J. Shanghai Jiaotong Univ. (Sci.), 2016, 21(3): 280-288
DOI: 10.1007/s12204-016-1723-2

Exploiting Parallelism in the Simulation of General Purpose Graphics Processing Unit Program

*ZHAO Xia** (赵　夏), *MA Sheng* (马　胜), *CHEN Wei* (陈　微), *WANG Zhiying* (王志英)
(State Key Laboratory of High Performance Computing; College of Computer,
National University of Defense Technology, Changsha 410072, China)

© Shanghai Jiaotong University and Springer-Verlag Berlin Heidelberg 2016

Abstract: The simulation is an important means of performance evaluation of the computer architecture. Nowadays, the serial simulation of general purpose graphics processing unit (GPGPU) architecture is the main bottleneck for the simulation speed. To address this issue, we propose the intra-kernel parallelization on a multicore processor and the inter-kernel parallelization on a multiple-machine platform. We apply these two methods to the GPGPU-sim simulator. The intra-kernel parallelization method firstly parallelizes the serial simulation of multiple compute units in one cycle. Then it parallelizes the timing and functional simulation to reduce the performance loss caused by the synchronization between different compute units. The inter-kernel parallelization method divides multiple kernels of a CUDA program into several groups and distributes these groups across multiple simulation hosts to perform the simulation. Experimental results show that the intra-kernel parallelization method achieves a speed-up of up to 12 with a maximum error rate of 0.009 4% on a 32-core machine, and the inter-kernel parallelization method can accelerate the simulation by a factor of up to 3.9 with a maximum error rate of 0.11% on four simulation hosts. The orthogonality between these two methods allows us to combine them together on multiple multi-core hosts to get further performance improvements.
Key words: general purpose graphics processing unit (GPGPU), multicore, intra-kernel, inter-kernel, parallel
CLC number: TP 302.7　　**Document code:** A

0 Introduction

Recently, the heterogeneous architecture becomes the mainstream of supercomputing. The general purpose graphics processing unit (GPGPU) is the main computation unit in the heterogeneous system and it shows strong computation ability in the Tianhe-1A supercomputer. The GPGPU is different from many other computation platforms since it has significantly more compute units. For example, there are 13 streaming multiprocessors (SMXs) and 192 CUDA cores per SMX in the K20 graphic card, which can execute 26 624 CUDA threads simultaneously[1]. A simulator is a software performance model of an architecture. The architecture that is modeled in the simulator is called the target architecture; running the simulator on the host machine can get the performance results. Simulation has the advantages that development is relatively cheap compared to building hardware prototypes, and it is typically more accurate than analytical models.

A good cycle-level simulator is important for the designer to explore the GPGPU architecture. Finding the bottleneck of the architecture, designing some new techniques to mitigate the bottleneck and verifying the ideas through modifying the simulator are common ways for the architecture exploration. In addition, for the GPGPU programmer, a good simulator can also help to understand the program hotspot and optimize the program based on the knowledge of the architecture.

Unfortunately, compared to the hardware running on a host machine, simulation costs unacceptable time and the GPGPU-sim (the GPGPU simulation model and the execution-driven simulation model) is no exception. Sampling and parallelization are two popular techniques to accelerate the simulation speed. The advantages of sampling are that it needs less hardware resources than parallelization and relies on a mathematic model to maintain its accuracy. Yet, sampling brings

Received date: 2014-09-01
Foundation item: the National Natural Science Foundation of China (Nos. 61572508, 61272144, 61303065 and 61202121), the National High Technology Research and Development Program (863) of China (No.2012AA010905), the Research Project of National University of Defense Technology (No. JC13-06-02), the Doctoral Fund of Ministry of Education of China (No. 20134307120028), and the Research Fund for the Doctoral Program of Higher Education of China (No. 20114307120013)
***E-mail:** xiazhao@nudt.edu.cn

unstability into the simulation process and sometimes causes big mistakes between sampling results and real results. Parallelization is different from sampling; it parallelizes the simulator so that the simulator can exploit the multiple thread contexts in the host machine to speed up simulation. However, since parallelization executes the whole program, it always gets an accurate result.

To keep the results accurate, in our design, we use parallelization to speed up CUDA programs' simulation. After analyzing the structure of CUDA program, we find that serial simulations of a single kernel function and a large number of kernel functions in a CUDA program are two main factors to slow down the simulation time. So we divide our parallelization work into two parts: intra-kernel parallelization and inter-kernel parallelization, and exploit these two techniques to the GPGPU-sim to speed up CUDA programs' simulation.

The intra-kernel parallelization method first parallelizes the serial simulation of multiple compute units in one cycle and distributes these simulation threads across multiple cores on the host machine. Then it parallelizes the timing and functional simulation to reduce the performance loss caused by the synchronization between different simulation threads. Although the cycle level synchronization causes performance degradation, we keep it to get the same cycle-accuracy as the typical serial simulation. The inter-kernel parallelization method divides the multiple kernels of a CUDA program into several groups and distributes these groups across multiple machines to perform the simulation. Compared with the serial simulation of multiple kernels, this method makes kernels in different groups to be simulated as early as possible. The experimental results show that the intra-kernel parallelization method achieves a speed-up of up to 12 with the maximum error rate of 0.009 4% on a 32-core host machine, and the inter-kernel parallelization method can accelerate the simulation by a factor of up to 3.9 with a maximum error rate of 0.11% on four simulation hosts.

