesses to the same memory. When these simultaneous accesses occur, the different execution orders of the memory load or store can cause different results or hard-to-reproduce errors depending on the situation. To detect the data race at the GPU, many researchers have suggested static, dynamic, hybrid algorithms or special hardware. GPUVerify is a static verifier [5], which translates a kernel into a sequential Boogie [2] program, and proves the correctness of the sequential program. LDetector is a static detector and uses a two-pass approach to detect write-write/read-write races [11]. LD statically detects a race with a two-pass detection algorithm that is atomic free and a memory-adaptive solution [12]. Oclgrind

is a dynamic simulator, which can identify the race through monitoring the memory accesses [18]. GRace [32] and GMRace [31] are hybrid mechanisms that combine the static and the dynamic ways, reduces the number of statements, and monitors only the survived memory accesses. HAccRG [9] and Hydra [30] designed special kinds of hardware for this problem. However, these implementations have their limitations for real use. If we want to use the GPUVerify, LD, LDetector, GRace or GMRace, the compilers should be modified. If we want to use hardware-based approaches such as HAccRG and Hydra, special hardware should be added. Furthermore, GPUVerify and LDetector do not support the atomic operations. Oclgrind just includes a simple access monitor and does not consider the GPU characteristics that all work-items within the same work-group run the same instruction. Section 3 will describe this limitation through an example in more detail.

Barrier divergence is one of the specific problems of the GPU, and can also cause unexpected results. The results can be different from an architecture to an architecture or a vendor to a vendor because the OpenCL specification does not clarify the results. The OpenCL 2.1 Reference Pages [3] and the OpenCL 2.2 Reference Guide [4] only mentions that "All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the work_group_barrier." "Work-items in a work-group must execute this before any can continue." and does not mention anything about the results. Therefore, an unexpected result can happen according to the implementation. GPUVerify and Oclgrind can detect this problem.

An infinite loop can occur at both the CPU and the GPU, but its causes can be different from each other. One of the GPU-specific reasons of infinite loop is that all work-items within the same work-group run the same operation in the GPU case. Section 3 will also describe the barrier divergence and the infinite loop through an example. To the best of our knowledge, no tools or algorithms can detect this problem at this time.

3 SPECIFIC CHARACTERISTICS AND ISSUES OF OPENCL PROGRAMMING

OpenCL programming is similar to the case of CPU programming in many aspects and uses a subset of C or C++ language. However, it has different characteristics and issues that this section describes.

3.1 GPU-Specific Programming Model

The GPU threads within a group run code simultaneously in lock-step method, which highlights its difference from the CPU threads. Given that the goal of a GPU is to maximize the parallelism, it internally has thousands of processing elements. Given this characteristic, it is hard for each processing element to run the entirely separate parts of codes. Therefore, all work-items within the same work-group execute the same instruction.

If a kernel includes a conditional statement, such as *if*, then the condition can be evaluated as *true* at some work-items, but *false* at the other work-items. For this different branch case within a single work-group, the GPU uses the predicated executions. A processing element executes the predicated instructions only if the condition is *true*. Otherwise, the processing element discards the instruction or does not commit the result according to the GPU's internal architecture. Figure 3 illustrates the example of a conditional statement, and Figure 4 shows the generated instructions with a predicated form after compiling Figure 3.

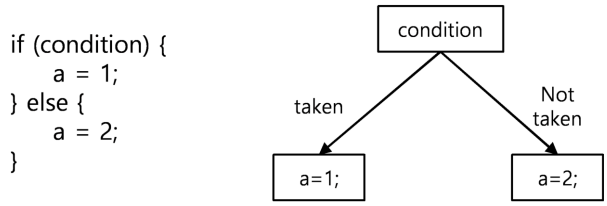

Figure 3. An example of a conditional statement

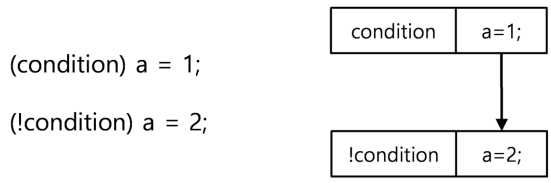

Figure 4. An example of predicated execution

Although the exact instructions can be different from architecture to architecture or vendor to vendor, Figure 4 illustrates the fundamental idea of predicated executions. In Figure 4, if the condition is *true*, the processing element runs $a = 1$. Otherwise, the processing element runs $a = 2$. With these predicated executions, all work-items within the same work-group can execute the same instruction at the same time. The loops, such as *for*, *do*, and *while*, also use these predicated executions. In this case, some processing elements can finish the loop earlier than other elements, but they cannot exit the loop and are still in the loop while ignoring the results of executed statements through this predicated executions.

