Parallelizing GPGPU-Sim for Faster Simulation with High Fidelity

Jun Wang[1] and Jiaquan Gao[2]

[1]Futurewei Technologies, Santa Clara, CA 95050, USA
[2]School of Computer Science and Technology, Nanjing Normal University, Nanjing, Jiangsu 210023, China

***Abstract*—This paper presents our work in parallelizing Gpgpu-sim for faster simulation without sacrificing fidelity. Based on profiling data, we first parallelize the simulation of the GPU cores, which is the most time-consuming part of the simulator. Secondly, we apply parallelization across clock domains to allow other major components of the model to be simulated in parallel with the cores. Our goal is two-fold: (1) To create a parallelized, faster simulator with minimal code changes, and (2) To achieve high fidelity to the sequential simulator. Experiments on a 4-core computer with 8 hardware threads show that speedup up to 2.65× is achieved and simulation statistics deviate from the sequential version by less than 1.4%.**
Keywords: GPGPU, GPU simulation, parallel simulation

I. Introduction

Ever since the first release of CUDA by NVidia in 2007, General-Purpose Graphics Processing Unit (GPGPU) has become a prominent type of platform for parallel computing. As is the case with any other computer architecture devices, researchers rely heavily on simulations to study GPGPUs. Efficient simulation tools are of great importance.

Gpgpu-sim [1] is one of the best-known cycle-level simulators for GPGPUs. Because of its high accuracy and its open-source license, Gpgpu-sim is widely used by computer architects for the advancement of GPGPU research. However, despite the fact that (1) GPGPUs are inherently a parallel architecture, and (2) Even the low-end desktop computers currently available to researchers for running Gpgpu-sim are equipped with multicore CPUs with four or more cores, Gpgpu-sim is essentially a sequential simulator. For example, it simulates the operations of the shader cores one by one is a sequential manner. This can be highly inefficient given that today's GPGPUs can include hundreds, even thousands of shader cores [2]. Simulating a large number of cores one by one for millions of cycles can be very time-consuming, leading to low productivity.

In this paper, we describe our efforts in parallelizing Gpgpu-sim for the purpose of improving simulation efficiency. Starting with a profiling of the execution time of Gpgpu-sim, we first identify which parts of the program to parallelize in order to achieve the best speedup. Then, two parallelization techniques are applied, resulting in two parallelized programs. In the first version, referred to as *gpusim-para1*, we use conventional multithreading to parallelize the execution of the shader core pipelines. This greatly improves simulation speed. With only one additional thread, we have achieved an average speedup of 1.47 on a set of benchmark CUDA programs. On top of this, in the second version, *gpusim-para2*, we apply parallelization across different clock domains, achieving speedup of up to 2.65 with 5 threads. Unlike some existing work that trades off accuracy for speedup, we strive to achieve speedup while maintaining high fidelity to the original sequential program in terms of simulation results. A user can therefore gain the speed advantage with high confidence in the results. Furthermore, in both programs, particularly *gpusim-para1*, we keep the code changes to a minimum (about 170 lines of code for *gpusim-para1*). To validate the parallel programs, we compare the statistics produced by the programs with those produced by the original program. In many cases the statistics are identical, and the worst-case differences are 0.22% and 1.4% respectively for the two programs.

II. Background and Related Work

Besides Gpgpu-sim, there exist a number of sequential GPGPU simulators for functional and/or timing simulation [3] [4]. However, very few pieces of research on parallelizing GPGPU simulation have appeared in the literature. The most relevant to our current work is probably the system reported in [5], where the focus is on developing a work-group based synchronization algorithm. As in our work, multiple threads are used to parallelize the sequential simulator. In order to maximize parallelism, the synchronization requirements are relaxed such that the threads only synchronize with each other at the end of the work-groups. However, this relaxation leads to significant simulation inaccuracy in terms of simulation cycles. Additional synchronization work has to be done to reduce simulation errors. By contrast, one of our goals for this work is to achieve simulation results, including simulation cycles, as close to those of the sequential simulator as possible. We make the threads synchronize every cycle to achieve this.

GpuTejas [6] is a Java-based GPGPU simulator. Unlike most GPGPU simulators, it starts out as a parallel simulator. As in [5] and our work, threads are the basic unit of parallelism. Similar to [5], it uses relaxed synchronization where the threads simulating the GPU cores synchronize every 1000 cycles. Advanced data structures are utilized to further reduce synchronization overhead. One important difference from Gpgpu-sim is that it does not include functional simulation. Instead it uses pre-generated program traces. Additionally, it proposes a scheduling algorithm to help parallelize the simulator, whereas we do not make any modifications to Gpgpu-sim in this regard. All the code changes in this work are limited to thread synchronization requirements.

