Parallel GPU Architecture Simulation Framework Exploiting Work Allocation Unit Parallelism

Sangpil Lee and Won Woo Ro
School of Electrical and Electronic Engineering
Yonsei University
Seoul, Republic of Korea
E-mail: {madfish, wro}@yonsei.ac.kr

***Abstract*—GPU computing is at the forefront of high-performance computing, and it has greatly affected current studies on parallel software and hardware design because of its massively parallel architecture. Therefore, numerous studies have focused on the utilization of GPUs in various fields. However, studies of GPU architectures are constrained by the lack of a suitable GPU simulator. Previously proposed GPU simulators do not have sufficient simulation speed for advanced software and architecture studies. In this paper, we propose a new parallel simulation framework and a parallel simulation technique called work-group parallel simulation in order to improve the simulation speed for modern many-core GPUs. The proposed framework divides the GPU architecture into parallel and shared components, and it determines which GPU component can be effectively parallelized and can work correctly in multithreaded simulation. In addition, the work-group parallel simulation technique effectively boosts the performance of parallelized GPU simulation by eliminating the synchronization overhead. Experimental results obtained using a simulator with the proposed framework show that the proposed parallel simulation technique has a speed-up of up to 4.15 as compared to an existing sequential GPU simulator on an 8-core machine providing minimized cycle errors.**

I. Introduction

A graphics processing unit (GPU) is a specialized many-core processor that provides considerable processing power for delivering high-performance graphics. Modern GPUs have hundreds or even thousands of processing elements in order to process massive complex operations. For example, NVIDIA's Fermi GPU contains up to 512 processing elements, and it can process thousands of threads in parallel [1]. AMD's Radeon HD 6000 series of GPUs contain more than 1000 processing elements on a single GPU die [2]. This architectural characteristic of GPUs has attracted considerable attention in computing-intensive fields, and studies have actively focused on utilizing GPU processing elements for high-performance computing. These studies have led to the concept of general-purpose computing on GPU (GPGPU), which is now widely used for parallel computing.

The increasing demands on GPUs necessitate a rapid increase in the number of processing elements they contain. This trend of architectural development on GPUs also has expedited several GPU architecture studies to maximize utilization of a large number of processing elements on GPU and to improve their performance. GPU architecture simulators also have been proposed to support these studies. ATTILA, [3], GPGPU-sim [4], and Multi2Sim [5] are simulators that support cycle-level GPU architecture simulation. Among these, GPGPU-sim is the first GPU simulator that can perform functional/cycle-level timing simulation for GPGPU applications using the Compute Unified Device Architecture (CUDA) [6] or Open Computing Language (OpenCL) [7].

However, a large number of processing elements cause another problem in GPU studies. The simulation speed for a large number of processing elements integrated on a GPU is too slow and practically not acceptable. This is because the modeling interactions among hundreds of processing elements and other hardware components are quite complicated. In fact, currently available GPU simulators are not parallelized, which mainly causes the performance degradation. For example, GPGPU-sim suffers from a 170,000 to 2,000,000× simulation slowdown due to its sequential structure.

To address the performance issues involved in typical many-core processors, several parallel simulation frameworks such as SlackSim [8] and Graphite [9] have been proposed. These parallel simulators simulate the parallel many-core architecture using multi-core processors and/or multiple simulation hosts. In fact, a GPU is also a type of many-core processor, and therefore, those previously proposed parallel simulation techniques might provide a possible solution to parallel GPU architecture simulation. However, there is architectural difference between GPUs and many-core CPUs, and this difference causes some restriction when we apply the previous parallel simulation techniques to the GPU simulation.

In fact, each processing element of a GPU is not a complete core as is the case with typical processors; actually, it is a simple execution unit which contains ALU/FPU pipelines. To minimize control complexity of hundreds of processing elements, the GPU has some hierarchies of control; several processing elements are grouped with minimal control logic for executing instructions, and work distribution/control for each group of processing elements is performed by a shared control unit in the GPU. In other words, the GPU has a centralized control structure to drive parallel computing resources. This is quite different from typical many-core processors, which have complete, standalone structured cores. Due to this characteristic of a GPU, prior studies cannot be adapted directly for the GPU architecture simulation. Therefore, a new simulation methodology is required for considering the structural characteristics of a GPU.

In this paper, we propose a new parallel simulation frame-

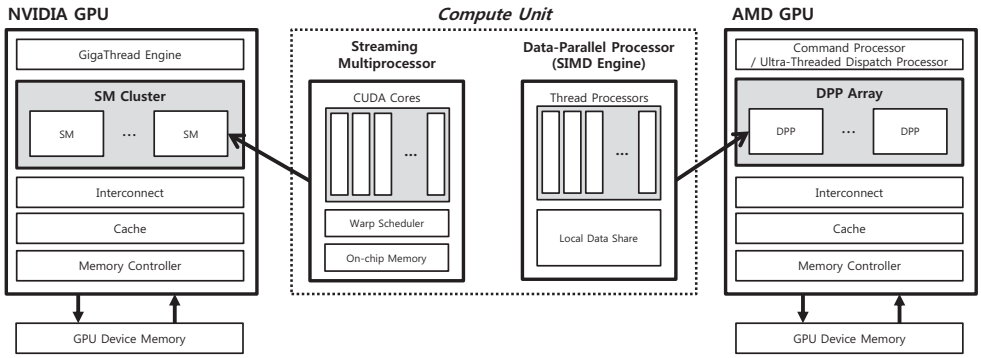


Fig. 1: GPU architecture block diagram.

work for improving GPU architecture simulation performance. The framework divides the entire GPU architecture into two parts: *parallel components* and *shared components*. The parallel components of the GPU architecture include processing elements, and they can be perfectly parallelized. The shared components include processor control units for the parallel components, interconnection networks, and memory subsystems. The shared components are defined since they are shared by the independent parallel components and should be simulated considering synchronization issues. This classification determines which component of the target GPU architecture can be simulated in parallel.

Another challenge of the GPU simulation is minimizing synchronization overhead on parallel simulation. Although *cycle-by-cycle* simulation is widely used for cycle-accurate simulation and the simulation speed would be accelerated by parallel cycle-by-cycle simulation, it suffers from synchronization overhead. Previous literatures introduced several techniques for relaxing synchronization overhead on parallel simulation [8], [9], but they did not define or address technical issues in simulating GPUs.

Based on the proposed parallel simulation framework, we propose the *work-group parallel simulation* technique, which improves the simulation speed by using a multi-core host machine. This technique focuses on minimizing synchronization overhead using the work allocation unit parallelism of a GPU. Consequently, the proposed technique effectively improves simulation speed with low cycle errors.