3.2 Data Race

Listing 1 shows an example of a data race in the GPU case. If we run Listing 1 after setting all values in the array, g, to 1, and the number of work-items and workgroups to 128 and 8, respectively, then the work-item 15 runs the *int gid = get_global_id(0)* and *gid* becomes 15 because the *get_global_id(0)* function returns the number of work-item. The work-item 15 runs the $g[15] = g[16] + g[17]$, and

the work-item 16 runs the $g[16] = g[17] + g[18]$. Because the work-items 15 and 16 are respectively included in work-groups 0 and 1, and the memory coherency is not guaranteed between work-groups, $g[15]$ can be two if $g[16]$ and $g[17]$ are one. $g[15]$ can also be three if $g[16]$ is one and $g[17]$ is two. $g[15]$ can also be four. The results are unpredictable depending on the GPU's internal implementation, the number of work-items, and the work-groups or the memory policy. Listing 2 illustrates the example that can resolve the data race in the Listing 1.

```
1 __kernel void data_race(__global int *g)
2 {
3     int gid = get_global_id(0);
4     g[gid] = g[gid + 1] + g[gid + 2];
5 }
```

Listing 1. An example of data race

```
1  __kernel void no_data_race_1(__global int * g)
2  {
3      int gid = get_global_id(0);
4
5      int temp1 = g[gid + 1];
6      int temp2 = g[gid + 2];
7
8      barrier(CLK_GLOBAL_MEM_FENCE);
9
10     g[gid] = temp1 + temp2;
11 }
```

Listing 2. An example of resolved data race

If a work-item meets the barrier function in Listing 2, the work-item waits until the other work-items within the same work-group also reach the barrier function. Therefore, work-item 15 stores the values of $g[16]$ and $g[17]$ into the private variables $temp1$ and $temp2$, respectively. The work-item 16 has the values of $g[17]$ and $g[18]$ in the variables $temp1$ and $temp2$. The variables $temp1$ and $temp2$ are private to each work-item. After the barrier function, the work-item 15 runs $g[15] = temp1 + temp2$, and the result will be two without any data race. The work-item 16 also runs $g[16] = temp1 + temp2$, and the result will be also two.

Data race can happen or not when a GPGPU kernel is executed with the different numbers of work-groups and work-items. Listing 3 shows an example.

If we run Listing 3 with four work-items and one work-group, all of the four work-items are included in the same work-group. Therefore, all of the four work-items perform the same instruction in the lock-step method, and the data race cannot

```
1 __kernel void no_data_race_2(__global int * g)
2 {
3     int gid = get_global_id(0);
4
5     int temp1 = g[gid + 1];
6     int temp2 = g[gid + 2];
7
8     g[gid] = temp1 + temp2;
9 }
```

Listing 3. An example of the data race according to the number of work-groups and work-items

happen because the private variables of each work-item, *temp*1 and *temp*2, are used to store the values of $g[gid+1]$ and $g[gid+2]$, differently from the Listing 1. However, if we run it with four work-items and four work-groups, all of the four work-items runs separately. Therefore, the data race can occur according to the scheduling, and we should avoid the four work-groups or use the barrier function. Oclgrind does not consider this GPU-specific lock-step execution and report the data race even in one work-group case.

3.3 Barrier Divergence

Listing 4 illustrates the barrier divergence at the GPU. If we run Listing 4 with 128 work-items and 8 work-groups, work-item 0 evaluates the condition as *true* at line 5. Therefore, the $g[0]$ will be zero by line 7, and work-item 0 executes the first barrier function. Work-item 1 evaluates the condition as *false*; therefore, it performs the second barrier function at line 12. If a work-item meets a barrier function, the work-item waits until the other work-items also reach the location; therefore, work-items 0 and 1 wait for each other at different places. In this case, the results are unpredictable according to the implementation.

3.4 Infinite Loop

Listing 5 illustrates the infinite loop caused by the lock-step execution. When we run the kernel *infloop* in Listing 5 with 128 work-items and 8 work-groups after setting g to zero, work-item 0 calls the *lock* function at line 11. In the *lock* function, the *atom_xchg* function changes the global variable g into one, and returns the old value, zero, of g. Therefore, the local variable o will be zero. Then, work-item 0 can exit the loop. Meanwhile, work-item 1 also calls the *lock* function, and the *atom_xchg* function changes the global variable, g, into one again, and returns the old value, one, of g because it was already modified by work-item 0. Therefore, the local variable o will be one. Then, work-item 1 continues the loop again because it

```
1  __kernel void barrier_divergence(__global int *g)
2  {
3      int gid = get_global_id(0);
4
5      if (gid % 2 == 0)
6      {
7          g[gid] = gid;
8          barrier(CLK_GLOBAL_MEM_FENCE);
9      }
10     else
11     {
12         barrier(CLK_GLOBAL_MEM_FENCE);
13         g[gid] = gid + g[gid - 1];
14     }
15 }
```

Listing 4. An example of the barrier divergence

does not satisfy the exit condition $o \leq 0$. Besides work-item 0, all work-items in the workgroup continue the loop. Work-item 0 satisfies the exit condition, but it cannot also exit the loop because it should run the same instruction in the lock-step similar to other work-items. As a result, all work-items will loop forever.

```
1  void lock(__global int* g)
2  {
3      int o;
4
5      do {
6          o = atom_xchg(g, 1);
7      } while (o > 0);
8  }
9
10 __kernel void infloop(__global int* g)
11 {
12     lock(g);
13 }
```

Listing 5. An example of infinite loop

We checked that Listing 5 caused the infinite loop, and the system sometimes halts at Intel's OpenCL implementation (HD Graphics 530, Driver version 20.19.15.4 531) and NVIDIA's implementation (GTX 960M, Driver version 378.78). This problem is quite common because this kind of codes, such as Listing 5, works well in the CPU case as the locking codes. In the CPU case, the thread 0 (work-item) does not

have to loop again; therefore, the CPU does not meet the infinite loop. However, the GPU is frozen by the lock-step execution.