From the methodology point of view, one related work is [7], where the focus is on converting existing uniprocessor simulators into parallelized multi-core simulators, whereas in our case the original simulator is created for many-core architectures at the outset. Like our work, their scope is limited to simulation on shared memory platforms, and a key objective is imposing minimal development effort in the parallelization process.

III. Design and Implementation

This section details the design and implementation of our parallelized versions of Gpgpu-sim. At the core of the work are the following two questions.

Q1: Which parts of the program should we parallelize?

Q2: How do we parallelize those parts?

We start with an examination of the basic structure of Gpgpu-sim and a profiling of its execution, and proceed to the parallelization techniques.

A. Basic Structure of Gpgpu-sim and Execution Profiling

In Gpgpu-sim, the system model for timing simulation consists of four major components: SIMT clusters, DRAM, interconnection network (icnt), and L2 cache. Correspondingly the system instantiates four clock domains. Fig. 1 shows the architecture of a generic GPU in Gpgpu-sim. As can be seen, at the highest level, the system consists of an interconnection network and a set of network nodes, each of which is either a SIMT cluster or a memory partition unit. Each SIMT cluster contains one or more shader cores. The cores are highly independent of each other – each has its own pipeline and L1 caches. A memory partition unit is made up of a DRAM unit as well as one or more memory subpartitions (MSPs). Each MSP in turn hosts an L2 slice.

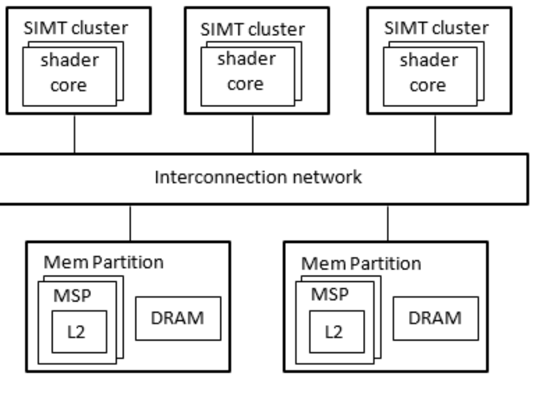

Fig. 1. Generic GPU architecture in Gpgpu-sim.

At runtime, the main function of Gpgpu-sim repeatedly calls the function `gpu_sim::cycle()` until the end of simulation. The simulation time moves from one clock edge to the next according to their chronological order. At each step, a bit-mask is used to indicate which clock edges are to be processed next. The corresponding timing model is then processed for one simulated clock cycle. Alg. 1 shows the pseudo-code for this function. Basically it is made up of six segments of code, in the particular order as shown. We summarize what each code segment does in the trailing comments. Each of the six segments is an *if* block that is executed when the corresponding bit for the clock domain is set in the bit mask. For example, the condition "if core bit set" (similarly for others) means the simulation time has moved to the next edge of the core clock. Therefore, the clock for the cores is now active and the corresponding simulation actions should be taken. Note that the per-cycle actions for the cores and the network are respectively divided into two parts. For our purposes we further divide *core-cycle-part2* into 2 sub-parts.

Algorithm 1 Pseudo-code for gpu_sim::cycle().

```
procedure GPU_SIM::CYCLE
    Update mask
    if core bit set then
        core-cycle-part1              ▷ receive packets from network
    if icnt bit set then
        icnt-cycle-part1 ▷ push pkts from mem controller to netwk
    if dram bit set then
        dram-cycle                                  ▷ dram cycle
    if l2 bit set then
        l2-cycle                                      ▷ l2 cycle
    if icnt bit set then
        icnt-cycle-part2                ▷ network internal transfer
    if core bit set then
        core-cycle-part2-1           ▷ pipeline cycle for all cores
        core-cycle-part2-2                      ▷ actions afterwards
```

To parallelize Gpgpu-sim, the first and foremost question we need to answer is which parts of the program we should parallelize in order to achieve the best speedup. To this end, we note that almost the entire running time of Gpgpu-sim is spent in `gpu_sim::cycle()`. Therefore, to determine which parts of the program to parallelize, we started by profiling the execution time of this function. We ran Gpgpu-sim on 9 of the benchmarks presented in [1] on a desktop computer using the configuration for the GTX480 GPU and measured the execution time of the segments listed in Alg. 1.

Table I shows the profiling results averaged over 5 runs. In addition to the execution time of the individual segments, in the last two columns we show the percentage of the execution time for *core-cycle-part2-1* and the icnt segments. Clearly, *core-cycle-part2-1*, where the pipelines of all the shader cores are simulated, dominates the simulation time, accounting for 84.3% (MUM) - 95.7% (AES) of the execution time of `gpu_sim::cycle()`. The execution time of the network in some cases also constitutes a significant portion, e.g. 9.2% for MUM, while the DRAM and L2 are less significant.

