Efficient Dynamic Automatic Memory Management and Concurrent Kernel Execution for General-Purpose Programs on Graphics Processing Units

A Thesis
Submitted For the Degree Of
Doctor of Philosophy
in the Faculty of Engineering

by
Sreepathi Pai

Supercomputer Education and Research Centre
Indian Institute of Science
BANGALORE – 560 012

November 2014
To my family
An ounce of algebra is worth a ton of verbal argument.

– J. B. S. HALDANE

The best way to predict the future is to invent it.

– ALAN KAY
# Contents

Acknowledgements xv
Abstract xvii
Publications xix

## 1 Introduction

1.1 Background ................................................. 3
1.2 Automatic Memory Management for GPUs in High-Level Languages 4
1.3 Elastic Kernels for Improved GPGPU Concurrency ................. 7
1.4 Structural Prediction for Improved Scheduling of Concurrent Kernels 9
1.5 Contributions of this Thesis .................................. 11
1.6 Outline .................................................. 12

## 2 Automatic Memory Management for GPUs

2.1 Introduction ................................................ 15
2.2 Background ................................................ 17
2.3 The Need for Efficient Memory Management ....................... 19
2.4 Overview of Automatic Memory Management using Software Coherence 23
  2.4.1 Implementing Coherence for Rails ....................... 23
  2.4.2 Maintaining Coherence on the GPU ....................... 24
  2.4.3 Maintaining Coherence on the CPU ....................... 24
  2.4.4 Putting it all together ................................... 25
  2.4.5 Coherence Checks versus Data Transfers ................. 27
2.5 X10CUDA+AMM ........................................... 27
  2.5.1 X10CUDA+AMM Programs ............................... 27
  2.5.2 Compiler Analyses and Transformations .................. 28
  2.5.3 Runtime Support ....................................... 32
2.6 CPU–GPU Communications Manager (CGCM) for X10CUDA .......... 34
2.7 Evaluation ................................................ 34
  2.7.1 Methodology ........................................... 36
  2.7.2 Performance of Individual Schemes ...................... 37
  2.7.3 Data Transferred by X10CUDA+AMM ..................... 38
  2.7.4 Data Transferred by X10CUDA+CGCM ................... 39
4.5 Thread Block Scheduling ........................................... 94
  4.5.1 Runtime Aware Policies ........................................ 94
  4.5.2 Runtime Unaware Policies ...................................... 96
4.6 Evaluation .............................................................. 97
  4.6.1 Benchmarks ....................................................... 97
  4.6.2 Simulator ......................................................... 98
  4.6.3 Methodology ..................................................... 98
  4.6.4 Results .......................................................... 99
  4.6.5 Sensitivity to Arrival Time .................................... 103
4.7 Related Work .......................................................... 103
4.8 Conclusion .............................................................. 104

5 Conclusion and Future Work ........................................... 105
  5.1 Summary of Results ................................................ 106
  5.2 Future Work ........................................................ 107
List of Tables

1.1 Resources per Streaming Multiprocessor across three NVIDIA GPU generations ......................................................... 7
2.1 A comparison of the ability of automatic memory management systems to eliminate redundant transfers ................................. 22
2.2 Differences between X10CUDA and X10CUDA+AMM language features for GPU programs ........................................................... 28
2.3 Rodinia benchmarks used in this study, descriptions adapted from the Rodinia documentation. Legend: KL=Kernel launches, A=Allocations, G=Gets to CPU, P=Putsto GPU ......................................................... 36
2.4 Number of coherence checks inserted and hoisted per benchmark .................................................................................. 42
2.5 Number of dynamic check_write() and check_read() executions ........................................................................... 42
3.1 Resource usage of Parboil2 kernels on the Fermi GPU ................................................................. 47
3.2 Concurrent Execution possible on Fermi (LEFTOVER policy) for Parboil2 kernels ............................................................... 49
3.3 Overall STP and ANTT results for two-program workloads ................................................................................ 68
3.4 Improvements in STP and ANTT and Multithreaded Speedup (MT-Speedup) ................................................................. 68
4.1 State maintained per-kernel in each SM ................................................................................................. 91
4.2 Grid Characteristics of ERCBench Kernels ......................................................................................... 97
4.3 Runtimes for ERCBench Kernels on the simulator .................................................................................. 98
4.4 Simulator Configuration .................................................................................................................. 98
4.5 Geomean STP, ANTT and Fairness for various scheduling policies ....................................................... 99
4.6 Geomean STP, ANTT and Fairness for various scheduling policies when second kernel arrives at 25\% and 50\% of first kernel’s runtime ................................................... 103
## List of Figures