4 CL-VIS: VISUALIZATION PLATFORM

This section describes our platforms for the visualization of running GPGPU codes (CL-Vis) and algorithms to find the issues described in Section 3.

4.1 Overview

When programming the GPU, it is hard to understand what each work-item is doing. Without knowing what went on, it is hard to fix the problems or the bugs within the software. If a developer can see what the work-item does, he/she can easily understand why the problem happens in many cases and can fix the problems. To help understand the internal operations of each work-item, we developed a visualization platform based on Oclgrind. To implement it, we used the architecture in Figure 5.

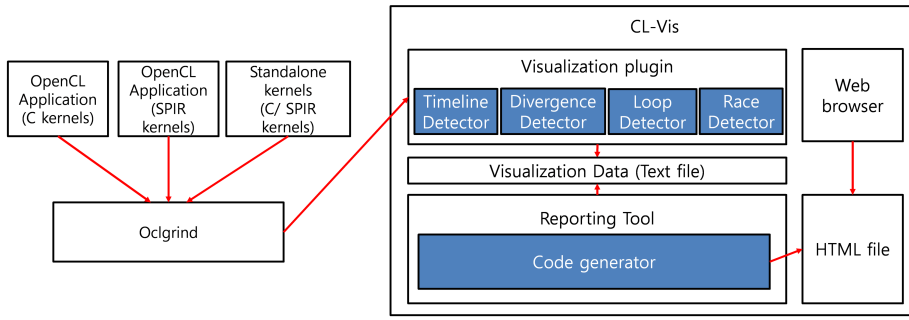

Figure 5. The overall architecture of CL-Vis

Our visualization platform, CL-Vis, largely consists of the following components: a visualization plugin and a reporting tool. As an input, Oclgrind receives the information file that includes the running kernel/file, the number of work-items and work-groups, and the parameter values, and runs the kernel. During the execution, it calls our visualization plugin that implements the callback functions. Oclgrind supports the following basic callback functions in Table 2 so that any user can develop additional tools based on Oclgrind. Besides the essential functions in Table 2, we extended Oclgrind so that we can obtain more information and added more functions in Table 3.

Through these callback functions in Tables 2 and 3, the visualization plugin records the information on the executed instructions and their times to thread-specific files. The timeline of each operation is detected through Algorithm 1.

Through Algorithm 1, the visualization plugin records the global id (the thread id), the time difference between the current time and the kernel's starting time, the C-based statement, the assembly-based instructions to a buffer, and writes the buffer to a file that includes the execution information of a work-item. It also detects the infinite loop, the barrier, and the data race through the algorithms described in more detail in Subsection 4.2. The reporting tool uses the file of each work-item for the visualization. The reporting tool receives the generated files by the visualization plugin as inputs and generates an HTML file as outputs. After these procedures, the HTML file can be viewed through a web browser. To summarize, our platform can visualize the codes after the execution completes. Figure 6 presents the result of our visualization.

ALGORITHM 1: Timeline detector

```
1: for each work-item w do
2:     for each start and end of an instruction do
3:         Record the w's id, time difference between the current time and the w's starting
           time, the statement and the executed instruction to a buffer
4:     end for
5:     Write the recorded buffer to a file
6: end for
```

In Figure 6, the horizontal axis shows the time, and the vertical axis lists the number of work-item and workgroup. Each blue rectangle represents an operation that a work-item performed, and the text in the box is the executed action at the time. If we scroll down the page, we can see other threads. We can also see the other operations if we move the page to the left or right. We can also zoom in/out the box through a mouse wheel. In many cases, a single line of the C-based statement consists of many lines of assembly-based statements, and many boxes can include the same text. Therefore, our platform generates the C-based statement first, and the assembly-based statement within a parenthesis next to avoid confusion.

Figure 6 shows that a single work-group includes 16 work-items, and the work-items run independently. We can also see the executed action if we move the mouse cursor on the box or horizontally increase the box. Through this visualization, we can check that the program operates as intended. For example, we can see that the work-item 0 runs the *true* case, and the other work-items run the *false* case at the *if* statement. We can also check the number of loops that the work-items run. Besides the checking, this visualization can also be used for the education purpose, and we can use this platform to describe the concepts of GPGPU programming.