Based on these data, our parallelization strategy consists of two steps: (1) Parallelize the execution of *core-cycle-part2-1*, i.e., the shader core pipelines, and (2) Run other components, i.e., icnt, l2, DRAM in parallel with the cores.

TABLE I
EXEC. TIME OF SEGMENTS OF GPU_SIM::CYCLE() IN μS: AVG. OF 5 RUNS

Bench	core1	core2-1	core2-2	icnt1	icnt2	dram	l2	core2-1%	icnt%
AES	1.0	809.0	16.5	1.0	10.2	3.6	3.8	95.7%	1.3%
BFS	1.2	296.9	20.9	2.5	19.6	2.9	4.3	85.2%	6.3%
CP	0.7	517.3	12.2	0.4	5.7	3.0	3.3	95.3%	1.1%
LIB	1.0	259.0	11.5	1.3	14.9	4.1	4.9	87.3%	5.5%
LPS	1.6	550.8	12.2	3.5	25.2	4.9	6.3	91.1%	4.7%
MUM	1.5	355.3	11.8	4.9	33.9	6.0	7.9	84.3%	9.2%
NN	0.8	345.1	13.6	1.1	10.3	2.7	3.4	91.5%	3.0%
RAY	1.1	517.2	13.5	1.2	12.9	3.1	3.9	93.5%	2.6%
STO	0.8	445.4	11.0	0.7	7.8	2.9	3.5	94.3%	1.8%

B. *gpusim-para1: Parallelizing core-cycle-part2-1*

The code segment *core-cycle-part2-1* essentially is a for-loop in which the `core_cycle()` function is called for each SIMT cluster, as shown in Listing 1. This function in turn calls the `shader_core_ctx()::cycle()` function for each core belonging to the cluster. The latter, as shown in Listing 2, exercises the pipeline stages one by one.

Listing 1 Partial source code for core-cycle-part2-1.

```
for (i=0; i<m_shader_config->n_simt_clusters; i++) {
    m_cluster[i]->core_cycle();
}
```

Listing 2 Partial source code for shader_core_ctx::cycle().

```
void shader_core_ctx::cycle() {
    writeback();
    execute();
    read_operands();
    issue();
    decode();
    fetch();
}
```

Upon careful examination of the pipeline functions in Listing 2, we find that one core's `shader_core_ctx::cycle()` function is highly, although not completely, independent of other cores. Therefore, our first step in parallelizing Gpgpu-sim is to use multiple threads to execute `core_cycle()` for the clusters. This amounts to running the `shader_core_ctx::cycle()` function for the cores in parallel. Specifically, we create one Leader thread and a number of Worker threads. The Leader thread is essentially the main simulation thread in the original program. The only differences are: (1) it only executes `core_cycle()` for its assigned clusters as opposed to for all the clusters, and (2) it has to synchronize with the Worker threads. The latter simply execute `core_cycle()` for their assigned clusters.

Both the Leader and the Worker threads run in loops. Alg. 2 shows the pseudo-code for one iteration for both types of threads. Although the threads execute the `core_cycle()` function for their assigned clusters in parallel, we force all the threads to synchronize at the end of each cycle in order to prevent any one thread from running too far ahead of the others. In other words, at any point of the simulation, all the threads are in the same simulated clock cycle. This is the key difference from existing works such as [5] and [6]. We impose this restriction because our main goal is high fidelity to the sequential simulation. As for implementation, it may be natural to use a barrier to achieve the per-cycle synchronization. However, we have found using semaphores [8] is much more efficient.

Algorithm 2 Pseudo-code for Leader and Worker threads.

(a) Leader thread

```
1: for each worker thread w do
2:     sem_post(work_sem[w])
3: for $i = start$ to $end$ do
4:     clusters[i]→core_cycle()
5: for $i = 1$ to $W$ do
6:     sem_wait(lead_sem)
7: do remaining tasks
```

(b) Worker thread

```
1: sem_wait(work_sem[w])
2: for $i = start$ to $end$ do
3:     clusters[i]→core_cycle()
4: sem_post(lead_sem)
```

As Alg. 2 shows, a Worker thread starts the cycle by waiting on its semaphore (Worker Line 1). Once the Leader calls `sem_post()` for all Workers (Leader Lines 1-2), all the threads execute `core_cycle()` in parallel (Leader Lines 3-4; Worker Lines 2-3). Once `core_cycle()` has completed, a Worker calls `sem_post()` to notify the Leader about the completion (Worker Line 4), and then goes back to the beginning of the loop to wait for the signal for the next cycle. On the other hand, after it has finished `core_cycle()` for its assigned clusters, the Leader waits for all the Workers to finish the cycle by waiting on its semaphore for W (the number of Workers) times (Leader Lines 5-6). Once all the Workers have finished their cycle, the Leader can move on to finish the remaining tasks of its cycle (Leader Line 7), including simulating other components of the model and updating statistics.