The performance gain of the proposed parallel simulation framework has been evaluated on an existing GPU simulator using various CUDA applications, and performance of the parallel simulation technique that successfully exploits multithreaded simulation has been demonstrated. In fact, an average speed-up of 3.39 has been achieved as compared to that in GPGPU-sim using 6 parallel threads (4 threads for the parallel components simulation and 2 threads for the shared components simulation). With the new parallel simulation technique and error handling, the proposed parallelized GPU simulator also achieved low simulation errors.

The remainder of this paper is organized as follows. Section II describes overview of the GPU architectures and defines the performance problems of a sequential simulator. In addition, design challenges for parallel GPU simulation is presented. In Section III, the proposed parallel simulation framework for GPUs is described in details. In Section IV, performance evaluation of the parallelized GPU simulator and analysis on the results are presented. Finally, prior related work is introduced in Section V and the conclusion is presented in Section VI.

II. Preliminary Study of Parallel GPU Architecture Simulation

In this section, we review the architectural characteristics of GPUs and investigate the performance issues of existing GPU simulators.

A. Overview of GPU Architecture

GPUs have architectural differences depending on the manufacturer. Fig. 1 shows the architecture block diagram of popular GPUs: NVIDIA's Tesla/Fermi GPU architecture and AMD's Evergreen/Northern Island family GPU architecture. These GPUs have a streaming multiprocessor (SM) cluster or a data-parallel processor (DPP) array, which corresponds to processor cores. Each SM has 8 to 48 CUDA cores that contain integer/floating point units, a warp scheduler for instruction scheduling, and several on-chip memories [10]. DPP has 16 thread processors that contain the VLIW architecture execution pipelines and on-chip memories for data sharing [2]. In this paper, each SM or DPP is termed as a compute unit (CU) for architecture-independent description.

In addition to the CUs, there is a control unit for operating the CUs in both GPU architectures: GigaThread Engine in the NVIDIA GPU [10] and command processor/ultra-threaded dispatch processor in the AMD GPU [2]. The GPUs also contain interconnection networks between the CUs and the memory subsystems including caches, memory controller, and GPU device memory.

The programming model for a GPGPU is designed to process a massive amount of data simultaneously using CUs in the GPU. It includes a *kernel*, which is a program executed on the GPU. In the CUDA programming model for NVIDIA GPUs, thousands of threads are instantiated by referring to the kernel code, and these are used to process data in parallel. The threads are grouped into multiple *cooperative thread arrays* (CTAs), and CTAs are allocated to and processed by CUs. The OpenCL execution model, which is an open-standard computing model used in AMD GPUs, differs slightly from the CUDA model. It uses a *work-item* and *work-group*, which correspond to a

TABLE I: Simulation performance of GPGPU-sim

Application	Native GPU execution time (ms)	Simulation time (ms)	Slowdown
matrixmul	0.128	30578	177784
MersenneTwister	11	7511800	682891
scan	0.337	253300	751632
QuasirandomGenerator	1.33	2766411	2080009
MonteCarlo	2.59	4213485	1646536
clock	0.037	6491	175432
scalarProd	0.177	94022	531201
BlackScholes	0.749	1574676	2102372

TABLE II: Characteristics of GPU computing applications

Application	Dynamic Instruction Mix (%)				GPU Component Simulation Workload (%)		
	INT/FP	Branch	Load	Store	CU	ICNT	DRAM
matrixmul	45.1	1.9	49.9	3.1	98.0	1.6	0.5
Mersenne Twister	89.2	3.6	3.6	3.6	96.2	2.9	0.9
scan	76.1	11.4	7.2	5.3	98.4	1.1	0.4
quasirandom Generator	79.0	15.8	4.3	0.9	98.7	0.9	0.4
MonteCarlo	86.6	6.7	6.6	0.1	97.2	2.2	0.6
clock	77.2	16.5	4.8	1.4	97.6	1.7	0.7
scalarProd	71.6	14.0	12.5	2.0	95.3	3.7	1.0
BlackScholes	93.7	1.0	3.2	2.1	97.5	1.9	0.6

thread and CTA in CUDA, and these are allocated to and processed by CUs as well. Although the internal hardware structure of the CU in the two GPU architectures is completely different, the high-level execution model is similar. In this study, a work-item/work-group is generally used for indicating the work allocation unit of a CU in both GPU architectures.

Current state-of-the-art GPUs integrate more execution units and more CUs in order to process more data in parallel. While this strategy effectively improves the computing power of the GPU, it also leads to a rapid increase in the architectural complexity of the GPU. This makes GPU architecture simulation even more challenging because a heavy computation overhead is inevitable in the architectural simulation of a GPU. Currently available GPU simulators are designed to be sequential and single-threaded, and therefore, they suffer from performance degradation.

B. Performance Bottleneck Analysis on Single-Threaded GPU Simulation

In this study, we propose a new parallel simulation framework for GPUs in order to accelerate simulation speed. To do so, we first analyze performance of the existing GPU simulator, the characteristics of GPU computing applications, and the simulation workload ratio of GPU components.

As a representative existing GPU simulator, we investigated the performance of GPGPU-sim. GPGPU-sim was developed by Ali *et al.* [4], and it provides cycle-level timing simulation as well as functional simulation of the Tesla architecture GPUs. We measure the execution time of 8 CUDA applications using a real NVIDIA GeForce GTX 285 and a simulated GeForce GTX 285 on GPGPU-sim using the performance simulation mode. The test applications are selected from NVIDIA GPU computing SDK and executed on a host machine with an Intel Xeon X5550 quad-core processor.

Table I shows simulation runtimes for 8 different CUDA applications. GPGPU-sim requires extremely long execution time for all applications showing 170,000× to 2,000,000× simulation slowdown. Although GPGPU-sim provides cycle-level simulation functionality for GPUs, the computation overhead required for detailed cycle-level modeling causes a critical delay in the simulation.

Table II shows the dynamic instruction mix of simulated applications. On average, the benchmark applications consist of 77% INT/FP instructions, 9% branch instructions, and 14% load/store instructions. As an extreme case, *BlackScholes* consists of 93.7% INT/FP instructions. This is distinctly different compared to applications used in general-purpose processor benchmarks. Typical CPU or many-core processor applications contain only up to 50% INT/FP instructions [11], [12]. Because GPU computing applications are basically developed for high-performance computing, they lay a greater focus on computation than other applications do. Based on the instruction mix, we presume that the simulation workload is biased toward CU simulation.

The workload of each GPU component during simulation is also shown in Table II. For an NVIDIA GPU, 97% of the total simulation runtime is spent on simulating the CUs (SMs), and only 1 – 2% of the simulation runtime is required for DRAM and interconnection network (ICNT) simulations. This means that CU simulation is a major performance bottleneck in the architectural simulation of GPUs. Our parallel simulation technique is designed based on the above characteristics of GPU simulations.