4.2 Detecting Algorithms

This subsection describes the algorithms used to detect the data race, the barrier divergence, and the infinite loop included in the CL-Vis. We designed them as

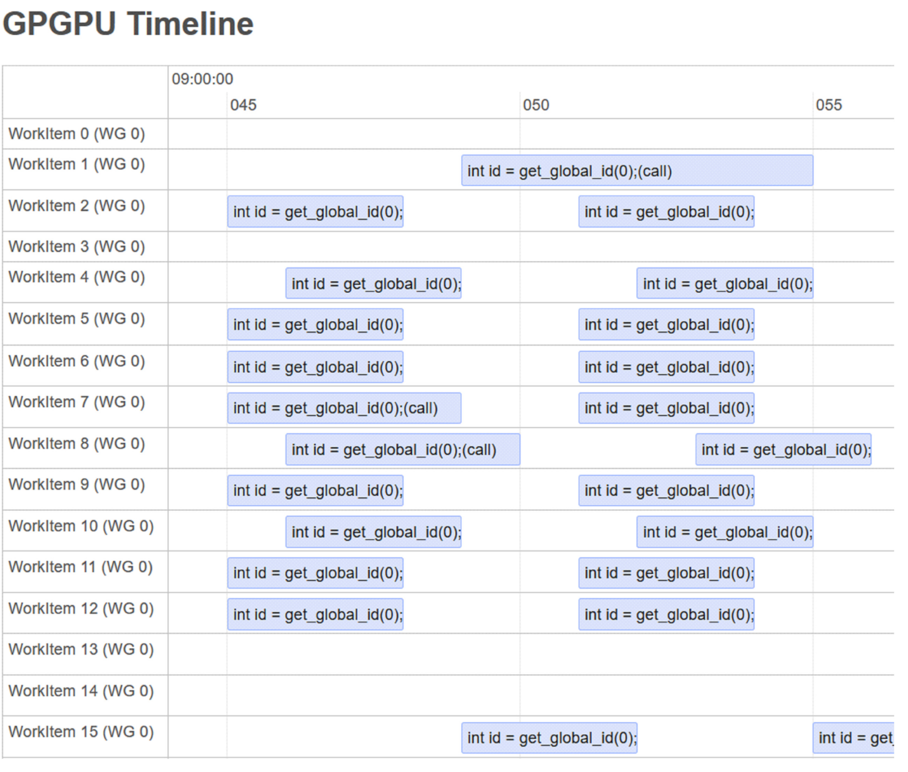

Figure 6. Visualization of the GPGPU program

a plugin of Oclgrind, and it is an event-based program. Therefore, we described each algorithm using events.

4.2.1 Race Detector

The visualization plugin includes the race detector that detects a data race through Algorithms 2 and 3. Algorithm 2 monitors all memory accesses of work-item w and reports the data race if the below conditions 1, 2, 3 are evaluated as *true*, and condition 4 or 5 is evaluated as *true*. Algorithm 3 changes the states of memory accesses into inactive if a work-item meets a barrier.

1. The first condition is that the address space of access is *global* or *local*. Work-items share only the global and local memories so that these types can have a data race.
2. The second condition is that the access to memory is active. If a work-item meets a barrier, all of the previous memory accesses are guaranteed to be committed. Therefore, the memory accesses become inactive in this case.

3. The third condition is that another thread accesses the same memory, and one of the access is the write. The data race can happen only in this case.
4. The fourth condition is that the accessing statement is the same in the case that the current work-item and the work-item of the previously access are in the same group. The work-items in the same work-group run an operation in lock-step; therefore, the data race will happen only if the accessing statements are the same.
5. The fifth condition is that the memory type is global if the work-items are in different groups. The work-items in different groups can be simultaneously executed based on the GPU's internal policy; therefore, the data race can happen.

ALGORITHM 2: Race detector

```
1: for each work-item w do
2:   for each memory access m do
3:     if address space of m is global or local then
4:       for each access entry e ∈ access table do
5:         if e is active, another thread also accesses m's address, and one of the
           accesses is the write operation then
6:           if w and the work-item of e are in the same group then
7:             if the accessing statement is the same then
8:               Mark the access m as a data race
9:             end if
10:          else
11:            if address space of m is global then
12:              Mark the access m as a data race
13:            end if
14:          end if
15:        end if
16:      end for
17:    end if
18:    Record m's address, size, statement, memory type (global/local), access type
       (read/write) and group number into the access table
19:  end for
20: end for
```

ALGORITHM 3: Access clearing

```
1: for each work-item w do
2:   if w meets a barrier then
3:     Change the all access states of w in the access table to inactive
4:   end if
5: end for
```

Most of the previous dynamic algorithms for detecting the data race do not include conditions 4 and 5. They just check conditions 1, 2 and 3, so have more false positives. To decrease the false positives, we added condition 4 utilizing that the work-items in a work-group run the code in lock-step. We also added condition 5 because the data race can happen in the global memory case even if the accessing statements are different.

4.2.2 Divergence Detector

The visualization plugin includes the divergence detector that detects a barrier divergence through Algorithms 4 and 5. Whenever a work-item meets a barrier function, Algorithm 4 records the global id (the thread id), the finishing time and statement. We record the statement because a kernel can have many barrier functions and barrier divergence, and we need to identify the barrier function. We also record the finishing time because our visualization platform should find the most recently cleared barrier function. Then, when Oclgrind clears the barrier function, Algorithm 4 marks the latest time among the met work-items as a barrier time because Oclgrind is a simulator based on the CPU, which sequentially executes a kernel differently from the GPU. Algorithm 5 uses the most recent time because all work-items should wait for the last finished work-item.