As in any multi-threaded program, accesses to shared variables must be protected by mutex. With Gpgpu-sim, the most common type of variables shared by the SIMT clusters is statistics. If a variable is updated too often, using a mutex will severely degrade performance. Fortunately, values of statistic variables in general are simply a sum of the contributions from individual clusters. If a variable is updated too often, our solution is to create a per-core version of the variable so they can be updated without any mutex, and the values are combined at the end of the simulation.

To achieve high fidelity to the original program, we strive to keep the code changes to the minimum. For *gpusim-para1*, the total changes are about 170 lines of code spread in 13 files, all of which are synchronization related.

C. *gpusim-para2: Parallelization Across Clock Domains*

In *gpusim_para1* we parallelized the simulation of the SIMT clusters. However, the other components of the system, i.e., the network, the L2 caches, and the DRAM, are still simulated sequentially within the Leader thread along with its assigned SIMT clusters. In *gpusim_para2* we try to run those components in parallel with the clusters to further reduce simulation time. Based on the profiling data in Table I and the structure of Gpgpu-sim, we choose to put the icnt, the L2, and the DRAM in a single thread, as depicted in Fig. 2. We refer to this thread as the *ILD-thread* and threads for the SIMT clusters the *C-threads*. The lines between the components in Fig. 2 represent dependence relationships in the form of data exchanges. For example, SIMT clusters and the icnt directly exchange network packets.

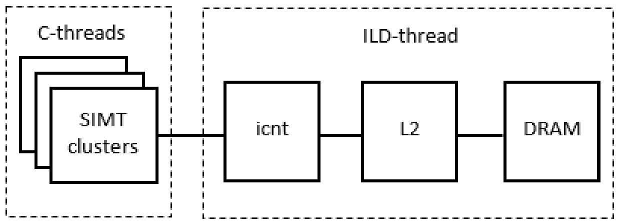

Fig. 2. Simulation threads.

Before creating the ILD-thread, however, we need first to modify the top-level function shown in Alg. 1 so as to separate the code for the clusters from the rest. The resulting functions are shown in pseudo-code in Alg. 3. We now have the code for each clock domain in a separate function. Note that the underlying timing models (e.g., cores) are not changed in any way. All that is changed is the order of the code segments in Alg. 1.

Algorithm 3 Re-structured gpu_sim::cycle().

(a) Functions for clock domains

```
procedure CORE_CYCLE
    core-cycle-part1
    core-cycle-part2
procedure ICNT_CYCLE
    icnt-cycle-part1
    icnt-cycle-part2
procedure DRAM_CYCLE
    dram-cycle
procedure L2_CYCLE
    l2-cycle
```

(b) Main function

```
procedure GPU_SIM::CYCLE
    Update mask
    if core bit set then
        CORE_CYCLE
    if icnt bit set then
        ICNT_CYCLE
    if dram bit set then
        DRAM_CYCLE
    if l2 bit set then
        L2_CYCLE
```

Next we replace the bit-mask based scheduling in Alg. 3(b) with the clock functionality from the Manifold framework [9], a parallel simulation framework for multicore architectures. Compared with the bit-mask, Manifold clock uses a callback mechanism that is more flexible. It also provides better determinism for simultaneous clock edges.

The resulting code is shown in Alg. 4. First we create the clocks (Line 2). Then we register the functions for the clock domains with their respective clocks as callbacks (Line 3-6). Scheduling is now simplified to (1) finding the next active clock (Line 8) and (2) calling the `process_edge` function for the clock (Line 9), which invokes the registered callbacks. Selecting the next active clock edge is transparently handled by the function `next_active_clock` (Line 8), which automatically identifies the next earliest clock edge among the set of clocks.

The ILD-thread poses a synchronization problem that did not occur in *gpusim-para1*. Such problems are the most crucial for any parallel simulation system [10]. As described, in

Algorithm 4 Function gpu_sim::cycle() re-written with Manifold clocks.

```
1: procedure GPU_SIM::INIT
2:     Create CoreClock, IcntClock, L2Clock, DramClock
3:     DramClock→REGISTER(dram_cycle)
4:     L2Clock→REGISTER(l2_cycle)
5:     IcntClock→REGISTER(icnt_cycle)
6:     CoreClock→REGISTER(core_cycle)
7: procedure GPU_SIM::CYCLE
8:     nextClock := NEXT_ACTIVE_CLOCK
9:     nextClock → PROCESS_EDGE
```

gpusim-para1 the work that is done in parallel by the C-threads is all within a single clock domain, i.e., the core clock. Synchronization is easily achieved at the end of each clock cycle. With the design in Fig. 2, however, there are three clock domains in the ILD-thread. Synchronization between the C-threads and the ILD-thread is more complicated. Consider the two clock edges A and C in Fig. 3. The clocks have different frequencies. If edge C is chronologically after A, this order is easily maintained when the two clocks are in the same thread. Maintaining this order is more complicated when the two clocks are in different threads.