III. Parallel Simulation Architecture for GPU

A. Simulation Framework

In this subsection, first, classification on the architectural components of GPUs is introduced to determine the paral-

lelization strategy for each component. Secondly, a parallel simulation framework is described in considering the characteristics of GPU architecture.

1) Parallel Components: In this paper, *parallel components* is used to refer to GPU hardware components that can be simulated in parallel. Components that satisfy the following conditions are defined as parallel components: 1) identical structure, 2) identical functionality, and 3) mutually independent executability. Generally, CUs are the only parallel component in GPUs because they satisfy all three requirements.

In the proposed parallel simulation framework, the simulation for parallel components is processed independently using multiple dedicated simulation threads. Each thread simulates the internal operations of a CU, including the instruction schedulers and on-chip memory components. The internal operations of a CU vary depending on the GPU architecture.

Parallel simulation for CUs can be realized employing architectural characteristics of the GPUs and independency of the GPU execution model. Internal structures of the CU, except for cache memories, are only used for processing work-items/work-groups that are assigned on the CU. In fact, on-chip memories (e.g., texture cache, shared memory) are not shared because they are read-only or dedicated units for each CU. Parallel simulation for CU is enabled by these features. In addition, the execution model of GPUs is designed to minimize communications between different work-groups in order to process multiple work-groups in parallel even though GPUs provide communication mechanisms for work-groups on other CUs [13]. When communications are required, they are handled explicitly in a high-level code. Therefore, CUs and their work allocation units (i.e., work-item and work-group) have a high level of parallelism and a low frequency of intercommunications except for explicit data sharing and communications. These characteristics of the GPU enable efficient simulation minimizing synchronization between threads.

2) Shared Components: In this paper, *shared components* is defined and used to refer to GPU hardware components that are mutually dependent on other components. Most GPU components, except for the CUs, are shared components.

In GPU simulations, the interconnection networks and memory subsystems have interdependency with the CUs to process memory requests. Therefore, some techniques require relaxing or removing this interdependency between shared components in order to enable parallel simulation for CUs. Existing parallel simulators such as SlackSim [8] and Graphite [9] employ event queues and the discrete event simulation methodology in order to decouple the core simulation from the simulation of other processor components. However, those simulators mainly target general purpose many-core processors.

Most conventional many-core processors have complete and standalone structured processor cores and therefore, do not have a centralized control unit for managing their cores. In contrast, work distribution and management for CUs in a GPU are performed by dedicated control units. For example, all DPPs of Evergreen and Northern Island family GPUs are fully controlled by the command processor and ultra-threaded dispatch processor. Although Tesla and Fermi architecture GPUs are more CPU-like and contain dedicated control and

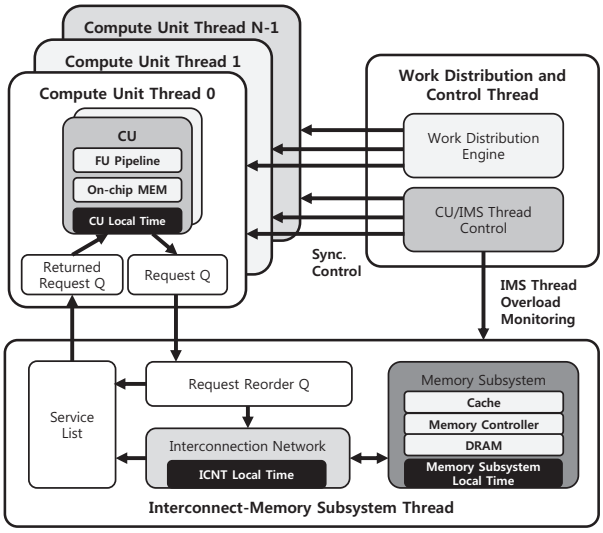


Fig. 2: Parallel simulation architecture for GPU.

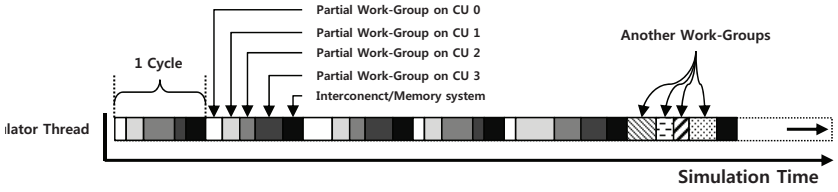
scheduling units on every SM, they also require the GigaThread Engine for managing SMs. In this paper, we call the centralized control unit as the *work distribution and control* (WDC). Because the activity of the WDC directly influences the operation and simulation of a CU, simulation for WDC needs to be carefully designed. As a matter of fact, the WDC should be considered as one of the shared components. It mimics the control units in the GPU and directly controls the CU threads. It also performs work distribution and reallocation for CUs, work monitoring, and task sequence control.

In addition to the WDC component, the proposed parallel simulation framework has another shared component that is *interconnect-memory subsystem* (IMS) component. The IMS component is also implemented using a dedicated thread. It simulates the interconnection between CUs and the GPU memory subsystem (cache, memory controllers, and DRAMs). The structure of the simulation architecture is described in detail in the following subsection.

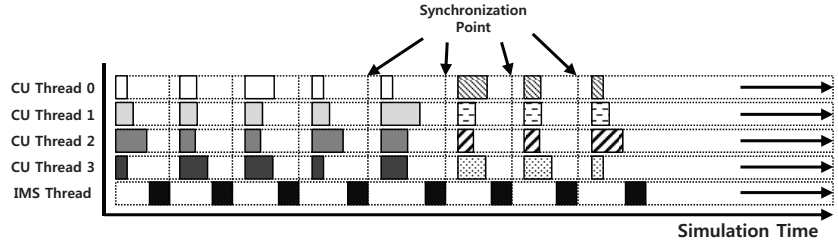
3) Parallel Simulation Architecture: The detailed architecture of the proposed parallel simulation framework is shown in Fig. 2. In this simulation architecture, three types of simulation threads are employed to simulate parallel/shared components.

As mentioned in Section III-A1, CU components are simulated using multiple dedicated threads. In this paper, we refer to these threads as CU threads. Each thread is responsible for simulating one or more CUs. Every CU thread has a local simulation time for each CU to compute its local cycle, a request queue, and a returned request queue. These are used for decoupling the CU simulation from the interconnection networks simulation and memory subsystem simulation.

