Accelerating GPGPU Simulation by Strategically Parallelizing the Compute Bottleneck

Jakob Sachs
Hamburg University of Technology, Germany

Tim Lühnen
Hamburg University of Technology, Germany

Sohan Lal
Hamburg University of Technology, Germany

Abstract

Cycle-accurate GPGPU simulators like GPGPU-Sim provide invaluable insights for hardware architecture research but suffer from extremely long runtimes, hindering research productivity. This paper addresses this critical bottleneck by proposing a strategy to accelerate GPGPU-Sim. We first perform a holistic profiling analysis across diverse GPGPU benchmarks to identify the primary performance bottleneck, pinpointing the SIMT-Core cluster execution within the CORE-clock cycle. Based on this, we implement a parallelization scheme that strategically targets this hotspot, utilizing a thread pool to manage concurrent execution of SIMT-Core clusters. Our approach prioritizes minimal modifications to the existing GPGPU-Sim codebase to ensure long-term maintainability. Evaluation of a simulated NVIDIA H100 model demonstrates an average simulation wall-time speedup of 3.58× with 8 worker threads, and a maximum up to 4.38×, while incurring a maximum cycle count error of 3.22%, with some other benchmarks exhibiting no error at all.

2012 ACM Subject Classification Computing methodologies → Parallel programming languages; Computing methodologies → Simulation tools; Computer systems organization → Parallel architectures

Keywords and phrases GPGPU, CUDA, Simulation, Computer Architecture, GPGPU-Sim, Parallel Simulation, Cycle-Accurate Simulation, Thread Pool

1 Introduction

Since the introduction of the GPGPU (*General Purpose Computing on GPUs*) paradigm, the need for accelerated computing has only grown, both in terms of scale and applications. This has increased the pressure on hardware architecture research to deliver higher performance/efficiency solutions and faster iteration on these new architectures. Simulation of hardware architectures is a widely used tool in the toolbox of hardware architecture researchers. For GPGPU, there are several established tools, of which a prominent one is GPGPU-Sim [8], which aims to be a cycle-accurate GPGPU simulator, meaning it directly models the target hardware's behavior on a clock-cycle-by-clock-cycle basis, offering deep insights into performance and internal operations (e.g., pipeline states, resource utilization, and cache behavior), particularly for NVIDIA GPUs. For our implementation, we build upon ClusterSim [11], an extension of GPGPU-Sim supporting modern architectural features.
However, such detailed simulation comes with significant performance impact, both due to the overhead inherent to all hardware simulations, and due to the shift in computational architecture from a SIMD processor (GPU) to a sequential CPU, leading to runtimes of hours and even days, as the parallelism of the GPU must be serialized for execution on the CPU.

To illustrate the scale of this challenge, one of our test cases (a 2048×2048 single-precision floating-point matrix multiplication kernel running on a simulated H100) originally took ≈ 9 hours to complete, while the real device runs the same kernel in 1.2 milliseconds. This dramatic difference in execution time represents a critical bottleneck in the research workflow and severely limits the search space of architectural parameters that researchers can explore with cycle-accurate simulators.

This critical bottleneck motivates our primary objective in this work: To reduce the "real" runtime (wall-time) of these simulations. Our empirical approach is based on profiling the original simulation and pragmatically parallelizing the identified hotspot, allowing us to reduce the overall amount of changes made to the code base, as opposed to a top-down strategy mandating large-scale structural refactoring from the outset.

This strategy offers two key advantages: First, it helps contain the project's scope, ensuring feasibility. Secondly, it significantly enhances the long-term maintainability of our parallelization extension. By limiting invasive changes, we simplify the process of integrating future updates or features and also leave open the possibility of future attempts at parallelizing other parts of the simulator. In this paper, we contribute the following:

- A holistic profiling of the GPGPU-Sim program across a diverse set of GPGPU benchmarks, to identify and verify the primary performance-limiting component: specifically the SIMT-Core execution inside the *CORE*-clock section.
- A parallelization scheme, based on our profiling results for the SIMT-Core execution. Our implementation distinctively prioritizes minimal modifications to the core GPGPU-Sim code base, to ensure long-term maintainability and ease of integration with future updates, while significantly decreasing simulation wall-times by an average $3.58\times$ with a maximum error of 3.22%.
- A comprehensive evaluation of our parallelization approach conducted on a simulated NVIDIA H100 model, which quantifies achieved simulation speedups against the original sequential implementation, measures, and analyzes the performance scaling of our implementation across different thread counts and measures the incurred error in simulated clock-cycle count across our set of benchmarks.

2 Background and Related Work

In this section, we will quickly cover the internal architecture of GPGPU-Sim and previous research on the same or related subjects.