1.1 Schematic of the system used in this work .......................... 3
1.2 Resource wastage due to allocation at Thread Block granularity. 8
2.1 Total data transferred by a naive automatic memory transfer scheme 19
2.2 Breakup of data transferred by the naive automatic memory manager 20
2.3 srad runtime control flow ............................................ 25
2.4 Data flow equations to determine placement of coherence checks 29
2.5 Normalized execution time of benchmarks using automatic memory management ......................................................... 38
2.6 Size of data transferred by all schemes compared to manual 39
2.7 Breakup of data transfers performed by X10CUDA+CGCM 40
2.8 Normalized execution time for OpenMPC and ADSM 42
3.1 Components that affect concurrency on the Fermi GPU ........... 50
3.2 Execution of kernels with a FIFO scheduler and a Concurrency-aware scheduler ................................................................. 52
3.3 Time-sliced memory transfers can reduce waiting times across programs 53
3.4 Example construction of a multithreaded concurrent CUDA workload 63
3.5 Overlap Categories ............................................................ 66
3.6 Runtimes of \texttt{GPU\_FFT\_Global} instances categorized into overlap categories ................................................................. 67
3.7 Fraction of execution time of kernels spent in different overlap categories per variant and per policy. .............................. 69
3.8 Average Kernel Waiting Time for two-program workloads ....... 70
3.9 System throughput for each two-program workload ................ 71
3.10 Average Normalized Turnaround Time for each two-program workload 72
3.11 Average number of running kernels for each two-program workload 73
3.12 Average utilization of threads for each two-program workload 73
4.1 System Throughput under the SJF, FIFO, and LJF policies ....... 79
4.2 Staircase Model Execution of \( N \) thread blocks on a single SM 82
4.3 Execution of SGEMM’s thread blocks from one SM ................. 83
4.4 Boxplots of Predictions from Linear Regression and Staircase Models 84
4.5 SGEMM execution as recorded on a different SM ................... 85
4.6 Boxplots of normalized thread block durations ...................... 86
4.7 Average thread block duration at various residencies .......................... 87
4.8 Total kernel runtime at various residencies ................................. 88
4.9 Average thread block durations from simulator for SAD mb_sad_calc kernel at different residencies when running with different kernels .... 89
4.10 Average thread block durations from simulator for SAD mb_sad_calc kernel when running with NLM2 ............................ 90
4.11 Accuracy of the Simple Slicing (SS) Predictor ............................... 93
4.12 Shortest Remaining Time First implementation .............................. 94
4.13 SRTF versus SRTF/Adaptive .................................................... 95
4.14 System Throughput (STP) for various scheduling policies ............... 100
4.15 Average Normalized Turnaround Time (ANTT) for various scheduling policies ................................................................. 102
4.16 Fairness (StrictF) for various scheduling policies ........................... 102
Acknowledgements

In the dim past, research advisors wrote and defended a student's dissertation while the student merely paid for its publication. Now, the thesis carries the name of a single author, the student. This thesis, however, would not have been if it were not for my advisors – Profs. R. Govindarajan and T. Matthew Jacob. Their high academic standards permeate the good parts of this work. But most of all, I am thankful to have worked with good human beings, who were incredibly supportive throughout my PhD, especially through the rough patches. I am also grateful to their families who hosted me numerous times at their homes.

Our ASPLOS paper benefited from answers by Sanjiv Satoor, Dibyapran Sanyal and Swapna Matwankar at NVIDIA, India to questions regarding their CUDA implementation. Our shepherd for the ASPLOS paper, Rodric Rabbah, prodded us to add a memory allocator which made that work more realistic. I acknowledge Microsoft Research India who funded my travel to ASPLOS 2013.

My labmates at the Lab for High Performance Computing – Girish, Kaushik, Santosh, Aditya, Abhishek, Sandya, Ashwin, Rupesh, Jayvant, Manikantan, Nagendra, Prasanna, Vaivaswatha, Radhika – provided a stimulating lab environment to work in and also supported me whenever I needed it.

I am thankful to the office staff at SERC who made sure I could focus on my thesis while they tackled the administrative machinery of the Institute.

