# Scalable Multi-Cache Simulation Using GPUs

Michael Moeng, Sangyeun Cho, and Rami Melhem
Department of Computer Science
University of Pittsburgh
{moeng,cho,melhem}@cs.pitt.edu

*Abstract*—**Software simulation is the primary tool used for evaluation of processor design. Simulation offers better accuracy than analytical models and is an important evaluation step before actually fabricating a chip. Unfortunately, simulator speeds are slow—a conventional cycle-accurate simulator will be unable to keep up with increasing core counts in modern processor design.**

**Parallel simulation is one method for improving simulation speeds. Two major areas of parallel simulation research are multithreaded simulators and FPGAs as simulation accelerators. Multithreaded simulators can only extract coarse-grained parallelism and must sacrifice accuracy in order to scale well. FPGA-based simulators can extract fine-grained parallelism, but are expensive and difficult to program.**

**We propose using GPUs for architectural simulation, which can take advantage of a high degree of fine-grained parallelism. In addition, they are inexpensive and easier to program compared to FPGAs. To demonstrate our ideas, we implement a trace-driven many-cache simulator using NVIDIA's CUDA toolkit. GPU-accelerated cache simulation displays remarkable scaling with number of simulated caches when compared to serial CPU-only simulation.**

***Keywords-Simulation; Parallel Architecture; Cache; General-Purpose GPU; CUDA***

## I. INTRODUCTION

Architecture simulation is the process of simulating a target machine's execution using a host machine. Simulation is crucial in the design of new microprocessors. Simulation allows computer architects to create a processor or alter an existing design before embarking on the expensive fabrication process. In addition, a simulator can be easily modified to monitor behavior that might be difficult to expose after fabrication.

Two of the key metrics used to evaluate simulators are performance—how fast the simulator executes simulated instructions; and accuracy—how closely the simulator models its target machine architecture. Simulation performance is unacceptable in the context of chip multiprocessors (CMPs). To achieve perfect accuracy, simulation occurs serially on a single host core. Parallel simulation is necessary in order to continue developing chips with higher core counts, and using a parallel simulator running on a host CMP is a growing area of research [2,3,4,6].

Multithreaded simulation using a host CMP is far from ideal. Speedups are limited by the number of host cores. This tends to be less than the number of cores on the target machine being simulated, especially if an architect is interested in manycore chip design. Furthermore, host threads must synchronize with one another in order to maintain correct relative timing between simulated cores. This synchronization is costly, forcing simulators to sacrifice some accuracy for reasonable speedups [2,3,5,6].

Another growing area of simulation development is the use of Field-Programmable Gate Arrays (FPGAs) as co-processors [7,8,9,10]. FPGAs have become popular because they can take advantage of the fine-grained parallelism between hardware structures. Some limitations of FPGAs include cost and a constrained area. The limited area on an FPGA can be dealt with by reusing FPGA structures [8], but this limits the total parallelism that can be achieved.

We propose the use of Graphics Processing Units (GPUs) to accelerate manycore architecture simulation. NVIDIA's CUDA technology allows a programmer to write code using the C++ syntax and run the code on a GPU [15,16]. Graphics processing is an endlessly parallelizable task, and Graphics Processing Units are built to have a high degree of fine-grained parallelism, high memory bandwidth, and relatively fast communication with the corresponding CPU. We believe these properties make General Purpose GPU processing a strong candidate for simulating the timing partition of a manycore simulator (the CPU would simulate the functional partition). A similar partitioning is used in [7], except using an FPGA to accelerate timing simulation for a single core.

We demonstrate our idea by implementing a trace-based cache simulator. Trace-driven simulation is similar to a functional/timing partitioned simulator, with the trace acting as the functional partition. As an example of the parallel scaling available on GPUs, Figure 1 illustrates an ideal case—simulating L1 private caches with no shared data†. Simulation with GPUs shows only a minor increase in simulation time while CPU-only simulation slows down super-linearly with the number of caches. Our experimental details are in Section V.

The remainder of our paper discusses our GPU-based cache simulator and obstacles faced while implementing the simulator in order to deal with non-ideal cases. Specifically, we focus on dealing with cache coherence for a multithreaded workload and two-level cache hierarchies using shared and private L2 organizations.

Section II covers relevant techniques used to improve simulation performance. Section III describes the architecture of a GPU and the programming model used for CUDA. In Section IV we outline our cache simulator and describe implementation issues. Section V details our experimental setup, while Section VI covers our results. Section VII concludes and touches on several ways we plan to expand this work.

† See Section V for details

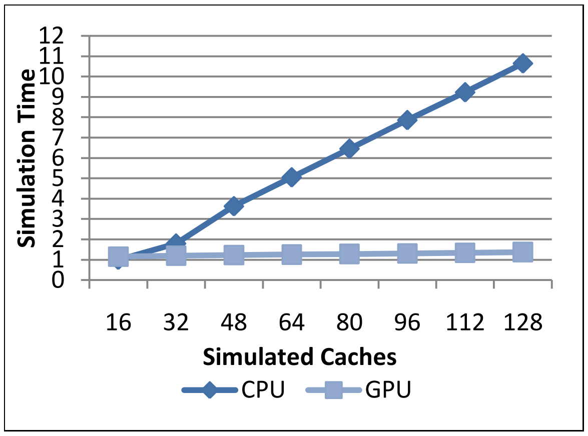

Figure 1. Simulation time as we vary the number of simulated caches, normalized to 16 caches in CPU-only simulation.

## II. BACKGROUND

The process of simulating a target machine can be broken into several components. Figure 2 shows the major parts shared by most cycle-accurate simulators. Functional simulation executes an application binary using the target machine's instruction set, dealing with control flow, system calls, and IO requests. Timing simulation models the hardware structures of interest to an architect. These can be core components such as a pipeline, or uncore components such as an on-chip network.

Much of the simulation state used in functional simulation, most notably data values, are not used during timing simulation. Similarly, much information obtained during timing—such as specific cache tags—are not needed by the functional simulator. This clean division of data results in many simulators being developed with a functional/timing partition, such as [3,6,7]. The functional partition does functional simulation and generates events relevant to the timing partition. The timing partition calculates how long these events take to resolve and notifies the functional partition if it needs to deviate from regular functional simulation (such as simulating the wrong path of a branch).