2.1 GPGPU-Sim

GPGPU-Sim is a cycle-accurate simulator for GPUs that supports simulating the execution of applications written using both CUDA and OpenCL. For the purposes of this work, we focus exclusively on CUDA programs. The simulator functions by intercepting the application's CUDA calls and replacing them with its own calls. This allows it to simulate the target GPU architecture in detail and collect performance statistics. This interception approach provides significant ease of usability, often requiring minimal application modification. GPGPU-Sim also includes a functional-only mode, which bypasses detailed timing simulation. It also incorporates a power model for energy analysis and tools for visualization. GPGPU-Sim's hardware model is designed to closely model contemporary NVIDIA GPUs. The main architecture features of the model are:

- **SIMT cores**: These are the main execution units on which thread blocks/CTAs are scheduled and are thus the equivalent of NVIDIA's Streaming Multiprocessors (*SMs*).
- **SIMT core clusters**: Clusters of multiple SIMT cores that share an interface to the memory interconnect, occupying the same place in the organizational hierarchy as NVIDIA's texture processing clusters (*TPCs*).
- **GPC**: A collection of SIMT core clusters. This level incorporates an internal high-speed network (the SM-to-SM network) connecting its constituent clusters without needing to go through the global interconnect. This addition of the GPC structure and its internal network enables the simulation of features like Hopper's distributed shared memory and thread block clusters.
- **Memory Partition**: Represents a slice of the device's memory subsystem, typically including a portion of DRAM and its associated L2 cache slice.
- **Interconnect**: The interconnect between the SIMT core clusters and the memory partitions. GPGPU-Sim utilizes BookSim [7] to simulate this interconnect.

The component layout is shown in Figure 1. The main driver for larger computing power is the increased component counts, especially the number of SMs, which is the dominant contributor to simulation time (Subsection 3.1). Historically, NVIDIA's flagship SM count grew from up to 16 (Fermi, 2010) [18] to up to 144 (Hopper, 2022) [1]. This trend is advantageous for our scheme, as it is the dimension along which we parallelize (Subsection 3.2). The simulation is controlled by a central clock, which serially steps through sub-clocks (*CORE*, *L2*, *DRAM*, *ICNT*, *SM_NETWORK*). Our optimizations focus entirely on the *CORE*-clock section, as this handles instruction execution and scales the most with grid size.

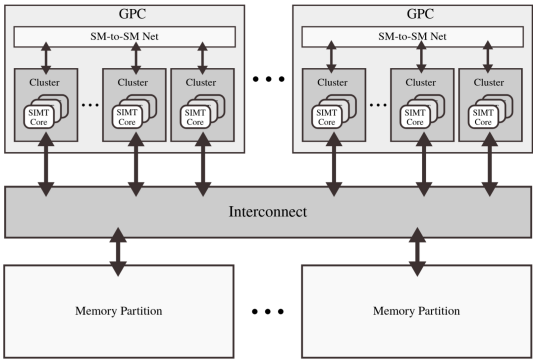


Figure 1 A simplified representation of the hierarchical abstract hardware model described in this section.

2.2 Related Work

The GPGPU simulation landscape extends beyond cycle-accurate simulators like GPGPU-Sim. Machine-learning-based models predict kernel performance [19] or the performance gain of porting applications from CPU to GPGPU [2]. Analytical approaches also statistically model architecture behavior based on execution traces [4, 9, 17]. Despite their utility, researching novel architectural designs often requires the higher fidelity of computationally intensive, cycle-accurate simulators. While valuable for research [6, 12, 14], the slow speed of cycle-accurate simulators like GPGPU-Sim limits study scope and motivates efforts to improve their performance. Early attempts to parallelize GPGPU-Sim, such as by Lee et al. [10],

split hardware into shared (memory, interconnect) and parallel (SIMT-Cores) components. This achieved up to 4.15× speedup (3.39× average with 6 threads) but incurred substantial cycle count errors of up to 5.78%.
Zhao et al. [20] implemented a two-level scheme, parallelizing "intra-kernel" (across SIMT-core-clusters) and "inter-kernel" (across machines), estimating a $10 - -40\times$ speedup with a 0.16% cycle error, though these results are just a theoretical combination of their two separate results without empirical verification of the combined approach. Wang et al. [16] proposed a two-step parallelization: first, evaluating SIMT cores in parallel ($1.61 - 2.24\times$ speedup, $\leq 0.22\%$ error), and second, separating *CORE*, *L2*, *Interconnect*, and *DRAM* clock domains using a "shadow-clock". However, such strategies often involve accuracy trade-offs or extensive code changes, hindering adaptability and upstream integration. More recently, Huerta et al. [5] accelerated Accel-Sim using OpenMP with minimal code modification, achieving an average speedup of 5.8× (up to 14×) with 16 threads.