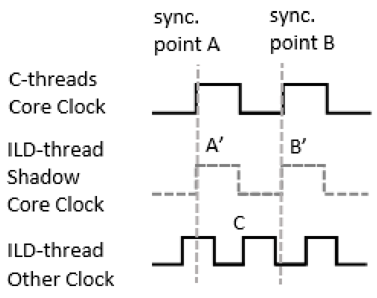

Fig. 3. Synchronization between C-threads and ILD-thread.

We address this problem in two steps. First, in the ILD-thread, we create a shadow core clock, whose sole purpose is to synchronize with the Leader C-thread. As Fig. 3 shows, now the C-threads and the ILD-thread synchronize through the core clocks. For example, edge A will synchronize with A'. The only change required for the Leader thread in Alg. 2(a) is to increase the number of calls of `sem_post` and `sem_wait` from W to $W+1$. However, naively adopting the synchronization method of the Worker threads in Alg. 2(b) for the ILD-thread would be problematic. In Alg. 5 we show one iteration of this incorrect implementation.

Algorithm 5 Incorrect ILD-thread iteration.

```
1: nextClock := NEXT_ACTIVE_CLOCK
2: if nextClock == CoreClock then
3:     sem_wait(ILD_sem)
4: nextClock→PROCESS_EDGE
5: if nextClock == CoreClock then
6:     sem_post(lead_sem)
```

We use the following sequence of events to illustrate the problem: (1) At edge A', because it is the core clock, we call `sem_wait` in Line 3 and wait for the Leader to call `sem_post` (Leader Line 2 in Alg. 2). (2) The leader has called `sem_post` and we proceed to Line 4. As this is the shadow clock, Line 4 is basically a no-op, and we proceed to Line 6 and call `sem_post`. This signals to the Leader that the cycle between points A' and B' is completed, which is incorrect, because, for example, edge C has not been processed yet. (3) Once the Leader has received all the signals, it will proceed to the next cycle. Therefore, edge B could be processed before C.

The key to this problem is the semantics of the call `sem_post(lead_sem)`. In Alg. 2(b) this call means the Worker has completed the current cycle. Therefore, when we make this call, we have to ensure that the current cycle is indeed completed. The correct ILD-thread is shown in pseudo-code in Fig. 6. Note that the call of `sem_post` appears before `sem_wait`. Now `sem_post` means the *previous* cycle has completed (semantics of `sem_post` for the Worker threads is not changed). Again we use the previous sequence: (1) At edge A', none of the C-threads has started processing edge A before the `sem_post` in the ILD-thread. Once it is called, they can process edge A, (2) After the C-threads have processed the cycle between A and B, they cannot process edge B before the ILD-thread calls `sem_post` at B', (3) When the ILD-thread calls `sem_post` at B', edge C would have already been processed. Therefore, C is processed before B, as it should have been.

Algorithm 6 The ILD-thread.

```
1: procedure ILD_THREAD
2:     Create clocks
3:     Register callbacks with clocks
4:     while simulation not ended do
5:         nextClock := NEXT_ACTIVE_CLOCK
6:         if nextClock == CoreClock then
7:             if not first cycle then
8:                 sem_post(lead_sem)
9:             sem_wait(ILD_sem)
10:        nextClock→PROCESS_EDGE
```

IV. Evaluation

A. Experimental Setup

All of our experiments, unless expressly stated, are conducted on a desktop computer that is equipped with an Intel Core™ i7-4790 3.6GHz quad-core processor with a total of eight hardware threads. We therefore limit the total number of threads of our programs to less than eight. The operating system is Ubuntu 14.04. Our parallelized Gpgpu-sim programs are based on Gpgpu-sim 3.2.2 [11] and built with the original build system. We select GTX480 as the GPGPU architecture for the experiments. We use the original configuration for GTX480 without any modifications. In this configuration there are 15 SIMT clusters and each has 1 shader core. There are 6 memory partitions with 2 sub-partitions each. Therefore, there are in total 21 network nodes, i.e., 15 clusters plus 6 memory partitions.

For inputs, we use 9 benchmarks from the set presented in [1]. These are selected based on two criteria: (1) They can be successfully built, and (2) They each run for more than 30 seconds with the original sequential program.

B. Experimental Results and Analysis

This section presents our experiment results and analysis thereof. Our focus here is on two quantities: speedup and fidelity. The former measures how fast the parallel programs are, and the latter how close the simulation results are, in comparison to the original sequential program.