1 Related Work

The research on computer architecture simulator is very important because simulator serves as an important tool for developing computer system architectures and software. As the number of cores in target system increases quickly, the simulation speed for a large number of cores is too slow and practically not acceptable. Several parallel simulation techniques have been proposed to address the performance issue. Parallel simulation has been an active research topic for several decades[2-3]. Long before its application to computer architecture simulator, parallelization is always used to speed up discrete event simulation (DES), which is called parallel discrete event simulation (PDES)[2]. In recent years, while much parallelization work has been done on central processing unit (CPU) simulators and gets a good performance, GPGPU simulator still suffers from the performance issues.

Wisconsin wind tunnel (WWT) is one of the earliest parallel simulators[4]. It calculates target program execution time on the parallel host with a distributed, discrete-event simulation algorithm, but it requires application to use an explicitly interface for shared memory and only runs on CM-5 machines. These restrictions make it impractical for modern usage. WWT II[5], the successor to the original WWT, overcomes the platform restriction, but it only models the target memory system and requires applications to be modified to explicitly allocate shared memory blocks.

Slacksim simulates each core of a target chip multiprocessor (CMP) in one thread and then spreads the threads across the hardware thread contexts of a host CMP[6]. It uses slack simulation scheme to give each thread some slack which allows it to continue simulating without synchronizing with other threads in every cycle. It uses a central manager to monitor all threads with shared memory which restricts it to running on a single host machine.

Graphite provides a distributed parallel multicore simulator infrastructure[7]. It uses various techniques to achieve the high performance needed for evaluation (including direct execution, multi-machine distribution, analytical modeling and lax synchronization). Compared with Slacksim, graphite has a good scalability and can be distributed across multiple host machines, but it only provides a simulation model for the large-scale multi-core processor architecture which differs significantly from graphics processing unit (GPU) architecture.

After analyzing the difference between CPU architecture and GPGPU architecture, Lee and Ro[8] pointed out the bottleneck of current GPGPU simulation and used different threads to represent different functional parts to parallel the GPGPU simulation. Although using relaxed synchronization between different threads achieves a high simulation speed, it causes loss of simulation accuracy. It is similar to the intra-kernel parallelization posed in this paper, but the intra-kernel parallelization uses cycle-by-cycle synchronization to avoid accuracy loss of parallelization and puts forward a parallelism model between timing/functional simulators to make up the performance loss caused by the synchronization.

2 Motivation and Background of Parallelization

In this section, we first illustrate our motivation and introduce two simulation models used in GPGPU-sim. Then, we review the GPGPU-sim structure, a popular

cycle-level GPGPU simulator we use in this paper.

2.1 Motivation

There are some popular GPGPU simulators such as ATTILA[9], GPGPU-sim[10] and Multi2Sim[11], and all of them can perform the cycle-level simulation. However, cycle-accurate simulation is extremely slow. It is a key concern in architecture research and development. There exists a big gap between the simulation time of an application on a GPGPU simulator and the execution time of the same application on GPGPU. Unfortunately, a large number of cores in the GPGPU and the serial simulation of these cores cause the GPGPU simulators to take much more execution time[12]. Table 1 shows the comparison of the GPGPU execution time and the simulation time of GPGPU-sim for several applications. The slowdown of CUDA programs running on the Intel Sandybridge is between 600 000 to 3 000 000 compared with the native NVIDIA K20. Even for kernels with the running time of a few seconds on the K20 graphic card, the simulation time can cost several days.

Table 1 Execution time comparison of benchmarks on GPGPU and GPGPU-sim

Benchmark	Execution time/s		Slowdown
	GPU	GPUPU-sim	
Stencil_default128	0.016 117	12 077	749 333
MM_large2	0.028 553	69 400	2 430 823
BlackScholes	0.465 92	304 298	653 112
Cutcp	0.014 991	19 385	1 293 109
MM_large	0.003 972	8 833	2 223 816
MergeSort	0.020 62	31 928	1 548 399
Mri-q	0.007 209	22 500	3 121 098
Stencil_default	0.130 128	210 200	1 615 332

Therefore, speeding up the cycle-level simulation is very important for the exploration of GPGPU architecture.

2.2 Two Simulation Models

Figure 1 illustrates the GPGPU simulation model. The streaming multiprocessor (SM) represents the compute unit in the GPGPU architecture. A GPGPU application may have multiple kernels, and each kernel consists of multiple thread blocks (TBs). To simplify the hardware scheduling mechanism, CUDA program requires that thread blocks should be able to execute independently in any order. All the TBs can run in parallel if there are enough hardware resources on the GPGPU. But in reality, only some TBs are allowed to run on SMs concurrently due to the resource limitation. A block is composed of hundreds of threads. Threads within one block can communicate through shared memory and can be synchronized at a barrier. Once a block is assigned to an SM, it is divided into 32-thread units called warps. Each warp represents a stream of instructions, and a warp is a basic unit of thread scheduling in SMs. In most cases, a CUDA program is a sequence of kernels, and each kernel completes execution before the next kernel begins.