When research became stressful, friends in the Institute kept me distracted. Anil helped me debug the swimming pool as co-convener. Madhu, Chetana, Siva, Arvind and Sumeer engaged me in the activities of the IISc Quiz Club among many other things. Nirmal, Abhijit, Mustafa, Alex, Prakash, Gokul, Karthik and many others were wonderful riding companions during our long bicycle rides, whenever I could keep up with them! Shyam, Shrisha, Lalit and many others provided stimulating mess-time conversations which I miss more than the mess food. I am privileged to have them as good friends.

The bicyclists of Bangalore, especially the members of IISc Randonneurs, the IISc Bikers Network and the Bangalore Bikers Club played a significant role in ensuring that I remained both mentally and physically fit throughout my research studies.

Lastly, this thesis would never have become a reality if my family had not supported me in all my endeavours so far. To my parents and my siblings, I thank you for your unconditional patience, love and understanding.
Abstract

Modern supercomputers now use accelerators to achieve their performance with the most widely used accelerator being the Graphics Processing Unit (GPU). However, achieving the performance potential of systems that combine a GPU and CPU is an arduous task which could be made easier with the assistance of the compiler or runtime. In particular, exploiting two features of GPU architectures – distributed memory and concurrent kernel execution – is critical to achieve good performance, but in current GPU programming systems, programmers must exploit them manually. This can lead to poor performance. In this thesis, we propose automatic techniques that: i) perform data transfers between the CPU and GPU, ii) allocate resources for concurrent kernels, and iii) schedule concurrent kernels efficiently without programmer intervention.

Most GPU programs access data in GPU memory for performance. Manually inserting data transfers that move data to and from this GPU memory is an error-prone and tedious task. In this work, we develop a software coherence mechanism to fully automate all data transfers between the CPU and GPU without any assistance from the programmer. Our mechanism uses compiler analysis to identify potential stale data accesses and uses a runtime to initiate transfers as necessary. This avoids redundant transfers that are exhibited by all other existing automatic memory management proposals for general purpose programs. We integrate our automatic memory manager into the X10 compiler and runtime, and find that it not only results in smaller and simpler programs, but also eliminates redundant memory transfers. Tested on eight programs ported from the Rodinia benchmark suite it achieves (i) a 1.06x speedup over hand-tuned manual memory management, and (ii) a 1.29x speedup over another recently proposed compiler–runtime automatic memory management system. Compared to other existing runtime-only (ADSM) and compiler-only (OpenMPC) proposals, it also transfers 2.2x to 13.3x less data on average.

Each new generation of GPUs vastly increases the resources available to GPGPU programs. GPU programming models (like CUDA) were designed to scale to use these
resources. However, we find that CUDA programs actually do not scale to utilize all available resources, with over 30% of resources going unused on average for programs of the Parboil2 suite. Current GPUs therefore allow concurrent execution of kernels to improve utilization. We study concurrent execution of GPU kernels using multiprogrammed workloads on current NVIDIA Fermi GPUs. On two-program workloads from Parboil2 we find concurrent execution is often no better than serialized execution. We identify lack of control over resource allocation to kernels as a major serialization bottleneck. We propose transformations that convert CUDA kernels into elastic kernels which permit fine-grained control over their resource usage. We then propose several elastic-kernel aware runtime concurrency policies that offer significantly better performance and concurrency than the current CUDA policy. We evaluate our proposals on real hardware using multiprogrammed workloads constructed from benchmarks in the Parboil2 suite. On average, our proposals increase system throughput (STP) by 1.21x and improve the average normalized turnaround time (ANTT) by 3.73x for two-program workloads over the current CUDA concurrency implementation.

Recent NVIDIA GPUs use a FIFO policy in their thread block scheduler (TBS) to schedule thread blocks of concurrent kernels. We show that FIFO leaves performance to chance, resulting in significant loss of performance and fairness. To improve performance and fairness, we propose use of the Shortest Remaining Time First (SRTF) policy instead. Since SRTF requires an estimate of runtime (i.e. execution time), we introduce Structural Runtime Prediction that uses the grid structure of GPU programs for predicting runtimes. Using a novel Staircase model of GPU kernel execution, we show that kernel runtime can be predicted by profiling only the first few thread blocks. We evaluate an online predictor based on this model on benchmarks from ERCBench and find that predictions made after the execution of single thread block are between 0.48x to 1.08x of actual runtime. We implement the SRTF policy for concurrent kernels that uses this predictor and evaluate it on two-program workloads from ERCBench. SRTF improves STP by 1.18x and ANTT by 2.25x over FIFO. Compared to MPMax, a state-of-the-art resource allocation policy for concurrent kernels, SRTF improves STP by 1.16x and ANTT by 1.3x. To improve fairness, we also propose SRTF/Adaptive which controls resource usage of concurrently executing kernels to maximize fairness. SRTF/Adaptive improves STP by 1.12x, ANTT by 2.23x and Fairness by 2.95x compared to FIFO. Overall, our implementation of SRTF achieves STP to within 12.64% of Shortest Job First (SJF, an oracle optimal scheduling policy), bridging 49% of the gap between FIFO and SJF.
Publications