### A. *Accelerating Functional Simulation*

Direct execution is a technique for improving simulation performance by performing functional simulation directly on a host machine and is used in [3]. This can be achieved using binary instrumentation [13]. Instructions in the application binary are instrumented to generate events which are simulated by the timing partition. Direct execution can greatly speed up functional simulation and is orthogonal to our work.

Functional simulation generates a stream of events that are sent to the timing partition, such as memory access addresses. Many events tend to not change even when the machine configuration is changed. Trace-driven simulation [11,12] takes advantage of this by putting all static information sent to the timing partition into a trace file. The information is collected once using a functional simulator and a trace of events is generated. Events in the trace can be simulated multiple times for different target machine configurations to amortize the cost of trace generation. Much like direct execution, trace-driven simulation is orthogonal to our overall work and we use trace-driven simulation in this work to evaluate our ideas.

### B. *Multithreaded Simulation*

Multithreaded Simulation has been used by computer architects for some time, and is gaining popularity as CMPs become the norm [2,3,4,5,6]. In general, a simulator will be divided into threads, each of which is assigned one or more cores to simulate. Difficulties arise when these threads residing on different cores need to communicate if the workload involves a shared memory space or because the cores access shared resources. All target cores need to be synchronized so they access resources in the same order as they would in a real machine. Using looser synchronization (where simulated cores progress at slightly different rates) reduces simulation accuracy but improves performance.

There are several popular techniques used to loosely synchronize simulated cores in parallel simulation. Quantum-based synchronization [5] periodically synchronizes all threads. Slack-based synchronization [2] keeps track of the slowest-progressing thread and ensures that no thread gets too far ahead of the slower thread. Point-to-point synchronization [3] forces each thread to periodically select a single other, random thread and ensure their progress is approximately the same. Our method of synchronization most closely matches quantum-based synchronization.

### C. *Acceleration with Coprocessors*

Han et al. introduced a GPU-based cache simulator in [1], but assign sets to threads to leverage parallel speedups on the GPU. The GPU first sorts trace accesses by set, and each set is processed independently. This means simulator processes cache accesses out of order temporally, so only hit/miss information can be gathered. Our work processes requests to each cache in order, which is essential if caches are to interact with one another for timing simulation. We extract our

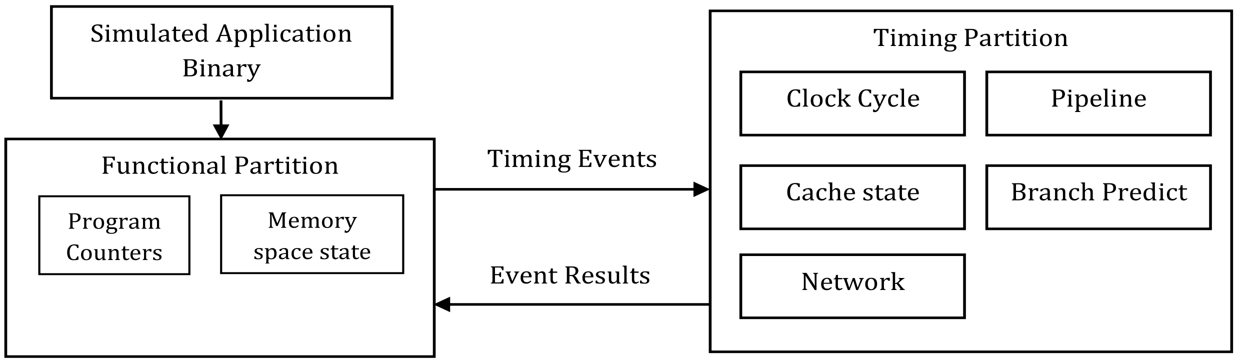

Figure 2. Diagram of Functional/Timing partitioned simulation.

parallelism through multiple caches rather than multiple sets. This means our single-cache simulation speed is significantly slower; however, more total caches can be simulated, which is a good fit for multicore/manycore timing simulation. In addition, [1] simulates a single L1 or single L1/L2 pair. We focus on the simulating multiple caches which interact with each other.

GPU acceleration has been proposed for low level circuit simulation. In [18], Chatterjee, DeOrio, and Bertacco first transform a circuit netlist into CUDA code before simulating the circuit. Gates are grouped into layers so CUDA threads can simulate gates from the same layer in parallel; only simulating gates if input values have changed. In [19], Nanjundappa et al. parallelize simulation of circuits modeled in SystemC. As with [18], the authors first transform code describing the circuit into CUDA code. Each step, all hardware modules with new inputs are simulated and their outputs are propagated.

Both [18] and [19] target low level circuit simulation, which is much slower but much more detailed than architectural simulation. Architectural simulation is more useful for evaluation of new architectures. Low level circuit simulation is also somewhat easier to parallelize on a GPU than architectural simulation. First, the time needed to simulate a layer can be balanced between threads (we know how many gates are in the layer). Second, the same instructions are executed each step by all gates, which reduces warp divergences, an important CUDA concept we discuss in Section III. In [20], Perumalla covers general discrete event simulation on GPUs, and discuss challenges when using specialized GPUs as opposed to general purpose CPUs for parallel discrete event simulation.

FPGAs allow hardware structures to be directly programmed into them. This results in fast simulation for several reasons. First, a hardware structure simulated on an FPGA takes many fewer FPGA clock cycles to simulate a target machine cycle than a general-purpose processor. Second, multiple hardware structures placed on an FPGA can quickly communicate and thus may be simulated in parallel without dealing with synchronization issues multithreaded simulation faces. Finally, FPGAs increase their transistor count with Moore's Law, which gives them better expected scalability for CMP simulation. In contrast, single-threaded simulation does not scale, as extra transistors gained through Moore's Law are mostly used for extra cores. These insights led to the RAMP project [9], which focuses on using FPGAs to accelerate CMP simulation.

The FAST simulator [7] is a single core simulator which performs functional simulation on a host machine. An FPGA performs a detailed timing simulation of a target machine's pipeline, branch predictor, and memory hierarchy. An FPGA can simulate a highly detailed pipeline very quickly. The drawbacks to using an FPGA include difficulty in developing the simulator and long communication latency with the host (compared to conventional single-threaded simulation). A significant part of the FAST project deals with the long latency between the functional and timing partitions. This involves roll-backs when the timing partition generates an event that is not part of pure functional simulation, such as simulating instructions for a mispredicted branch.

