GpuTejas: A Parallel Simulator for GPU Architectures

Geetika Malhotra, Seep Goel, and Smruti R. Sarangi
Department of Computer Science,
Indian Institute of Technology,
Hauz Khas, New Delhi, India
E-mail: {mcs122798, mcs132582, srsarangi}@cse.iitd.ac.in

***Abstract*—In this paper, we introduce a new Java-based parallel GPGPU simulator, *GpuTejas*. *GpuTejas* is a fast trace driven simulator, which uses relaxed synchronization, and non-blocking data structures to derive its speedups. Secondly, it introduces a novel scheduling and partitioning scheme for parallelizing a GPU simulator. We evaluate the performance of our simulator with a set of Rodinia benchmarks. We demonstrate a mean speedup of 17.33x with 64 threads over sequential execution, and a speedup of 429X over the widely used simulator GPGPU-Sim. We validated our timing and simulation model by comparing our results with a native system (NVIDIA Tesla M2070). As compared to the sequential version of *GpuTejas*, the parallel version has an error limited to $<$7.67% for our suite of benchmarks, which is similar to the numbers reported by competing parallel simulators.**

***Keywords*-GPU; Simulator; Timing model; Cycle-level; Parallel Architectural Simulation;Nvidia;Tesla;**

I. Introduction

GPUs are increasingly being regarded as first class citizens in the world of processors. Before NVIDIA released the CUDA API, GPUs were exclusively being used for computer graphics based applications. However, off late GPUs have become general purpose and it is possible to use them for a variety of numerical and scientific applications, bio-informatics, and even data-analytics based applications [1]. Consequently, GPUs are nowadays termed as GPGPUs (general purpose GPUs). GPUs are not restricted to add-on cards any more. Intel and AMD [2], [3], [4], [5] have already integrated them on chip, and now it is possible for general purpose users to write programs that run on the GPU.

In response to this market trend, both computer architects as well as processor manufacturers are increasingly focusing on the hardware and software aspects of designing GPUs. Till a few years ago, GPUs were being considered primarily as graphics engines that additionally support numerical computations. However, perceptions are changing, and nowadays GPUs are being designed exclusively for high performance computing applications. In fact, 2 out of the top 10 fastest supercomputers (Titan, Piz Daint, Nov'13 list) are built with GPUs. Consequently, it is necessary for students, researchers, and professionals to understand the design and research issues in building fast and power efficient GPUs.

Traditionally, for studying, and designing new architectures, an *architecture simulator* has been the main workhorse in the computer architecture community. Hence, for working with GPUs, the computer architecture community uses several popular GPU simulators that perform both functional as well as timing simulation. Some of the notable simulators in this space are GPGPU-Sim [6], Ocelot [7], Barra [8], and Attila [9]. Most of these simulators have been validated with native systems, and have proved to be very useful for studying the features of GPU based workloads, and in proposing both the software and hardware enhancements to the GPU design and associated software stack. Along with exclusive GPU simulators, the computer architecture community has developed simulators that can simulate both the CPU and GPU [10], [11], [12], [13], [14]. These are useful for applications that have computations in the GPU, as well as in the CPU. These simulators are also used to study the interaction between the CPU, and GPU.

We believe that there is a need for a new and scalable simulator in this space. This is because modern GPUs have hundreds to thousands of processing elements. For example, NVIDIAs Kepler GPU contains more than 2500 processing elements, and it can process thousands of threads in parallel [4]. AMDs Radeon HD 6000 [5] series of GPUs contain more than 1000 processing elements on a single GPU die. Sequential GPGPU architecture simulators suffer from performance issues while simulating these massively parallel GPUs. With the exponential rise in the processing elements in a GPU, there is a consensus view that *parallel simulators* are required since the slow simulation speed of sequential simulators [6] proves to be prohibitive for teaching, and designing GPUs. Additionally, given the fact that most of the time we run embarrassingly parallel benchmarks on a GPU, the interaction between threads in the memory system is not significant. We believe that this feature of GPU benchmarks should be leveraged to design, fast parallel GPU simulators. To the best of our knowledge, the proposal by Ro et. al. [15] is the only work that focuses on designing parallel GPU timing simulators. They propose a simulation technique where they minimize the synchronization overhead by simulating the parallel components of the GPU architecture independently using multiple simulation threads. They reported a speedup of

4.15x using an 8-core system.

In this paper, we present the design of a soon to be released Java-based, open source, GPGPU simulator called *GpuTejas* that scales to at least 64 threads. It is a part of the broad Tejas simulation framework [16] that can simulate complex multicore CPUs (sequentially and also in parallel). The salient features of this simulator is that it uses novel data structures for obtaining speedups. In specific, we use phasers (advanced barriers with support for phases), and lock free parallel ports (non-blocking structures for managing contention), for designing our simulator. We show that by using these novel concurrent data structures, we can design a highly parallel simulator that can simulate the timing aspect of GPUs. It is important to note that we do not perform functional simulation, i.e., execute the behavior of the instructions. We use the Ocelot framework [7] to execute CUDA based programs, and then we simulate the collected traces to estimate the timing.

Moreover, our simulator is written in Java. We chose Java for satisfying the dual objectives of having an efficient research tool that is easy to design, maintain, and extend, and also to create an educational tool that does not require a lot of time to learn. We collected informal feedback from students in Indian educational institutions. The students indicated that they prefer Java over C/C++ because it is easier to write and debug programs. We noted some additional advantages such as the built in support for multiple threads, availability of lock-free data structures, extensive library support, and support for garbage collection. Secondly, the quintessential argument that Java is significantly slower than C++ is debatable [17], [18], [19]. With modern JIT compiler based JVMs, the performance difference has narrowed to less than 20% [18], and sometimes researchers have reported speedups with heavily object oriented benchmarks [18].

Let us succinctly, list our contributions.