3 Profiling and Parallelization Strategy

While from a purely architectural point of view, there are some clear ideas about which parts of the simulation can be easily parallelized, we decided to focus on a highly empirical approach, by profiling the original implementation to ensure that our beliefs are correct and we don't fall into the trap of premature optimization. So next we will briefly describe our profiling setup and the hotspot analysis results.

3.1 Profiling and Bottleneck Analysis

The original implementation was profiled using Linux's `perf` tool. We profiled our benchmarks on an AMD EPYC 9124 running at 3 GHz and up to 3.70 GHz in turbo and then evaluate the results by examining the percentage of samples recorded inside each function as a proxy for time spent inside. As anticipated, our profiling results reveal that almost the entirety of the runtime is spent with cycling the simulator clock. On average 98.3% of all recorded samples are within the `gpgpu_sim::cycle()` function and its child calls (*SGEMM*: 99.2%, *SM2SM_HIST*: 98.1%, *MONTE_CARLO*: 98.6%, *VECTOR_ADD*: 97.2%).
To get a clearer understanding of this, we observe the child calls of `gpgpu_sim::cycle()`, given its role as the primary simulation clock. Analyzing these child calls reveals a consistent pattern: the same set of functions are the primary contributors to the runtime of `gpgpu_sim::cycle()` across all benchmarks. Among these, one function consistently emerges as the dominant contributor, although its exact percentage contribution varies per benchmark.

Benchmark	*SGEMM*	*SM2SM_HIST*	*MONTE_CARLO*	*VECTOR_ADD*
`clst.core_cycle()`	**61.36**	**87.93**	**92.91**	**86.03**
`clst.getCacheStats()`	27.49	6.15	4.24	5.83
`LocalIcnt_transfer()`	5.17	1.18	0.65	1.36

Table 1 Percentage of total sample count for the top three child calls of `gpgpu_sim::cycle()`, highlighting the largest hotspot across all benchmarks.

Profiling the major child calls of `gpgpu_sim::cycle()` (as detailed in Table 1) confirms that the *CORE*-Clock, handled by `simt_core_cluster::core_cycle()`, is the dominant contributor to runtime. This confirmation validates our focus and allows us to proceed with the design and implementation of the parallelization scheme.

3.2 Parallelization Scheme

We'll now outline our approach to parallelizing the *CORE*-clock cycle and explain why our method is preferable. Given that profiling identified `simt_core_cluster::core_cycle()` as the primary contributor within the *CORE*-clock's execution breakdown, we must first examine its surrounding code structure to inform our parallelization strategy. The main simulator clock is composed of different sub-clocks, which execute based on which one is active. `gpgpu_sim::cycle()` represents this main simulation clock in code, managing the execution of the separate sub-clocks, as structurally simplified in Algorithm 1.

Algorithm 1 Simplified pseudo-code showing the approximate structure inside the `gpgpu_sim::cycle()` function, where the functions are meant as placeholders for the actual code.

```
clock = get_next_clock_domain()
if clock == L2 then
    L2_cycle()
if clock == CORE then
    core_cycle()                                  ▷ hotspot section
if clock == DRAM then
    dram_cycle()
if clock == ICNT then
    icnt_cycle()
if clock == SM_NETWORK then
    sm2smnet_cycle()
```

Based on the profiling results in Subsection 3.1, the *CORE*-clock section represented by `core_cycle()` contains the primary hotspot function. We will now examine the actual internal structure of this `core_cycle()` section.

Algorithm 2 Simplified pseudo-code showing the insides of the placeholder `core_cycle()`, and displaying the context around the hotspot `simt_core_cluster::core_cycle()`.

```
for clst in clusters do
    if clst.is_active or work_remaining() then
        clst.core_cycle()                         ▷ hotspot function
    update_stats_for_cluster(clst)
```

As illustrated in Algorithm 2, the compute-heavy portion of each cycle is largely independent across the different SIMT-clusters, with dependencies limited primarily to control logic (scheduling and statistics). Similar to related work covered in Subsection 2.2, we chose to parallelize along this cluster dimension: the set of all clusters is split, and the `clst.core_cycle()` function for each is evaluated in parallel. This approach is the least invasive method for addressing the identified hotspot and minimizes synchronization overhead compared to parallelizing individual SIMT-Cores within the clusters.
Parallelizing each individual `<...>::core_cycle()` by launching a new software thread is infeasible due to the substantial overhead from frequent thread creation and destruction in latency-sensitive code. To quantify this overhead, consider the *SGEMM* 1024×1024 benchmark, which runs for 2.4 hrs and involves 1.5×10^6 *CORE*-cycles.