HAsim [8] deals with area constraints of FPGAs. An FPGA quickly runs out of space when simulating a CMP if one tries to replicate hardware structures for each core. HAsim reuses hardware structures in order to accurately simulate the entire CMP. By overlapping some of the computation when simulating multiple cores, simulation throughput improves somewhat as the number of simulated cores increases. However, beyond 4 target cores HAsim experiences a roughly linear slowdown with simulated core count.

## III. NVIDIA's CUDA

In 2006 NVIDIA introduced its CUDA technology [15], which uses NVIDIA GPUs for general purpose computation usually performed by CPUs. In this section, we describe enough of GPU architectures and the CUDA programming model to explain our work; for more information on CUDA, readers should refer to [16].

In the CUDA programming model, work is sent from the host (CPU+main memory) to the device (GPU). Host memory is explicitly transferred to the device; GPU code is started by calling a kernel function. Each kernel function runs under a Single Instruction, Multiple Data (SIMD) paradigm. Each kernel is broken into a number of blocks, each of which has a number of threads. Threads within a block can communicate using a dedicated shared memory and have access to a fast barrier primitive. Figure 3 is a diagram showing the organization of threads and blocks in a kernel. On the GPU, each block in the kernel is executed on one of the GPUs vector multiprocessor. Multiple blocks can be co-scheduled on a multiprocessor and share its resources.

Threads from a block are grouped by the GPU scheduler and run together as a warp of 32 threads. These threads should be executing the exact same instructions because they are running together on a vector multiprocessor. When threads in the same warp follow different control paths, each control path must be executed serially—threads from a single control path take up the multiprocessor while threads in other control paths sit idle. This *warp divergence* severely impacts performance, and should be avoided whenever possible.

CUDA's memory hierarchy includes *global*, *local*, and *shared* memory. Both global and local memory map to a GPU's graphics RAM; global memory is accessible by all threads in the kernel and local memory is thread-private. Shared memory is block-private—all threads within a block have access to shared memory allocated for that block. We used a Fermi card in our experiments, which features a L1/L2 cache hierarchy that caches global and local memory requests. The L1 cache uses the same type of memory as shared memory, which is why shared memory is not cached. Threads in different blocks may only communicate through global memory. CUDA provides atomic instructions to allow for consistent memory accesses.

Note that CUDA syntax calls code running on the CPU "host code" and code running on the GPU "device code". Because this somewhat conflicts with simulation syntax of host

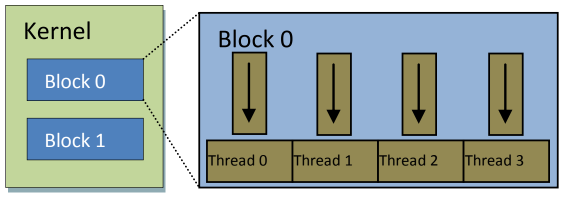

Figure 3. Diagram of thread / block organization within a CUDA kernel.

machine and target machine, we will use CPU and GPU instead of standard CUDA syntax.

## IV. GPU Cache Simulation

In this section we describe how we perform cache simulation using a GPU. Each cache processes a sequence of memory accesses stored in a trace file. Simulation occurs in intervals: a chunk of each trace is read from file; then memory accesses in that chunk are simulated. The process to fetch memory accesses is the same for both the CPU-only simulation we use as a baseline and for the GPU-based simulation. Below we describe the changes made to accommodate simulation using CUDA.

We start by describing the logical simulation flow, illustrated in Figure 4. After reading a chunk of trace files into memory, trace data (addresses, instruction timestamps, and access types) are transferred to GPU memory. Once memory transfer completes, the "L1 kernel" is started. During its execution, the L1 kernel outputs miss addresses to a buffer residing in device memory. Once the L1 kernel completes, the "L2 kernel" is started, which processes the L1 miss addresses. After the L2 kernel finishes, statistics for both the L1 and L2 are copied back to the host and simulation moves to the next interval.

### *A. Thread-to-Block Mapping*

Each thread in our kernel simulates a cache way. This allows us to parallelize the address lookup process. Threads simulating the same cache perform their tag comparisons to check for a hit simultaneously. The threads then communicate via shared memory whether there was a hit or not. If no hit occurred, eviction follows the Least Recently Used (LRU) policy by checking each way's position on an LRU list. The first thread in the cache performs the check for a victim serially. We experimented with a parallel tree-based reduction to find a victim, but this had worse results at all associativity levels we evaluated. Whether or not an eviction occurred, all threads then update their LRU position in parallel.

To help guide our thread/block mapping, we used the recommended CUDA Occupancy Calculator [17]. Mapping just one cache to a block results in better performance with a small number of simulated caches, but does not scale as well to a larger cache count. Increasing the number of caches improves scaling until register or shared memory usage exceeds a block's capacity. In addition, caches mapped to the same block are kept perfectly synchronized (see Part F). The Occupancy Calculator determined that 64 threads per block allowed for the maximum parallelism for our GPU. Therefore, for most experiments, we map 4 caches to each block, with each cache spawning a number of threads equal to the cache's associativity. When simulating 16-way caches, we end up with 64 threads per block.

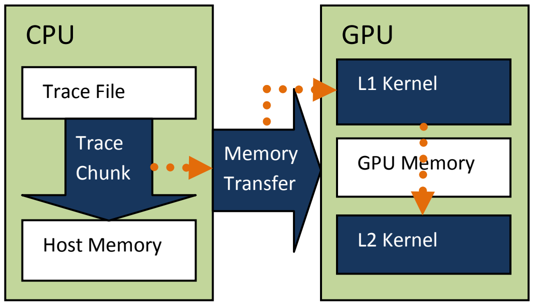

Figure 4. Diagram of Logical Simulation Flow.

### *B. Memory Organization*

Cache state is stored in global memory. Initially we copied tag information to local memory at the start of each kernel, but leaving tags and other metadata like the LRU position into global memory helped us deal with cache coherency (see Part D). Cache parameters such as associativity, cache size, and block size are initially in global memory and are copied into local memory to cut down on contention between threads accessing the same information.

Initially we tried placing tag and LRU position data in shared memory, but this quickly filled up the maximum shared memory available to each physical multiprocessor and reduces the number of caches that can be simulated. In addition, the L1/L2 caching added to Fermi GPUs means global memory accesses are nearly as fast as shared memory accesses, provided they have good locality (which cache tag lookups tend to have).