Execution-driven simulation is one of the most popular simulation models used widely in CPU and GPGPU simulator. Mauer et al.[13] presented four different ways to couple the functional and timing simulator to manage simulator complexity. Timing-directed simulation is one way. In timing-directed simulation, the functional model keeps track of the architecture state such as memory values, and the timing model has no notion of values. The functional model can be viewed as a set of functions, and the timing model invokes them to get the effective addresses, the warp's active mask, and so on. The serial relation between timing simulator and functional simulator can restrict the simulation speed.

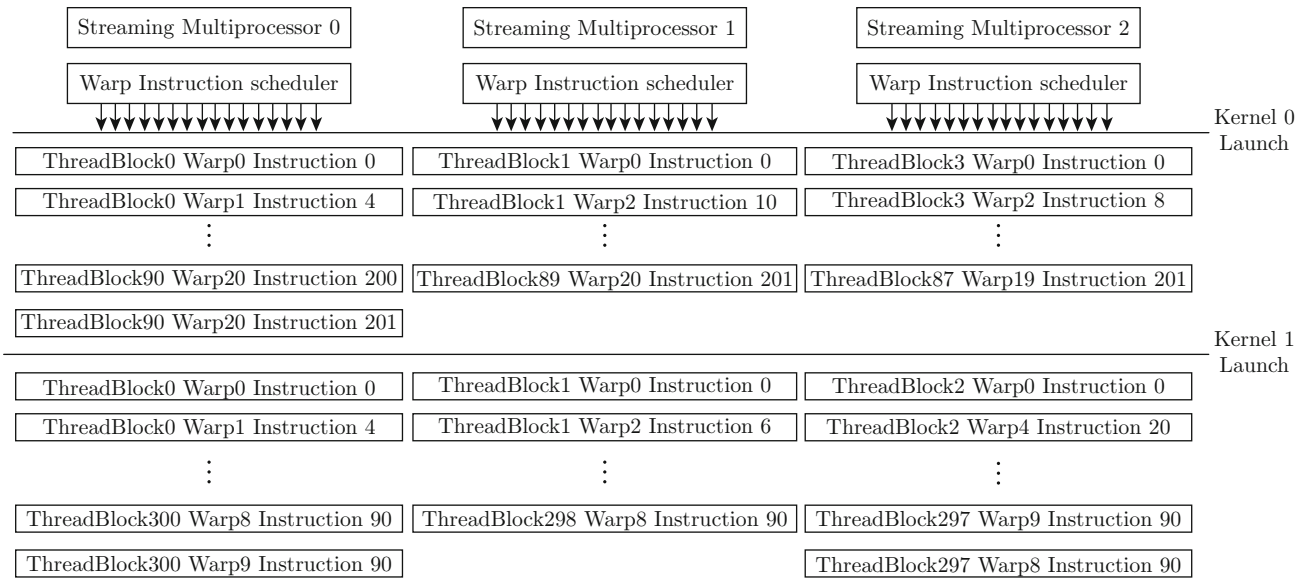


Fig. 1 GPGPU simulation overview

2.3 GPGPU-sim

GPGPU-sim developed by Bakhoda et al.[10] provides cycle-level timing simulation as well as functional simulation of GPGPU. As shown in Fig. 2, GPGPU-sim consists of four parts: the simultaneous multithreading (SIMT) core cluster, the L2 cache, the dynamic random-access memory (DRAM) and the interconnection network (ICNT). The SIMT core cluster consists of several SIMT cores which represent the compute units in GPGPU, and it uses the timing-directed simulation to simulate the execution of warp instructions. The L2 cache and the DRAM represent the storage units, and the ICNT models the communication backbone between compute units and storage units.

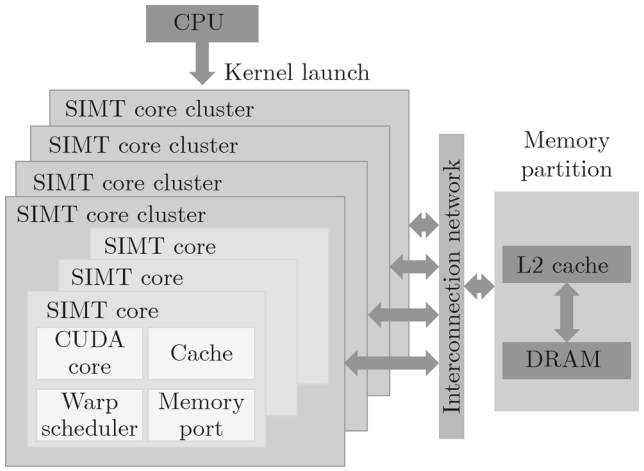


Fig. 2 The structure of GPGPU-sim

GPGPU-sim supports four independent clock domains: the SIMT core cluster clock domain, the ICNT clock domain, the L2 cache clock domain, and the DRAM clock domain. The four clock domains are used to drive the simulation of four different parts. Units in adjacent clock domains communicate with each other by filling and draining shared buffers. For example, when SIMT core cluster cycle comes, SIMT core cluster advances each SIMT core's state and writes memory accesses into the shared buffer between SIMT core cluster and ICNT. When ICNT cycle comes, ICNT reads these memory accesses and regards them as the injections of the interconnection network.

3 Intra-Kernel Parallelization

In the simulation of a kernel, the serial simulation of multiple compute units is a factor to slow down the simulation time. In this section, the serial simulation of clusters in one cycle and the serial relation between the timing/functional simulators are parallelized to reduce the simulation time of a kernel.

