Analyzing CUDA Workloads Using a Detailed GPU Simulator

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong and Tor M. Aamodt
University of British Columbia,
Vancouver, BC, Canada
{*bakhoda,gyuan,wwlfung,henryw,aamodt*}*@ece.ubc.ca*

Abstract

Modern Graphic Processing Units (GPUs) provide sufficiently flexible programming models that understanding their performance can provide insight in designing tomorrow's manycore processors, whether those are GPUs or otherwise. The combination of multiple, multithreaded, SIMD cores makes studying these GPUs useful in understanding tradeoffs among memory, data, and thread level parallelism. While modern GPUs offer orders of magnitude more raw computing power than contemporary CPUs, many important applications, even those with abundant data level parallelism, do not achieve peak performance. This paper characterizes several non-graphics applications written in NVIDIA's CUDA programming model by running them on a novel detailed microarchitecture performance simulator that runs NVIDIA's parallel thread execution *(PTX) virtual instruction set. For this study, we selected twelve non-trivial CUDA applications demonstrating varying levels of performance improvement on GPU hardware (versus a CPU-only sequential version of the application). We study the performance of these applications on our GPU performance simulator with configurations comparable to contemporary high-end graphics cards. We characterize the performance impact of several microarchitecture design choices including choice of interconnect topology, use of caches, design of memory controller, parallel workload distribution mechanisms, and memory request coalescing hardware. Two observations we make are (1) that for the applications we study, performance is more sensitive to interconnect bisection bandwidth rather than latency, and (2) that, for some applications, running fewer threads concurrently than on-chip resources might otherwise allow can improve performance by reducing contention in the memory system.*

1. Introduction

While single-thread performance of commercial superscalar microprocessors is still increasing, a clear trend today is for computer manufacturers to provide multithreaded hardware that strongly encourages software developers to provide explicit parallelism when possible. One important class of parallel computer hardware is the modern *graphics processing unit* (GPU) [22,25]. With contemporary GPUs recently crossing the teraflop barrier [2,34] and specific efforts to make GPUs easier to program for non-graphics applications [1, 29, 33], there is widespread interest in using GPU hardware to accelerate non-graphics applications.

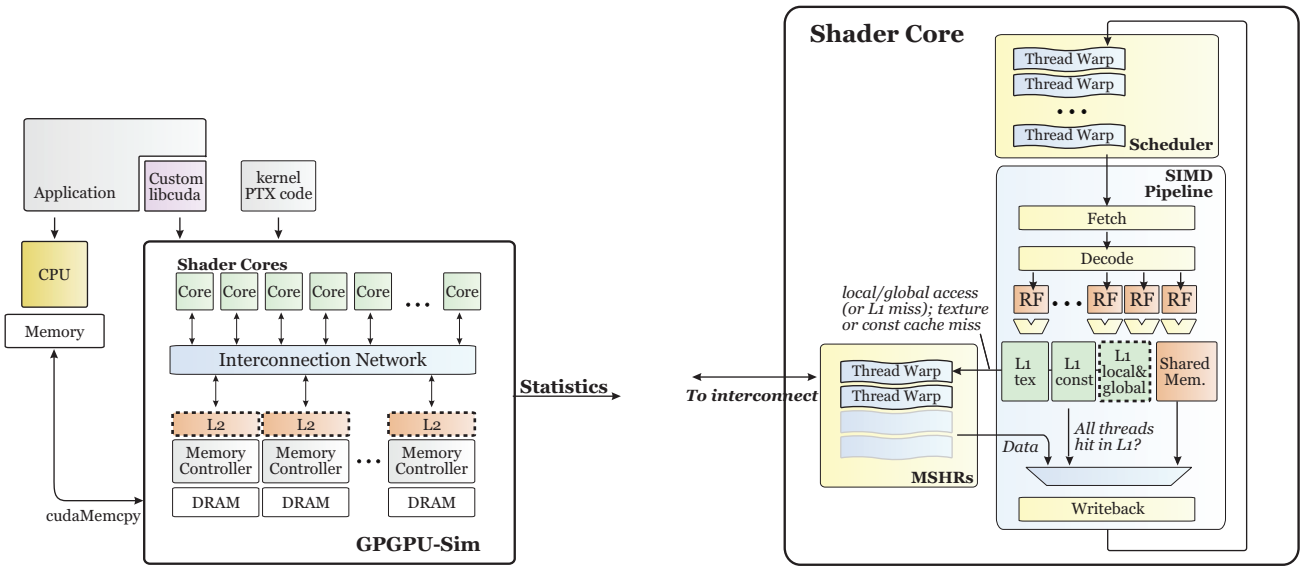
Since its introduction by NVIDIA Corporation in February 2007, the CUDA programming model [29,33] has been used to develop many applications for GPUs. CUDA provides an easy to learn extension of the ANSI C language. The programmer specifies parallel threads, each of which runs *scalar* code. While short vector data types are available, their use by the programmer is not required to achieve peak performance, thus making CUDA a more attractive programming model to those less familiar with traditional data parallel architectures. This execution model has been dubbed a *single instruction, multiple thread* (SIMT) model [22] to distinguish it from the more traditional *single instruction, multiple data* (SIMD) model. As of February 2009, NVIDIA has listed 209 third-party applications on their CUDA Zone website [30]. Of the 136 applications listed with performance claims, 52 are reported to obtain a speedup of 50× or more, and of these 29 are reported to obtain a speedup of 100× or more. As these applications already achieve tremendous benefits, this paper instead focuses on evaluating CUDA applications with reported speedups below 50× since this group of applications appears most in need of software tuning or changes to hardware design.

This paper makes the following contributions:

- It presents data characterizing the performance of twelve existing CUDA applications collected on a research GPU simulator (GPGPU-Sim).
- It shows that the non-graphics applications we study tend to be more sensitive to bisection bandwidth versus latency.
- It shows that, for certain applications, decreasing the number of threads running concurrently on the hardware can improve performance by reducing contention for on-chip resources.
- It provides an analysis of application characteristics including the dynamic instruction mix, SIMD warp branch divergence properties, and DRAM locality characteristics.

We believe the observations made in this paper will provide useful guidance for directing future architecture and software research.

The rest of this paper is organized as follows. In Section 2 we describe our baseline architecture and the microarchitecture design choices that we explore before describing our simulation infrastructure and the benchmarks used in this study.



(a) Overview (b) Detail of Shader Core

Figure 1. Modeled system and GPU architecture [11]. Dashed portions (L1 and L2 for local/global accesses) omitted from baseline.

Our experimental methodology is described in Section 3 and Section 4 presents and analyzes results. Section 5 reviews related work and Section 6 concludes the paper.

2. Design and Implementation

In this section we describe the GPU architecture we simulated, provide an overview of our simulator infrastructure and then describe the benchmarks we selected for our study.

2.1. Baseline Architecture

Figure 1(a) shows an overview of the system we simulated. The applications evaluated in this paper were written using CUDA [29, 33]. In the CUDA programming model, the GPU is treated as a co-processor onto which an application running on a CPU can launch a massively parallel compute kernel. The kernel is comprised of a *grid* of scalar threads. Each thread is given an unique identifier which can be used to help divide up work among the threads. Within a grid, threads are grouped into *blocks*, which are also referred to as *cooperative thread arrays* (CTAs) [22]. Within a single CTA threads have access to a common fast memory called the *shared memory* and can, if desired, perform barrier synchronizations.

Figure 1(a) also shows our baseline GPU architecture. The GPU consists of a collection of small data-parallel compute cores, labeled *shader cores* in Figure 1, connected by an interconnection network to multiple memory modules (each labeled *memory controller*). Each shader core is a unit similar in scope to a *streaming multiprocessor* (SM) in NVIDIA terminology [33]. Threads are distributed to shader cores at the granularity of entire CTAs, while per-CTA resources, such as registers, shared memory space, and thread slots, are not freed until all threads within a CTA have completed execution. If resources permit, multiple CTAs can be assigned to a single shader core, thus sharing a common pipeline for their execution. Our simulator omits graphics specific hardware not exposed to CUDA.