1) We design a GPGPU compute pipeline, and a configurable cache/memory model for GPGPUs.
2) We propose a configurable block scheduling algorithm, which evenly schedules the blocks across SMs.
3) We design a fast transfer medium using files to transfer traces between Ocelot, and our Java based simulator threads.
4) The pipelines operate in parallel. The rendezvous points between the threads are in the memory system. The crux of our technique for implementing a parallel memory system is to use a novel concurrent data structure called a *parallel port*. The parallel port helps us to model the contention in the memory system accurately, and avoid costly clock synchronization operations between the threads.
5) We propose a simulation algorithm that uses *phasers*(see Section IV-B) to dynamically reduce the time that threads spend in synchronization. This ensures that the time wasted in waiting for a barrier is significantly reduced.
6) For improving the performance of Java programs, we propose a host of Java specific optimizations namely selection of appropriate data structures, fine grained locking mechanisms, and pooling techniques.
7) We demonstrate a mean, 17.33X speedup with 64 threads from our sequential simulation.
8) We demonstrate a 429X speedup with respect to the widely used sequential simulator, GPGPU-Sim [6].

II. Related work

A. Sequential GPU Simulators

Attila [9] was one of the earliest software based simulators for graphics programs. It is a detailed execution-driven cycle accurate GPU simulator. It is highly configurable, which makes it a generic simulator that is independent of the GPU manufacturer. It simulates GPU pipelines by collecting dynamic traces from an OpenGL application. It does not support CUDA and the GPGPU framework.

The widely used simulator, GPGPU-Sim [6], is a detailed general-purpose GPU (GPGPU) simulator that supports functional and cycle-level timing simulation for NVIDIA GPUs. It models GPGPU compute units (CUs) called streaming multiprocessors by NVIDIA and the GPU memory system. It has been extended to include PTX as an instruction set. It can run applications without source code modifications, but requires access to the source code.

Gem5-GPU [12] is a heterogeneous full-system CPU-GPU simulator, tightly integrated with the Gem5 [20] simulation infrastructure. It can simulate programs meant for the CPU, and programs meant for the GPU simultaneously. It can be used to study the behavior of benchmarks that jointly use the CPU and GPU. The MacSim [11] simulator provides models of both CPU and GPU cores. MacSim is a trace driven heterogeneous architecture simulator that can simulate x86 and PTX traces. It can simulate both the ISAs simultaneously.

Multi2Sim [10] is a simulation framework for heterogeneous computing, and it includes models for superscalar, multithreaded, multicore, and graphics processors. It provides cycle-level simulation for the AMD Evergreen family of GPU architectures. It provides a parallel simulation framework for CPU architecture simulation; however, its GPU simulator is sequential. FusionSim [14] is an open-source modeling framework that is capable of cycle-accurate simulation of a complete x86-based computer system with a CPU and a GPU. It models an x86 out-of-order CPU and a CUDA-capable GPU that operate concurrently.

Ocelot [7] is a dynamic compilation and emulation framework that executes CUDA binaries without modifying the source code. It provides a binary translator, which is capable of translating PTX into x86 and other ISAs using the LLVM compiler. It does not have a timing model. *GpuTejas* uses Ocelot for functional simulation and trace generation.

B. Parallel GPU Simulators

Barra-Sim [8] is a parallel functional GPGPU emulator for the NVIDIA Tesla architecture. It emulates the CUDA driver and maps all the function calls destined for the GPU to itself. It decodes the ISA and emulates the instructions. It does not have a timing model.

Ro et. al. [15] parallelized GPGPU-Sim. They divide the functional units into shared and parallel components. They propose two synchronization algorithms. The first method, performs a cycle by cycle synchronization across the threads, which makes it very slow. In the second algorithm, the threads synchronize after the end of a work-group and the parallel components are simulated without any synchronization overhead. They demonstrate a speedup of 4.15x over the sequential GPGPU-Sim on a 8-core system. On the other hand, *GpuTejas* is 429.89x faster than GPGPU-Sim on an average(see Section V-G). Additionally, *GpuTejas* has a different model of parallelization. We parallelize at the level of blocks. To the best of our knowledge, [15] is the only work in the area of parallel GPU timing simulators.

III. System Architecture

A. Overview

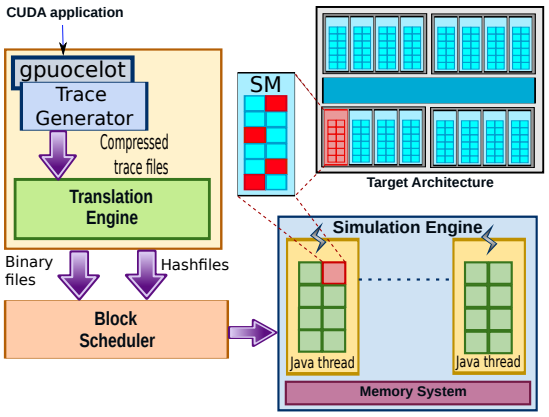

Figure 1. System Overview of *GpuTejas*

In this paper, we show how *GpuTejas* can simulate the NVIDIA Tesla [2] architecture. Note that *GpuTejas* is a highly configurable simulator and it can seamlessly simulate advanced GPU architectures such as Tesla, Fermi, and Kepler. The overview of the design of *GpuTejas* is shown in Figure 1. A CUDA executable is given as an input to the simulator. We first use an instrumented version of Ocelot [7] to run the executable, and then generate a set of trace files. These trace files primarily contain information regarding the instructions being executed, including the instruction type, instruction pointer (IP), and the corresponding PTX instruction. Note that (PTX) (Parallel Thread Execution) is an intermediate device language, which has to be converted into device specific binary code for native execution. The trace additionally contains the list of memory addresses (if it is a load/store instruction) accessed by an instruction. We also embed some metadata along with every trace file. The metadata lists the number of kernels, the grid sizes in each kernel, and the number of blocks present in each kernel.