3.1 Cycle-by-Cycle Parallelization

Figure 3(a) shows the original simulation of multiple compute units. The simulator sequentially simulates all the clusters in every SIMT core cluster cycle; this serial process restricts the simulation speed and cannot fully utilize the power of multi-core processor. Figure 3(b) shows that multiple threads are used to simulate different clusters. Each cluster thread advances the state of a cluster and simulates operations during the cycle. Because the cycle-by-cycle parallelization synchronizes all simulation threads in every cycle, it has almost the same cycle-accuracy as the typical single-threaded simulation.

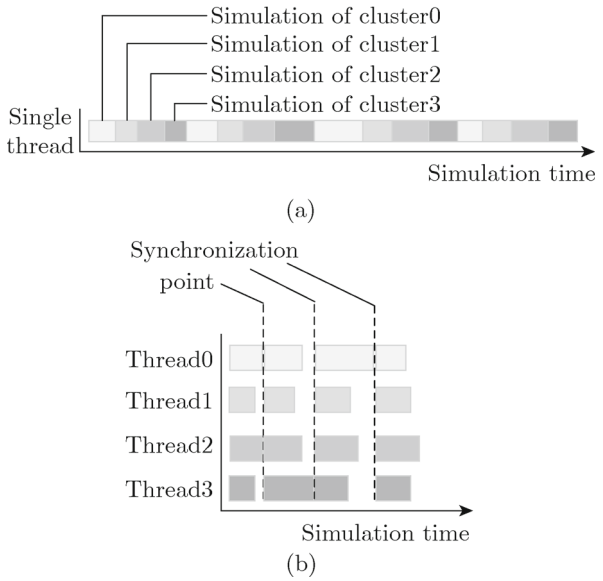


Fig. 3 Cycle-by-cycle parallelization

The cycle-by-cycle parallelization is a well-known parallel simulation technique for its accuracy, but the synchronization in every cycle restricts its speed and its usage. Using relaxed synchronization scheme to break the cycle-by-cycle synchronization brings speedup. However, it causes unacceptable errors. In the next section, a parallelism model between timing simulator and functional simulator is proposed to overcome the performance degradation of the cycle-by-cycle parallelization. The parallelism model does not affect the simulation accuracy while further accelerates the cycle-by-cycle parallelization.

3.2 Parallelism Model Between Timing and Functional Simulators

As we mention earlier, the timing-directed simulation method is used in the simulation of SIMT core cluster. Figure 4(a) shows that the timing simulator calls the functional model in the simulation of each warp instruction to get values such as program counter (PC) and memory address. Although every function call costs little time, too many function calls can degrade the

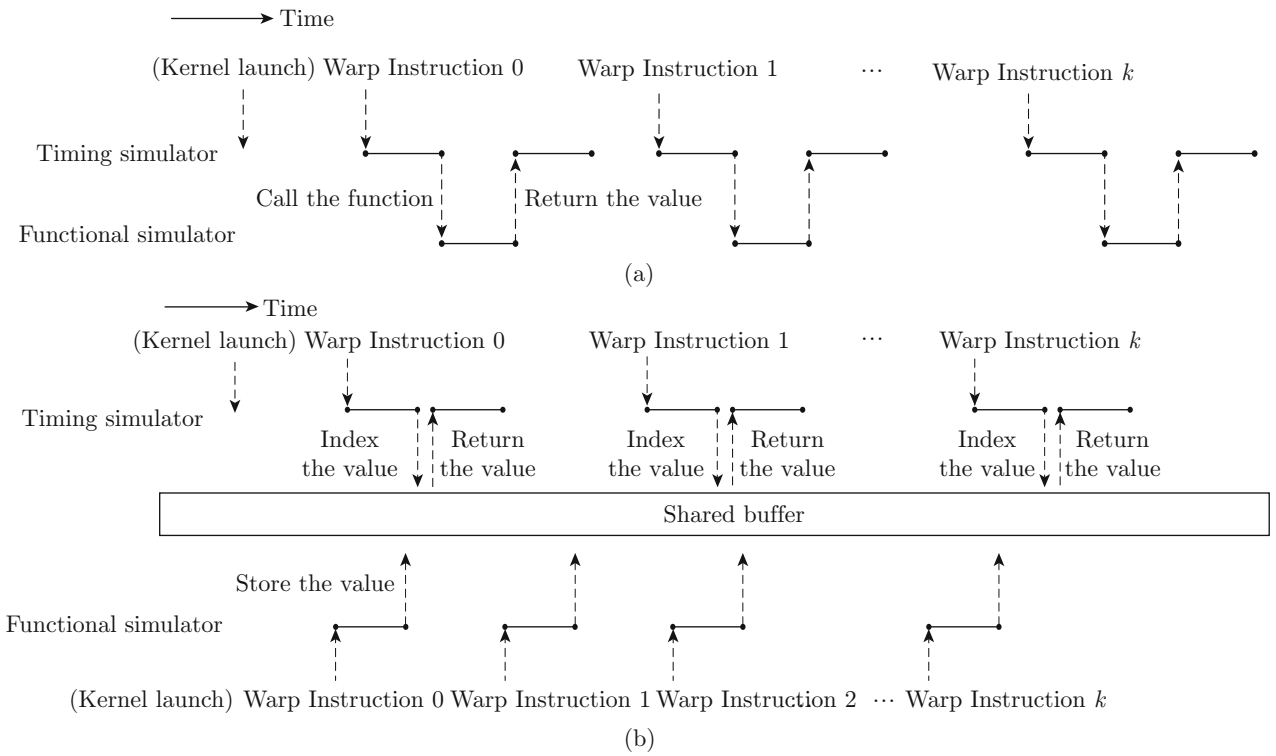


Fig. 4 Simulation of warp instructions

simulation performance, and there are always millions of warp instructions in CUDA programs.