Figure 1(b) shows the detailed implementation of a single shader core. In this paper, each shader core has a SIMD width of 8 and uses a 24-stage, in-order pipeline without forwarding. The 24-stage pipeline is motivated by details in the CUDA Programming Guide [33], which indicates that at least 192 active threads are needed to avoid stalling for true data dependencies between consecutive instructions from a single thread (in the absence of long latency memory operations). We model this pipeline with six logical pipeline stages (fetch, decode, execute, memory1, memory2, writeback) with super-pipelining of degree 4 (memory1 is an empty stage in our model). Threads are scheduled to the SIMD pipeline in a fixed group of 32 threads called a *warp* [22]. All 32 threads in a given warp execute the same instruction with different data values over four consecutive clock cycles in all pipelines (the SIMD cores are effectively 8-wide). We use the immediate post-dominator reconvergence mechanism described in [11] to handle branch divergence where some scalar threads within a warp evaluate a branch as "taken" and others evaluate it as "not taken".

Threads running on the GPU in the CUDA programming model have access to several memory regions (global, local, constant, texture, and shared [33]) and our simulator models accesses to each of these memory spaces. In particular, each shader core has access to a 16KB low latency, highly-banked per-core shared memory; to global texture memory with a per-core texture cache; and to global constant memory with a per-core constant cache. Local and global memory accesses always require off chip memory accesses in our baseline configuration. For the per-core texture cache, we implement a 4D blocking address scheme as described in [14], which essentially permutes the bits in requested addresses to promote

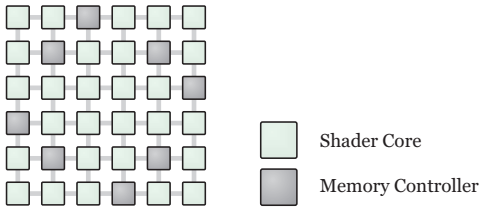


Figure 2. Layout of memory controller nodes in mesh[3]

spatial locality in a 2D space rather than in linear space. For the constant cache, we allow single cycle access as long as all threads in a warp are requesting the same data. Otherwise, a port conflict occurs, forcing data to be sent out over multiple cycles and resulting in pipeline stalls [33]. Multiple memory accesses from threads within a single warp to a localized region are coalesced into fewer wide memory accesses to improve DRAM efficiency[1]. To alleviate the DRAM bandwidth bottleneck that many applications face, a common technique used by CUDA programmers is to load frequently accessed data into the fast on-chip shared memory [40].

Thread scheduling inside a shader core is performed with zero overhead on a fine-grained basis. Every 4 cycles, warps ready for execution are selected by the warp scheduler and issued to the SIMD pipelines in a loose round robin fashion that skips non-ready warps, such as those waiting on global memory accesses. In other words, whenever any thread inside a warp faces a long latency operation, all the threads in the warp are taken out of the scheduling pool until the long latency operation is over. Meanwhile, other warps that are not waiting are sent to the pipeline for execution in a round robin order. The many threads running on each shader core thus allow a shader core to tolerate long latency operations without reducing throughput.

In order to access global memory, memory requests must be sent via an interconnection network to the corresponding memory controllers, which are physically distributed over the chip. To avoid protocol deadlock, we model physically separate send and receive interconnection networks. Using separate logical networks to break protocol deadlock is another alternative, but one we did not explore. Each on-chip memory controller then interfaces to two off-chip GDDR3 DRAM chips[2]. Figure 2 shows the physical layout of the memory controllers in our 6x6 mesh configuration as shaded areas[3]. The address decoding scheme is designed in a way such that successive 2KB DRAM pages [19] are distributed across different banks and different chips to maximize row locality while spreading the load among the memory controllers.

1. When memory accesses within a warp cannot be coalesced into a single memory access, the memory stage will stall until all memory accesses are issued from the shader core. In our design, the shader core can issue a maximum of 1 access every 2 cycles.

2. GDDR3 stands for Graphics Double Data Rate 3 [19]. Graphics DRAM is typically optimized to provide higher peak data bandwidth.

3. Note that with area-array (i.e., "flip-chip") designs it is possible to place I/O buffers anywhere on the die [6].

2.2. GPU Architectural Exploration

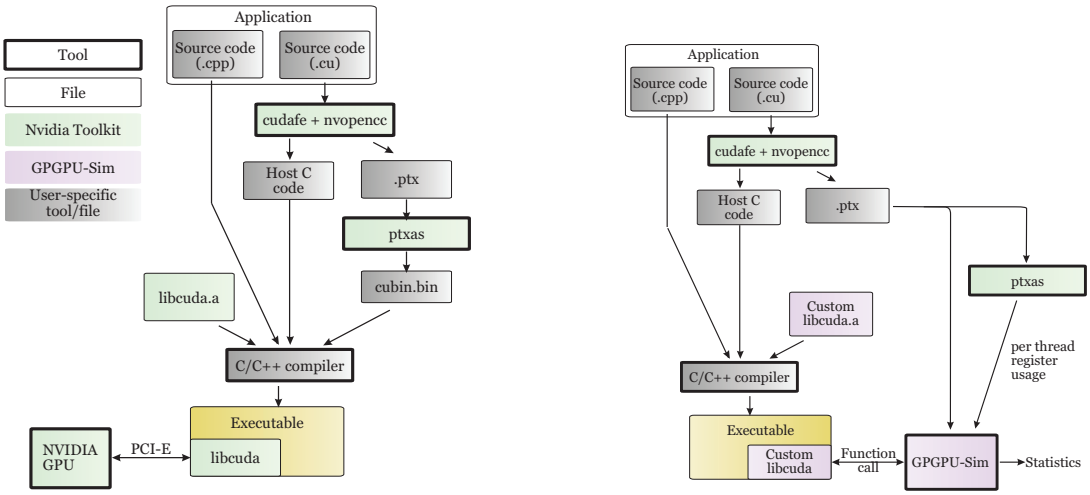
This section describes some of the GPU architectural design options explored in this paper. Evaluations of these design options are presented in Section 4.

2.2.1. Interconnect. The on-chip interconnection network can be designed in various ways based on its cost and performance. Cost is determined by complexity and number of routers as well as density and length of wires. Performance depends on latency, bandwidth and path diversity of the network [9]. (Path diversity indicates the number of routes a message can take from the source to the destination.)

Butterfly networks offer minimal hop count for a given router radix while having no path diversity and requiring very long wires. A crossbar interconnect can be seen as a 1-stage butterfly and scales quadratically in area as the number of ports increase. A 2D torus interconnect can be implemented on chip with nearly uniformly short wires and offers good path diversity, which can lead to a more load balanced network. Ring and mesh interconnects are both special types of torus interconnects. The main drawback of a mesh network is its relatively higher latency due to a larger hop count. As we will show in Section 4, our benchmarks are not particularly sensitive to latency so we chose a mesh network as our baseline while exploring the other choices for interconnect topology.

2.2.2. CTA distribution. GPUs can use the abundance of parallelism in data-parallel applications to tolerate memory access latency by interleaving the execution of warps. These warps may either be from the same CTA or from different CTAs running on the same shader core. One advantage of running multiple smaller CTAs on a shader core rather than using a single larger CTA relates to the use of barrier synchronization points within a CTA [40]. Threads from one CTA can make progress while threads from another CTA are waiting at a barrier. For a given number of threads per CTA, allowing more CTAs to run on a shader core provides additional memory latency tolerance, though it may imply increasing register and shared memory resource use. However, even if sufficient on-chip resources exist to allow more CTAs per core, if a compute kernel is memory-intensive, completely filling up all CTA slots may reduce performance by increasing contention in the interconnection network and DRAM controllers. We issue CTAs in a breadth-first manner across shader cores, selecting a shader core that has a minimum number of CTAs running on it, so as to spread the workload as evenly as possible among all cores.