These traces undergo another pass to reduce the size of the files. It was observed that all the blocks of a kernel contain the same set of instructions. Thus, redundant information was getting saved in the trace files for every block. We stored the information regarding these instructions separately in a hashfile. Our post-processing scripts subsequently generate new trace files that only contain the instruction pointers of instructions. These instruction pointers map to the actual instructions in the hashfile. Note that, these instructions are translated to specific instruction classes before storing them into the hashfiles. This further reduces the space occupied by the traces being generated. The trace files generated in the second pass are read by the waiting Java simulation threads. The Java based simulator threads model the GPU, and the memory system. They are responsible for generating the timing information, and detailed execution statistics for each unit in the GPU, and the memory system.

The NVIDIA Tesla GPU contains a set of TPCs (Texture Processing Clusters), where each TPC contains a texture cache, and two SMs (Streaming Multiprocessors). Each SM contains 8 cores (Stream Processors (SPs)), instruction and constant caches, shared memory and two special function units (can perform integer, FP, and transcendental operations). A typical CUDA computation is divided into a set of kernels (function calls to the GPU). Each kernel conceptually consists of a large set of computations. The computations are arranged as a grid of blocks, where each block contains a set of threads. The NVIDIA GPU defines the notion of a *warp*, which is a set of threads (typically in the same block) that are supposed to execute in a SIMD fashion. We simulate warps, blocks, grids, and kernels in *GpuTejas*.

We parallelize the simulation by allocating a set of SMs to each thread. In our simulation, each SM has its own local clock for maintaining the timing of instructions. Memory instructions are passed to the memory system, which supports private SM caches, instruction caches, constant caches, shared memory, local memory and global memory. The important point to note here is that different Java threads do not operate in lock step. This can potentially create issues in the memory system, where we need to model both causality (load-store) order and contention. We shall present novel solutions to model both of these issues in Section IV-A.

B. Trace Generation using Ocelot

Ocelot [7] is a dynamic compilation framework that replaces the CUDA runtime API library, which links with CUDA applications. Ocelot traps the CUDA library calls,

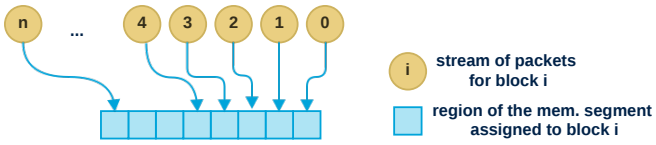

Figure 2. Structure of the shared memory segment

and simulates their execution. Note that it simply emulates the functionality of the GPU. It does not model the timing and architectural aspects of the GPU, or its components. We instrumented Ocelot to catch various events and trigger custom functions, and then generate packets of information. Each packet contains the *instruction type*, *instruction pointer*, and the corresponding *PTX* instruction. Additionally, a memory-based instruction contains a *memory addresses* vector. Other events considered for handshaking purposes include the $kernel_{start}$, and $kernel_{end}$ events. These packets are used to simulate the sequential ordering of kernels among Java simulator threads.

C. Transfer Mechanism

Packets generated by the *trace generator* (see Section III-B) need to be subsequently transferred to the Java simulation threads. We conducted an experiment to evaluate different transfer mechanisms. We evaluated various mechanisms such as UNIX sockets, shared memory, and files. We measured the time taken by each mechanism to transfer 5 GB of data from Ocelot to Java threads on an Intel Core i7 desktop(3.1 GHz processor, 4 GB RAM, Ubuntu Linux 12.04).

1) **Sockets**: In the case of sockets, we use a dedicated TCP socket between the two processes. We transfer a stream of packets, each of size 1.5KB (Ethernet MTU). We observed that the communication with sockets is the slowest of all the mechanisms.(See Figure 4).
2) **Shared Memory**: We implemented a shared memory model similar to that used by ParTejas [16]. We used a single shared memory segment divided into n contiguous regions for n blocks. Each region contains a header, and a circular queue. Packets in a block are written to its corresponding slot, and read by the Java threads simultaneously (see Figure 2). For multi-kernel applications, the memory segment is flushed whenever a new kernel arrives, and after all the packets of the previous kernel have been read. Note that this is an online mechanism where packets need to be consumed by simulator threads in an in-vivo manner.
   We observed several limitations of this mechanism. Firstly, Ocelot being sequential in nature generates the dynamic instruction trace block-by-block for each kernel. This makes the producer process very slow, and makes it a sequential bottleneck. Having a slow feeder limits the total simulation speed and scalability as well. Figure 3 shows the execution speed of various CUDA executables running with the base version of Ocelot, and with the instrumented version of Ocelot. The results are normalized to the CUDA emulation execution time (on a native system) for each executable. We observe a mean slowdown of 9.5X, and 286.56X for the base and instrumented versions respectively.