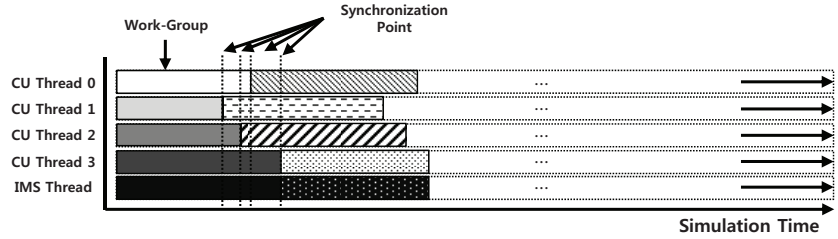
For example, if a memory access request is generated during execution of a load/store instruction on a CU, the CU thread spawns a new memory request entry and records the timestamp using the local simulation time. A memory event contained in the memory request entry is pushed into the request queue. After pushing the memory request, the CU thread continues the execution of other instructions irrespective of the interconnection networks and memory subsystem simu-



(a) Single-threaded simulation



(b) Cycle-by-cycle parallel simulation



(c) Work-group parallel simulation

Fig. 3: Parallel GPU simulation techniques.

lation. The memory events in the request queue are processed by the IMS thread and processed events are returned to the returned request queue. Finally, the CU thread checks the arrived memory events in the returned request queue, and it performs completion of the memory instructions.

Simulation for shared components is performed by two different threads: a WDC thread and an IMS thread. As described in Section III-A2, the WDC thread performs monitoring, control, and management for CU threads.

At the beginning of simulation, the WDC thread assigns a maximum possible number of work-groups to each CU. Thereafter, the WDC thread triggers the CU threads to start the CU simulation. When a CU completes the simulation of one of the assigned work-groups, a CU thread, which has responsibility for simulating the CU, changes the CU state to *issuable* (which means that another work-group can be issued to the CU), and the CU thread temporally stops simulation of the CU. The WDC thread continuously watches for CUs that are in the *issuable* state. When it detects such a CU, it collects the statistical data of CU simulation, assigns remaining work-groups to the CU, and restarts the simulation of the CU. This sequence is repeated until all tasks of the GPU computing application are completed.

The IMS thread simulates the interconnection networks, cache, memory controller, and DRAM activities. In order to simulate memory access events generated from CUs, it has a memory request reorder queue and a service list. It checks all request queues in the CUs in every simulation cycle. If there exist memory request events, it brings events to the memory request reorder queue. Collected memory request events are sorted according to their timestamps. Thereafter, the events are injected into the interconnection networks in the generation-time order.

After injecting memory request events to the interconnection networks, those injected events are managed by the service list until they return. The IMS thread simulates the latency of memory request procedure and obtains how many simulation cycles are elapsed for processing the request. Finally, the memory request event returns to the CU and the latency is applied to the simulation results.

B. Parallel GPU Simulation Scheme

Fig. 3 shows how the simulation is performed when a work-group is assigned to each CU. A typical cycle-accurate simulator sequentially simulates all the hardware components in every simulation cycle, as shown in Fig. 3(a). This is the most accurate simulation technique because all simulated components are synchronized on a global simulation cycle. However, a GPU has a large number of execution units, in which case this simulation technique causes a critical delay in the simulation, as analyzed in Section II-B. To address this performance issue, we propose a new simulation technique based on our parallel simulation framework.

TABLE III: Cycle gap and standard deviations of work-group completion cycle.

Applications	Standard Deviation of WG Completion cycle	Maximum Completion Cycle Gap
matrixmul	54	234
MersenneTwister	499	1352
scan	31	109
quasirandomGenerator	3089	12306
MonteCarlo	0	0
clock	89	278
scalarProd	1096	3482
BlackScholes	570	1954

1) Cycle-by-Cycle Parallel Simulation: Cycle-by-Cycle parallel simulation is a parallel simulation technique which concurrently simulates behaviors of GPU operations in every simulation cycle. In the case of CU simulation, each CU thread advances the state of a CU by one cycle and it simulates operations during the cycle. In the case of shared component simulation, the IMS thread sequentially processes 1 cycle worth of interconnection networks with data transfer, memory events on the memory controller, and DRAM activities. Because the cycle-by-cycle parallel simulation synchronizes all simulation threads in every cycle, it can precisely maintain a global cycle. Therefore, it has the same cycle-accuracy as the typical single-threaded simulation.

Fig. 3(b) shows how work-groups are processed by the cycle-by-cycle parallel simulation. The performance of this simulation could be degraded because all simulation threads need to be synchronized in every cycle. Although work-groups have the same instruction sequence, each simulation thread can execute different parts of a work-group at a given simulation point. Therefore, the time required for simulating 1 cycle differs across the threads. If a thread requires a long time for execution, then other threads need to wait until it has finished. This can lead to severe performance degradation.

The cycle-by-cycle parallel simulation is a well-known parallel simulation technique, and it has been adapted in the previous studies for many-core processor simulators [8], [9]. In this study, we use the simulation performance of the single-threaded simulation and cycle-by-cycle simulation as the performance baseline.

2) Work-Group Parallel Simulation: As described in Section III-A1, CU component simulation and the simulation for work-groups assigned on a CU can be processed by multiple threads in parallel. A parallel simulation technique for a GPU should consider these architectural characteristics and minimize the synchronization overhead in order to maximize the efficiency of the parallel simulation.

To overcome the performance degradation of the cycle-by-cycle simulation, we propose *work-group parallel simulation*. This parallel simulation scheme synchronizes simulation threads at the end of work-group execution, as shown in Fig. 3(c) . In contrast to the cycle-by-cycle parallel simulation, a synchronization point of the work-group parallel simulation scheme changes dynamically according to the size of the work-group and the execution speed of the CU threads.

In the work-group parallel simulation, synchronization at the end of work-group execution is performed in order to reduce work-group distribution difference and simulated cycle errors. Although each work group contains identical sets of instructions and its completion cycle is expected to be similar as the completion cycle of other work groups, the cycle gap among finished work-groups can be varying according to the applications. Table III shows that the completion cycle of the each work-group is scattered in indeterminate range. This cycle gap causes the simulation error on the parallel GPU simulation using relaxed synchronization scheme. Because the difference of running speed of CU threads makes variance on the work-group completion order, work-group distribution cannot be consistently performed in breadth-first manner.

Figure 4 shows how the work-group parallel simulation scheme employs two types of dynamic synchronization points to improve work-group distribution consistency: 1) *base synchronization point* is set as the local simulation cycle of CU which reaches the end point of a work-group, and 2) *top synchronization point* is set as the largest local simulation cycle when the base synchronization point is firstly set. At the base synchronization point, CU threads that exceed the cycle of the base synchronization point are stopped, and remaining CU threads continue the simulation until reaching the base synchronization point. At this moment, if a CU finishes a work-group but it still does not reach the base synchronization point, the WDC thread immediately stops all CU threads and checks the local simulation cycle of the CU. If the local simulation cycle is minimum, then the WDC thread assigns new work-groups on the CU and restarts all CU threads. If not, the base synchronization point is reset as the local simulation cycle of the CU which finishes a work-group execution intermediately, and the WDC thread restarts CU threads which do not reach the new base synchronization point. Finally a CU thread which has the smallest local simulation cycle reaches the base synchronization point, then the WDC thread resets the base synchronization point as the second smallest local simulation cycle and continues these procedures until reaching the top synchronization point. According to this synchronization method, work-group assignment is performed in work-group completion cycle order. Therefore, work-group distribution is kept in a breadth-first manner.