2.2.3. Memory Access Coalescing. The minimum granularity access for GDDR3 memory is 16 bytes and typically scalar threads in CUDA applications access 4 bytes per scalar thread [19]. To improve memory system efficiency, it thus makes sense to group accesses from multiple, concurrently-issued, scalar threads into a single access to a small, contiguous memory region. The CUDA programming guide indicates that parallel memory accesses from every half-warp of 16



(a) CUDA Flow with GPU Hardware

(b) GPGPU-Sim

Figure 3. Compilation Flow for GPGPU-Sim from a CUDA application in comparison to the normal CUDA compilation flow.

threads can be coalesced into fewer wide memory accesses if they all access a contiguous memory region [33]. Our baseline models similar *intra*-warp memory coalescing behavior (we attempt to coalesce memory accesses from all 32 threads in a warp).

A related issue is that since the GPU is heavily multi-threaded a balanced design must support many outstanding memory requests at once. While microprocessors typically employ *miss-status holding registers* (MSHRs) [21] that use associative comparison logic merge simultaneous requests for the same cache block, the number of outstanding misses that can be supported is typically small (e.g., the original Intel Pentium 4 used four MSHRs [16]). One way to support a far greater number of outstanding memory requests is to use a FIFO for outstanding memory requests [17]. Similarly, our baseline does not attempt to eliminate multiple requests for the same block of memory on cache misses or local/global memory accesses. However, we also explore the possibility of improving performance by coalescing read memory requests from later warps that require access to data for which a memory request is already in progress due to another warp running on the same shader core. We call this *inter*-warp memory coalescing. We observe that inter-warp memory coalescing can significantly reduce memory traffic for applications that contain data dependent accesses to memory. The data for inter-warp merging quantifies the benefit of supporting large capacity MSHRs that can detect a secondary access to an outstanding request [45].

2.2.4. Caching. While coalescing memory requests captures spatial locality among threads, memory bandwidth requirements may be further reduced with caching if an application contains temporal locality or spatial locality within the access pattern of individual threads. We evaluate the performance impact of adding first level, per-core L1 caches for local and global memory access to the design described in Section 2.1. We also evaluate the effects of adding a shared L2 cache on the memory side of the interconnection network at the memory controller. While threads can only read from texture and constant memory, they can both read and write to local and global memory. In our evaluation of caches for local and global memory we model non-coherent caches. (Note that threads from different CTAs in the applications we study do not communicate through global memory.)

2.3. Extending GPGPU-Sim to Support CUDA

We extended *GPGPU-Sim*, the cycle-accurate simulator we developed for our earlier work [11]. GPGPU-Sim models various aspects of a massively parallel architecture with highly programmable pipelines similar to contemporary GPU architectures. A drawback of the previous version of GPGPU-Sim was the difficult and time-consuming process of converting/parallelizing existing applications [11]. We overcome this difficulty by extending GPGPU-Sim to support the CUDA Parallel Thread Execution (PTX) [35] instruction set. This enables us to simulate the numerous existing, optimized CUDA applications on GPGPU-Sim. Our current simulator infrastructure runs CUDA applications without source code modifications on Linux based platforms, but does require access to the application's source code. To build a CUDA application for our simulator, we replace the *common.mk* makefile used in the CUDA SDK with a version that builds the application to run on our microarchitecture simulator (while other more complex build scenarios may require more complex makefile changes).

Figure 3 shows how a CUDA application can be compiled for simulation on GPGPU-Sim and compares this compilation flow to the normal CUDA compilation flow [33]. Both compilation flows use *cudafe* to transform the source code of a CUDA application into host C code running on the CPU and device C code running on the GPU. The GPU C code is then compiled into PTX assembly (labeled ".ptx" in Figure 3) by *nvopencc*, an open source compiler provided by NVIDIA based on Open64 [28, 36]. The PTX assembler (*ptxas*) then assembles the PTX assembly code into the target GPU's native

ISA (labeled "cubin.bin" in Figure 3(a)). The assembled code is then combined with the host C code and compiled into a single executable linked with the CUDA Runtime API library (labeled "libcuda.a" in Figure 3) by a standard C compiler. In the normal CUDA compilation flow (used with NVIDIA GPU hardware), the resulting executable calls the CUDA Runtime API to set up and invoke compute kernels onto the GPU via the NVIDIA CUDA driver.

When a CUDA application is compiled to use GPGPU-Sim, many steps remain the same. However, rather than linking against the NVIDIA supplied libcuda.a binary, we link against our own libcuda.a binary. Our libcuda.a implements "stub" functions for the interface defined by the header files supplied with CUDA. These stub functions set up and invoke simulation sessions of the compute kernels on GPGPU-Sim (as shown in Figure 3(b)). Before the first simulation session, GPGPU-Sim parses the text format PTX assembly code generated by nvopencc to obtain code for the compute kernels. Because the PTX assembly code has no restriction on register usage (to improve portability between different GPU architectures), *nvopencc* performs register allocation using far more registers than typically required to avoid spilling. To improve the realism of our performance model, we determine the register usage per thread and shared memory used per CTA using *ptxas*[4]. We then use this information to limit the number of CTAs that can run concurrently on a shader core. The GPU binary (cubin.bin) produced by *ptxas* is not used by GPGPU-Sim. After parsing the PTX assembly code, but before beginning simulation, GPGPU-Sim performs an immediate post-dominator analysis on each kernel to annotate branch instructions with reconvergence points for the stack-based SIMD control flow handling mechanism described by Fung et al. [11]. During a simulation, a PTX functional simulator executes instructions from multiple threads according to their scheduling order as specified by the performance simulator. When the simulation completes, the host CPU code is then allowed to resume execution. In our current implementation, host code runs on a normal CPU, thus our performance measurements are for the GPU code only.

2.4. Benchmarks

Our benchmarks are listed in Table 1 along with the main application properties, such as the organization of threads into CTAs and grids as well as the different memory spaces on the GPU exploited by each application. Multiple entries separated by semi-colons in the grid and CTA dimensions indicate the application runs multiple kernels.

For comparison purposes we also simulated the following benchmarks from NVIDIA's CUDA software development kit (SDK) [32]: Black-Scholes Option Pricing, Fast Walsh Transform, Binomial Option Pricing, Separable Convolution, 64-bin Histogram, Matrix Multiply, Parallel Reduction, Scalar Product, Scan of Large Arrays, and Matrix Transpose. Due to space limitations, and since most of these benchmarks already perform well on GPUs, we only report details for Black-Scholes (BLK), a financial options pricing application, and Fast Walsh Transform (FWT), widely used in signal and image processing and compression. We also report the harmonic mean of all SDK applications simulated, denoted as *SDK* in the data bar charts in Section 4.

Below, we describe the CUDA applications not in the SDK that we use as benchmarks in our study. These applications were developed by the researchers cited below and run unmodified on our simulator.

AES Encryption (AES) [24] This application, developed by Manavski [24], implements the Advanced Encryption Standard (AES) algorithm in CUDA to encrypt and decrypt files. The application has been optimized by the developer so that constants are stored in constant memory, the expanded key stored in texture memory, and the input data processed in shared memory. We encrypt a 256KB picture using 128-bit encryption.

Graph Algorithm: Breadth-First Search (BFS) [15] Developed by Harish and Narayanan [15], this application performs breadth-first search on a graph. As each node in the graph is mapped to a different thread, the amount of parallelism in this applications scales with the size of the input graph. BFS suffers from performance loss due to heavy global memory traffic and branch divergence. We perform breadth-first search on a random graph with 65,536 nodes and an average of 6 edges per node.

Coulombic Potential (CP) [18,41] CP is part of the Parboil Benchmark suite developed by the IMPACT research group at UIUC [18,41]. CP is useful in the field of molecular dynamics. Loops are manually unrolled to reduce loop overheads and the point charge data is stored in constant memory to take advantage of caching. CP has been heavily optimized (it has been shown to achieve a 647× speedup versus a CPU version [40]). We simulate 200 atoms on a grid size of 256×256.

gpuDG (DG) [46] gpuDG is a discontinuous Galerkin time-domain solver, used in the field of electromagnetics to calculate radar scattering from 3D objects and analyze wave guides, particle accelerators, and EM compatibility [46]. Data is loaded into shared memory from texture memory. The inner loop consists mainly of matrix-vector products. We use the 3D version with polynomial order of N=6 and reduce time steps to 2 to reduce simulation time.