As Gpgpu-sim reports the execution time at the end of each simulation run, in the following, all the simulation time results are those reported by Gpgpu-sim. For measuring fidelity, we use selected statistics produced by Gpgpu-sim.

1) Results for gpusim-para1: The left part of Table II shows the simulation time of *gpusim-para1* averaged over 10 runs. We used 2 to 6 threads (1 Leader and $n-1$ Workers). Assignment of the SIMT clusters to the threads is done manually with load-balance in mind.

In Fig. 4 we show the speedup values of *gpusim-para1* relative to the sequential program. As can be seen, with just 2 threads, we have reduced simulation time significantly. The speedup ranges between 1.37 (RAY) and 1.59 (CP) with a median of 1.46 and an average of 1.47. Best results are generally obtained with 5 threads, which produced speedup between 1.61 (LIB) and 2.24 (CP) with a median of 1.87 and an average of 1.91. Results with 6 threads are very close to those with 5 threads. From 2 to 5 threads, the general trend is that the more threads the faster the simulation. However, in a few cases we see that the 3-thread program has better results than using 4 threads.

TABLE II
SIMULATION TIME (S): *gpusim-para1* VS. SEQUENTIAL, AVG. OF 10 RUNS.

Bench	Seq	2-thrd	3-thrd	4-thrd	5-thrd	6-thrd	Seq	15-thrd
AES	33.3	22.9	19.3	18.3	16.7	16.0	44.7	12.0
RAY	69.4	50.5	43.2	41.5	38.7	39.0	100.6	30.3
LPS	102.6	65.2	53.6	56.5	49.5	52.6	140.8	39.1
CP	125.6	79.0	61.8	63.1	56.0	56.1	185.0	44.5
STO	183.1	125.3	108.1	103.9	96.1	96.1	242.8	84.2
BFS	266.4	175.8	150.8	161.6	142.4	147.7	431.7	147.7
NN	676.4	458.7	365.1	412.1	365.6	378.6	1047.6	364.5
MUM	870.9	597.4	502.2	559.0	479.0	504.3	1472.8	460.6
LIB	1477.2	1114.5	1007.1	1020.7	918.4	940.1	2375.6	868.8

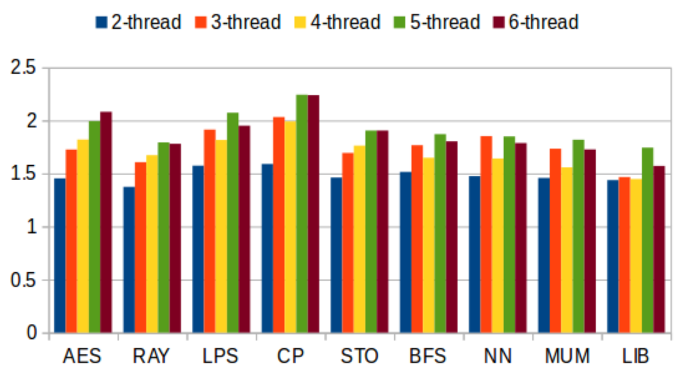

Fig. 4. Speedup of gpusim-para1 with 2 to 6 threads.

When the CPU resource is available, further speedup can be achieved. To demonstrate this, we run *gpusim-para1* with

15 threads on a 16-core Intel Xeon® E5-2620 2.1GHz CPU. In the right part of Table II we compare the results with those of the original sequential program. Speedup values between 2.73 (LIB) and 4.16 (CP) are achieved with a median of 3.20 and average 3.27. Unfortunately but expectedly, due to the limitation set by Amdahl's law and the synchronization overhead, the speedup efficiency decreases as the number of threads increases. With 2 threads, the average efficiency is 0.74, a rather high value, while it becomes 0.38 and 0.22 with 5 and 15 threads respectively.

To verify the simulation results fidelity to the sequential simulator, we selected the following 6 statistics produced by Gpgpu-sim, each from a different category.

- `gpu_sim_cycle`: the number of simulated shader core cycles.
- `L1I_total_cache_accesses`: the number of accesses to the L1 instruction cache.
- `L2_total_cache_accesses`: the number of accesses to the L2 cache.
- `icnt_total_pkts_mem_to_simt`: the number of network packets originated from the memory partitions.
- `total_reads`: the number of reads to the DRAM.
- `gpgpu_n_tot_w_icount`: the number of warps that have completed the pipeline.

Table III shows the deviations produced by the 5-thread *gpusim-para1* from the sequential version. The number 0 means the results are identical. We can see the fidelity is very high. In all cases except BFS, the statistics are identical to those of the sequential version. With BFS, the worst difference is 0.22%.