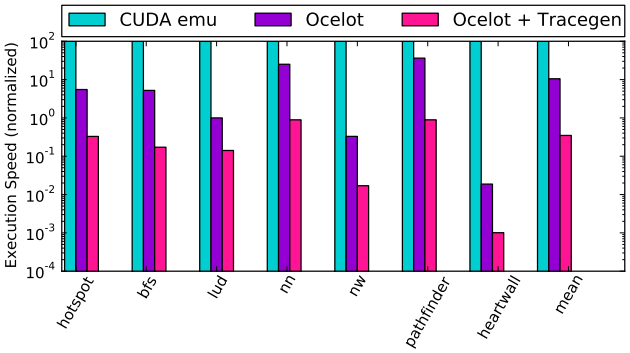

Figure 3. Comparison of execution speed

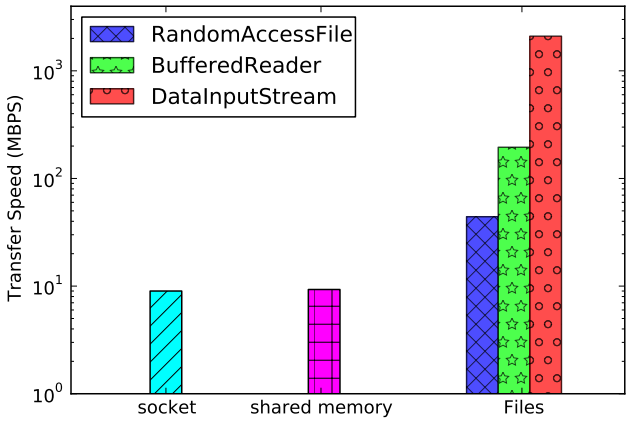

Figure 4. Comparison of various transfer mechanisms

3) **Compressed Trace Files**: Instead of online mechanisms, we tried an offline mechanism that decouples emulation and simulation. The basic approach is to write the traces to a set of files, post-process them, and use the files for detailed timing simulation. We experimented with various *java.io* packages. We conducted experiments with the classes: *RandomAccessFile*, *BufferedReader*, and *DataInputStream*. *RandomAccessFile* uses a minimal amount of in-memory caching; hence, it proved to be a very slow method. Subsequently, we experimented with the *BufferedReader* class. The only drawback with the *BufferedReader* class is that it does not provide any provision to read objects directly. We found *DataIn-*

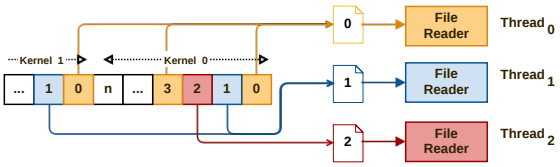

Figure 5. Block distribution with compressed trace files

putStream to be the fastest and most versatile. We additionally used Java's built in features to couple it with a buffering mechanism. We read data in chunks of 64KB, which reduced the IO overhead considerably. The results are shown in Figure 4.

D. Post Processing the Traces

Generating the traces is a two pass process. In the first pass, we create a set of trace files. If we decide to have n_{sim} simulator threads, then we create n_{sim} trace files, one for each thread. Packets from the trace generator are written to these trace files. Blocks of a kernel are distributed uniformly across the files. If we have n_b blocks in a kernel, then n_b/n_{sim} blocks are simulated by each Java thread. Packets of $block_i$ of a kernel are written in the $trace\_file_{i\%n_{sim}}$. This method of partitioning is shown in Figure 5.

In the second pass, these files are further compressed to reduce the space before being fed to the simulation engine. We assign a RISC Instruction class (similar to GPU device specific RISC instructions) to each instruction that is read from the trace file. The assignment is done on the basis of the opcode of the PTX instruction. This mapping is similar to that used by MacSim [11]. An important point to note here is that all the blocks in the kernel are executing the same set of instructions and consequently, we do not need to repeat the same translation procedure for all the blocks of a kernel. We define a mapping of *instruction pointer* $\rightarrow$ *instruction class*, and store it in a Hashfile, $hashfile\_kernel_{num}$. Next, we create a new set of binary trace files containing only instruction pointers and memory addresses (in case of memory instructions) using the *DataOutputStream* (writer counterpart of *DataInputStream*). These files are then passed to the simulation threads. The total size of these traces ranges from 250KB for a small benchmark such as nn to 2.2GB for a large benchmark such as $heartwall$. On an average, the size is around 550MB for the entire benchmark.

In *GpuTejas*, each Java thread independently reads all the blocks assigned to it for simulation from its corresponding compressed binary trace files. All the Java threads are physically mapped to separate cores on the host machine. Threads use the hashfile of the currently executing kernel to get the instruction class from the instruction pointer. After that, the instruction objects are instantiated and a stream of instructions is formed and fed to the GPGPU pipeline. These instructions get executed after getting scheduled over the SMs. Section III-E discusses the scheduling of the blocks over the SMs of the simulated GPU architecture.

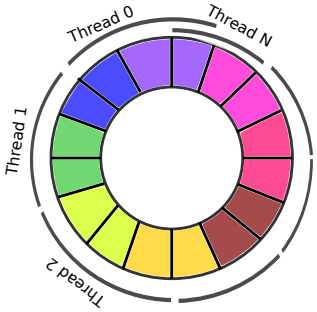

Figure 6. Block scheduling on SMs (conceptual view)

E. Block Allocation Mechanism

We propose a two layer scheduling algorithm for the execution of the blocks over the simulated SMs. In the first layer, we statically assign an SM to a block. This static binding is performed before the simulation begins (simulates the GigaThread engine in NVIDIA GPUs). In effect, each simulation thread gets a set of SMs for the execution of the blocks assigned to it. Figure 6 illustrates the static allocation of SMs. Here, each color in the ring denotes a part of an SM (split equally between Java threads), and the number of slots of each color denotes the number of warps that can be scheduled by the SM in each cycle (because Tesla can schedule at most two warps on an SM). Since we map blocks to threads, each SM may get requests from multiple simulation threads. We shall resolve such conflicts in the second layer of the scheduling algorithm. The algorithm discussed up till now is shown in Algorithm 1.

In the second layer, we dynamically assign a set of processing elements from the assigned SM to an active warp of every executing block. Algorithm 2 illustrates the dynamic assignment of processing elements (SPs, SFUs, load/store units, and double precision units). Note that in our scheme it is possible to have an SM shared between two Java threads. In this case, it is necessary to synchronize their accesses. We use a simple CAS (compare-and-set) based mechanism to partition the issue slots between the threads. This models the dynamic contention that happens in SMs when we have multiple warps competing for issue slots.

Initially, all the warps to be scheduled are placed inside a *ready queue* of each simulation thread. During the simulation, warps are dequeued from the ready queue. The dequeued warp tries to get a set of functional units from the assigned SM of that block, using an atomic *requestFU()* operation. If the resources are available, then the operation returns true and the warp gets enqueued to the *execute queue*. Subsequently, the *oneCycleOperation()* method is invoked for all the warps present inside the execute queue.