3D Laplace Solver (LPS) [12] Laplace is a highly parallel finance application [12]. As well as using shared memory, care was taken by the application developer to ensure coalesced global memory accesses. We observe that this benchmark suffers some performance loss due to branch divergence. We run one iteration on a 100x100x100 grid.

LIBOR Monte Carlo (LIB) [13] LIBOR performs Monte

4. By default, the version of ptxas in CUDA 1.1 appears to attempt to avoid spilling registers provided the number of registers per thread is less than 128 and none of the applications we studied reached this limit. Directing ptxas to further restrict the number of registers leads to an increase in local memory usage above that explicitly used in the PTX assembly, while increasing the register limit does not increase the number of registers used.

Table 1. Benchmark Properties

Benchmark	Abr.	Grid Dimensions	CTA Dimensions	Concurrent CTAs/core	Total Threads	Instruction Count	Shared Memory?	Constant Memory?	Texture Memory?	Barriers?
AES Cryptography [24]	AES	(257,1,1)	(256,1,1)	2	65792	28M	Yes	Yes	1D	Yes
Graph Algorithm: Breadth First Search [15]	BFS	(128,1,1)	(512,1,1)	4	65536	17M	No	No	No	No
Coulombic Potential [18,41]	CP	(8,32,1)	(16,8,1)	8	32768	126M	No	Yes	No	No
gpuDG [46]	DG	(268,1,1); (268,1,1); (603,1,1)	(84,1,1); (112,1,1); (256,1,1)	5 6 4	22512; 30016; 154368	596M	Yes	No	1D	Yes Yes No
3D Laplace Solver [12]	LPS	(4,25,1)	(32,4,1)	6	12800	82M	Yes	No	No	Yes
LIBOR Monte Carlo [13]	LIB	(64,1,1)	(64,1,1)	8	4096	907M	No	Yes	No	No
MUMmerGPU [42]	MUM	(782,1,1)	(64,1,1)	3	50000	77M	No	No	2D	No
Neural Network Digit Recognition [5]	NN	(6,28,1); (50,28,1); (100,28,1); (10,28,1)	(13,13,1); (5,5,1); (1,1,1); (1,1,1)	5 8 8 8	28392; 35000; 2800; 280	68M	No	No	No	No No No No
N-Queens Solver [37]	NQU	(223,1,1)	(96,1,1)	1	21408	2M	Yes	No	No	Yes
Ray Tracing [26]	RAY	(16,32,1)	(16,8,1)	3	65536	71M	No	Yes	No	Yes
StoreGPU [4]	STO	(384,1,1)	(128,1,1)	1	49152	134M	Yes	No	No	No
Weather Prediction [27]	WP	(9,8,1)	(8,8,1)	3	4608	215M	No	No	No	No
Black-Scholes option pricing [32]	BLK	(256,1,1)	(256,1,1)	3	65536	236M	No	No	No	No
Fast Walsh Transform [32]	FWT	(512,1,1); (256,1,1);	(256,1,1); (512,1,1)	4 2	131072; 131072	240M	Yes	No	No	Yes Yes

Carlo simulations based on the London Interbank Offered Rate Market Model [13]. Each thread reads a large number of variables stored in constant memory. We find the working set for constant memory fits inside the 8KB constant cache per shader core that we model. However, we find memory bandwidth is still a bottleneck due to a large fraction of local memory accesses. We use the default inputs, simulating 4096 paths for 15 options.

MUMmerGPU (MUM) [42] MUMmerGPU is a parallel pairwise local sequence alignment program that matches query strings consisting of standard DNA nucleotides (A,C,T,G) to a reference string for purposes such as genotyping, genome resequencing, and metagenomics [42]. The reference string is stored as a suffix tree in texture memory and has been arranged to exploit the texture cache's optimization for 2D locality. Nevertheless, the sheer size of the tree means high cache miss rates, causing MUM to be memory bandwidth-bound. Since each thread performs its own query, the nature of the search algorithm makes performance also susceptible to branch divergence. We use the first 140,000 characters of the Bacillus anthracis str. Ames genome as the reference string and 50,000 25-character queries generated randomly using the complete genome as the seed.

Neural Network (NN) [5] Neural network uses a convolutional neural network to recognize handwritten digits [5]. Pre-determined neuron weights are loaded into global memory along with the input digits. We modified the original source code to allow recognition of multiple digits at once to increase parallelism. Nevertheless, the last two kernels utilize blocks of only a single thread each, which results in severe under-utilization of the shader core pipelines. We simulate recognition of 28 digits from the Modified National Institute of Standards Technology database of handwritten digits.

N-Queens Solver (NQU) [37] The N-Queen solver tackles a classic puzzle of placing N queens on a NxN chess board such that no queen can capture another [37]. It uses a simple backtracking algorithm to try to determine all possible solutions. The search space implies that the execution time grows exponentially with N. Our analysis shows that most of the computation is performed by a single thread, which explains the low IPC. We simulate N=10.

Ray Tracing (RAY) [26] Ray-tracing is a method of rendering graphics with near photo-realism. In this implementation, each pixel rendered corresponds to a scalar thread in CUDA [26]. Up to 5 levels of reflections and shadows are taken into account, so thread behavior depends on what object the ray hits (if it hits any at all), making the kernel susceptible to branch divergence. We simulate rendering of a 256x256 image.

StoreGPU (STO) [4] StoreGPU is a library that accelerates hashing-based primitives designed for middleware [4]. We chose to use the sliding-window implementation of the MD5 algorithm on an input file of size 192KB. The developers minimize off-chip memory traffic by using the fast shared memory. We find STO performs relatively well.

Weather Prediction (WP) [27] Numerical weather prediction uses the GPU to accelerate a portion of the Weather Research and Forcast model (WRF), which can model and predict condensation, fallout of various precipitation and related thermodynamic effects of latent heat release [27]. The kernel has been optimized to reduce redundant memory transfer by storing the temporary results for each altitude level in the cell in registers. However, this requires a large amount of registers, thus limiting the maximum allowed number of threads per shader core to 192, which is not enough to cover global and local memory access latencies. We simulate the kernel using the default test sample for 10 timesteps.

3. Methodology

Table 2 shows the simulator's configuration. Rows with multiple entries show the different configurations that we have simulated. Bold values show our baseline. To simulate the mesh network, we used a detailed interconnection network model, incorporating the configurable interconnection network

Table 2. Hardware Configuration

Number of Shader Cores	**28**
Warp Size	**32**
SIMD Pipeline Width	**8**
Number of Threads / Core	256 / 512 / **1024** / 1536 / 2048
Number of CTAs / Core	2 / 4 / **8** / 12 / 16
Number of Registers / Core	4096 / 8192 / **16384** / 24576 / 32768
Shared Memory / Core (KB)	4/8/**16**/24/32 (16 banks, 1 access/cycle/bank)[5]
Constant Cache Size / Core	**8KB** (2-way set assoc. 64B lines LRU)
Texture Cache Size / Core	**64KB** (2-way set assoc. 64B lines LRU)
Number of Memory Channels	**8**
L1 Cache	**None** / 16KB / 32KB / 64KB 4-way set assoc. 64B lines LRU
L2 Cache	**None** / 128KB / 256KB 8-way set assoc. 64B lines LRU
GDDR3 Memory Timing	t_{CL}=**9**, t_{RP}=**13**, t_{RC}=**34** t_{RAS}=**21**, t_{RCD}=**12**, t_{RRD}=**8**
Bandwidth per Memory Module	**8** (Bytes/Cycle)
DRAM request queue capacity	**32** / 128
Memory Controller	**out of order (FR-FCFS)** / in order (FIFO) [39]
Branch Divergence Method	**Immediate Post Dominator** [11]
Warp Scheduling Policy	**Round Robin among ready warps**

Table 3. Interconnect Configuration

Topology	**Mesh** / Torus / Butterfly / Crossbar / Ring
Routing Mechanism	**Dimension Order** / Destination Tag
Routing delay	**1**
Virtual channels	**2**
Virtual channel buffers	**4**
Virtual channel allocator	**iSLIP** / PIM
Alloc iters	**1**
VC alloc delay	**1**
Input Speedup	**2**
Flit size (Bytes)	8 / **16** / 32 / 64

simulator introduced by Dally et al. [9]. Table 3 shows the interconnection configuration used in our simulations.