### *C. Concurrent Execution*

CUDA supports asynchronous memory transfers. In addition, CUDA kernels are spawned in a non-blocking fashion, so CPU code can still be run. We use these features to speed up execution with the help of CUDA streams. A stream is a serial sequence of commands that has no ordering restrictions with respect to other streams. In addition, asynchronous commands in a CUDA stream can also execute concurrently with blocking commands running on the CPU.

After a trace chunk is read from a trace file, the transfer of data from CPU's memory space to the GPU's memory space is started asynchronously before the next trace chunk is read. This process uses a single stream dedicated to this memory transfer. More transfer streams did not benefit performance as each transfer uses the maximum transfer bandwidth.

In addition to using streams for memory transfer, we use streams to concurrently execute all three major components of the trace simulation:

- Trace I/O and memory transfer
- GPU kernel simulating the L1 cache
- GPU kernel simulating the L2 cache

These are overlapped in a pipelined manner shown in Figure 5. Two streams are used—each with its own trace buffer and L1 miss address buffer. While the L1 kernel reads trace information from Buffer trace-A and outputs misses to Buffer miss-A, the trace I/O will be copying the next chunk into trace-B and the L2 kernel will be processing misses from Buffer miss-B.

Note that simultaneous execution of kernels is supported only by newer NVIDIA GPUs (Fermi GPUs). Without simultaneous kernel execution, the L1 and L2 kernels must be executed serially. However, kernels still spawn asynchronously, so the trace file IO and memory transfer can still execute concurrently with the L1 or L2 kernel.

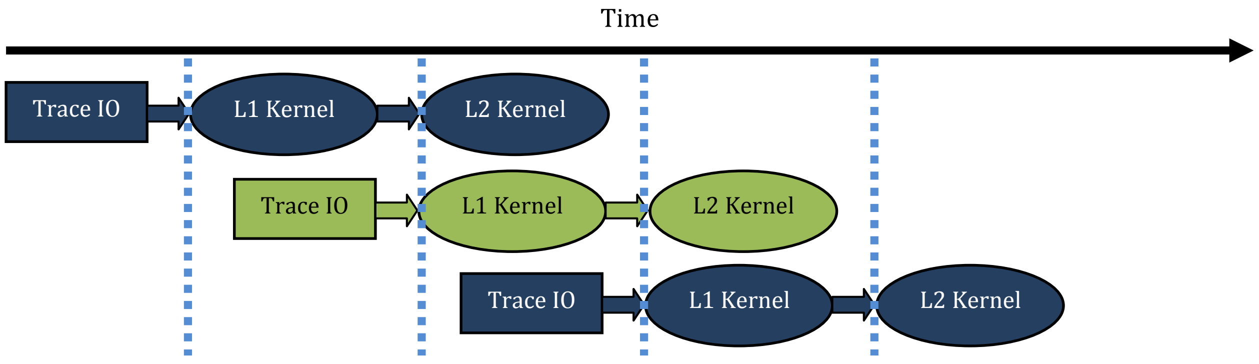

Figure 5. Diagram of pipelined kernel execution. Coloring denotes which buffer is being used for by kernel. Vertical dotted lines denote synchronization points.

### D. *Cache Coherence*

For multithreaded workloads, it is necessary to keep caches coherent. We implement a snooping protocol to gauge the effect of inter-cache communication on GPU simulation. Each cache keeps track of a list of other caches snooping on it. When a write is encountered, the writing cache can check the tags of other snooping caches because tag and meta state are stored in global memory. If a match is found, the writing cache can directly invalidate the block in other caches by atomically modifying the meta state. An atomic operation is necessary so multiple caches do not overcount simultaneous invalidations.

When all caches in a multithreaded workload are mapped to the same block (this is possible when there are 4 or fewer threads that need to snoop on each other), all caches are kept in sync by calling CUDA's fast barrier primitive each cycle. In this case, direct invalidation results in negligible deviation from the host. This deviation results from same-cycle writes that cause race conditions. Inaccuracies arise only when there are more caches snooping on each other than we can fit onto a block (more than 4 caches).

### E. *Shared Memory*

Private L2 cache slices receive all misses from a single L1, so there is no interaction between L1 slices when misses occur. For shared caches, this is not the case. In order to simulate L2 caches using the shared cache policy, we make use of CUDA's atomic instructions. Each L2 cache has an associated miss address buffer. An L1 kernel thread determines which L2 slice a miss address belongs to using the lowest bits of the block address. The kernel then uses an atomic operation to increment the miss address buffer index for the destination L2 cache before placing its miss address into the buffer. Each simulated cache in the L2 kernel has a sequence of memory accesses and can then process its accesses in the same manner as the L1 kernel.

A significant issue with this approach occurred when trying to synchronize threads at the end of the L1 kernel. This was necessary in order to add a delimiter marking the end of the miss address buffer for each L2 slice. The scheduler currently used for CUDA cores does not swap out spinning threads, so implementing a barrier using global memory will often result in deadlock. Once a kernel terminates, however, threads are guaranteed to be synchronized. We used a short cleanup kernel to add this delimiter, avoiding the need for cross-block synchronization within the L1 kernel.

### F. *Synchronization*

Both techniques used in Parts D and E (although especially Part D) rely on threads in a kernel proceeding at approximately the same rate to be accurate. This is not a problem for caches in the same block, because for each trace item we execute CUDA's block-wide barrier so all caches mapped to the block are synchronized. However, as discussed above, inter-block barriers within a kernel are impractical. For inter-block synchronization, we turn to quantum-based synchronization [5]. CUDA kernels are a natural fit for quantum synchronization, because in between kernels all threads are automatically synchronized. We vary the size of a trace chunk in order to evaluate the effect of quantum size on our simulation. Because threads within a block are synchronized every access, our quantum synchronization is hierarchical—quantum size for caches in the same block is 1, and quantum size for caches In different blocks is the trace chunk size. Much like classic quantum-based synchronization, processing fewer trace elements in a kernel (using a smaller quantum) results in a worse performance but better accuracy.

## V. EXPERIMENTAL SETUP

All our experiments are performed on a machine detailed in Table I. We implemented our own trace-driven cache simulator running on the CPU to serve as a baseline. Execution times are collected with the UNIX time command. We use a low-mid range GPU, a GeForce GTS 450, for our experiments. Fermi cards have an L1/L2 cache hierarchy that can be configured to take up a portion of the shared memory. We found giving the GPU 48KB cache and 16KB shared performed slightly better than 16KB cache and 48KB shared; however, it was important to set both the L1 and L2 kernels to the same cache configuration.