In the method, *oneCycleOperation()*, the instruction is sent to the pipelines of the functional units for execution. We can assume that different units in an SM might have a different number of stages. The memory requests are handled using events in *GpuTejas*. *GpuTejas* supports a flexible cache model, and incorporates the constant, instruction, and shared caches in an SM. If an access misses in the caches, then it is sent to the DRAM based global memory.

After the execution, each warp can be in an active, inactive, or executed state. Each active warp is enqueued back to the ready queue, while inactive warps (wrong path of a branch) are enqueued to the *inactive queue*. Inactive warps get enqueued back to the ready queue, when they get into the active state again. The process goes on till all the queues get empty. As we observe in Algorithm 2, for each call to *oneCycleOperation*, FUs are assigned to the active warps of blocks in a round-robin fashion. These FUs are released after the cycle gets completed so that the waiting active warps can get scheduled in the next cycle. Being a multi-threaded environment, the request, and the release procedures are atomic (implemented using compare-and-set instructions).

Algorithm 1: Static Allocation of SMs to the Application Blocks

```
1 partition (activeThreads , blocksPerThread,
  noOfSms,noOfBlocksPerSM )
2 totalBlocks ← activeThreads * blocksPerThread
3 Declare an array assignedSM[totalBlocks] to hold the allocated
  SM number
4 currResource ← 0
5 for i ← 0;i < totalBlocks;i ← i+1 do
6 |   assignedSM[i] ← currResource / noOfBlocksPerSM
7 |   currResource ← ( currResource + 1 ) % (noOfSms *
  |   noOfBlocksPerSM)
8 end
```

Algorithm 2: Dynamic Allocation of FUs to the active warps

```
1  allocate (warpSize , blocksPerThread, noOfSms,
   noOfBlocksPerSM, threadsPerBlock,
   warp[blocksPerThread][threadsPerBlock/warpSize])
2  for i ← 1;i ≤ blocksPerThread; i ← i + 1 do
3  |   for j ← 1;j ≤ threadsPerBlock/warpSize; j ← j + 1 do
4  |   |   ReadyQ.enqueue(warp[i][j])
5  |   end
6  end
7  while true do
8  |   while ReadyQ.notEmpty() do
9  |   |   warpSelected ← ReadyQ.dequeue()
10 |   |   if warpSelected.requestFU() then
11 |   |   |   ExecuteQ.enqueue(warpSelected)
12 |   |   end
13 |   end
14 |   while ExecuteQ.notEmpty() do
15 |   |   warpSelected ← ExecuteQ.dequeue()
16 |   |   oneCycleOperation(warpSelected)
17 |   |   warpSelected.releaseFU()
18 |   |   if warpSelected.active then
19 |   |   |   ReadyQ.enqueue(warpSelected)
20 |   |   end
21 |   |   else if warpSelected.inActive then
22 |   |   |   InActiveQ.enqueue(warpSelected)
23 |   |   end
24 |   end
25 |   while InActiveQ.notEmpty() do
26 |   |   warpSelected ← InActiveQ.dequeue()
27 |   |   if warpSelected.active then
28 |   |   |   ReadyQ.enqueue(warpSelected)
29 |   |   end
30 |   |   else
31 |   |   |   InActiveQ.enqueue(warpSelected)
32 |   |   end
33 |   end
34 |   if ReadyQ.empty() and InActiveQ.empty() and
   |   ExecuteQ.empty() then
35 |   |   BREAK
36 |   end
37 end
```

IV. Parallel Simulation

A. Parallel Memory System and Parallel Ports

This section focuses on the methods employed to model contention in *GpuTejas*. Cache and memory structures are shared across threads. In *GpuTejas*, we use a parallel port for every cache. A parallel port was originally used in the ParTejas parallel architectural simulator [16], which was based on non-blocking slot schedulers proposed by Aggarwal and Sarangi [21].

The basic idea of a parallel port is as follows. Let us consider a reservation matrix of slots as shown in Figure 7. The number of rows is equal to the number of ports in the structure, and the columns correspond to cycles. Initially, the status of all the slots is free. Let us now assume that a request arrives at slot number 10, and it requires 5 cycles. If the slots are free, then we can schedule the request from cycles 11-15 on any port. Gradually, it will become difficult to schedule requests because slots will become busy. The non-blocking slot scheduler ensures that we get a set of contiguous slots within a bounded number of time steps. Furthermore, this data structure guarantees sequential semantics and is linearizable (the request appears to have been performed instantaneously). The overheads of having this structure are minimal, and it has been used successfully in a parallel multicore simulator [16]. This structure allows each thread to use its local clock while accessing elements in the memory system. Each element in the memory system maintains a reservation matrix and assigns a set of slots to every request. This structure allows us to effectively model contention because while scheduling requests we can never have more requests per cycle than the number of rows (ports).

B. Barriers and Phasers for Synchronization

The kernels are scheduled one after the other sequentially in the case of the NVIDIA Tesla GPU [2]. In *GpuTejas*, all the n_{sim} simulator threads simulate the same kernel

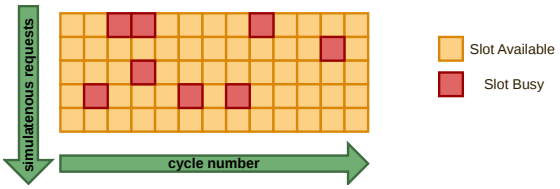

Figure 7. Structure of a parallel port

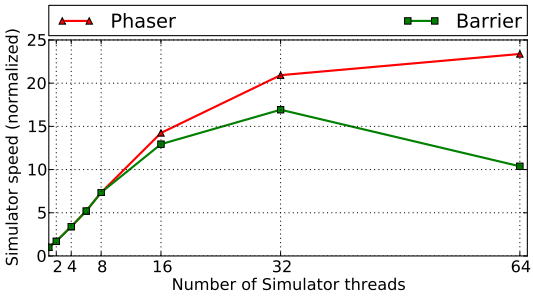

Figure 8. Performance benefit with phasers

at a time. The threads synchronize at the end of a kernel with a barrier. Next, we require synchronization during the simulation of a kernel to ensure that threads do not indefinitely deviate from each other. We maintain a local clock for each SM. Threads synchronize at the end of every epoch (1000 cycles). Sadly, when there are a lot of threads, there can be a substantial amount of imbalance between them. Barriers will introduce a lot of idleness in this case. Hence, we propose to replace barriers with *phasers* that allow fast threads to move ahead and do work that is unrelated to the barrier.