Figure 4(b) shows that a shared buffer is added to break the call/return relation between the timing simulator and the functional simulator. After the buffer is added, the timing simulator and the functional simulator can simulate the kernel concurrently. When the kernel launches, the timing simulator and the functional simulator begin the simulation at the same time. After the functional simulation of a warp instruction is completed, the functional simulator stores the values into the buffer before simulating the next warp instruction. The timing simulator indexes the buffer and then gets the values directly when it needs the values. This parallelism model reduces simulation time of each warp instruction without affecting the simulation accuracy.

After the cycle-by-cycle parallelism, the accelerated timing simulator runs faster than the original functional simulator. If the timing simulator cannot find the values which it needs in the buffer, it stops and waits for the functional simulator to store the values.

In order to avoid the potential bottleneck caused by the functional simulator, we put up a new paralleled functional simulator to improve the performance of the parallelism model in advance. The main idea of the functional parallelism is that there are no relations between different TBs in kernel. If there are enough hardware resources on the GPGPU, all the blocks can be executed at the same time. In the paralleled functional simulator, the simulator divides different blocks in a kernel into several groups and begins every group's functional simulation at the same time.

4 Inter-Kernel Parallelization

The sequential simulation of kernel functions in CUDA program is another factor to slow down the simulation time. This section accelerates the simulation by splitting the simulation progress into a set of N chunks and distributing them over the same number of machines. Implementing simulation splitting and distribution is rather direct, except for the accuracy error.

Because of the CUDA characteristic and the way kernels use to launch on GPGPU, when the GPGPU simulator simulates the current kernel, it uses little timing information produced by the simulation of previous kernels. But the functional information must be transformed from the previous kernels to the current kernel to assure the accurate execution of CUDA programs.

Figure 5 shows the principle of distributed simulation. Assume that one wants to run a simulation of N kernels with 2 machines. Each machine simulates $N/2$ kernels. The first $N/2$ kernels are simulated on the first machine. The second $N/2$ kernels are simulated on the second machine. Each machine runs the CUDA program from the beginning, but starts simulating at the first kernel of the specified group. Thus, each machine must emulate the kernels (functional simulation

only) until it has reached the first kernel of the specified group. So the first machine starts simulating immediately, and the second machine emulates the first $N/2$ kernels before it starts simulating.

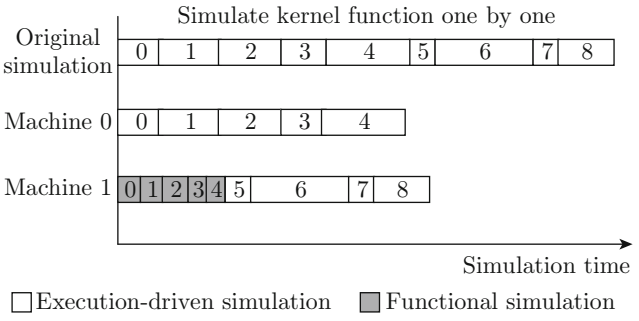


Fig. 5 Simulation of kernel functions

5 Evaluation and Analysis

5.1 Experimental Environment

The intra-kernel parallelization experimental results are obtained on power leader server PR48040R with 4 Intel(R) Xeon(R) E5-4620 CPU processors running at 2.67 GHz and 512 GB of DRAM. The inter-kernel parallelization experimental results are obtained on a system with four machines, and each machine has an i3-2130 CPU running at 3.2 GHz and 4 GB of DRAM. The operating system is Red Hat Server6.0 with kernel 2.6.4. The version of GPGPU-sim is 3.4. The simulators and CUDA benchmark applications are compiled by gcc 4.3.3 and NVCC 2.3. Table 2 shows the CUDA applications which are selected from the NVIDIA GPU computing software (Development Kit, Parboil, Rodinia and other sources). The target GPU architecture parameters are summarized in Table 3.

Table 2 Benchmarks for evaluation

Benchmark	Source	Input	Kernel count
Cutcp	Ref. [14]	Small data set	11
MM	Ref. [14]	Large: (1 024, 992) × (992, 1 056) Large 2: (2 048, 1 984) × (1 984, 2 112)	1
Mri-q	Ref. [14]	Large data set	3
Stencil	Ref. [14]	Default128: <128, 128, 64, 32, 2, 100> Default: <512, 512, 64, 32, 8, 100>	100
Blacksholes	Ref. [15]	4 million options	512
Mergesort	Ref. [15]	$N = 4 \times 1\,048\,576$	49
Libor	Ref. [16]	NPATH = 96 000	3