Although our host machine has less RAM than a typical server, our simulator's working set size is much smaller than 2GB—the main memory is never taxed. For our machine, we empirically found a chunk size of 2048 to work well for both CPU and GPU simulation. Later we evaluate the effect of a varied trace chunk size.

## A. *Target Machine*

We evaluate three timing models. The first is one level of ***L1*** caches. The next two are L1/L2 cache hierarchies using ***Private*** L2 caches (where addresses are mapped to the same cache tile as the L1), and ***Shared*** L2 caches (where addresses are mapped to tiles based on the lower bits of the block address). Unless otherwise stated, L1 caches are 16-way; L2 caches are always 16-way. We implement snoop-based coherence, where each cache keeps a list of snoopers and invalidates written cache blocks for copies residing in other caches on the snoop list. Table II lists our other target parameters. We assume the caches are connected as a 2D mesh, although we do not simulate the on-chip network in this work. Because our simulated caches interact with one another, direct comparison with [1] is impossible.

TABLE I. HOST MACHINE

| **Host Machine Specifications** | |
|---|---|
| Processor | Intel Core2 Duo |
| Clock Rate | 3.0 GHz |
| Memory | 2GB |
| Operating System | Ubuntu 10.04 |
| ***GPU Specifications*** | |
| Model | GeForce GTS 450 |
| Multiprocessors | 4 |
| CUDA Cores | 192 cores |
| Clock Rate | 925 MHz |
| Graphics Memory | 1GB |
| CUDA Version | 3.20 |

TABLE II. TARGET TIMING MODEL

| **Target Timing Model Specifications** | |
|---|---|
| Block Size | 64 Bytes |
| Associativity | 16-way |
| Replacement Policy | Least Recently Used (LRU) |
| L1 Size (per slice) | 64kB |
| L2 Size (per slice) | 512kB |
| Timing models | *L1, Private, Shared* |

## B. *Workloads*

We use a combination of workloads derived from a few of the PARSEC benchmarks: blackscholes, streamcluster, and canneal [14], using the large simulation inputs. Traces are generated by instrumenting the benchmarks with PIN [13]. Each trace item contains the memory address, memory access type, and dynamic instruction count.

During simulation, each cache or cache tile simulates 10 million accesses. During trace generation, each benchmark is fast forwarded until all working threads are spawned. Each thread spawned for the benchmark writes to its own trace file. Table III lists the benchmark combinations we evaluate in this work, including the number of trace files generated for each workload and the number of instructions fast forwarded (in millions). Workloads G, H, and I are used to evaluate accuracy in Part E.

Because we simulate a large number of caches, trace files are reused. In these cases, an offset is specified that is greater than the number of accesses in a trace chunk. A new file pointer is opened for a repeated trace, and is offset by an amount equal to the offset. For example, if we simulate 32 caches and have 4 trace files, each file is reused 8 times. If we use Trace T eight times, we open eight file pointers; the first is not offset, the second is offset by OFFSET, the third is offset by 2 × OFFSET, and so on. In addition, we mimic virtual address translation by putting addresses into a numbered address space. Addresses from multithreaded traces share address spaces; but repeated traces and multiprogrammed traces have different address spaces.

TABLE III. WORKLOADS

| *Label* | *Benchmark* | *Trace Files* | *Fast Forward (instructions)* |
|---|---|---|---|
| A | streamcluster | 1 | 150M |
| B | streamcluster | 1 | 150M |
| | blackscholes | 1 | 600M |
| C | streamcluster | 4 | 150M |
| D | blackscholes | 4 | 600M |
| E | streamcluster | 4 | 150M |
| | blackscholes | 4 | 600M |
| F | streamcluster | 4 | 150M |
| | blackscholes | 4 | 600M |
| | canneal | 4 | 5500M |
| G | streamcluster | 8 | 150M |
| H | blackscholes | 8 | 600M |
| I | canneal | 8 | 5500M |

## VI. EXPERIMENTAL RESULTS

### A. *Single-level Caches*

Figure 6 shows execution times for *L1* simulations where only the L1 kernel is running. We show results for 32, 64, and 96 simulated caches—normalized to 32 caches under CPU simulation. CPU-only simulation slows down superlinearly with cache count—average execution time increases by 3.4x from 32 to 96 caches. GPU accelerated execution time increases by just 1.1x. GPU performance is the worst in workload F—this workload is the most heterogeneous and must deal with cache coherency. Out of all our workloads, this behavior causes the most warp divergences—threads following different execution paths; such divergences have a negative effect on parallelism for CUDA multiprocessors.

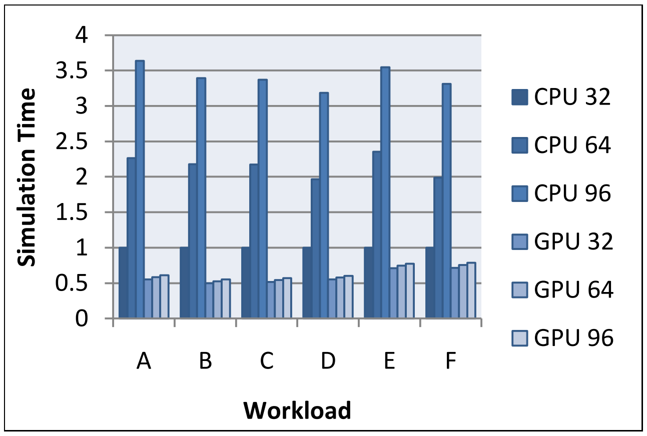

Figure 6. Execution time when simulating *L1* for 32, 64, and 96 caches. First three bars for each workload represent CPU-only simulation with increasing cache count; next three bars represent GPU simulation.

## *B. Hierarchical Caches*

Figure 7 shows execution times for *Private* simulation where L1 misses are passed to the L2 on the same tile. As in Part A, results are normalized to 32 tiles in CPU-only simulation. Parallel scaling is worse at 96 simulated tiles. The slowdown when scaling from 32 to 96 tiles increases from 10% (L1) to 34% (Private). Worse scaling for L1/L2 simulation means 96 L1/L2 tiles exceeds the limit of parallel improvements possible for our 192-core GPU.