A phaser has two distinct points. After a thread reaches the first point, it informs the rest of the threads. No thread can cross the second point, unless all the threads have reached the first point. We use phasers to consider two epochs at a time. This allows us to finish some work of the second epoch before all the threads have finished the first epoch. We initialize the number of columns of each parallel port to 1000 (epoch size), and consider two ports at a time (when the implementation uses phasers). We conducted a small experiment to quantify the performance gain using phasers over barriers. Figure 8 shows the results for a representative Rodinia benchmark ($pathfinder$). We observe slowdowns of more than 100% for 64 threads.

C. Java Specific Optimizations

1) Memory Specific Optimizations: We tried to reduce false sharing, used $ThreadLocal$ variables (to reduce the number of global variables), and read the trace files in chunks of 64 KB (take advantage of block transfer and DMA mechanisms in the system). We subsequently observed that frequent calls to the garbage collector degraded the overall performance. The garbage collector should be used only for those data structures that are not initialized very frequently. In our case, classes such as the *Instruction* class get instantiated billions of times in a single simulation. We thus wrote a custom pooling mechanism that maintains a pool of $Instruction$ instances. Moreover, we limit the overall heap size of *GpuTejas* to avoid any memory leaks using the *-Xmx* JVM parameter. Lastly, we try to reuse the same object as much as possible. For example, instead of declaring multiple instances of the $Event$ class, we reuse the same object for different types of events for a complex operation such as a request to global memory.

2) Concurrency Specific Optimizations: Read-only structures shared across threads were declared as *static* and *final* (only one read-only copy). The Hashfile that stores the instruction information (see Section III-D) is implemented using HashMaps rather than HashTables, because HashMaps are lock free structures, whereas HashTables use locks.

V. Results

A. Experimental Setup

We evaluated *GpuTejas* on a four socket, 64 bit, Dell PowerEdge R820 server. It had four 8 core 2.20GHz Intel(R) Xeon(R) cpus (with hyper-threading enabled), 16 MB L2 cache, and 64 GB of main memory. We thus had a maximum of 64 threads visible to software. This server runs Ubuntu Linux 12.10 using the generic 3.5.0-17 kernel. All our code is written in Java 6 using Sun OpenJDK 1.6.0_27 with the latest patches. We use CUDA toolkit v4.0.17, Ocelot v2.1, Flex v2.5.35, Bison v2.4.1, LLVM v3.1, and Boost v1.4.6. All the native executions, were performed on a system with an NVIDIA Tesla M2070 card. We used the $nvprof$ (CUDA 5.5) profiler to get the timing statistics of only the GPU part of the computation.

Table I shows the details of the simulated GPU architecture. For all our experiments, we use a 64-thread simulation system, unless otherwise stated. We simulate 7 benchmarks from the Rodinia benchmark suite v2.1 [22] ($hotspot$, bfs, lud, nn, nw, $pathfinder$, and $heartwall$). We use an epoch size of 1000 cycles for all our simulations.

B. Performance Results

Our performance results are shown in Figure 12. For all our experiments, we perform simulations with 1, 8, 16, 32 and 64 threads respectively. When we show results with n cores, we instantiate n Java threads.

We observe a mean speedup of 17.33x with 64 threads over sequential simulation. $heartwall$, $pathfinder$, nw, bfs, and $hotspot$ show a speedup of 6-7x for 8 threads, 10-14x for 16 threads, 13-21x for 32 threads, and 16-26x for 64 threads. An exceptional behavior is observed with lud. It scales well till 32 threads and gives a speedup of 12x. For the case of 64 threads, the speedup tapers off. The

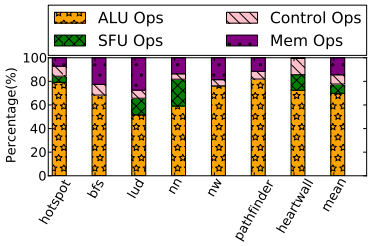

Figure 9. Instruction classification

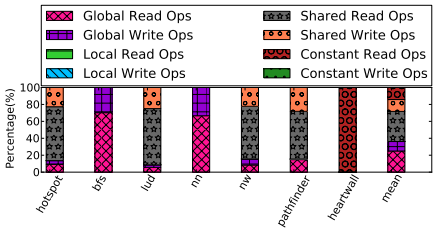

Figure 10. Classification of memory instructions

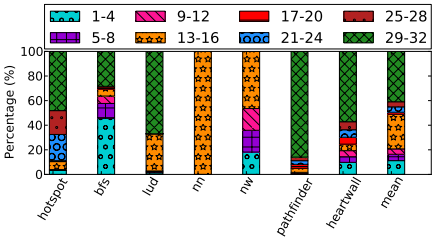

Figure 11. Warp occupancy

Simulation Parameters	
Number of Simulator Threads	1/8/16/32/64
Block Scheduling Policy	Static
Warp Scheduling Policy	Round Robin
System Parameters	
Number of TPCs	8
Number of SMs per TPC	2
Number of SPs per SM	8
Number of Blocks per SM	4
Warp Size	32
Number of Warp Schedulers per SM	2
Shared Memory Size per SM	16 KB
Constant Cache Size per SM	8KB (2 way set associative, 64 Byte line)
Instruction Cache per SM	4 KB (4 way set associative, 32 Byte line)
Latency Parameters	
Main memory	100 cycles
iCache /Constant Cache	1/1 cycle
Integer ALU / MUL / DIV operations	1/2/4 cycles
Float ALU / MUL / DIV operations	1/4/8 cycles
BRANCH / CALL / RETURN / EXIT operations	1/1/1/1 cycle