We simulate all benchmarks to completion to capture all the distinct phases of each kernel in the benchmarks, especially the behavior at the tail end of the kernels, which can vary drastically compared to the beginning. If the kernels are relatively short and are frequently launched, the difference in performance when not simulating the benchmark to completion can be significant.

We note that the breadth-first CTA distribution heuristic described in Section 2.2.2 can occasionally lead to counter-intuitive performance results due to a phenomina we will refer to as *CTA load imbalance*. This CTA load imbalance can occur when the number of CTAs in a grid exceeds the number that can run concurrently on the GPU. For example, consider six CTAs on a GPU with two shader cores where at most two CTAs can run concurrently on a shader core. Assume running one CTA on one core takes time T and running two CTAs on one core takes time 2T (e.g., no off-chip accesses and six or more warps per CTA—enough for one CTA to keep our 24 stage pipeline full). If each CTA in isolation takes equal time T, total time is 3T (2T for the first round of four CTAs plus T for the last two CTAs which run on separate shader cores). Suppose we introduce an enhancement that causes CTAs to run in time 0.90T to 0.91T when run alone (i.e., faster). If both CTAs on the first core now finish ahead of those on the other core at time 1.80T versus 1.82T, then our CTA distributor will issue the remaining 2 CTAs onto the first core, causing the load imbalance. With the enhancement, this actually causes an overall slowdown since now 4 CTAs need to be completed by the first core, requiring a total time of at least 3.6T. We carefully verified that this behavior occurs by plotting the distribution of CTAs to shader cores versus time for both configurations being compared. This effect would be less significant in a real system with larger data sets and therefore grids with a larger number of CTAs. Rather than attempt to eliminate the effect by modifying the scheduler (or the benchmarks) we simply note where it occurs.

In Section 4.7 we measure the impact of running greater or fewer numbers of threads. We model this by varying the number of concurrent CTAs permitted by the shader cores, which is possible by scaling the amount of on-chip resources available to each shader core. There are four such resources: Number of concurrent threads, number of registers, amount of shared memory, and number of CTAs. The values we use are shown in Table 2. The amount of resources available per shader core is a configurable simulation option, while the amount of resources required by each kernel is extracted using *ptxas*.

4. Experimental Results

In this section we evaluate the designs introduced in Section 2. Figure 4.1 shows the classification of each benchmark's instruction type (dynamic instruction frequency). The *Fused Multiply-Add* and *ALU Ops (other)* sections of each bar show the proportion of total ALU operations for each benchmark (which varies from 58% for NQU to 87% for BLK). Only DG, CP and NN utilize the Fused Multiply-Add operations extensively. *Special Function Unit* (SFU)[6] instructions are also only used by a few benchmarks. CP is the only benchmark that has more than 10% SFU instructions.

The memory operations portion of Figure 4.1 is further broken down in terms of type as shown in Figure 5. Note that "param" memory refers to parameters passed through the GPU kernel call, which we always treat as cache hits. There is a large variation in the memory instruction types used among benchmarks: for CP over 99% of accesses are to constant memory while for NN most accesses are to global memory.

4.1. Baseline

We first simulated our baseline GPU configuration with the bolded parameters shown in Table 2. Figure 6 shows the performance of our baseline configuration (for the GPU only) measured in terms of scalar instructions per cycle (IPC). For comparison, we also show the performance assuming a perfect memory system with zero memory latency. Note that the maximum achievable IPC for our configuration is

5. We model the shared memory to service up to 1 access per cycle in each bank. This may be more optimistic than what can be inferred from the CUDA Programming Guide (1 access/2 cycles/bank) [33].

6. The architecture of the NVIDIA GeForce 8 Series GPUs includes a *special function unit* for transcendental and attribute interpolation operations [22]. We include the following PTX instructions in our SFU Ops classification: `cos`, `ex2`, `lg2`, `rcp`, `rsqrt`, `sin`, `sqrt`.

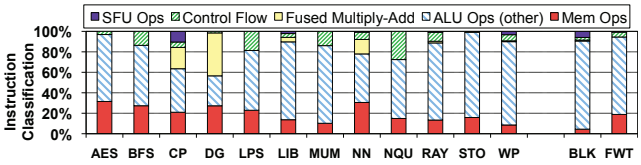


Figure 4. Instruction Classification.

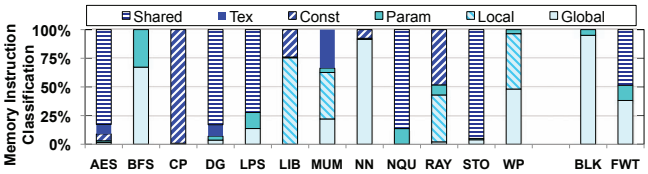


Figure 5. Memory Instructions Breakdown

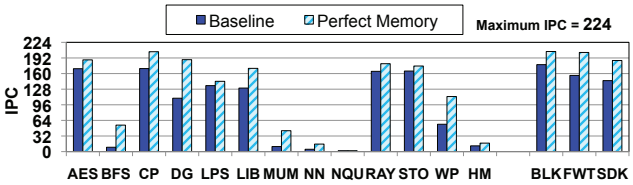


Figure 6. Baseline performance (HM=Harmonic Mean)

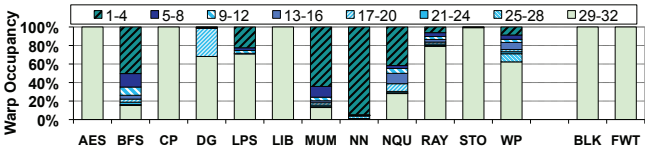


Figure 7. Warp Occupancy

224 (28 shader cores x 8-wide pipelines). We also validated our simulator against an Nvidia Geforce 8600GTS (a "low end" graphics card) by configuring our simulator to use 4 shaders and two memory controllers. The IPC of the GPU hardware, as shown in Figure 8(a), was estimated by dividing the dynamic instruction count measured (in PTX instructions) in simulation by the product of the measured runtime on hardware and the shader clock frequency [31]. Figure 8(b) shows the scatter plot of IPC obtained with our simulations mimicking the 8600GTS normalized to the theoretical peak IPC versus the normalized IPC data measured using the 8600GTS. The correlation coefficient was calculated to be 0.899. One source of difference, as highlighted by the data for CP which actually achieves a normalized IPC over 1, is likely due to compiler optimizations in *ptxas* which may reduce the instruction count on real hardware[7]. Overall, the data shows that applications that perform well in real GPU hardware perform well in our simulator and applications that perform poorly in real GPU hardware also perform poorly in our simulator. In the following sections, we explore reasons why some benchmarks do not achieve peak performance.