ALGORITHM 4: Barrier recording at the divergence detector

```
1: for each work-item w do
2:     if w meets a barrier then
3:         Record the w's id, the finishing time and the statement
4:     end if
5: end for
```

ALGORITHM 5: Barrier clearing at the divergence detector

```
1: for each barrier b do
2:     Mark the last finished work-item's time among the met work-items as a barrier
       time
3: end for
```

Algorithms 4 and 5 can find all of the barrier divergences without a false positive because Algorithm 4 records all of the work-items that meet a barrier, and Algorithm 5 marks to all of the met work-items when a barrier is cleared.

4.2.3 Loop Detector

The visualization plugin includes the loop detector that detects a possible infinite loop through Algorithm 6. In Algorithm 6, whenever a work-item w meets a loop l, our visualization plugin records the used variables if they are global and modified

within the loop. Our plugin also records the affected variable by a global variable. Then, our plugin checks that the changed or affected variables are used at the exit condition of the loop.

ALGORITHM 6: Loop detector

```
1: for each work-item w do
2:   for each loop l do
3:     Record the used variables if they are global and modified within the loop l
4:     Record the affected variables by a global variable if they are private or local,
       and modified within the loop l
5:     if the global or affected variables are used as an exit condition then
6:       Mark the exit condition as an infinite loop
7:     end if
8:   end for
9: end for
```

If a work-item changes the global variable and uses the variable at the exit condition, then other work-items can also be affected because the exit condition can also be modified. The count of a loop can be shorter, longer or even infinite. In Listing 5, the global variable, g, is modified within the loop and affects the private variable o. The other work-items are also affected by the modification of g, and the exit condition is also affected by the change of o. The infinite loop at the GPU happens in this case; therefore, our plugin detects the changes of an exit condition.

Algorithm 6 may have a false positive if a global variable is modified or a local/private variable is modified by a global variable within a loop, and the variable is used as an exit condition. But, we have never met such case in many kernels until now.

4.3 Detailed Implementation

To verify our architecture and algorithms, we implement the visualization plugin and the reporting tool. Besides the function additions in Table 3, we also disabled the optimization of a GPGPU kernel code because we should obtain the statement information in our algorithms.

We implemented the algorithms in Section 4 at our visualization plugin and the reporting tool in the Linux environment (Ubuntu 14.04), and verified the results using Firefox 52.0. To visualize the results, we used a timeline within the *vis.js* library [23] that supports powerful display functions for the web browser. To determine whether two statements are the same or not in Algorithm 2, we used their line numbers as statement information.

5 RESULTS OF DETECTING ALGORITHMS

This section presents the results of suggested architecture and algorithms.

5.1 Data Race

We run Listing 1 with 128 work-items and 8 work-groups. Figure 7 presents the results. In Figure 7, our Algorithm 2 detects the data race at the $g[gid] = g[gid + 1] + g[gid + 2]$ statement in Listing 1, and marks them with yellow color (Red box). Therefore, any researcher or developer can easily notice it. We also run Listing 2 with 128 work-items and 8 work-groups. Our Algorithms 2 and 3 did not detect the race because the barrier function cleared all of the previous memory accesses. However, Oclgrind reports some data races.

Figure 7. The result of detecting data race at Listing 1. Algorithm 2 detects the data race at the $g[gid] = g[gid + 1] + g[gid + 2]$ statement in Listing 1, and marks them with yellow color (red box).

Figure 8 illustrates the results of no race and barrier.

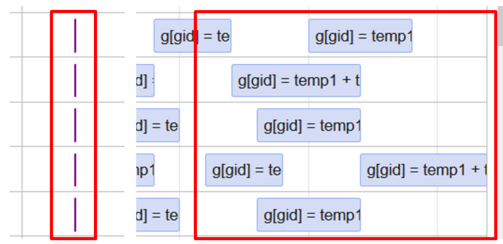

Figure 8. The result of detecting data race at Listing 2. Algorithms 2 and 3 did not detect the race at the statement that Oclgrind reports some data races but they cannot happen in the real scenario (red box in the right side).

In Figure 8, our CL-Vis platform does not report any data races with yellow color (the right red box) because the barrier function cleared all of the previous memory accesses and correctly marks the barrier function with magenta color (the left red box).

We also run Listing 3 with one work-group and four work-groups. Our Algorithms 2 and 3 can correctly detect the race from the lock-step execution only in the four work-group, as shown in Figures 7 and 8.

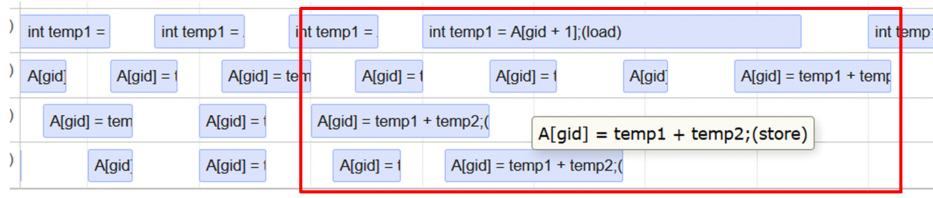

Figure 9. The result of detecting data race with one work-group at Listing 3. Given that the accessing statements are at different locations, Algorithm 2 does not report it as a data race (red box).