Table 3 GPGPU-sim configuration

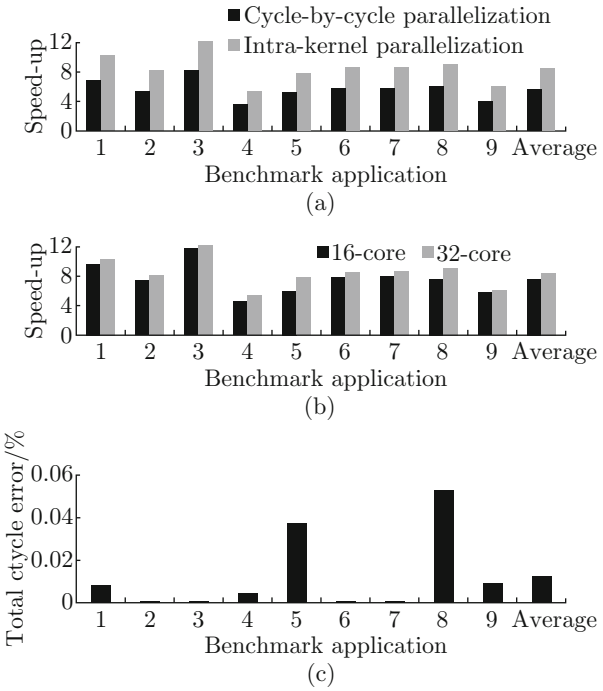
Parameter	Definition
Number of clusters	10
Number of core per cluster	3
Warp size	32
SIMD pipeline width	8
Number of threads per core	1 024
Number of CTAS per core	8
Number of registers per core	16 384
Shared memory per core/KB	16
Constant cache size per core/KB	8
Number of memory channels	8
L1 data cache	None
L2 data cache/KB	256
Memory controller type	Out of order (FR-FCFS)
Branch divergence method	Immediate post dominator
Warp scheduling policy	Round robin among ready warps

5.2 Performance of Intra-kernel Parallelization

Figure 6 shows the speed-up and the total simulated cycle error of the intra-kernel parallelization, as compared with the single-threaded simulation. The results are presented for 9 CUDA benchmark applications, and they are normalized to the performance with a single host core. Figure 6(a) shows that the average speed-up of the cycle-by-cycle parallelization is 6 after using 10 threads to represent each cluster's simulation in GPGPU-sim. The speed-up of this parallelization is smaller than 10 because the 10 simulation threads need to be synchronized in every cycle. Although every warp has the same instruction stream, each simulation thread can simulate different instruction at a given simulation cycle. Therefore, the time required for simulating 1 cycle differs across the threads. If a thread requires a long time for execution, the other threads must wait until it has finished.

Based on combining the cycle-by-cycle parallelization and the parallelism model between timing/functional simulators together, the average speed-up of intra-kernel parallelization is 8.1, ranging from 5 (Stencil_default128) to 12 (Mri-q). The intra-kernel parallelization accelerates the simulation of compute part in GPGPU-sim; with the increase of the kernel's simulation time, the percentage of the compute part

increases too, so the intra-kernel parallelization performs well to accelerate the kernel which has a long simulation time. But the long simulation time of Stencil_default128 mainly derives from the large number of kernels and the single kernel costs less time compared with other benchmarks, so the intra-kernel parallelization doesn't perform well in Stencil_default128. However, as shown later, the inter-kernel parallelization addresses this problem and accelerates the simulation speed of Stencil_default128 significantly. To evaluate the scalability of intra-kernel parallelization mechanism, we increase the number of cores on the host machine from 16 to 32. As shown in Fig. 6(b), with the increase of the cores on the host, the performance of intra-kernel parallelization does not be improved dramatically. There are two reasons: ① the cycle-by-cycle parallelization only uses 10 threads to represent each cluster's simulation, so it has achieved its maximum performance on a 16-core host; ② although exploiting more cores improves the performance of the parallelism model between timing/functional simulators, it does not have obvious effect on the whole performance of the intra-kernel parallelization mechanism.



1—Cutcp, 2—MM-larger, 3—Mri-q, 4—Stencil-default128, 5—Stencil-default, 6—Libor, 7—MM-larger2, 8—Mergesort, 9—Blackscholes

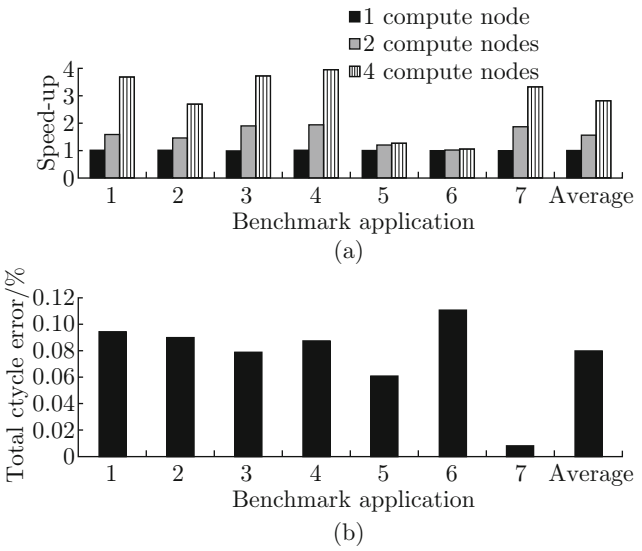
Fig. 6 Performance of intra-kernel parallelization