4.2. Branch Divergence

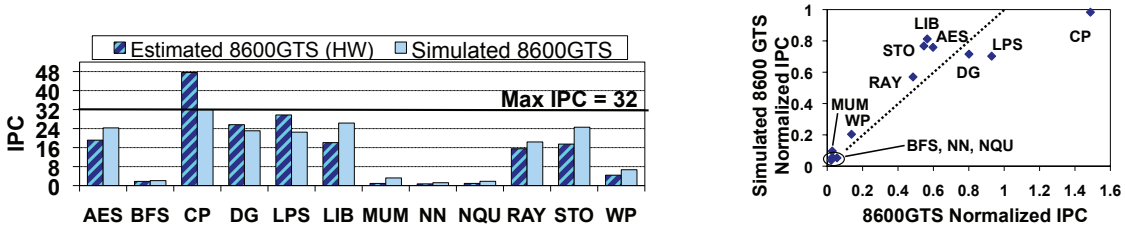
Branch divergence was highlighted by Fung et al. as a major source of performance loss for multithreaded SIMD architectures with intra-warp branch divergence [11]. Figure 7 shows warp occupancies (number of active threads in an issued warp) over the entire runtime of the benchmarks. This metric can be seen as a measure of how much GPU throughput potential is wasted due to unfilled warps. The control flow portion of the bars in Figure 4 shows that BFS, LPS, and NQU contain from 13% to 28% control flow operations. However, intensive control flow does not necessarily translate into high branch divergence; it depends more on whether or not all threads in a warp branch in the same direction. NN has the lowest warp occupancy while it contains only 7% control flow operations. On the other hand, LPS with 19% control flow has full warp occupancy 75% of the time. It is best to analyze Figure 7 with Table 1 in mind, particularly in the case of NN. In NN, two of the four kernels have only a single thread in a block and they take up the bulk of the execution time, meaning that the unfilled warps in NN are not due to branch divergence. Some benchmarks (such as AES, CP, LIB, and STO) do not incur significant branch divergence, while others do. MUM experiences severe performance loss in particular because more than 60% of its warps have less than 5 active threads. BFS also performs poorly since threads in adjacent nodes in the graph (which are grouped into warps) behave differently, causing more than 75% of its warps to have less than 50% occupancy. Warp occupancy for NN and NQU is low due to large portions of the code being spent in a single thread.

4.3. Interconnect Topology

Figure 9 shows the speedup of various interconnection network topologies compared to a mesh with 16 Byte channel bandwidth. On average our baseline mesh interconnect performs comparable to a crossbar with input speedup of two for the workloads that we consider. We also have evaluated two torus topologies: "Torus - 16 Byte Channel BW", which has double the bisection bandwidth of the baseline "Mesh" (a determining factor in the implementation cost of a network); and "Torus - 8 Byte Channel BW", which has the same bisection bandwidth as "Mesh". The "Ring" topology that we evaluated has a channel bandwidth of 64. The "Crossbar" topology has a *parallel iterative matching* (PIM) allocator as opposed to an iSLIP allocator for other topologies. The two-stage butterfly and crossbar employ destination tag routing while others use dimension-order routing. The ring and mesh networks are the simplest and least expensive networks to build in terms of area.

As Figure 9 suggests, most of the benchmarks are fairly insensitive to the topology used. In most cases, a change in topology results in less than 20% change in performance from the baseline, with the exception of the Ring and Torus with 8 Byte channel bandwidth. BLK experiences a performance gain with Ring due to the CTA load imbalance phenomena described in Section 3. BLK has 256 CTAs. For the Ring configuration, the number of CTAs executed per shader core varies from 9 to 10. However, for the baseline configuration,

7. We only simulate the input PTX code which, in CUDA, *ptxas* then assembles into a proprietary binary format that we are unable to simulate.



(a) (b)

Figure 8. Performance Comparison with 8600GTS

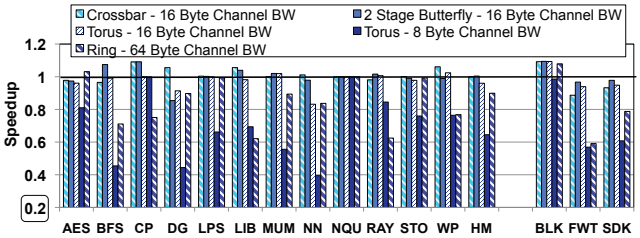


Figure 9. Interconnection Network Topology

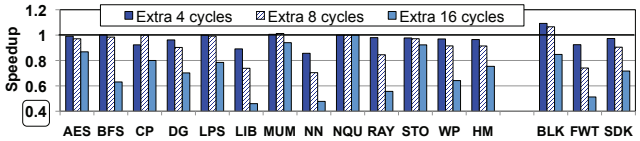


Figure 10. Interconnection Network Latency Sensitivity

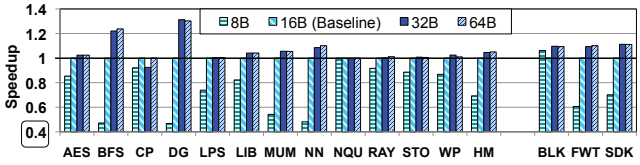


Figure 11. Interconnection Network Bandwidth Sensitivity

one of the shader cores is assigned 11 CTAs due to small variations in time coupled with our greedy CTA distribution heuristic. When more CTAs run on a shader core, all CTAs on that shader core take longer to complete, resulting in a performance loss for the baseline configuration for BLK.

As we will show in the next section, one of the reasons why different topologies do not change the performance of most benchmarks dramatically is that the benchmarks are not sensitive to small variations in latency, as long as the interconnection network provides sufficient bandwidth.

4.4. Interconnection Latency and Bandwidth

Figure 10 shows the IPC results for various mesh network router latencies. Without affecting peak throughput, we add an extra pipelined latency of 4, 8, or 16 cycles to each router on top of our baseline router's 2-cycle latency. An extra 4 cycle latency per router is easily tolerated for most benchmarks and causes only 3.5% performance loss when harmonically averaged across all benchmarks. BLK and CP experience a performance gain due to the CTA load imbalance phenomena described in Section 3. With 8 extra cycles of latency per router, the performance degradation is noticeable (slowdown by 9% on average) and becomes much worse (slowdown by 25% on average) at 16 cycles of extra latency. Note that these experiments are only intended to show the latency sensitivity of benchmarks.

We also modify the mesh interconnect bandwidth by varying the channel bandwidth from 8 bytes to 64 bytes. Figure 11 shows that halving the channel bandwidth from 16 bytes to 8 bytes has a noticeable negative impact on most benchmarks, but doubling and quadrupling channel bandwidth only results in a small gain for a few workloads i.e., BFS and DG.

DG is the most bandwidth sensitive workload, getting a 31% speedup and 53% slowdown for flit sizes of 32 and 8 respectively. The reason why DG does not exhibit further speedup with flit size of 64 is because at this point, the interconnect has already been overprovisioned. Our analysis shows that for the baseline configuration, the input port to the return interconnect from memory to the shader cores is stalled 16% of the time on average. Increasing the flit size to 32 completely eliminates these stalls, which is why there is no further speedup for interconnect flit size of 64. Note that our memory read request packet sizes are 8 bytes, allowing them to be sent to the memory controllers in a single flit for all of the configurations shown in Figure 11.

Overall, the above data suggests that performance is more sensitive to interconnect bandwidth than to latency for the non-graphics workloads that we study. In other words, restricting channel bandwidth causes the interconnect to become a bottleneck.

4.5. DRAM Utilization and Efficiency

In this section we explore the impact that memory controller design has on performance. Our baseline configuration uses an *Out-of-Order* (OoO) *First-Ready First-Come First-Serve* (FR-FCFS) [39] memory controller with a capacity of 32 memory requests. Each cycle, the OoO memory controller prioritizes memory requests that hit an open row in the DRAM over requests that require a precharge and activate to open a new row. Against this baseline, we compare a simple *First-In First-Out* (FIFO) memory controller that services memory requests in the order that they are received, as well as a more aggressive FR-FCFS OoO controller with an input buffer capacity of 128 (OoO128). We measure two metrics besides performance: The first is *DRAM efficiency*, which is the percentage of time spent sending data across the pins of DRAM over the time when there are any memory requests being serviced or pending in the memory controller input buffer; the second is *DRAM*

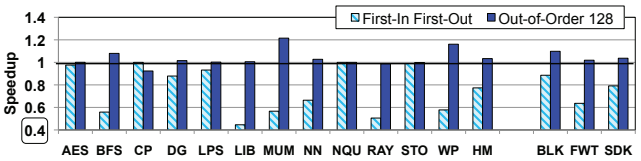


Figure 12. Impact of DRAM memory controller optimizations

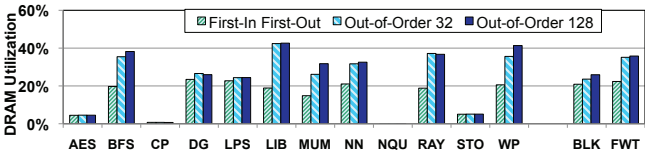


Figure 13. DRAM Utilization

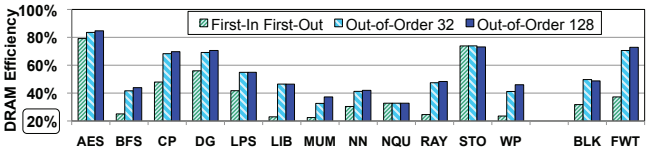


Figure 14. DRAM Efficiency

utilization, which is the percentage of time spent sending data across the DRAM data pins over the entire kernel execution time. These two measures can differ if an application contains GPU computation phases during which it does not access DRAM (e.g., if it has been heavily optimized to use "shared memory").