Assuming an optimistic 10 μs overhead per thread-launch, the total wall-time added solely by thread creation across the simulation's 57 clusters is:

$$57 \text{ clusters} \cdot 10\,\mu\text{s} \cdot 1.5 \times 10^6 \text{ cycles} \approx 14.25\,\text{min}$$

This overhead represents approximately 10% of the total runtime and becomes a bottleneck; any performance improvements must first overcome this initial $\approx$ 10% slowdown, which is likely underestimated. To address this bottleneck in thread management, we instead opted for a thread reuse scheme, implemented using the C++ thread pool by [15]. This dynamic thread pool approach allows us to control the number of software threads used for cycle execution, enabling precise tuning based on both the host hardware and the simulated hardware model. The pool is initialized once at startup and currently handles only the SIMT-Cluster parallelization.

Algorithm 3 Simplified pseudo-code showing the actual dispatching of the cluster-cycles to the worker threads, as mostly handled inside the thread pool.

```
1: chunks = split_up_evenly(clusters)
2: for chunk in chunks do
3:     dispatch_to_thread(chunk)              ▷ following code is ran in worker thread
4:     for clst in chunk do
5:         if clst.is_active or work_remaining() then
6:             clst.core_cycle()
                                                                    ▷ in main-thread
7: wait_for_worker_threads()
8: for clst in clusters do
9:     update_stats_for_cluster(clst)
```

Algorithm 3 shows the simplified thread pool structure, detailing how tasks are dispatched to worker threads and which components are parallelized. We explicitly wait for all clusters to complete their core cycle before updating statistics based on their finished state. This statistics collection step is kept sequential, as it operates on small pieces of shared state and parallelization is unlikely to provide benefit while potentially introducing race conditions. Access to the shared state manipulated by `simt_core_cluster::core_cycle()` is primarily managed using `std::mutex` from the C++ standard library. This static, even partitioning of clusters can be suboptimal under partial system loads. When only a fraction of SIMT-Clusters are active, threads assigned to inactive clusters are underutilized. While detecting and reassigning empty cycling clusters to a single thread could improve speedup in these non-full occupancy scenarios, the additional scheduling logic required would add latency, potentially reducing speedup in the optimal, near-full occupancy case. Given that high-occupancy scenarios correspond to the longest sequential wall-times and are the primary optimization target, we chose to optimize for the highest occupancy rather than implementing dynamic load balancing.

4 Experimental Setup

To evaluate our parallelization scheme, we implemented our proposed solution into ClusterSim [11], which is a modified version of GPGPU-Sim, that extends the base simulator with modern features, such as the SM-to-SM-network introduced in the Hopper architecture, to support functionality like thread block clusters and distributed shared memory.

4.1 Modeled GPU configuration

The model configuration used for all tests is designed to replicate the NVIDIA H100 architecture. Specifically, the simulation aimed for a replica of the H100 PCIe 80 GB configuration, which features 114 SMs, as briefly discussed in Subsection 2.1. The primary focus is the number of SIMT clusters and SIMT cores (SMs) in the model. Since parallelization occurs across these clusters, the high cluster count is expected to positively influence speedup results.

4.2 Hardware & Software environment

The benchmarks were executed on a compute node equipped with dual Intel Xeon Gold 6130 CPUs. Unlike the AMD login node used for profiling, these nodes offer a SMP configuration of their cores, which is more suitable for our workload. These nodes provide a total of 32 physical cores (2 sockets $\times$ 16 cores/socket, with hyper-threading disabled) operating at a 2.10 GHz base clock speed (3.70 GHz turbo), supported by 192 GB of RAM split across the two NUMA nodes. For configurations with $N \leq 16$ worker threads, we pinned the process to a single NUMA node using `numactl --cpunodebind=0 --membind=0`. For $N > 16$, threads were allowed to span both NUMA nodes under default Linux scheduling.

4.3 Benchmarks

To evaluate the performance impact of our parallelization strategy across diverse computational patterns, we selected four CUDA kernels. These kernels represent a range of common GPGPU workload characteristics, from compute-intensive tasks to memory-bound operations.

- *MONTE_CARLO*: Estimates π via Monte Carlo using a simple LCG generating single-precision pseudo-random (x, y) pairs. Results are aggregated within thread-blocks via a shared memory reduction, and final summation happens on the host CPU.
- *SGEMM*: For our evaluation, we employ an SGEMM (Single-Precision General Matrix Multiplication) kernel based on [3], which operates on two randomized square matrices using 2D shared-memory tiling.
- *SM2SM_HIST*: A histogram implementation adapted from NVIDIA's official guide [13] for using the distributed-shared-memory feature. This is our benchmark targeted to check for incurred cycle error in heavy use of the SM-to-SM net model. The thread-block-clusters are statically configured to run at the maximum size possible on the *sm90* architecture (16 thread-blocks per cluster).
- *VECTOR_ADD*: A kernel that performs element-wise addition on vectors of single-precision floating-point values. It was selected primarily because it exhibits high memory intensity, requiring only one arithmetic operation per element.