TABLE III
SIMULATION RESULTS DIFFERENCES: 5-THREAD GPUSIM-PARA1 VS. SEQUENTIAL (CYCLE: GPU_SIM_CYCLE; L1I: L1I_TOTAL_CACHE_ACCESSES; L2: L2_TOTAL_CACHE_ACCESSES; ICNT: ICNT_TOTAL_PKTS_MEM_TO_SIMT; DRAM: TOTAL READS; WARP: GPGPU_N_TOT_W_ICOUNT).

Bench	Cycle	L1I	L2	Icnt	Dram	Warp
AES	0	0	0	0	0	0
BFS	0.18%	0.02%	-0.01%	-0.01%	-0.22%	0.02%
CP	0	0	0	0	0	0
LIB	0	0	0	0	0	0
LPS	0	0	0	0	0	0
MUM	0	0	0	0	0	0
NN	0	0	0	0	0	0
RAY	0	0	0	0	0	0
STO	0	0	0	0	0	0

It is important to point out that the differences in Table III for BFS do not mean there are simulation errors. They exist only because the order we simulate the shader core pipelines (Listing 2) is different from the order used by Gpgpu-sim. As shown in Listing 1, Gpgpu-sim simulates the SIMT clusters in ascending order of their IDs. This means, for example, all the pipeline stages of a core in cluster 1 are simulated before a core in cluster 2 is simulated. This is an arbitrary choice. With our parallel programs, simulation of the pipeline stages of a core in one cluster and a core in another cluster can be concurrent, although for any given core, its pipeline stages are still simulated in the order given in Listing 2.

To further demonstrate this, we tested BFS with the original simulator except at the beginning of the simulation we give the clusters a random order. This was run for 10 times and Table IV shows the results as compared with the original program. It can be seen the differences are similar to the ones in Table III.

TABLE IV
BFS: RANDOM CLUSTER ORDER VS. ORIGINAL.

Diff	Cycle	L1I	L2	Icnt	Dram	Warp
Min	0.03%	-0.02%	-0.01%	-0.02%	-0.08%	-0.02%
Max	0.26%	0.19%	0.01%	0.02%	0.01%	0.17%

2) Results for gpusim-para2: Table V shows simulation time of *gpusim-para2* with 3 to 7 threads. For ease of comparison with *gpusim-para1*, the thread count shown is for the C-threads only. Keep in mind that there is one additional thread, i.e., the ILD-thread. The sequential version, named *Seq2*, is based on the restructured program shown in Alg. 3. It is between 0.3% and 13.9% faster than the original program when tested with the 9 benchmarks.

TABLE V
SIMULATION TIME (S): *gpusim-para2* VS. *Seq2*, AVG. OF 10 RUNS.

Bench	Seq2	2-thrd	3-thrd	4-thrd	5-thrd	6-thrd
AES	33.1	21.3	17.4	16.2	14.1	14.1
RAY	69.2	46.3	38.8	36.3	33.0	32.3
LPS	98.1	56.1	45.1	42.5	38.7	38.6
CP	117.2	72.9	58.2	56.4	49.5	49.2
STO	157.6	116.6	102.0	95.9	87.4	85.6
BFS	237.1	176.9	147.5	153.9	140.7	152.0
NN	613.3	417.6	334.8	359.2	312.1	292.6
MUM	839.4	603.0	489.0	497.7	454.2	487.8
LIB	1443.6	962.2	848.9	820.1	745.3	769.0

Compared with *gpusim-para1*, we see further improvements in simulation speed. In Fig. 5 we show the speedup values relative to the original sequential program.

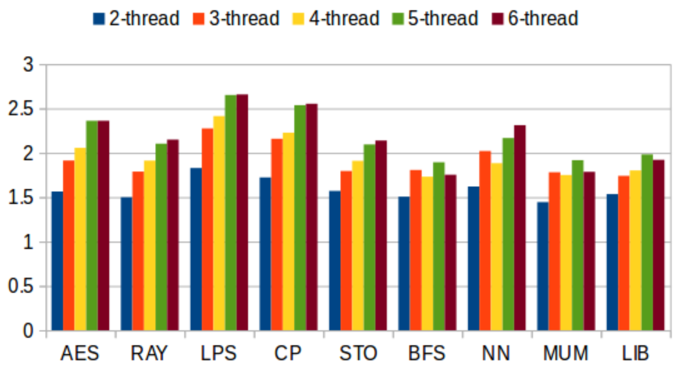

Fig. 5. Speedup of gpusim-para2 with 2 to 6 C-threads.

With 2 C-threads, speedup against the original Gpgpu-sim ranges between 1.44 (MUM) and 1.83 (LPS) with a median of 1.57. This is a significant improvement compared with *gpusim-para1*'s speedup range of 1.37 to 1.59. The program with 5 C-threads produced the best or close to best results, with speedup between 1.89 and 2.65 and a median of 2.10.