Figure 12 compares the performance of our baseline to FIFO and OoO128. We observe that AES, CP, NQU, and STO exhibit almost no slowdown for FIFO. Figure 14 shows AES and STO obtain over 75% DRAM efficiency. Close examination reveals that at any point in time all threads access at most two rows in each bank of each DRAM, meaning that a simple DRAM controller policy suffices. Furthermore, Figure 13 shows that AES and STO have low DRAM utilization despite the fact that they process large amounts of data. Both these applications make extensive use of shared memory (see Figure 5). NQU and CP have very low DRAM utilization, making them insensitive to memory controller optimizations (CP slows down for OoO128 due to variations in CTA load distribution). Performance is reduced by over 40% when using FIFO for BFS, LIB, MUM, RAY, and WP. These benchmarks all show drastically reduced DRAM efficiency and utilization with this simple controller.

4.6. Cache Effects

Figure 15 shows the effects on IPC of adding caches to the system. The first 3 bars show the relative speedup of adding a 16KB, 32KB or 64KB cache to each shader core. The last two bars show the effects of adding a 128KB or 256KB L2 cache to each memory controller in addition to a 64KB L1 cache in each shader. CP, RAY and FWT exhibit a slowdown with the addition of L1 caches. Close examination shows that CP experiences a slowdown due to the CTA load imbalance phenomena described in Section 3, whereas RAY and FWT experience a slowdown due to the way write misses and evictions of dirty lines are handled. For the baseline (without

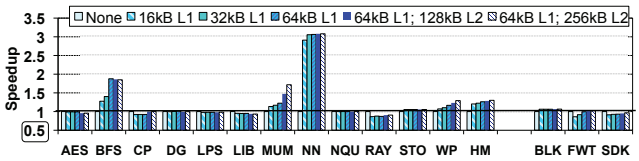


Figure 15. Effects of adding an L1 or L2

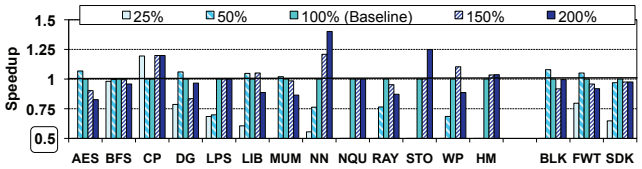


Figure 16. Effects of varying number of CTAs

caches for local/global accesses) writes to memory only cause the memory controller to read data out of DRAM if a portion of a 16B is modified due to writes that are not coalesced. When caches are added for local and global accesses, for simplicity, a write miss prevents a warp from being scheduled until the cache block is read from DRAM. Furthermore, when a dirty line is evicted, the entire line is written to DRAM even if only a single word of that line is modified. We leave exploration of better cache policies to future work.

Benchmarks that make extensive use of "shared memory", namely AES, LPS, NQU, and STO, do not respond significantly to caches. On the other hand, BFS and NN have the highest ratio of global memory instructions to all instructions (at 19% and 27% respectively) and so they experience the highest speedup among workloads.

4.7. Are More Threads Better?

Increasing the number of simultaneously running threads can improve performance by having a greater ability to hide memory access latencies. However, doing so may result in higher contention for shared resources, such as interconnect and memory. We explored the effects of varying the resources that limit the number of threads and hence CTAs that can run concurrently on a shader core, without modifying the source code for the benchmarks. We vary the amount of registers, shared memory, threads, and CTAs between 25% to 200% of those available to the baseline. The results are shown in Figure 16. For the baseline configuration, some benchmarks are already resource-constrained to only 1 or 2 CTAs per shader core, making them unable to run using a configuration with less resources. We do not show bars for configurations that for this reason are unable to run. NQU shows little change when varying the number of CTAs since it has very few memory operations. For LPS, NN, and STO, performance increases as more CTAs per core are used. LPS cannot take advantage of additional resources beyond the baseline (100%) because all CTAs in the benchmark can run simultaneously for the baseline configuration. Each CTA in STO uses all the shared memory in the baseline configuration, therefore increasing shared memory by half for the 150% configuration results in no increase in the number of concurrently running CTAs. AES and MUM show clear trends in *decreasing* performance

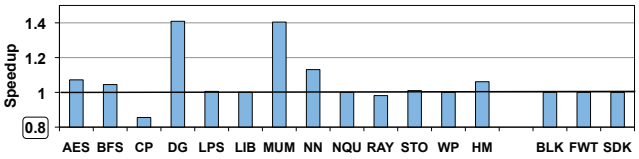


Figure 17. Inter-Warp Memory Coalescing

as the number of CTAs *increases*. We observed that with more concurrent CTAs, AES and MUM experience increased contention in the memory system resulting in 8.6× and 5.4× worse average memory latency, respectively comparing 200% resources vs. 50%. BFS, RAY, and WP show distinct optima in performance when the CTA limit is at 100%, 100%, and 150% of the baseline shader, respectively. Above these limits, we observe DRAM efficiencies decrease and memory latencies increase, again suggesting increased contention in the memory system. For configuration with limits below the optima, the lack of warps to hide memory latencies reduces performance. CP suffers CTA load imbalance due to CTA scheduling for the 50% and 100% configurations. Similarly, DG suffers CTA load imbalance in the 150% configuration.

Given the widely-varying workload-dependent behavior, always scheduling the maximal number of CTAs supported by a shader core is not always the best scheduling policy. We leave for future work the design of dynamic scheduling algorithms to adapt to the workload behavior.

4.8. Memory Request Coalescing

Figure 17 presents data showing the improvement in performance when enabling inter-warp memory coalescing described in Section 2.2.3. The harmonic mean speedup versus intra-warp coalescing is 6.1%. CP's slowdown with inter-warp coalescing is due to load imbalance in CTA distribution. Accesses in AES, DG, and MUM are to data dependent locations which makes it harder to use the explicitly managed shared memory to capture locality. These applications use the texture cache to capture this locality and inter-warp merging effectively eliminates additional requests for the same cache block at the expense of associative search hardware.

It is interesting to observe that the harmonic mean speedup of the CUDA SDK benchmarks is less than 1%, showing that these highly optimized benchmarks do not benefit from inter-warp memory coalescing. Their careful program optimizations ensure less redundancy in memory requests generated by each thread.

5. Related Work

Existing graphics-oriented GPU simulators include Qsilver [43], which does not model programmable shaders, and ATTILLA [10], which focuses on graphics specific features. Ryoo et al. [41] use CUDA to speedup a variety of relatively easily parallelizable scientific applications. They explore the use of conventional code optimization techniques and take advantage of the different memory types available on NVIDIA's 8800GTX to obtain speedup. While their analysis is performed by writing and optimizing applications to run on actual CUDA hardware, we use our novel performance simulator to observe the detailed behavior of CUDA applications upon varying architectural parameters.

There have been acceleration architectures proposed besides the GPU model that we analyze in this paper. Mahesri et al. introduce a class of applications for visualization, interaction, and simulation [23]. They propose using an accelerator architecture (xPU) separate from the GPU to improve performance of their benchmark suite. The Cell processor [7, 38] is a hardware architecture that can function like a stream processor with appropriate software support. It consists of a controlling processor and a set of SIMD co-processors each with independent program counters and instruction memory. Merrimac [8] and Imagine [3] are both streaming processor architectures developed at Stanford.