Figure 8 shows execution times for Shared simulation, where an address belongs to a tile based on the lowest bits of its block address. A consequence of using shared L2 cache slices is a single slice can receive a disproportionate number of the L1 misses. This can cause a heavy load imbalance for GPU simulation, as seen especially in workload D. This highlights the fact that a more efficient cache hierarchy can also result in faster simulation. Performance for Shared is governed by the degree of load imbalance, which is why GPU performance is erratic. Long execution times mean more cache accesses were directed to the same cache slice. Because we take the mod of a miss access's block address to get its destination tile, mappings change with core counts. Some mappings, such as the mapping that occurs at 64 caches, cause heavier load imbalances than others. We performed some profiling of this load imbalance for 64 caches—for a trace chunk size of 2048 accesses to each L1, a single L2 slice can receive over 10k misses during one kernel.

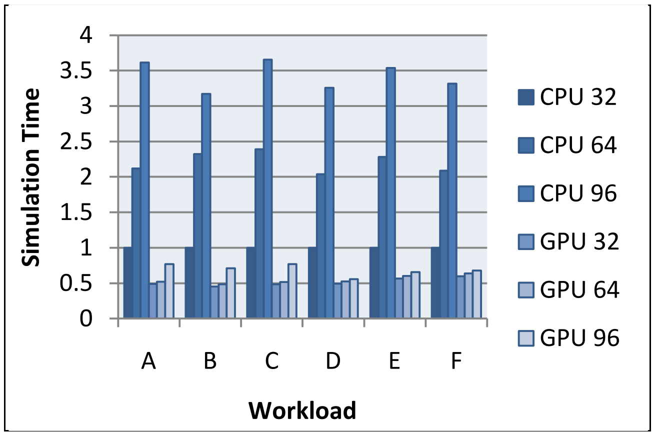

Figure 7. Execution time when simulating Private for 32, 64, and 96 tiles.

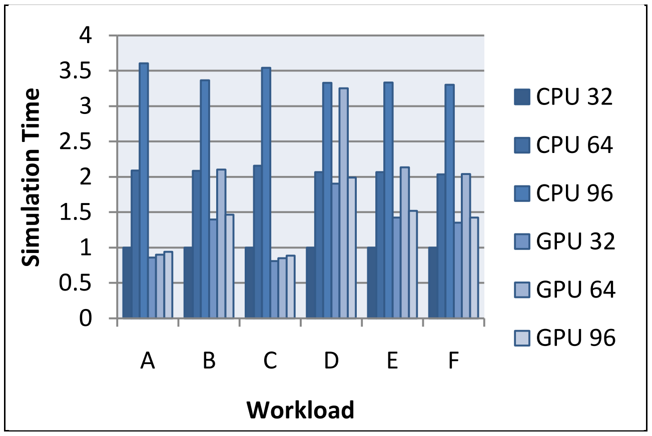

Figure 8. Execution time when simulating *Shared* for 32, 64, and 96 tiles.

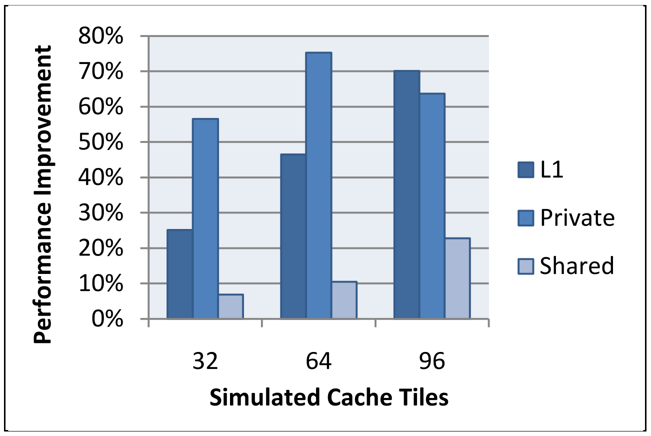

Figure 9. Average performance improvement of concurrent over serialized execution. Results are for *L1*, *Private*, and *Shared*.

## *C. Concurrent Execution*

To measure the impact of using overlapping CUDA streams, we measure performance for L1, Private, and Shared with and without concurrent execution. When CUDA streams are disabled, simulation follows the same flow as the logical from Figure 4. Performance improvements for concurrent over serialized simulation are shown in Figure 9. The benefits from concurrent execution increase as we simulate more caches, because there is more data to transfer to the GPU. The exception is for Private simulation with 96 tiles. As we saw in Part B, for 96 L1/L2 tiles the GPU has saturated its cores and execution is already being partially serialized—lessening the benefit of concurrently executing the L1 and L2 kernels. Shared simulation benefits very little from concurrent execution, because load imbalance to heavily accessed L2 slices serializes much of the simulation process.

## *D. Caches per Block*

We perform a study on the effect of the number of caches mapped to each CUDA block. Mapping more caches to a block results in:

- Better scaling to high cache counts (provided other resources are not taxed)
- Better accuracy (less inter-thread communication)
- Worse performance with low cache counts

We found four caches per block to give the best overall performance. Beyond four, our GPU was limited in other areas (registers). It is possible that more caches per block would be suitable for some studies, however. Caches on the same block can synchronize within a kernel launch, which makes error negligible if all caches that interact with one another fit onto the same CUDA block.

To illustrate the effect of cache-to-block mapping on performance, Figure 10 shows execution time—averaged across all workloads—for different numbers of caches per CUDA block for L1 simulation. At 32 simulated caches, mapping only one cache to each CUDA block improves performance by 80% over 4 caches per block, because there are fewer warp divergences. Specifically, mapping one cache to a CUDA block removes warp divergences caused when some

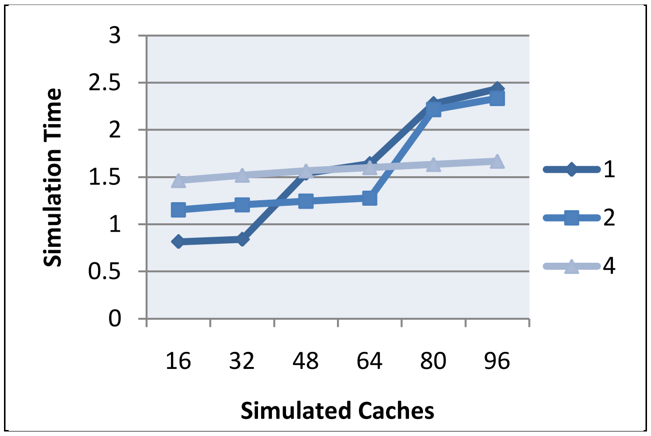

Figure 10. Average execution time *when* simulating *L1*, normalized to CPU simulation of 16 caches. Lines show performance scaling when mapping 1, 2, or 4 caches to each CUDA block.

threads in the same block are resolving a miss while others are resolving a hit.