The cycle-by-cycle parallelization accelerates the simulation of instructions in a cycle, and the parallelism model accelerates the simulation of each instruction. These two ways do not affect the number of simulated instructions. Figure 6(c) shows the total simulated cycle error of intra-kernel parallelization, as compared with the single-threaded simulation. Compared with the serial simulation, the parallelization causes violation to the memory access order which induces the cycle errors. In the serial simulation, when the SIMT core cluster cycle comes, the GPGPU-sim sequentially simulates SIMT core cluster to advance each SIMT core's state. In this process, if an SIMT core wants to access the memory, it will inject the memory access into the shared buffer between the SIMT core cluster model and the ICNT model. The order of these memory accesses is fixed. After exploiting the cycle-by-cycle parallelization, the injection order depends on the speed of the simulation threads. If a thread runs slowly than other threads, it might inject the memory access lastly. Because of the violation of the memory accesses order, the memory access latency and the cycles used to simulate an instruction might be different with the serial simulation. The intra-kernel parallelization shows 0.00%—0.051% of total simulated cycle errors and the high simulation accuracy benefits from the cycle-by-cycle synchronization.

5.3 Performance of Inter-Kernel Parallelization

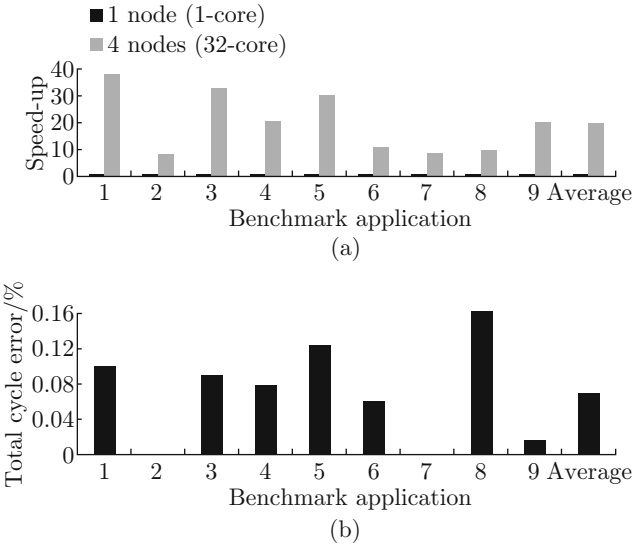
Figure 7 shows the speed-up and the total simulated cycle error of the inter-kernel parallelization, as compared with the single host simulation. The benchmark MM has only one kernel, so it is not listed below. Figure 7(a) shows that the performance of inter-kernel parallelization scales up with the increase of the machine count. On condition of using four machines, the average speed-up is 3.2, ranging from 1.2 (Libor) to 3.9 (Stencil_default128). Benchmark Stencil_default128 has many kernels, and the simulation time for each is almost the same. These two characteristics make it benefit a lot from the inter-kernel parallelization, so it can get almost linear speed-up with the increase of host machines. In contrast, Libor has only three kernels and the third kernel costs most of the simulation time. The long simulation time of the third kernel makes intra-kernel parallelization perform well in Libor, but it limits the performance of inter-kernel parallelization. The poor performance of Mergesort has the same reason, because its first kernel costs most of the simulation time.

The inter-kernel parallelization splits the kernels into a set of N groups and distributes them over the same number of machines. The parallelization simulates all the instructions in the kernels, so the sum of the total simulated instructions is the same as the serial simulation. Figure 7(b) shows the total simulated cycle error of inter-kernel parallelization, as compared with the single host simulation. The simulated cycle error varies from about 0.06% to 0.11%. This value is a little bigger than intra-kernel parallelization. The error derives from the cold-start problem[17]. For example, the inter-kernel parallelization uses 2 machines to simulate



1—Cutcp, 2—Mri-q, 3—Stencil-default128, 4—Stencil-default, 5—Libor, 6—Mergesort, 7—Blackscholes

Fig. 7 Performance of inter-kernel parallelization



1—Cutcp, 2—MM-larger, 3—Mri-q, 4—Stencil-default128, 5—Stencil-default, 6—Libor, 7—MM-larger2, 8—Mergesort, 9—Blackscholes

Fig. 8 Performance of kernel parallelization

N kernels. The first $N/2$ kernels are simulated on the first machine, and the second $N/2$ kernels are simulated on the second machine. In the serial simulation, when the GPGPU-sim begins to simulate the second $N/2$ kernels, the microarchitecture state in the simulator is warmed up by the simulation of the first $N/2$ kernels. In contrast, in the inter-kernel parallelization, although the second machine emulates the first $N/2$ kernels, the microarchitecture state is still cold when the machine starts simulating the second $N/2$ kernel. Although there are many techniques posed before to solve the cold-start problem[18-19], the inter-kernel parallelization doesn't need to adapt any solutions. It is because the microarchitecture warmed before is only some L2 cache states and these states have little impact on the memory access latency in the simulation of the next kernel. This can by proved by the fact that the maximum error is only 0.11%.

The intra-kernel parallelization accelerates the simulation of a CUDA program by reducing the simulation time of each kernel, and the inter-kernel parallelization accelerates the simulation by splitting the simulation process into a set of N chunks and distributing them over the same number of machines. These two parallelization techniques are orthogonal, so further performance improvements can be achieved by combining these two techniques together. Due to the limitation of the experimental environment, Fig. 8 only shows the theoretical speed-up and the simulated cycle errors by combining these two techniques. It is assumed that the simulation process is distributed over several machines and the simulation of each kernel on the machine is accelerated, so the speed-up is the product of these two techniques and the error is the sum of these two.