Khailany et al. [20] explore VLSI costs and performance of a stream processor as the number of streaming clusters and ALUs per cluster scales. They use an analytical cost model. The benchmarks they use also have a high ratio of ALU operations to memory references, which is a property that eases memory requirements of streaming applications. The UltraSPARC T2 [44] microprocessor is a multithreading, multicore CPU which is a member of the SPARC family and comes in 4, 6, and 8 core variations, with each core capable of running 8 threads concurrently. They have a crossbar between the L2 and the processor cores (similar to our placement of the L2 in Figure 1(a)). Although the T1 and T2 support many concurrent threads (32 and 64, respectively) compared to other contemporary CPUs, this number is very small compared to the number on a high end contemporary GPU (e.g., the Geforce 8800 GTX supports 12,288 threads per chip).

We quantified the effects of varying cache size, DRAM bandwidth and other parameters which, to our knowledge, has not been published previously. While the authors of the CUDA applications which we use as benchmarks have published work, the emphasis of their papers was not on how changes in the GPU architecture can affect their applications [4, 5, 12, 13, 15, 24, 26, 27, 37, 41, 42, 46]. In terms of streaming multiprocessor design, all of the above-mentioned works have different programming models from the CUDA programming model that we employ.

6. Conclusions

In this paper we studied the performance of twelve contemporary CUDA applications by running them on a detailed performance simulator that simulates NVIDIA's parallel thread execution (PTX) virtual instruction set architecture. We presented performance data and detailed analysis of performance bottlenecks, which differ in type and scope from application to application. First, we found that generally performance of these applications is more sensitive to interconnection network bisection bandwidth rather than (zero load) latency: Reducing interconnect bandwidth by 50% is even more harmful than increasing the per-router latency by 5.3× from 3 cycles

to 19 cycles. Second, we showed that caching global and local memory accesses can cause performance degradation for benchmarks where these accesses do not exhibit temporal or spatial locality. Third, we observed that sometimes running fewer CTAs concurrently than the limit imposed by on-chip resources can improve performance by reducing contention in the memory system. Finally, aggressive inter-warp memory coalescing can improve performance in some applications by up to 41%.

Acknowledgments

We thank Kevin Skadron, Michael Shebanow, John Kim, Andreas Moshovos, Xi Chen, Johnny Kuan and the anonymous reviewers for their valuable comments on this work. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Advanced Micro Devices, Inc. *ATI CTM Guide*, 1.01 edition, 2006.

[2] Advanced Micro Devices, Inc. *Press Release: AMD Delivers Enthusiast Performance Leadership with the Introduction of the ATI Radeon HD 3870 X2*, 28 January 2008.

[3] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the Imagine stream architecture. In *Proc. 31st Int'l Symp. on Computer Architecture*, page 14, 2004.

[4] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *Proc. 17th Int'l Symp. on High Performance Distributed Computing*, pages 165–174, 2008.

[5] Billconan and Kavinguy. A Neural Network on GPU. http://www.codeproject.com/KB/graphics/GPUNN.aspx.

[6] P. Buffet, J. Natonio, R. Proctor, Y. Sun, and G. Yasar. Methodology for I/O cell placement and checking in ASIC designs using area-array power grid. In *IEEE Custom Integrated Circuits Conference*, 2000.

[7] S. Clark, K. Haselhorst, K. Imming, J. Irish, D. Krolak, and T. Ozguner. Cell Broadband Engine interconnect and memory interface. In *Hot Chips 17*, Palo Alto, CA, August 2005.

[8] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proc. 2003 ACM/IEEE Conf. on Supercomputing*, page 35, 2003.

[9] W. J. Dally and B. Towles. *Interconnection Networks*. Morgan Kaufmann, 2004.

[10] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures. *Int'l Symp. on Performance Analysis of Systems and Software*, pages 231–241, March 2006.

[11] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. 40th IEEE/ACM Int'l Symp. on Microarchitecture*, 2007.

[12] M. Giles. Jacobi iteration for a Laplace discretisation on a 3D structured grid. http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/laplace3d.pdf.

[13] M. Giles and S. Xiaoke. Notes on using the NVIDIA 8800 GTX graphics card. http://people.maths.ox.ac.uk/~gilesm/hpc/.

[14] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. In *Proc. 24th Int'l Symp. on Computer Architecture*, pages 108–120, 1997.

[15] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, pages 197–208, 2007.

[16] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium® 4 Processor. *Intel® Technology Journal*, 5(1), 2001.

[17] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In *Proc. SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 1998.

[18] Illinois Microarchitecture Project utilizing Advanced Compiler Technology Research Group. Parboil benchmark suite. http://www.crhc.uiuc.edu/IMPACT/parboil.php.

[19] Infineon. 256Mbit GDDR3 DRAM, Revision 1.03 (Part No. HYB18H256321AF). http://www.infineon.com, December 2005.

[20] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owens, and B. Towles. Exploring the VLSI scalability of stream processors. In *Proc. 9th Int'l Symp. on High Performance Computer Architecture*, page 153, 2003.

[21] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proc. 8th Int'l Symp. Computer Architecture*, pages 81–87, 1981.

[22] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[23] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. Tradeoffs in designing accelerator architectures for visual computing. In *Proc. 41st IEEE/ACM Int'l Symp. on Microarchitecture*, 2008.

[24] S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSPC 2007: Proc. of IEEE Int'l Conf. on Signal Processing and Communication*, pages 65–68, 2007.

[25] Marco Chiappetta. ATI Radeon HD 2900 XT - R600 Has Arrived. http://www.hothardware.com/printarticle.aspx?articleid=966.

[26] Maxime. Ray tracing. http://www.nvidia.com/cuda.

[27] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. *IPDPS 2008: IEEE Int'l Symp. on Parallel and Distributed Processing*, pages 1–7, April 2008.

[28] M. Murphy. NVIDIA's Experience with Open64. In *1st Annual Workshop on Open64*, 2008.

[29] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar.-Apr. 2008.

[30] NVIDIA. CUDA ZONE. http://www.nvidia.com/cuda.

[31] NVIDIA. Geforce 8 series. http://www.nvidia.com/page/geforce8.html.

[32] NVIDIA Corporation. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.

[33] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 1.1 edition, 2007.

[34] NVIDIA Corporation. *Press Release: NVIDIA Tesla GPU Computing Processor Ushers In the Era of Personal Supercomputing*, 20 June 2007.

[35] NVIDIA Corporation. *PTX: Parallel Thread Execution ISA*, 1.1 edition, 2007.

[36] Open64. The open research compiler. http://www.open64.net/.

[37] Pcchen. N-Queens Solver. http://forums.nvidia.com/index.php?showtopic=76893.

[38] D. Pham, S. Asano, M. Bolliger, M. D. , H. Hofstee, C. Johns, J. Kahle, A.Kameyama, J. Keaty, Y. Masubuchi, D. S. M. Riley, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. W. D. Wendel, T.Yamazaki, and K. Yazawa. The design and implementation of a first-generation Cell processor. *Digest of Technical Papers, IEEE Int'l Solid-State Circuits Conference (ISSCC)*, pages 184–592 Vol. 1, 10-10 Feb. 2005.

[39] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proc. 27th Int'l Symp. on Computer Architecture*, pages 128–138, 2000.

[40] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S.-Z. Ueng, J. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proc. 6th Int'l Symp. on Code Generation and Optimization (CGO)*, pages 195–204, April 2008.

[41] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 73–82, 2008.

[42] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007.

[43] J. W. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 85–94, 2004.

[44] Sun Microsystems, Inc. *OpenSPARC^{TM} T2 Core Microarchitecture Specification*, 2007.

[45] J. Tuck, L. Ceze, and J. Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *Proc. 39th IEEE/ACM Int'l Symp. on Microarchitecture*, pages 409–422, 2006.

[46] T. C. Warburton. Mini Discontinuous Galerkin Solvers. http://www.caam.rice.edu/~timwar/RMMC/MIDG.html.