However, as the number of simulated caches increase the GPU reaches its limit for active blocks. With fewer caches per block, the GPU saturates faster. Therefore, scaling is better when there are more caches per block. At 1 or 2 caches per block, we see jumps in execution time corresponding to block execution being serialized, but at 4 caches per block execution time increases only marginally up to 96 simulated caches.

### *E. Accuracy*

We evaluate GPU simulation's impact on accuracy for cache coherency and for the shared cache organization. For cache coherency, we measure the number of invalidated blocks for multithreaded workloads with more snooping caches than will fit onto a single CUDA block. For a shared L2 organization, we record L2 miss counts. Accuracy metrics are compared against CPU-only simulation.

To evaluate invalidation-error, we evaluate multithreaded schemes running L1 simulation. We simulate 64 caches running workloads G, H and I; these workloads use more threads sharing a memory space than there are caches mapped to a CUDA block and therefore have nontrivial error rates. Figures 11, 12, and 13 show accuracy (measured in number of invalidated cache blocks) and performance tradeoffs for varying quantum sizes (measured in trace items per kernel launch). Performance is relative to GPU performance at a chunk size of 2048; CPU performance for a chunk size of 2048 is plotted for reference. Quantum-based synchronization works well to control invalidation count error with moderate overheads for Workload G and I. Workload H has low error rates for all chunk sizes. As an aside, trace chunk size had no significant impact on CPU-only simulation performance for the chunk sizes we used.

Cache coherency error encompasses all error for private L1 and private L2 caches. Error for shared caches comes from the ordering of misses sent from all L1 caches. To evaluate error when simulating shared L2 caches, we compare L2 miss counts with CPU-only execution. Figure 14 shows error in miss counts for each workload using the default trace chunk size of 2048. Error rates are very low, below 1% for all workloads, although in the future we'd like to examine other metrics to evaluate the accuracy of our shared cache implementation.

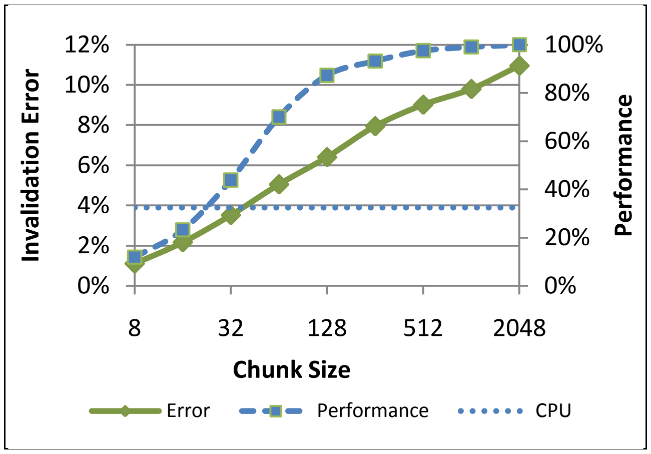

Figure 11. Error (left axis, solid line) and Performance (right axis, dotted line) when simulating *L1* with varied trace chunk size for workload G.

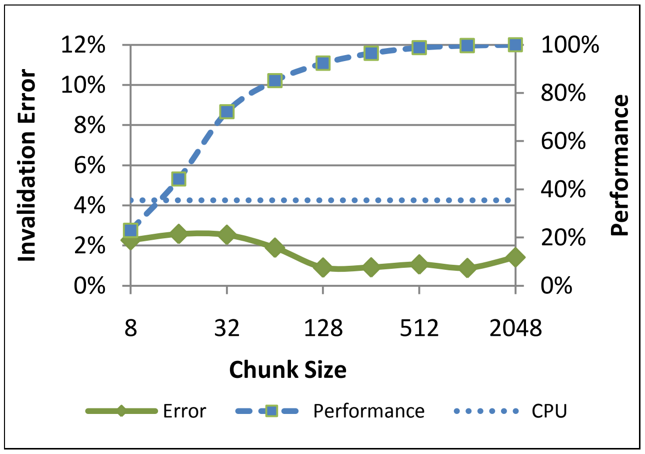

Figure 12. Error (left axis, solid line) and Performance (right axis, dotted line) when simulating *L1* with varied trace chunk size for workload H.

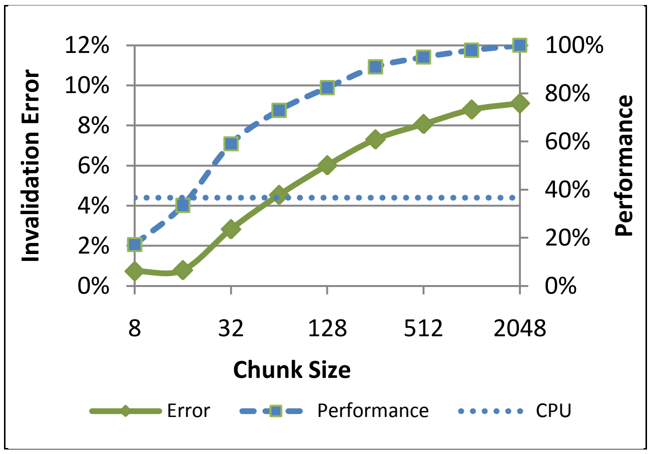

Figure 13. Error (left axis, solid line) and Performance (right axis, dotted line) when simulating *L1* with varied trace chunk size for workload I.

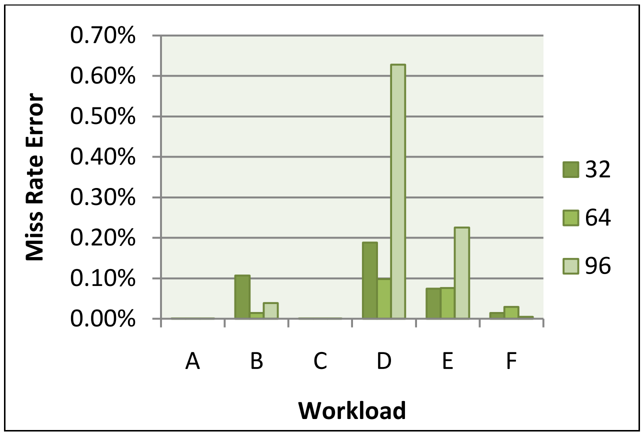

Figure 14. Error in L2 miss counts when simulating *Shared*.