Table VI shows deviations of *gpusim-para2* (5 C-threads) from the original Gpgpu-sim in terms of the 6 selected statistics. As expected, the differences are bigger than *gpusim-para1*. However they are in general still quite small. We can see that most of the results are less than 0.5%, with the worst case being 1.39%. This attests to the high fidelity of the parallel program to the original.

TABLE VI
SIMULATION RESULTS DIFFERENCES: *gpusim-para2* (5 C-THREADS) VS. *Original* (CYCLE: GPU_SIM_CYCLE; L1I: L1I_TOTAL_CACHE_ACCESSES; L2: L2_TOTAL_CACHE_ACCESSES; ICNT: ICNT_TOTAL_PKTS_MEM_TO_SIMT; DRAM: TOTAL READS; WARP: GPGPU_N_TOT_W_ICOUNT).

Bench	Cycle	L1I	L2	Icnt	Dram	Warp
AES	-0.59%	-0.05%	0.09%	0.06%	0	0
BFS	0.19%	0.10%	-0.14%	-0.13%	0.13%	0.08%
CP	0.14%	-0.00%	-0.10%	-0.17%	0	0
LIB	-0.22%	-0.00%	0.00%	0.00%	-0.00%	0
LPS	0.04%	-0.00%	0.01%	0.01%	0.41%	0
MUM	0.32%	-0.00%	0.02%	0.02%	-0.02%	0
NN	-0.06%	0.00%	1.34%	1.39%	0.26%	0
RAY	-0.22%	-0.12%	0.20%	0.10%	-0.68%	0
STO	-0.28%	0.00%	0.27%	0.48%	0.04%	0

V. OBSERVATIONS AND FUTURE WORK

We make some observations about the current work and point out possible future work.

One remaining source of non-determinism in *gpusim-para2* is simultaneous events across the two thread groups. In the current Gpgpu-sim, network packets are sent/received with direct function calls. This is fine for sequential programs, or for parallel programs if the network nodes and the network are in the same thread. However, in general, direct function calls across different components pose synchronization problems for parallel programs. A better modeling is to have some delay between the sending and the receiving. That way, simultaneous events in the icnt can have a more deterministic order even if the sender/receiver is in a different thread.

As stated, our objective for this work is to parallelize Gpgpu-sim with minimal changes and to attain high fidelity to sequential simulation. To this end we force the threads to synchronize every clock cycle. Removing this restriction is likely to produce higher speedup, but may require more substantial code changes. We leave this effort for future work.

VI. CONCLUSION

This paper presents two techniques used in parallelizing Gpgpu-sim. Unlike existing works that trade off accuracy for speedup, one of our main goals is to maintain high fidelity to the original program. First we use multithreading to parallelize the most time-consuming part of the program with minimal code changes and obtain significant speedup. Secondly, we parallelize across clock domains and address the problems therein. Tested on a set of benchmark programs, our parallel programs achieve speedup up to 2.65 while deviate from the sequential program by less than 1.4% in terms of key simulation statistics.

REFERENCES

[1] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2009, pp. 163–174.
[2] "List of nvidia graphics processing units," https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units.
[3] V. M. Del Barrio, C. González, J. Roca, A. Fernández, and E. Espasa, "Attila: a cycle-level execution-driven simulator for modern gpu architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 231–241.
[4] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.
[5] S. Lee and W. W. Ro, "Parallel gpu architecture simulation framework exploiting work allocation unit parallelism," in *IEEE Int'l Symp. on Perf. Analysis of Systems and Software*, 2013, pp. 107–117.
[6] G. Malhotra, S. Goel, and S. R. Sarangi, "Gputejas: A parallel simulator for gpu architectures," in *International Conference on High Performance Computing (HiPC)*, 2014, pp. 1–10.
[7] J. Donald and M. Martonosi, "An efficient, practical parallelization methodology for multicore architecture simulation," *IEEE Computer Architecture Letters*, vol. 5, no. 2, 2006.
[8] K. A. Robbins and S. Robbins, *Practical UNIX programming: a guide to concurrency, communication, and multithreading*. Prentice-Hall, Inc., 1995.
[9] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao *et al.*, "Manifold: A parallel simulation framework for multicore systems," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 106–115.
[10] R. M. Fujimoto, *Parallel and distributed simulation systems*. Wiley New York, 2000.
[11] "Gpgpu-sim distribution," https://github.com/gpgpu-sim.

Jun Wang is a senior researcher at Futurewei Technologies. He holds a PhD in Computer Science from McGill University. His current research interests include computer architecture, parallel computing, mobile computing, and optimizing compilers.

Jiaquan Gao is a professor of Computer Science at Nanjing Normal University. He holds a PhD in Computer Science from the Chinese Academy of Sciences. His current research interests include high-performance computing, parallel algorithms on heterogeneous platforms, computational intelligence, and information visualization.