These benchmarks should equip us to make relatively general statements about the scaling behavior of our implementation of the parallelization scheme.

4.4 Metrics

The primary objective of this work is reducing the wall-time T_{wall} of GPGPU-Sim simulations, making it the main performance metric. Although factors like total CPU time, utilization, or energy efficiency may be relevant, they are considered secondary to minimizing T_{wall}.

To evaluate the effectiveness and scaling of our parallelization approach, we calculate the speedup S_N achieved using N threads:

$$S_N = \frac{T_{\text{seq}}}{T_N}$$

where T_N is the wall-time of the parallelized simulator running with N threads. To ensure functional correctness and verify that our parallelization does not alter the simulation semantics, we measure the total number of simulated GPU cycles C reported by the simulator. We compare the cycle-count from the parallel execution C_N with the reported count by the original sequential execution C_{seq} and measure the absolute percentage deviation:

$$\text{Absolute Relative Error } (\%) = \frac{|C_N - C_{\text{seq}}|}{C_{\text{seq}}} \times 100\%$$

While GPGPU-Sim reports many more statistics than just the total simulated GPU cycles, this is a good proxy for the other statistics, and also allows us to compare our results with the other works covered in Subsection 2.2.

4.5 Experimental Design

Benchmark	T_8 **Evaluation**	**Scaling Study**
MONTE_CARLO	$2^{27} = 134$M samples	$2^{24} = 16.8$M samples
SGEMM	$2^{11} = 2048 \times 2048$ matrix	$2^{10} = 1024 \times 1024$ matrix
SM2SM_HIST	$2^{23} = 8.4$M elements	$2^{21} = 2.1$M elements
VECTOR_ADD	$2^{24} = 16.8$M floats	$2^{23} = 8.4$M floats

Table 2 Benchmark input sizes for both configurations. The exponent denotes the input parameter passed to each benchmark; the resulting workload size is shown alongside.

Using the hardware, software, simulated GPU configuration, and benchmarks described above, we performed the following experiments to evaluate our parallelized implementation and the base version:

1. We measured the wall-time for the original sequential version (T_{seq}) and our implementation with 8 worker threads (T_8). We set the input-sizes of each benchmark so that (T_{seq}) was at least 60 minutes. We also record the simulated cycles so we can quantify the incurred cycle-error (as described in Subsection 4.4).
2. We measured the wall-time for the sequential version (T_{seq}) and our implementation with different worker thread counts, from 2 up-to and including 32 in increments of 2 ($T_2, T_4, \ldots, T_{32}$). Due to the increased amount of runs for this benchmark, we reduced the input-sizes slightly. This experiment allows us to observe the performance scaling characteristic of our implementation and potential impacts of the NUMA architecture of our dual-socket machine (as detailed in Subsection 4.2). We also measure the cycle-error in the same way as described above.

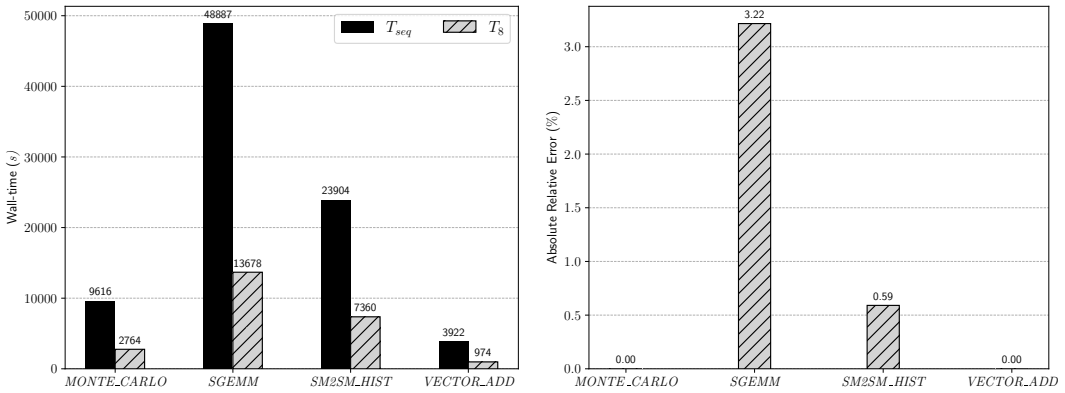
The input sizes for each benchmark can be seen in Table 2. For both sets, we average the resulting metrics over three separate runs.

5 Results and Discussion

In this section, we cover the results of our measurements as described in Subsection 4.4, beginning with the longer benchmarks for a fixed worker thread count of 8.