Table I
PARAMETERS OF THE SIMULATED ARCHITECTURE

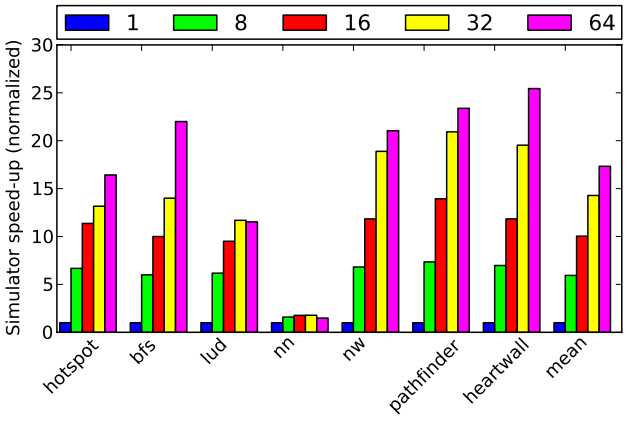

Figure 12. Simulation speed

kernels in *lud* have 30 blocks on an average. When *lud* is simulated using 64 simulation threads, half of the threads remain idle. Hence, we do not observe a speedup after 32 threads. *nn* does not scale with multiple threads. Though, there are 2673 blocks in its kernel, but there are only a few instructions in each block. The simulation threads do not do much work while simulating the blocks. Most of the time gets spent on setting up the computation.

C. Analysis of the Speedups

Table II shows the break up of the total time in terms of the average (across all threads) time taken to wait for the barriers and phasers, the time taken to access memory structures (caches, DRAM), the time taken in scheduling and the time the pipeline takes to execute. We instrumented the code with calls to read the current time for measuring these parameters.

We observed that on an average 14.55% of time is spent in scheduling the warps over FUs, 12.31% of the time is spent in simulating the pipelines, 23.72% of the time is spent in the memory system, 28.6% of the time is spent waiting for barriers at the end of a kernel, and 20.8% of the time is spent at phasers used for periodic synchronization.

D. Application Characterization using GpuTejas

Figure 9 classifies the dynamic instruction mix of applications. As we can see from the dynamic instruction mix of the simulated applications, on an average there are 69.84% of integer instructions, 7.89% of floating-point instructions, 7.81% of control instructions, and 14.46% of memory instructions. We can observe that a majority of instructions are integer or floating-point operations. *heartwall* has 85.66% of integer/floating-point operations. Similarly, Figure 10 gives a breakdown of all the memory operations in the benchmarks. On an average, there are 36.19% global memory operations, 0% local memory operations, 49.57% shared memory operations, and 14.24% constant memory operations. Global and local memory requests go to the main memory for data. On the other hand, shared and constant memory requests go to their respective caches first.

Figure 11 shows the warp occupancies over the entire runtime of the benchmarks. Warp occupancy is defined as the

Application	KIPS		#Instructions	Sched. Overhead (%)	Pipe-line (%)	Mem. Sys (%)	Phaser wait(%)	Barrier Wait (%)
	16	64						
hotspot	495.70	716.34	601,355	25.28	23.44	43.06	0.40	7.82
bfs	724.34	1593.54	4,181,380	27.43	8.24	0	37.56	26.77
lud	360.43	437.24	635,968	1.35	3.50	16.17	16.13	62.85
nn	159.80	134.16	72,172	32.48	13.96	0	3.30	50.26
nw	1218.88	2166.29	20,422,464	3.70	12.12	27.94	17.94	38.30
pathfinder	609.57	1022.46	2,934,570	9.08	23.92	46.29	15.56	5.15
heartwall	374.14	804.04	121,606,107	2.56	0.99	32.61	54.75	9.09
mean	563.26	982.01	21,493,430.86	14.55	12.32	23.73	20.80	28.60

Table II
SIMULATOR SPEED, AND BREAK-UP OF THE TIME TAKEN FOR 64 THREADS

number of active threads in an issued instruction. This metric can be seen as a measure of how much GPU throughput is wasted due to unfilled warps. It is often believed that an intensive control flow results into high branch divergence. However, it depends more on whether or not all threads in a warp branch in the same direction. *heartwall* has the largest number of control instructions, but it has full warp occupancy 57.17% of time.

E. Framework Validation

In this section, we discuss the validation of *GpuTejas* with a native machine(see Section V-A). First, we recorded the total kernel execution time for each benchmark using the NVIDIA CUDA profiler, *nvprof* [23]. This ignores the time to transfer data from the CPU to the GPU and vice versa, and the computation on the CPU.

Figure 13 shows the comparison between simulated execution time with *GpuTejas*, and the total kernel execution time in the native system's GPU. The simulation error is shown in the yellow boxes. We believe that the error is due to the lack of knowledge about the latency of instructions, associativity of the memory structures, and the type of the interconnection network used in the native system. These parameters have been obtained from other open source simulators [11], and they might vary from the exact values.

Note that the relative performance of the benchmarks in *GpuTejas* and the native system follow the same trend. Table II shows the total number of instructions in each benchmark. Recall that the percentage of memory instructions was shown in Figure 9. We can conclude that an increase in total instructions, as well as an increase in the number of memory instructions has similar effects for native and simulated execution. The authors of Multi2sim [10] also validated their simulation on the basis of observed trends. Their mean simulation error was 20%, whereas *GpuTejas* has a mean simulation error of 15%.