By using the work-group parallel simulation scheme, each CU thread is allowed to continue simulation until the assigned work-groups are completed. As a result, the synchronization overhead and thread waiting are significantly reduced. However, as a trade-off, relaxing synchronization among CUs might cause simulation cycle errors in the memory instructions. The latency of INT/FP instructions can be simulated precisely when the CU threads are executed independently, because they have fixed execution latency. However, the latency of memory instructions is determined by the round-trip delay of memory events, which traverse the interconnection networks and memory subsystems. Because the interconnection networks, memory controllers, and DRAMs of GPU are shared by multiple CUs, the latency of memory instructions of a CU can be affected by other CUs that try to process memory instructions at the same simulated cycle. The single-threaded simulation and the cycle-by-cycle parallel simulation are designed to model the contention between CU and the shared component in every cycle using the global simulation

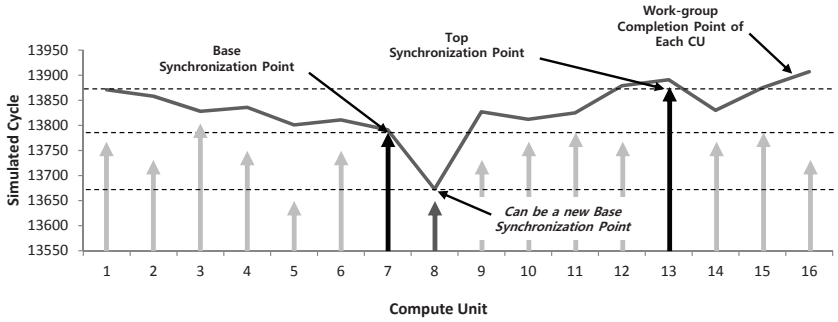


Fig. 4: Synchronization method of the work-group parallel simulation scheme.

cycle, therefore, it is possible to compute the precise latency of memory instructions. On the contrary, in the work-group parallel simulation which uses the relaxed synchronization scheme, the following problems can occur:

- Memory request order inversion should occur because each CU thread uses the local simulation cycle.
- Contention modeling of the interconnection networks and memory subsystem might not be same as the results of the single-threaded simulation.
- Imbalance of the simulation speed and workload on simulation threads can lead simulation violations because our parallel simulation framework uses multiple CU threads and an IMS thread. For example, if an application intensively generates many memory access requests in a specific kernel section, CU threads can lead to unnecessary execution stalls or task rescheduling because of execution delay on the IMS thread.

To improve accuracy of the interconnection network and memory subsystem simulation on the work-group parallel simulation scheme, two additional synchronization mechanisms are applied. Firstly, delayed-dequeuing mechanism is used on the memory request reorder queue to prevent memory request order inversion in parallel GPU simulation. The IMS thread monitors the local simulation time of each CU, and the thread delays dequeuing memory request events until local simulation time of all CUs is greater than the timestamp of the first event in the memory request reorder queue. This solution is time critical because additional latency error can occur if memory request events in the queue are processed too late. However, because the workload of IMS thread is significantly lighter than that of the CU threads, queued events can be processed without cycle errors.

Secondly, the IMS thread overload prevention technique is considered for minimizing the effect of simulation thread imbalance. This technique is implemented as limiting the size of the request reorder queue of the IMS thread, and also supports the operation of delayed-dequeuing on the memory request reorder queue. If memory request events reach the limit of the request reorder queue, all CU threads stop simulation temporarily and wait until already generated memory request events are completed. Once the number of memory request events is reduced under the limit, CU simulation is restarted. The maximum size of the request reorder queue varies depending on the number of load/store units in the target GPU architecture and performance of the host system. In this study, it is experimentally determined to be two times the maximum number of memory requests that can be generated from CUs in a cycle.

TABLE IV: CUDA applications for Evaluation

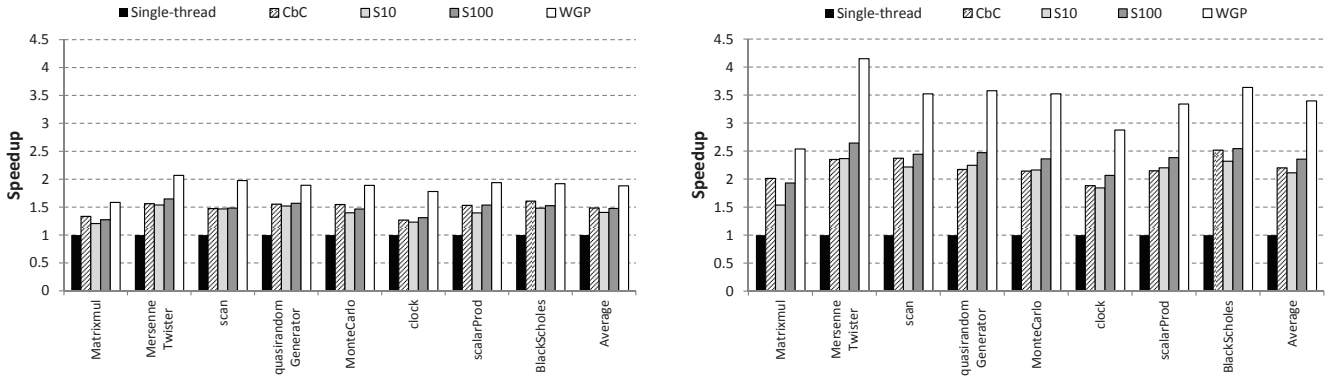
Application	Description	Configuration
matrixmul	Classic matrix multiplication	$(160\times96)\cdot(96\times128)$
Mersenne Twister	Pseudorandom number generator	24M random numbers
scan	Parallel prefix sum	512 elements
quasirandom Generator	Low-discrepancy sequences generator	1M random numbers
MonteCarlo	Option pricing (Monte Carlo)	256 options, 256K paths/option
clock	Clock function on a kernel	64 CTAs/kernel
scalarProd	Calculate scalar products	256 vectors, 4096 elements/vector
BlackScholes	Option pricing (blackscholes)	4M options

Because the work-group parallel simulation technique processes the CU simulation independently using multiple simulation threads, the processing sequence of memory events can differ from that in cycle-by-cycle simulation, and it might lead to simulation cycle errors. However, the errors can be minimized by applying the above mentioned techniques.