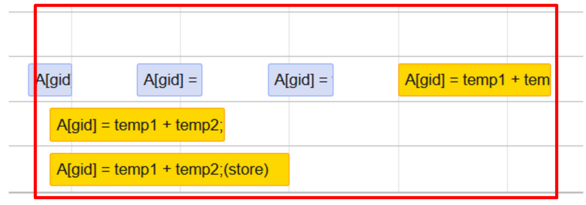

Figure 10. The result of detecting data race with four work-groups at Listing 3. The work-item can run the separate instruction in the different work-group case; therefore, our algorithm reports the data race (red box).

In the one work-group case, such as Figure 9, all work-items in the same work-group run the same instruction, and our Algorithm 2 checks that the accessing statement is the same. Given that the accessing statements are at different locations, our Algorithm 2 does not report it as a data race. However, in the different work-group case, such as Figure 10, the work-item can run the separate instruction; therefore, our algorithm reports the data race (red box).

5.2 Barrier Divergence

We run Listing 4 with four work-items and four work-groups, detect the barrier divergence through Algorithms 4 and 5, and mark it with magenta as shown in Figure 11 (red box). In Figure 11, the bar with magenta color shows that work-items 0 and 2 run the first barrier in Listing 4, but work-items 1 and 3 execute the second barrier in Listing 4. If all of the work-items meet the same barrier function, then only one bar with a magenta color exists. Therefore, any researcher or developer can see that the barrier divergence happened.

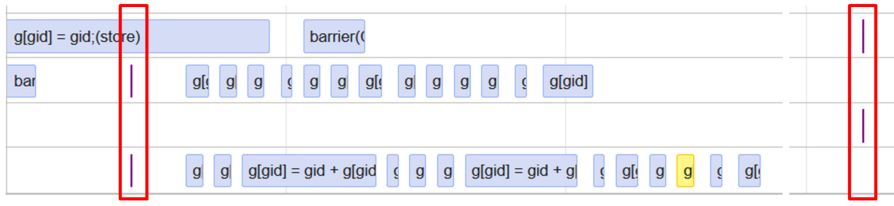

Figure 11. The result of detecting barrier divergence at Listing 4. Algorithms 4 and 5 mark the barrier divergence with magenta in the case of four work-items and four work-groups.

5.3 Infinite Loop

We run Listing 5 with 128 work-items and 8 work-groups as shown in Figure 12. In Figure 12, our Algorithm 6 detects the infinite loop and marks it with red color (red box). It records the global or the affected variables within a loop, then checks that the exit condition uses the variables. Through this algorithm, our platform detects the infinite loop, and shows it at the statement $while(o > 0)$ in Listing 5.

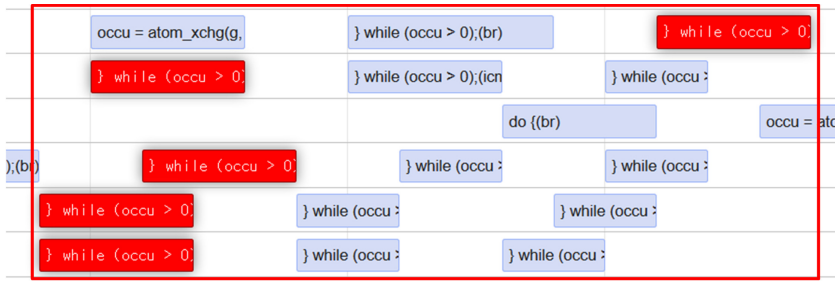

Figure 12. The result of detecting infinite loop at Listing 5. Algorithm 6 records the global or the affected variables within a loop, and checks that the exit condition uses the variables. It also marks the infinite loop with red color.

5.4 Others

Besides these primary results through the kernels in this paper, we also verify that our architecture and algorithms can successfully visualize 15 kernels within Oclgrind. Table 1 summarizes the tested kernels and their sizes of global/local works. Among the kernels in Table 1, Oclgrind wrongly reports the data races in the *global_only_fence*, *intragroup_hidden_race*, *local_read_write_race* cases because it does not check the lock-step execution at the GPU. Furthermore, Oclgrind reports the data races in the *local_only_fence* because it does not clear the memory access

before the barrier function. However, our algorithm correctly finds no data race in all of those cases.

Group	File	Global Size	Local Size
Barrier	barrier_different_instructions	4, 1, 1	4, 1, 1
Divergence	barrier_divergence	4, 1, 1	4, 1, 1
Data Race	broadcast	4, 1, 1	1, 1, 1
	global_fence	16, 1, 1	4, 1, 1
	global_only_fence	4, 1, 1	4, 1, 1
	global_read_write_race	4, 1, 1	4, 1, 1
	global_write_write_race	4, 1, 1	1, 1, 1
	increment	4, 1, 1	1, 1, 1
	intergroup_hidden_race	2, 1, 1	1, 1, 1
	intergroup_race	8, 1, 1	4, 1, 1
	intragroup_hidden_race	2, 1, 1	2, 1, 1
	local_only_fence	16, 1, 1	4, 1, 1
	local_read_write_race	4, 1, 1	4, 1, 1
	local_write_write_race	4, 1, 1	4, 1, 1
	uniform_write_race	4, 1, 1	4, 1, 1

Table 1. Tested kernels

6 CONCLUSIONS

GPU programming is known to be difficult due to several reasons: difficulty in understanding the GPU characteristics, which are different from the CPU, and few of debugging tools compared to the CPU. To reduce these problems and help in checking the GPGPU programs, we developed the CL-Vis based on Oclgrind. We also suggest the algorithms for automatic detection of data race with lock-step execution and barrier function, barrier divergence, and infinite loop. These algorithms are included in the CL-Vis and verified through various kernels. To the best of our knowledge, our paper is the first research that visualizes the executed functions of GPU and suggests an algorithm for detecting infinite loops automatically. Furthermore, our paper suggests the algorithm for detecting data races at the GPU-specific executions.