5.1 Comparing Sequential and Parallel implementations

We begin by examining the results for our long-running benchmarks, comparing the base version and our implementation with 8 worker threads.



(a) The measured Wall-Times comparing T_{seq} and T_8.

(b) Absolute-Relative-Cycle error (%) of the parallelized implementation.

Figure 2 Results for both the wall-time measurement (sub-figure 2a) and the error analysis (sub-figure 2b) of the long running benchmarks.

Figure 2a presents the measured wall-times for both of these (T_{seq} and T_8) across the different benchmarks. These are averaged over three separate runs, as detailed in Subsection 4.4.

Our parallelization scheme achieved a notable reduction in wall-time over all the benchmarks. The speedups ($S_8 = T_{\text{seq}}/T_8$), calculated from these average wall-times, were 3.48× for *MONTE_CARLO*, 3.57× for *SGEMM*, 3.25× for *SM2SM_HIST*, and 4.03× for *VECTOR_ADD*. The difference between benchmarks is likely explained by the different computational and data-access patterns of the different benchmarks.

Across these four benchmarks, our implementation demonstrated an average speedup of 3.58×. This speedup, while remarkable, is below an ideal linear value of 8×. This is to be expected due to factors such as inherent sequential portions of the simulation, synchronization overheads, and communication costs between threads, as discussed in Subsection 3.2. The exact scaling behavior of our implementation is discussed below (in Subsection 5.2).

5.1.1 Error discussion

While parallelization improved speed, we must quantify accuracy using the absolute relative error in total simulation cycles, as shown in Figure 2b. The results vary by benchmark complexity. *SGEMM* shows the highest error (3.22%), which we hypothesize is an inherent consequence of the parallel execution ordering altering L2-cache access patterns, but deeper investigation into this is needed for a definite confirmation.

Similarly, the minor error in *SM2SM_HIST* (0.59%) appears to stem from unsynchronized access to the intra-GPC SM-to-SM-network, potentially changing the order in which messages are sent/received. While stricter synchronization could likely mitigate these deviations, the current error margins are deemed acceptable trade-offs for rapid architectural design space exploration. *MONTE_CARLO* and *VECTOR_ADD* notably exhibit 0.0% error. We attribute this stability to their specific computational characteristics: *MONTE_CARLO* relies on intra-core shared memory and order-independent global atomics, while *VECTOR_ADD*'s single-use access pattern effectively allows it to bypass the L2-cache.

5.2 Scaling with thread-count

To assess how effectively our parallelization strategy utilizes increasing computational resources, we evaluated its scaling behavior. Figure 3 illustrates the measured speedup S_N, relative to the sequential base version, for all the benchmarks as the number of worker threads N was varied from 2 up to 32.

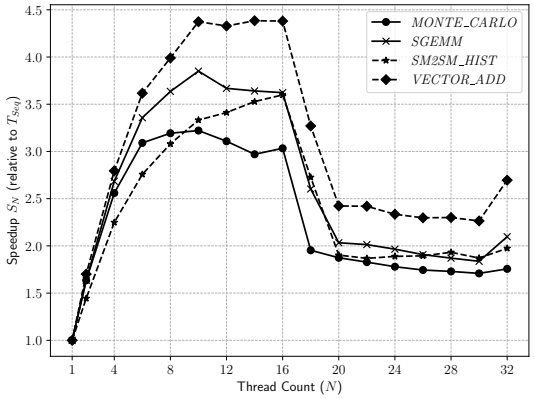


Figure 3 The measured speedup (from S_1 up to S_{32}) over the base implementation for each benchmark.

We can see in Figure 3 that our implementation scales well up to a certain amount of worker threads, but starts delivering diminishing returns around $N \geq 8$. The benchmarks peak at different thread counts: *MONTE_CARLO* and *SGEMM* both top out at $N = 10$ with 3.22× and 3.85× respectively, while *VECTOR_ADD* doesn't top out until $N = 14$ with a speedup of 4.38× and *SM2SM_HIST* only tops out at $N = 16$ with a speedup of 3.60× over the sequential base version.

We can also see a drop in speedup as the thread count increases above 16. The reason for this is quite clearly the nature of our hardware platform (described in Subsection 4.2) having 2 NUMA Nodes, each with 16 CPU cores. So what we are likely observing here is the increase in NUMA-latency as the program is spread across the two NUMA nodes (we manually restricted the program to a single NUMA node if $N \leq 16$).

The diminishing returns effect we observe around 8-10 worker threads is likely due to the increase in synchronization overhead beginning to outweigh the additional speedup of more threads. As mentioned above, we would not expect perfect speedup for any worker thread count, since there are necessary sequential sections to the simulator, and also the

mentioned synchronization and communication overhead from the parallelization scheme itself.