IV. Performance Evaluation

A. Experimental Environment

The experimental results provided in this section are obtained on a system with two-socket quad-core Intel Xeon X5550 processors running at 2.67 GHz and 16 GB of DRAM. The operating system used is Ubuntu Linux with kernel version 2.6.28. To obtain correct results, all processor power management features and Intel Hyper-Threading technology are disabled. To support parallel simulation of the GPU architecture, our modified version of GPGPU-sim (base version 2.1.1b) has been used. The simulators and CUDA benchmark applications are compiled using gcc version 4.3.3 and NVCC version 2.3 with the -O3 optimization option. The CUDA applications are selected from the NVIDIA GPU computing software development kit and their details are summarized in Table IV. The target GPU architecture parameters are summarized in Table V.



(a) 2 CU threads

(b) 4 CU threads

Fig. 5: Simulation speedup of parallel simulation techniques.

TABLE V: Target GPU architecture parameters

Parameter	Value
Number of CUs	16
Warp Size	32
SIMD Pipeline Width	8
Number of Threads/CU	1024
Number of CTAs/CU	8
Number of Registers/CU	16384
Shared Memory/CU	16KB
Constant Cache Size/CU	8KB
Texture Cache Size/CU	64KB
Number of Memory Channels	8
L1 Cache	None
L2 Cache	None
Bandwidth per Memory Module	8
Memory Controller Type	Out of Order (FR-FCFS)
Branch Divergence Method	Immediate Post Dominator
Warp Scheduling Policy	Round Robin among Ready Warps
Interconnection Network Topology	Crossbar

B. Performance Results

Fig. 5 shows the performance results of three parallel simulation techniques with the proposed parallel simulation framework. The results are presented for 8 CUDA benchmark applications, and they are normalized to the performance of the original GPGPU-sim using a single host core. The parallelized GPGPU-sim uses 2 or 4 CU threads for simulation of parallel components, a WDC thread, and an IMS thread.

The speedup of the cycle-by-cycle parallel simulation (CbC) continuously increases when it uses more CU threads, as shown in Fig. 5. However, the improvement is not proportional to the number of CU threads due to the synchronization overhead. Note that the maximum speed up is limited as 2.52 using 4 CU threads. Although the workload in a cycle is effectively parallelized by the CU threads, frequent synchronizations in every cycle diminish the effectiveness of parallelization.

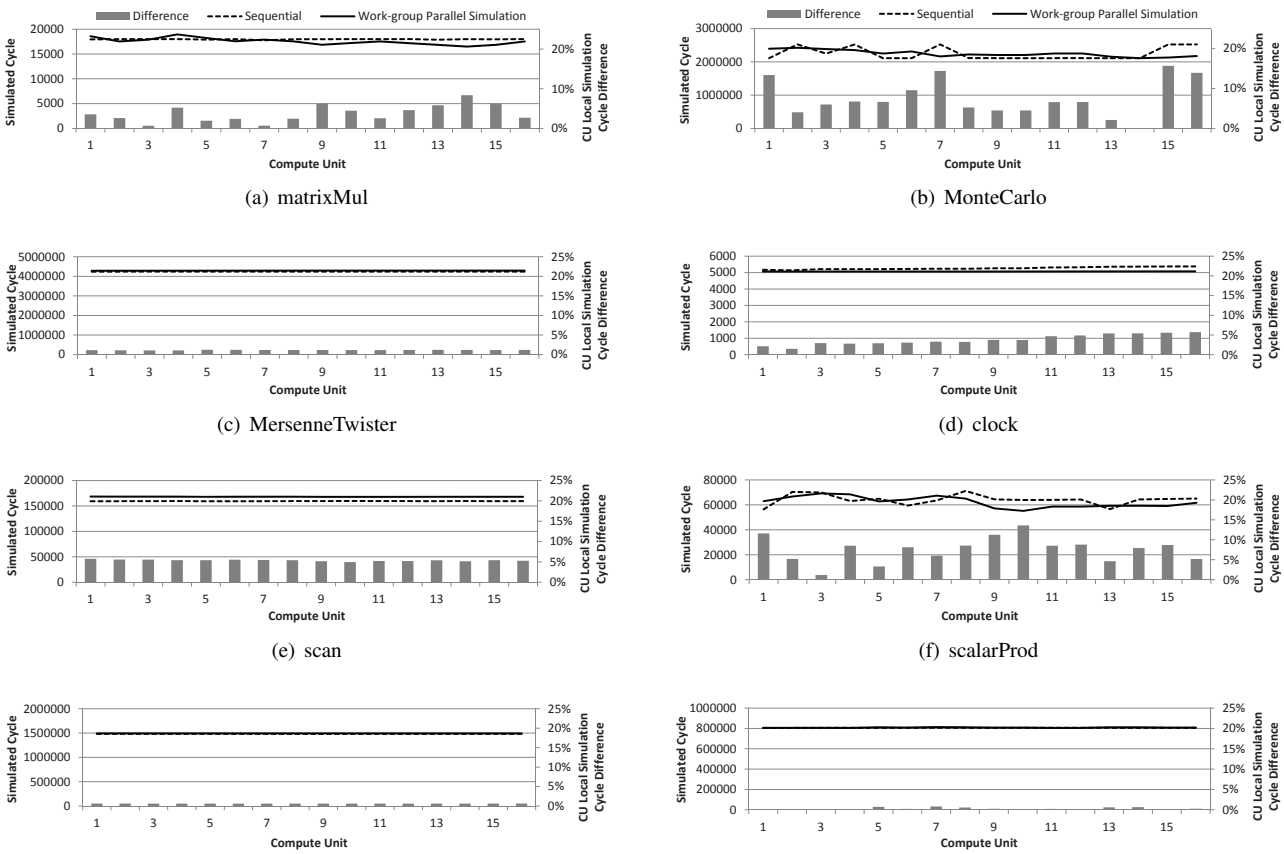
The speedup of the bounded slack simulation with 10-cycle slack (S10) and 100-cycle slack (S100) also shows similar tendencies as that of the cycle-by-cycle parallel simulation. The reason of the low performance on the bounded slack simulation is imbalance of the thread workload during simulation. In the original SlackSim, the simulation manager thread also performs the memory subsystem simulation [8]. The manager thread easily and effectively synchronizes other core threads and itself. However, for the GPU simulation, the memory subsystem simulation is performed by the dedicated IMS thread to implement isolated local time domain from the CU and the WDC unit for the interconnect/memory subsystem simulation. The WDC thread should concurrently control the CU threads and the IMS thread but these two types of threads have different workload and simulation speed. In case of the GPU simulation, the workload of the IMS thread is lighter than the CU threads, thus the IMS thread runs always faster than the CU threads and exceeds the slack bound frequently. As a result, the slack simulation incurs synchronization operations very frequently to synchronize the IMS thread and the CU threads. It weakens the efficiency of the slack simulation scheme.