However, our CL-Vis and algorithms can be improved more in the following aspects: our plugin is based on Oclgrind, which is an OpenCL simulator run on the CPU. However, the real working environment with the GPU can be different. For example, the actual execution time can be different. If the GPU vendor develops its simulator that is more similar to its internal operations, then we can visualize the running codes based on it and see the instructions more correctly. Furthermore, our race detection algorithm is not aware of the barrier function with a local memory option. If a work-item meets a barrier function, only the local memory accesses within the same group should be cleared. The current implementation makes all

of the global/local memory accesses inactive. Also, our visualization platform has a limitation in the number of threads due to the restricted memory. Finally, if we can see the result of each action by a work-item, it can also be helpful to check the OpenCL program. We have plans to improve these aspects and find other ways to relieve the burden of GPU users.

Acknowledgements

This research was supported by the MSIT (Ministry of Science, ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2015-0-0445) supervised by the IITP(Institute for Information and communications Technology Promotion). SeongKi Kim was supported by NRF in Korea (NRF-2017R1A1A1A05069806).

REFERENCES

[1] Apple Coporation. Availaible at: `https://www.apple.com/`.

[2] Barnett, M.—Chang, B.-Y. E.—DeLine, R.—Jacobs, B.—Leino, K. R. M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F. S., Bonsangue, M. M., Graf, S., de Roever, W. P. (Eds.): Formal Methods for Components and Objects (FMCO 2005). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4111, 2006, pp. 364–387, doi: 10.1007/11804192_17.

[3] OpenCL 2.1 Reference Pages. Availaible at: `https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/`.

[4] OpenCL 2.2 Reference Guide. Availaible at: `https://www.khronos.org/files/opencl22-reference-guide.pdf`.

[5] Betts, A.—Chong, N.—Donaldson, A.—Qadeer, S.—Thomson, P.: GPUVerify: A Verifier for GPU Kernels. ACM SIGPLAN Notices – OOPSLA '12, Vol. 47, 2012, No. 10, pp. 113–132, doi: 10.1145/2398857.2384625.

[6] Bond, M. D.—Coons, K. E.—McKinley, K. S.: PACER: Proportional Detection of Data Races. ACM SIGPLAN Notices – PLDI '10, Vol. 45, 2010, No. 6, pp. 255–268, doi: 10.1145/1809028.1806626.

[7] Fernando, R.: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Pearson Higher Education, 2004.

[8] Flanagan, C.—Freund, S. N.: FastTrack: Efficient and Precise Dynamic Race Detection. ACM SIGPLAN Notices – PLDI '09, Vol. 44, 2009, No. 6, pp. 121–133, doi: 10.1145/1543135.1542490.

[9] Holey, A.—Mekkat, V.—Zhai, A.: HAccRG:: Hardware-Accelerated Data Race Detection in GPUs. Proceedings of the 2013 42nd International Conference on Parallel Processing (ICPP '13), 2013, pp. 60–69, doi: 10.1109/icpp.2013.15.

[10] Howes, L. (Ed.): The OpenCL Specification Version: 2.1. Document Revision: 24, 2018. `https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf`.

[11] Li, P.—Ding, C.—Hu, X.—Soyata, T.: LDetector: A Low Overhead Race Detector for GPU Programs. 5^{th} Workshop on Determinism and Correctness in Parallel Programming (WoDET 2014), Salt Lake City, UT, March 2014.

[12] Li, P.—Hu, X.—Chen, D.—Brock, J.—Luo, H.—Zhang, E. Z.—Ding, C.: LD: Low-Overhead GPU Race Detection Without Access Monitoring. ACM Transactions on Architecture and Code Optimization, Vol. 14, 2017, No. 1, Art. No. 9, 25 pp., doi: 10.1145/3046678.

[13] Nickolls, J.—Buck, I.—Garland, M.—Skadron, K.: Scalable Parallel Programming with CUDA. Queue – GPU Computing, Vol. 6, 2008, No. 2, pp. 40–53, doi: 10.1145/1365490.1365500.

[14] NVIDIA Corporation. Driving Innovation. Availaible at: `http://www.nvidia.com/object/drive-automotive-technology.html`.

[15] Khronos Group. Availaible at: `https://www.khronos.org/`.

[16] Ouril, B. (Ed.): The SPIR Specification Version 2.0 – Provision. 2014.

[17] Pharr, M.—Fernando, R.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (GPU Gems). Addison-Wesley Professional, 2005.

[18] Price, J.—McIntosh-Smith, S.: Oclgrind: An Extensible OpenCL Device Simulator. Proceedings of the 3^{rd} International Workshop on OpenCL (IWOCL '15), New York, NY, USA, 2015, Art. No. 12, doi: 10.1145/2791321.2791333.