5.3 Comparison to Related Approaches

As discussed in Subsection 2.2, there have been previous efforts in parallelizing GPGPU-Sim. The comparison reveals distinct trade-offs between performance and accuracy across different parallelization strategies.

Table 3 Comparison of GPU Simulation Parallelization Approaches.

Implementation	Average Speedup	Min. Error	Max. Error
Work-Group Parallel [10]	3.39	0.05%	5.78%
gpusim-para1 [16]	1.91	0.00%	0.18%
gpusim-para2 [16]	2.1	0.04%	0.59%
Our Approach	3.58	0.00%	3.22%

Among prior approaches, *Work-Group Parallel* achieves the highest average speedup at 3.39×, but exhibits significantly higher error rates, with maximum errors reaching 5.78%. This suggests that while aggressive parallelization can yield substantial performance gains, it may compromise simulation fidelity.

In contrast, both *gpusim-para1* and *gpusim-para2* prioritize accuracy, maintaining maximum errors below 0.6%. However, this reduced error comes at the cost of reduced speedup, with *gpusim-para1* achieving 1.91× and *gpusim-para2* reaching 2.1× median speedup (the paper only reports median for this specific result). The *gpusim-para1* implementation demonstrates exceptional precision with some benchmarks showing zero error, indicating that highly accurate parallel simulation is achievable but requires more conservative parallelization strategies. We exclude the work of Zhao et al. [20] from this comparison, because the reported 10 – 40× speedup is not empirically measured but rather a theoretical product of their separate intra-kernel and inter-kernel results.

Comparing implementation complexity, our approach required only 10 lines of critical path changes versus Wang et al.'s [16] 170 lines for their simpler version. This minimal modification approach, combined with our balanced performance-accuracy profile, is particularly valuable for research requiring long-term maintainability and frequent updates to the code base.

6 Future Work

While this work demonstrates significant performance improvements, several promising directions exist for future research and development. One potential area involves enhancing the simulation model's fidelity. Specifically, investigating how stricter synchronization within the simulated GPCs could help reduce the cycle error, particularly for benchmarks sensitive to inter-SM communication (e.g., *SM2SM_HIST*).
This would, as previously discussed, likely come with a noticeable decrease in speedup. Additionally, testing with more complex benchmarks could help further narrow down the error source for the *SGEMM* and *SM2SM_HIST* benchmark, if so desired. Alternatively, exploring more relaxed synchronization strategies for the worker threads presents another avenue. While potentially complex to implement, allowing worker threads greater asynchronicity (as demonstrated by [10]) in completing their tasks might bring even greater speedup in

simulation time, especially on highly parallel host systems. Testing different simulated model configurations might also provide more insight into possible optimizations, especially regarding the impact that SIMT core cluster counts can have on the amount of speedup achieved through parallelization.

7 Conclusion

In this paper, we demonstrate the effectiveness of a relatively simple and non-invasive parallelization scheme for GPGPU-Sim. We benchmarked our implementation across varied GPGPU workloads, measuring both wall-time and overall simulation cycles to quantify the error introduced by parallelization. Our implementation achieves an average speedup of 3.58× over the sequential version, with a maximum relative cycle error of 3.22%, while some benchmarks display no error at all. We also analyzed the scaling behavior across different worker thread counts on a dual-node NUMA system, providing guidelines for optimal hardware/software configuration.

Our approach parallelizes the CORE-cycle execution phase by distributing work across worker threads managed by a thread pool. We based our approach on thorough profiling that identified the execution hotspots, particularly in the SIMT-Core cluster execution within the CORE-clock cycle. By reducing simulation wall-times, our parallelization enables researchers to explore significantly larger architectural design spaces within practical time constraints. This acceleration is particularly valuable for iterative design processes requiring multiple parameter sweeps. Our strategy of minimal code modifications ensures these benefits can be maintained as GPGPU-Sim evolves, while serving as a foundation for further parallelization attempts.

While achieving substantial speedups, there are some limitations. The cycle count errors in benchmarks like *SGEMM* (3.22%) likely stem from different execution orderings impacting shared components like L2-cache or the SM-to-SM network, representing the trade-off between simulation speed and determinism. Our scaling analysis reveals diminishing returns beyond 16 worker threads due to NUMA architecture constraints, suggesting that optimal performance requires careful consideration of the underlying hardware platform. The code for this paper can be found in the ClusterSim [11] code base.

References

1 Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. NVIDIA Hopper Architecture In-Depth, March 2022. URL: `https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/`.

2 Newsha Ardalani, Urmish Thakker, Aws Albarghouthi, and Karu Sankaralingam. A Static Analysis-based Cross-Architecture Performance Prediction Using Machine Learning, June 2019. arXiv:1906.07840 [cs]. URL: `http://arxiv.org/abs/1906.07840`, `doi:10.48550/arXiv.1906.07840`.