In comparison, work-group parallel simulation (WGP) shows better speedups in all configurations, as shown in Fig. 5. Unlike the cycle-by-cycle parallel simulation and the bounded slack simulation, which shows low improvement on multi-threaded simulation, the work-group parallel simulation accelerates simulation speed more effectively when the simulator employs more CU threads. It shows almost linear speedups for up to 4 CU threads on several applications that have a high INT/FP instruction ratio. The best speedup of 4.15 is achieved in *MersenneTwister*.

In all simulation techniques, applications that have a high INT/FP instruction ratio show better performance improvement than those that have a high memory instruction ratio, such as *matrixMul* and *scalaProd*. *clock* is an exceptional case even though it has an INT/FP instruction ratio of 77.2%; this is because of overhead for enabling work-group parallel simulation. Multi-threaded simulation overhead for operating CU/WDC/IMS threads is negligible if simulation duration is sufficiently long. However, *clock* has the shortest runtime in the benchmarks, and each thread of the simulator runs only for 2,800 to 4,200 cycles. In this case, simulation overhead affects on performance of the simulator and the efficiency of work-group parallel simulation is decreased.

C. Total Simulated Cycle Error Analysis

Table VI shows the total simulated cycle error of work-group parallel simulation and bounded slack simulation with



(a) matrixMul (b) MonteCarlo

(c) MersenneTwister (d) clock

(e) scan (f) scalarProd

(g) quasirandomGenerator (h) BlackScholes

Fig. 6: Differences in CU local simulation cycles across applications.

TABLE VI: Total simulated cycle errors of work-group parallel simulation

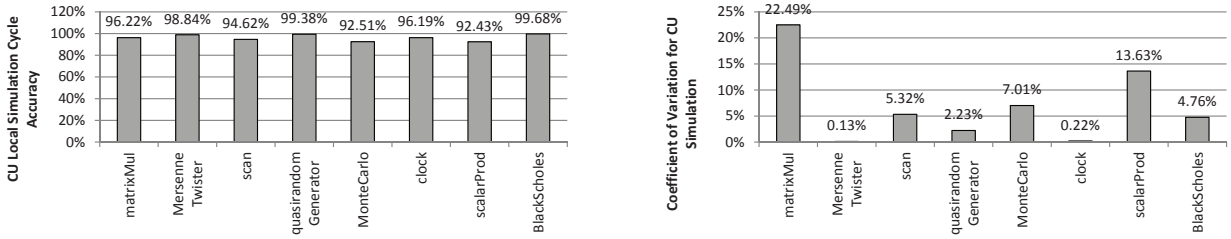
Application	Error (%)		CV (%)	
	S10	WGP	S10	WGP
matrixmul	18.29	26.61	0.34	8.10
MersenneTwister	0.36	1.28	0.04	0.05
scan	0.12	5.78	0.15	5.72
quasirandomGenerator	0.01	0.10	0.00	0.11
MonteCarlo	0.84	0.05	0.07	0.21
clock	1.23	5.46	0.45	0.51
scalarProd	0.41	1.07	0.50	3.69
BlackScholes	0.29	4.16	0.05	8.08

10-cycle slack as compared with the single-threaded simulation. The total simulated cycle error is represented by the number of GPU execution cycles required to complete the kernel code of benchmark application. The coefficient of variation (CV) represents the simulation consistency at each run. It is defined as the ratio of the standard deviation to the mean.

Work-group parallel simulation shows 0.05 – 5.78% of total simulated cycle errors and low CVs in *MersenneTwister*, *scan*, *quasirandomGenerator*, *MonteCarlo*, *clock*, *scalarProd*, and *BlackScholes*. Although the work-group parallel simulation scheme shows lower cycle accuracy than the bounded slack simulation, the results mean that work-group parallel simulation is an effective parallelization method for complex and computation-intensive applications. These applications contain a large number of memory instructions in their billions of instructions, but they show high cycle accuracy. However, *matrixMul*, *scan*, and *clock* result in lower cycle accuracy. As analyzed in Section II-B, simulations for these applications perform more memory event simulations than others. Latency estimation error of memory events could cause cycle errors. We investigate the reason of errors in the following subsection.

D. Effect of Local Simulation Cycle Difference

Fig. 6 shows the local simulation cycle difference of each CU between work-group parallel simulation (the solid lines) and single-threaded simulation (the dotted lines). Bar graphs show the relative error of the CU local simulation cycle between the two simulation methods. In the case of *MersenneTwister*, *scan*, *quasirandomGenerator*, and *BlackScholes*, all local simulation cycles of CU come close to the baseline with in less than 5% of error rate. *matrixMul* and *clock* show slightly more errors in the local simulation cycle of the CU; however, they also show similar cycle trends as the results of sequential simulations. In contrast, *MonteCarlo* and *scalarProd* show relatively large errors in several CUs, although some CUs have accurate local simulation cycles. These errors are induced by the difference in work-group distribution, as discussed in Section III-B2. However, the total simulated cycle errors of *matrixMul*, *MonteCarlo*, and *scalarProd* are quite different



(a) CU local simulation cycle accuracy

(b) Coefficient of variation for CU local simulation

Fig. 7: Accuracy of CU local simulation.

from their CU local simulation cycle errors. The reason is investigated in the following analysis.

The average local simulation cycle accuracy of the CU and CV of each CU simulation using the work-group parallel simulation technique is shown in Fig. 7. Data are collected from ten runs of each simulation. *MersenneTwister*, *quasirandomGenerator*, and *BlackScholes* show accurate CU local simulation cycles and low CV for each CU simulation. *matrixMul*, *MonteCarlo*, and *scalarProd* show low CU local simulation cycle accuracy and high CV, however, *MonteCarlo*, and *scalarProd* also show very low total simulated cycle error. The reason of these contrastive results is the work-group distribution difference in the work-group parallel simulation. *matrixMul* and *MonteCarlo* have very narrow cycle gap of each work-group completion point, as shown in Table III. Therefore, if the work-group parallel simulation scheme incurs small errors during a memory instruction latency simulation, then the work-group distribution order on CUs might be changed. If there remains insufficient amount of work-groups to distribute over all CU, remaining work-groups are assigned on arbitrary CUs. As a result, the local cycle of CUs is different at each run. Nevertheless, this difference does not directly cause the total simulated cycle errors because it simply changes the CU that has the maximum simulation cycle. *scalarProd* and *MonteCarlo* is that case.

matrixMul is combined case of the estimation error of memory latency and the work-group distribution difference. Especially, the work-group parallel simulation can unfairly delay specific memory requests due to the reordering operation. This delay results in long memory access latency and appears especially when the application contains a large number of memory instructions. In fact, 53% of dynamic instructions in executing *matrixMul* are load/store instructions as shown in Table II. These errors on modeling memory access delay would cause work-group allocation errors as well and continuously cause repeated errors on the simulation.