F. Simulation Errors and Deviation

Figure 14 presents the average deviation as compared to the sequential execution. We observe that the deviation in mean IPC varies from 3.4 to 7.67% and the mean cache

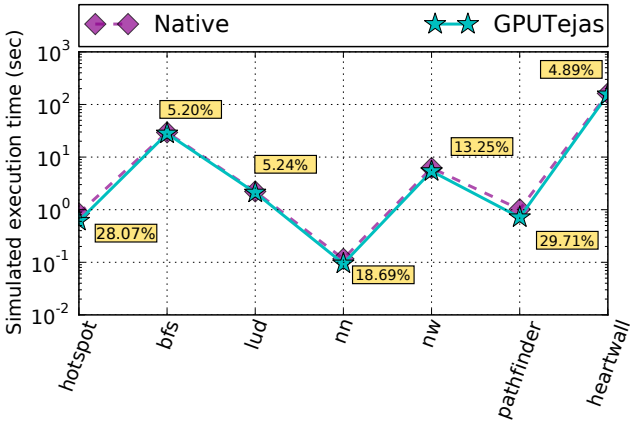

Figure 13. Validating *GpuTejas* with a native system

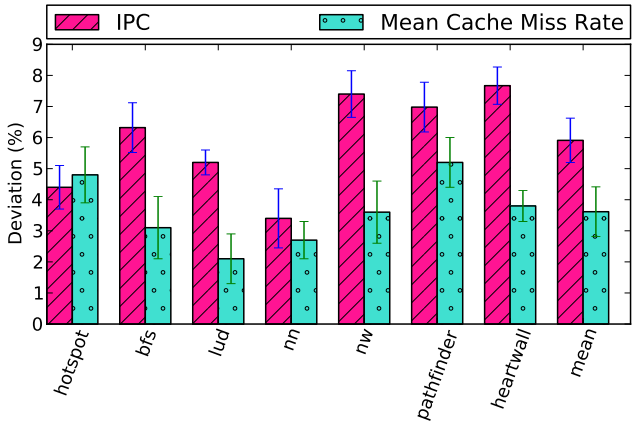

Figure 14. Average deviation from sequential execution

miss rate (all kinds of caches) varies from 2.1 to 5.2%. There is some variance ($< 0.95\%$) for different runs of the same benchmark (shown in error bars).

G. Comparison of Execution times

Table III presents a comparison of execution time of *GpuTejas* with a native system, and GPGPU-Sim. As we

can observe, *GpuTejas* is 1132.99x slower than native execution, while GPGPU-Sim is 487056.65x slower than native execution. In effect, GPGPU-Sim is 429.89x slower than *GpuTejas*. We can thus conclude that our benefits from parallelizing our simulator are substantial.

Benchmark	Execution Time		
	Native Execution(ms)	GPGPUsim(s)	*GpuTejas* (s)
hotspot	0.85	200	4.54
bfs	29.48	4517	5.36
lud	2.22	168	16.31
nn	0.12	3	0.86
nw	6.21	1673	27.76
pathfinder	1.02	280	23.51
heartwall	162.77	91861	151.24
mean	28.95 ms	14100.29 s	32.80 s

Table III
COMPARISON OF EXECUTION TIME

VI. CONCLUSION

We presented the design of a parallel Java-based GPGPU simulator, *GpuTejas*. We demonstrate a mean speedup of 17.33x over sequential execution for 64 simulation threads. We rely on relaxed synchronization using phasers, non-blocking structures and lock free implementations to derive our speedups. We have validated our timing model against an NVIDIA Tesla M2070 GPU. Our current approach is trace driven, and relies on a sequential GPU emulator to produce traces. Parallelizing the emulator and integrating it into our system is a part of future work.

REFERENCES

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer Graphics Forum*, vol. 26, 2007.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, March 2008.

[3] Nvidia, "Nvidias next generation cuda compute architecture, fermi," 2009.

[4] ——, "Nvidias next generation cuda compute architecture, kepler gk 110," 2012.

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

[6] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009.

[7] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *PACT*, 2010.

[8] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for gpgpu," in *MASCOTS*, 2010.

[9] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa, "Attila: a cycle-level execution-driven simulator for modern gpu architectures," in *ISPASS*, March 2006.

[10] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *PACT*, 2012.

[11] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework," *Georgia Institute of Technology*, 2012.

[12] J. Power, J. Hestness, M. Orr, M. Hill, and D. Wood, "Gem5-gpu: A heterogeneous cpu-gpu simulator," *Computer Architecture Letters (accepted)*, 2014.

[13] J. Meng and K. Skadron, "A reconfigurable simulator for large-scale heterogeneous multicore architectures," in *ISPASS*, 2011.

[14] V. Zakharenko, T. Aamodt, and A. Moshovos, "Characterizing the performance benefits of fused cpu/gpu systems using fusionsim," in *DATE*, 2013.

[15] S. Lee and W. W. Ro, "Parallel gpu architecture simulation framework exploiting work allocation unit parallelism," in *ISPASS*, 2013.

[16] G. Malhotra, P. Aggarwal, A. Sagar, and S. R. Sarangi, "ParTejas: A parallel simulator for multicore processors," in *ISPASS*, 2014.

[17] R. A. Vivanco and N. J. Pizzi, "Scientific computing with java and c++: a case study using functional magnetic resonance neuroimages," *Software: Practice and Experience*, vol. 35, no. 3, pp. 237–254, 2005.

[18] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, "Benchmarking java against c and fortran for scientific applications," in *ACM ISCOPE*, 2001.

[19] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence, "Java programming for high-performance numerical computing," *IBM Systems Journal*, vol. 39, no. 1, pp. 21–56, 2000.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[21] P. Aggarwal and S. Sarangi, "Lock-free and wait-free slot scheduling algorithms," in *IPDPS*, 2013.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[23] Nvidia, "Nvidia cuda toolkit v5.5," 2013.