[19] Silver, D.—Huang, A.—Maddison, C. J.—Guez, A.—Sifre, L.—van den Driessche, G.—Schrittwieser, J.—Antonoglou, I.—Panneershelvam, V.—Lanctot, M.—Dieleman, S.—Grewe, D.—Nham, J.—Kalchbrenner, N.—Sutskever, I.—Lillicrap, T.—Leach, M.—Kavukcuoglu, K.—Graepel, T.—Hassabis, D.: Mastering the Game of Go with Deep Neural Networks and Tree Search. Nature, Vol. 529, 2016, No. 7587, pp. 484–489, doi: 10.1038/nature16961.

[20] Stone, J. E.—Gohara, D.—Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science and Engineering, Vol. 12, 2010, No. 3, pp. 66–73, doi: 10.1109/MCSE.2010.69.

[21] Stone, S. S.—Haldar, J. P.—Tsao, S. C.—Hwu, W.-M. W.—Sutton, B. P.—Liang, Z.-P.: Accelerating Advanced MRI Reconstructions on GPUs. Journal of Parallel and Distributed Computing, Vol. 68, 2008, No. 10, pp. 1307–1318, doi: 10.1016/j.jpdc.2008.05.013.

[22] TOP 500 The List. Availaible at: `https://www.top500.org/`.

[23] vis.js. Availaible at: `http://http://visjs.org/`.

[24] NVIDIA Nsight Visual Studio Edition. `https://developer.nvidia.com/nsight-visual-studio-edition`.

[25] Allinea DDT. `https://developer.nvidia.com/allinea-ddt`.

[26] CUDA-MEMCHECK. `https://docs.nvidia.com/cuda/cuda-memcheck/index.html`.

[27] Profiler. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[28] Voung, J. W.—Jhala, R.—Lerner, S.: RELAY: Static Race Detection on Millions of Lines of Code. Proceedings of the the 6^{th} Joint Meeting of the Euro-

pean Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07), New York, NY, USA, 2007, pp. 205–214, doi: 10.1145/1287624.1287654.

[29] Yang, J.—Goodman, J.: Symmetric Key Cryptography on Modern Graphics Hardware. In: Kurosawa, K. (Ed.): Advances in Cryptology – ASIACRYPT 2007. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4833, 2007, pp. 249–264, doi: 10.1007/978-3-540-76900-2_15.

[30] Zhang, W.—Yu, S.—Wang, H.—Dai, Z.—Chen, H.: Hardware Support for Concurrent Detection of Multiple Concurrency Bugs on Fused CPU-GPU Architectures. IEEE Transactions on Computers, Vol. 65, 2016, No. 10, pp. 3083–3095, doi: 10.1109/TC.2015.2512860.

[31] Zheng, M.—Ravi, V. T.—Qin, F.—Agrawal, G.: GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. IEEE Transactions on Parallel and Distributed Systems, Vol. 25, 2014, No. 1, pp. 104–115, doi: 10.1109/TPDS.2013.44.

[32] Zheng, M.—Ravi, V. T.—Qin, F.—Agrawal, G.: GRace: A Low-Overhead Mechanism for Detecting Data Races in GPU Programs. ACM SIGPLAN Notices – PPoPP '11, Vol. 46, 2011, No. 8, pp. 135–146, doi: 10.1145/2038037.1941574.

SeongKi Kim is Assistant Professor at Keimyung University. He received his Ph.D. degree in computer science and engineering from Seoul National University in 2009. He researched and developed software for the GPU, the GPGPU and dynamic voltage and frequency scaling (DVFS) at the Samsung Electronics from 2009 to 2014. He also worked at the Ewha Womans University and SangMyung University from 2014 to 2017. His current research interests include the areas of graphics/game algorithms, an algorithm optimization through the GPU, and high-performance computing with CPU and GPU.

HyukSoo Han is Professor of computer science at SangMyung University. He received his M.Sc. degree in computer science and statistics from Seoul National University in 1987 and his Ph.D. degree in computer engineering from South Florida University in 1992. His current research interests include software process, software quality, software safety, software education and human computer interaction.

Appendices

Name	Description
hostMemoryLoad	Called when a host memory is loaded
hostMemoryStore	Called when a host memory is stored
instructionExecuted	Called when an instruction is executed
kernelBegin	Called when a kernel starts
kernelEnd	Called when a kernel ends
log	Called when a log message is outputted
memoryAllocated	Called when a memory is allocated
memoryAtomicLoad	Called when an atomic memory is loaded
memoryAtomicStore	Called when an atomic memory is stored
memoryDeallocated	Called when a memory is deallocated
memoryLoad	Called when a memory is loaded
memoryMap	Called when a memory is mapped
memoryStore	Called when a memory is stored
memoryUnmap	Called when a memory is unmapped
workGroupBarrier	Called when a barrier is cleared
workGroupBegin	Called when a work-group begins
workGroupComplete	Called when a work-group completes
workItemBegin	Called when a work-item begins
workItemComplete	Called when a work-item completes

Table 2. Callback functions

Name	Description
instructionBeforeExecuted	Called before an instruction is executed
workGroupBarrierBeforeClear	Called before a barrier is cleared
workItemBarrier	Called when a work-item clears a barrier

Table 3. Added Callback functions