In summary, parallel GPU architecture simulation that requires work-group distribution inevitably involves a work-group distribution difference due to the asynchronous CU simulation. The difference causes fluctuations in the local simulation cycle of the CU and affects accuracy. The work-group parallel simulation technique implies these erroneous conditions; however, it still can provide the high cycle accuracy.

V. Related Work

Various types of simulators have been developed because these serve as important tools for developing computer system architectures and software. However, while many general-purpose processor simulators have been developed, few GPU architecture simulators have been developed because GPUs have evolved as special-purpose processors for 3D graphics.

Several studies have attempted to provide insights into architectural modeling on a GPU. Wong et al. [14] analyzed architectural characteristics of the NVIDIA GT200 GPU. Hong and Kim [15] proposed an analytical model that estimates the execution time of GPU applications. Although this research suggested approaches to analyze the behavior of GPGPU applications and GPUs, a GPU simulator is an essential tool for more detailed studies of the GPU architecture.

Barra [16] provides a functional simulation for the real ISA of the NVIDIA Tesla GPU architecture. It can also perform parallel functional simulation using a multi-core processor or SIMD instruction sets to improve the simulator performance. However, it cannot perform performance analysis for GPU architectures because it does not support cycle-level timing simulation.

GPGPU-sim [4] is a simulator that supports functional and cycle-level timing simulation for NVIDIA GPUs. However, it suffers from performance problems during cycle-level simulation because of its sequential structure, as described previously in this paper.

Multi2Sim [5] is a simulation framework for heterogeneous computing, including models for superscalar, multithreaded, multi-core, and graphics processors. It provides cycle-level simulation for the AMD Evergreen family GPU architecture. However, it also suffers from the same problem as GPGPU-sim. It does not support parallel GPU architecture simulation although it provides a parallel simulation framework for CPU architecture simulation.

Simulation architectures for general-purpose many-core processor can be used for GPU simulators; however, they also suffer from similar performance issues. To address this performance problem, parallel many-core simulation frameworks such as BigSim [17], P-GAS [18], SlackSim [8], Graphite [9], TaskSim [19], and Sniper [20] have been proposed. These simulators show good simulation performance for many-core processor architecture simulation using multi-core processors and multiple host machines. However, we primarily focus on developing an accurate, parallel simulation architecture

for a GPU and propose an optimized simulation technique considering the characteristics of a GPU. In fact, the slack simulation scheme introduced in SlackSim has also performance issues when the target system has a large number of cores and the simulation thread have different workload. Although Graphite provides a simulation model for the large-scale many-core processor architecture, it differs significantly from GPU architectures.

VI. Conclusions and Future Work

This study proposed a new parallel simulation technique for GPU architecture simulation. We focused on the simulator slowdown of previously proposed GPU simulators when performing cycle-level timing simulation. To improve the performance of GPU simulation, we proposed a new parallel simulation framework and an effective parallel simulation technique called the work-group parallel simulation. The framework has been developed through an intensive analysis of GPU architectures, and it provides a new simulation architecture by considering the characteristics of a GPU. Based on this new simulation framework, the work-group parallel simulation technique effectively accelerates simulation speed with relaxed synchronization between simulation threads.

We have performed detailed performance evaluations with various benchmarks, and observed that the parallelized version of GPGPU-sim, which is applied to our parallel simulation architecture, achieved an average speed-up of 3.39 as compared to the original GPGPU-sim GPU simulator using 6 simulation threads.

The work-group parallel simulation technique shows synergistic performance improvement and low relative cycle errors. To improve adaptability of the work-group parallel simulation technique for various types of applications such as the persistent thread execution model [21], our future work will focus on developing techniques to preserve integrity on the atomic operations during parallel simulation with low synchronization/restoration overhead. With this enhancement and our parallel simulation architecture, the GPU simulator should serve as a helpful simulation tool for studies of GPU architectures and GPGPU software.

Acknowledgment

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea, which is funded by the Ministry of Education, Science and Technology [2009-0070364].

References

[1] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, Mar/Apr 2011.

[2] AMD. (2011) HD 6900 Series Instruction Set Architecture. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/AMD_HD_6900_Series_Instruction_Set_Architecture.pdf

[3] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E, "ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures," in *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS 2006)*, Mar. 2006, pp. 231–241.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS 2009)*, Apr. 2009, pp. 163–174.

[5] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*, Sep. 2012.

[6] NVIDIA. (2012) NVIDIA CUDA C Programming Guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[7] Khronos. (2010) OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. [Online]. Available: http://www.khronos.org/opencl/

[8] J. Chen, M. Annavaram, and M. Dubois, "SlackSim: A Platform for Parallel Simulations of CMPs on CMPs," *SIGARCH Computer Architecture News*, vol. 37, pp. 20–29, Jul. 2009.

[9] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *Proc. IEEE 16th Int'l Symp. High Performance Computer Architecture (HPCA '10)*, Jan. 2010, pp. 1–12.

[10] NVIDIA. (2009) NVIDIAs Next Generation CUDA Compute Architecture: Fermi. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[11] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite," in *Proc. 34th Int'l Symp. Computer Architecture (ISCA '07)*, 2007, pp. 412–423.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '08)*, 2008, pp. 72–81.

[13] AMD. (2011) OpenCL and the AMD APP SDK. [Online]. Available: http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/

[14] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Proc. IEEE Int'l Symp. on Performance Analysis of Systems Software (ISPASS 2010)*, march 2010, pp. 235 –246.

[15] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness," in *Proc. 36th Int'l Symp. Computer Architecture (ISCA '09)*, 2009, pp. 152–163.

[16] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," in *Proc. IEEE Int'l Symp. Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, Aug. 2010, pp. 351–360.

[17] G. Zheng, G. Kakulapati, and L. Kale, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS '04)*, Apr. 2004, p. 78.

[18] H. Lv, Y. Cheng, L. Bai, M. Chen, D. Fan, and N. Sun, "P-GAS: Parallelizing a Cycle-Accurate Event-Driven Many-Core Processor Simulator Using Parallel Discrete Event Simulation," in *Proc. IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, May. 2010, pp. 1–8.

[19] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero, "On the simulation of large-scale architectures using multiple application abstraction levels," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 36:1–36:20, jan 2012.

[20] T. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int'l Symp. High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 1 –12.

[21] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *Proc. Innovative Parallel Computing*, May 2012, p. 14.