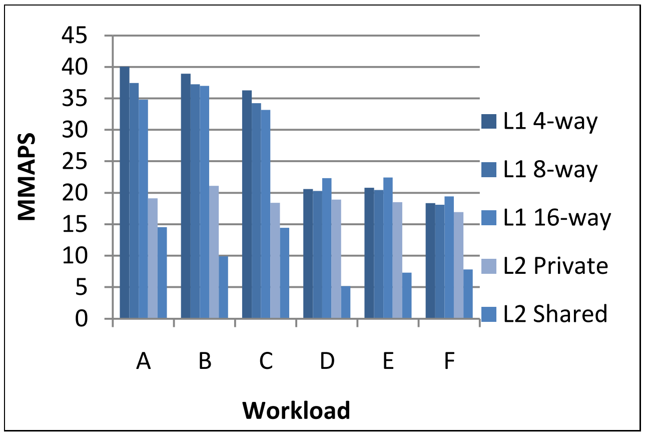

Figure 15. Simulation throughput, measured in Million Memory Accesses per Second (MMAPS). Shown are *L1* configurations at varying associativities, as well as *Private* and *Shared*. All configurations use 96 tiles.

### *F. Absolute Performance, Associativity*

Figure 15 shows our simulator's absolute performance, measured in Million Memory Accesses per Second (MMAPS), when simulating 96 tiles. We measure performance for *Private* and *Shared* simulation. In addition, we also vary L1 associativities when simulating *L1*. For *L1* and *Private* Simulation, GPU simulation can simulate over 15 million accesses per second. For *Shared*, GPU simulation still maintains over 5 million accesses per second.

## VII. Conclusions

Using GPUs for the timing partition of architectural simulation can achieve much better scaling with core count than other techniques used to simulate multicore or manycore systems. Multithreaded simulation is limited because CPUs on host machines have lower core counts than a researcher is interested in. FPGAs are similarly constrained as programmable gates are less area efficient. Because GPUs are designed with very high core counts, parallel simulation using GPUs can stay ahead of Moore's Law. To prove this hypothesis, we implemented a trace-driven cache simulator running on a GPU, which outstrips CPU-only simulation when we are interested in many caches.

As future work, we plan to implement an on-chip network simulator in CUDA. Furthermore, we plan to move from trace-driven cache simulation to more detailed trace-driven simulation similar to TPTS [11] or execution-driven simulation where the functional simulation takes place on a host machine's CPU much like the FAST simulator [7].

## Acknowledgment

This work was supported in part by the US National Science Foundation under CCF-1064976 and CNS-1059202.

## References

[1] W. Han, G. Xiaopeng, L. Xiang, and C. Xianqin, "Using GPU to Accelerate a Pin-based Multi-level Cache Simulator," Proceedings of the 2010 Spring Simulation Multiconference, 2010.

[2] J. Chen, M. Annavaram, and M. Dubois, "Slacksim: a platform for parallel simulations of cmps on cmps," SIGMETRICS Performance Evaluation Review, vol. 37, no. 2, pp. 77–78, 2009.

[3] J. Miller, H. Kasture, G. Kurian, C. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in HPCA, January 2010.

[4] M. Lis, K. S. Shim, M. H. Cho, P. Ren, O. Khan, and S. Devadas, "DARSIM: a parallel cycle-level NoC simulator," in Sixth Workshop on Modeling, Benchmarking, and Simulation (MoBS), June 2010.

[5] S. S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood, "Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator," IEEE Concurrency, Vol.8 No. 4, pp. 12-20, 2000.

[6] M. Chidester and A. George, "Parallel simulation of chip-multiprocessor architectures," ACM Trans. Model. Comput. Simul., vol. 12, no. 3, pp. 176–200, 2002.

[7] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies FAST: Fast, full-system, cycle-accurate simulators," in MICRO, 2007.

[8] M. Pellaur, M. Adlery, M. Kinsy, A. Parashary, and J. Emer, "HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing," in HPCA, 2011.

[9] D. Patterson, Arvind, K. Asanovi´c, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, , M. Oskin, J. Rabaey, and J. Wawrzynek. "RAMP: Research Accelerator for Multiple Processors." In Proceedings of Hot Chips 18, Palo Alto, CA, Aug. 2006.

[10] C. Ihrig, R. Melhem and A. Jones. "Automated Modeling and Emulation of Interconnect Designs for Many-Core Chip Multiprocessors," DAC, Anaheim, CA, 2010.

[11] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, M. Moeng. "TPTS: A Novel Framework for Very Fast Manycore Processor Architecture Simulation," Proceedings of the Int'l Conference on Parallel Processing (ICPP), pp. 446~453, Portland, Oregon, September 2008.

[12] R. A. Uhlig and T. N. Mudge. "Trace-Driven Memory Simulation: A Survey," ACM Computing Surveys, 29(2): 128–170, June 1997.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Programming Language Design and Implementation (PLDI), Chicago, IL, June 2005, pp. 190-200.

[14] C. Bienia. "Benchmarking Modern Multiprocessors," Ph.D. Thesis. Princeton University, January 2011.

[15] NVIDIA. 2010. CUDA Technology; http://www.nvidia.com/CUDA.

[16] NVIDIA. 2010. CUDA Programming Guide 3.2; http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

[17] NVIDIA. 2010. CUDA Occupancy Calculator; http://developer.download.nvidia.com/compute/cuda/3_2_prod/sdk/docs/CUDA_Occupancy_Calculator.x

[18] D. Chatterjee, A. DeOrio and V. Bertacco. "Event-Driven Gate-Level Simulation with GP-GPUs," DAC '09, San Francisco, CA, July 2009, pp. 557-562.

[19] M. Nanjundappa, H. Patel, B. Jose, and S. Shukla. "SCGPSim: A Fast SystemC Simulator on GPUs," ASP-DAC '10, Taipei, Taiwan, Jan. 2010, pp. 149-154.

[20] K. Perumalla. "Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)," PADS '06