3 Simon Boehm. How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog, December 2022. URL: `https://siboehm.com/articles/22/CUDA-MMM`.

4 Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S. Lee. GPUMech: GPU Performance Modeling Technique Based on Interval Analysis. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–279, Cambridge, United Kingdom, December 2014. IEEE. URL: `http://ieeexplore.ieee.org/document/7011394/`, `doi:10.1109/MICRO.2014.59`.

5 Rodrigo Huerta and Antonio González. Parallelizing a modern gpu simulator. *arXiv preprint arXiv:2502.14691*, 2025.

6 Vishwesh Jatala, Jayvant Anantpur, and Amey Karkare. Improving GPU Performance Through Resource Sharing, June 2015. arXiv:1503.05694 [cs]. URL: http://arxiv.org/abs/1503.05694.

7 Nan Jiang, James Balfour, Daniel U. Becker, Brian Towles, William J. Dally, George Michelogiannakis, and John Kim. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 86–96, Austin, TX, USA, April 2013. IEEE. URL: http://ieeexplore.ieee.org/document/6557149/, doi:10.1109/ISPASS.2013.6557149.

8 Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, May 2020. URL: https://ieeexplore.ieee.org/document/9138922, doi:10.1109/ISCA45697.2020.00047.

9 Jounghoo Lee, Yeonan Ha, Suhyun Lee, Jinyoung Woo, Jinho Lee, Hanhwi Jang, and Youngsok Kim. GCoM: a detailed GPU core model for accurate analytical modeling of modern GPUs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 424–436, New York New York, June 2022. ACM. URL: https://dl.acm.org/doi/10.1145/3470496.3527384, doi:10.1145/3470496.3527384.

10 Sangpil Lee and Won Woo Ro. Parallel GPU architecture simulation framework exploiting work allocation unit parallelism. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 107–117, Austin, TX, USA, April 2013. IEEE. URL: http://ieeexplore.ieee.org/document/6557151/, doi:10.1109/ISPASS.2013.6557151.

11 Tim Lühnen, Jyotirman Behera, Devashree Tripathy, and Sohan Lal. Clustersim: Modeling thread block clusters in hopper gpus. In *IEEE International Symposium on Workload Characterization, IISWC*, 2025. doi:10.15480/882.15858.

12 Shuai Mu, Yandong Deng, Yubei Chen, Huaiming Li, Jianming Pan, Wenjun Zhang, and Zhihua Wang. Orchestrating Cache Management and Memory Scheduling for GPGPU Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(8):1803–1814, August 2014. URL: http://ieeexplore.ieee.org/document/6595566/, doi:10.1109/TVLSI.2013.2278025.

13 NVIDIA. CUDA C++ Programming Guide (for CUDA version 12.8.1), March 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#distributed-shared-memory.

14 Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. APRES: improving cache efficiency by exploiting load characteristics on GPUs. *ACM SIGARCH Computer Architecture News*, 44(3):191–203, October 2016. URL: https://dl.acm.org/doi/10.1145/3007787.3001158, doi:10.1145/3007787.3001158.

15 Barak Shoshany. A C++17 Thread Pool for High-Performance Scientific Computing. *SoftwareX*, 26:101687, May 2024. arXiv:2105.00613 [cs]. URL: http://arxiv.org/abs/2105.00613, doi:10.1016/j.softx.2024.101687.

16 Jun Wang and Jiaquan Gao. Parallelizing GPGPU-Sim for Faster Simulation with High Fidelity. *IEEE Design & Test*, 37(4):83–91, August 2020. URL: https://ieeexplore.ieee.org/document/9062555/, doi:10.1109/MDAT.2020.2986738.

17 Lu Wang, Magnus Jahre, Almutaz Adileho, and Lieven Eeckhout. MDM: The GPU Memory Divergence Model. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1009–1021, Athens, Greece, October 2020. IEEE. URL: https://ieeexplore.ieee.org/document/9251853/, doi:10.1109/MICRO50266.2020.00085.

18 Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, 31(2):50–59, 2011.

19 Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st*

International Symposium on High Performance Computer Architecture (HPCA), pages 564–576, Burlingame, CA, USA, February 2015. IEEE. URL: http://ieeexplore.ieee.org/document/7056063/, doi:10.1109/HPCA.2015.7056063.

20 Xia Zhao, Sheng Ma, Wei Chen, and Zhiying Wang. Exploiting parallelism in the simulation of general purpose graphics processing unit program. *Journal of Shanghai Jiaotong University (Science)*, 21(3):280–288, June 2016. URL: http://link.springer.com/10.1007/s12204-016-1723-2, doi:10.1007/s12204-016-1723-2.