6 Conclusion

After analyzing the structure of CUDA program, we find that serial simulations of a single kernel and a large number of kernels are two main factors to slow down the simulation time. The inter-kernel parallelization and the intra-kernel parallelization are proposed to overcome the bottleneck of serial simulation through using the computing power of multiple multi-core hosts.

We perform detailed performance evaluations with various benchmarks, and observe that the intra-kernel parallelization achieves a speed-up of up to 12 with the maximum error rate of 0.009 4% on a 32-core host machine, and the inter-kernel parallelization can accelerate the simulation by a factor of up to 3.9 with a maximum error rate of 0.11% on four simulation hosts. In fact, the intra-kernel parallelization and the inter-kernel parallelization are orthogonal, so these two ways can be combined together to get a further performance improvement.

Some benchmarks (such as MM) only have a single kernel function, so the performance cannot benefit from the inter-kernel parallelization. The experimental results also show that MM performs badly when it is combined with the inter-kernel and intra-kernel parallelization techniques together. In order to take advantage of multiple machines, our future work will focus on how to split the simulation process of blocks within a kernel into several chunks and distribute them over the

same number of machines. The main challenge is the cold-start problem and it seriously impacts the simulation accuracy in this situation. The problem must be dealt with properly in the future work. With this enhancement and our parallel simulation techniques, the GPGPU-sim should serve as a helpful cycle-level simulation tool for the studies of GPGPU architectures.

References

[1] NVIDIA. Tesla® Kepler™ GPU accelerator [EB/OL]. (2014-09-01). http://www.nvidia.com/content/tesla/pdf/ Tesla-KSeries-Overview-LR.pdf.

[2] AYANI R. Parallel simulation [C]//*Performance Evaluation of Computer and Communication Systems*. Berlin Heidelberg: Springer, 1993: 1-20.

[3] NICOL D, FUJIMOTO R. Parallel simulation today [J]. *Annals of Operations Research*, 1994, **53**(1): 249-285.

[4] REINHARDT S K, HILL M D, LARUS J R, et al. The Wisconsin wind tunnel: Virtual prototyping of parallel computers [C]// *Proceedings of the 1993 ACM SIGMETRICS Conference*. New York: ACM, 1993: 1-3.

[5] MUKHERJEE S S, REINHARDT S K, FALSAFI B, et al. Wisconsin wind tunnel II: A fast, portable parallel architecture simulator [J]. *IEEE Concurrency*, 2000, **8**(4): 12-20.

[6] CHEN J W, ANNAVARAM M, DUBOIS M. SlackSim: A platform for parallel simulations of CMPs on CMPs [J]. *ACM SIGARCH Computer Architecture News*, 2009, **37**(2): 20-29.

[7] MILLER J E, KASTURE H, KURIAN G, et al. Graphite: A distributed parallel simulator for multicores [C]//*Proceedings of 16th International Symposium on High Performance Computer Architecture*. Washington: IEEE, 2010: 1-12.

[8] LEE S, RO W W. Parallel GPU architecture simulation framework exploiting work allocation unit parallelism [C]//*2013 IEEE International Symposium on Performance Analysis of Systems and Software*. Washington: IEEE, 2013: 107-117.

[9] DEL BARRIO V M, GONZÁLEZ C, ROCA J, et al. ATTILA: A cycle-level execution-driven simulator for modern GPU architectures [C]//*2006 IEEE International Symposium on Performance Analysis of Systems and Software*. Washington: IEEE, 2006: 231-241.

[10] BAKHODA A, YUAN G L, FUNG W W L, et al. Analyzing CUDA workloads using a detailed GPU simulator [C]// *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. Washington: IEEE, 2009: 163-174.

[11] UBAL R, JANG B, MISTRY P, et al. Multi2Sim: A simulation framework for CPU-GPU computing [C]//*Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. New York: ACM, 2012: 335-344.

[12] YU Z B, EECKHOUT L, GOSWAMI N, et al. Accelerating GPGPU architecture simulation [C]// *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*. New York: ACM, 2013: 331-332.

[13] MAUER C J, HILL M D, WOOD D A. Full-system timing-first simulation [C]// *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*. New York: ACM, 2002: 108-116.

[14] Illinois Microarchitecture Project utilizing Advanced Compiler Technology Research Group. Parboil benchmark suite [EB/OL]. (2014-09-01). http://impact.crhc.illinois.edu/Parboil/parboil.aspx.

[15] NVIDIA Corporation. NVIDIA CUDA SDK code samples [EB/OL]. (2014-09-01). http://docs.nvidia.com/cuda/cuda-samples.

[16] MIKE GILES. Libor [EB/OL]. (2014-09-01). http://people.maths.ox.ac.uk/gilesm/cuda.html.

[17] EECKHOUT L. Computer architecture performance evaluation methods [J]. *Synthesis Lectures on Computer Architecture*, 2010, **5**(1): 1-145.

[18] LUO Y, JOHN L K, EECKHOUT L. Self-monitored adaptive cache warm-up for microprocessor simulation [C]// *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*. [s.l.]: IEEE, 2004: 10-17.

[19] HASKINS JR J W, SKADRON K. Accelerated warmup for sampled microarchitecture simulation [J]. *ACM Transactions on Architecture and Code Optimization*, 2005, **2**(1): 78-108.