Portions of this thesis have appeared in the following publications.


Chapter 1

Introduction

In 2006, MDGRAPE-3 became the first supercomputer to break through the petaflop barrier [RIKEN et al., 2006]. It used a specialized molecular dynamics accelerator to achieve this milestone. Since then, supercomputers have increasingly used accelerators to meet performance goals. Also known as co-processors, accelerators are computational devices that are distinct from the main CPU. They use specialized hardware logic and parallel processing elements to perform some computations faster than their host CPUs who can then offload some types of processing to them. Unlike MDGRAPE-3’s specialized accelerator, most supercomputers today use general-purpose accelerators. These accelerate a wider range of computations, e.g. SIMD-style computations, but still lack features like I/O support to qualify as CPUs. Since 2006, five of the eight fastest supercomputers in the Top500 list – Roadrunner, Jaguar, Tianhe-1A, Titan and Tianhe-2 – have used general-purpose accelerators to supplement the performance of their CPU cores [TOP500No1]. In the June 2013 Top500 list [TOP500List], 54 supercomputers use accelerators, and nearly 35% of all TOP500 performance in Flops can be attributed to them [TOP500Stats]. While the Tianhe-2, currently at the top of the list, uses the newly available Intel Xeon Phi as an accelerator, the major general-purpose accelerators used are the NVIDIA Fermi [Glaskowsky, 2009] and the NVIDIA Kepler [NVKWP], both graphics processing units (GPUs).

Originally, GPUs implemented the so-called graphics pipeline using fixed-function graphics logic for the pipeline. However, as graphics programs demanded increased visualization capabilities, fixed-function logic (“shaders”) gave way to partially programmable shaders. Ultimately, this trend towards increased programmability culminated in the unified shader, a GPU architecture that uses general-purpose ALUs organized in a SIMD fashion to implement the graphics pipeline. The unified shader allowed
non-graphics programs to exploit the performance of GPUs naturally – the GPU was essentially a very wide SIMD machine. Thus, GPUs ended up being one of the most widely used accelerators.

Combining a CPU and a GPU results in a heterogeneous architecture. Programming systems for these heterogeneous architectures are still very primitive. Originally, programmers used GPUs by writing their non-graphics code as OpenGL shaders [Owens et al., 2007], an approach that severely limited what computations could run on GPUs. NVIDIA’s CUDA [NVCUDA], a C++ dialect targeting GPUs, allowed more general programs to be specified. As CUDA is proprietary to NVIDIA’s GPUs, the Open Computing Language (OpenCL) was proposed as “the open standard for parallel programming of heterogeneous systems” [KOCL]. Despite the difference in nomenclature, OpenCL’s data parallel constructs are largely inherited from CUDA constructs and have similar behaviour. This thesis focuses on NVIDIA GPUs and CUDA, partly because of the increased maturity of NVIDIA’s tools. However, the problems described in this thesis apply to OpenCL as well.

Although CUDA programs can contain both the CPU-side code and the GPU-side code in a single file, this is merely for convenience. NVIDIA’s nvcc compiler separates out the GPU and CPU portions of the source code. It compiles the GPU code while the CPU code is handed over to the system compiler, usually gcc (or g++). In the process, nvcc replaces calls to GPU code with invocations to the CUDA Runtime API. Therefore, to gcc, the CPU code looks no different from a CPU-only program because the API calls look like ordinary function calls. This organization has an unfortunate effect: no individual compiler has a complete picture of the interactions between the CPU and the accelerator. For some problems, like instruction scheduling, this is of little consequence. However, many problems that lie at the interface of the CPU and accelerator end up being ignored by both the compilers used.

For example, the problems considered in this thesis – memory management for GPUs and concurrent execution of GPU kernels – are both expressed using the CUDA API. Since CUDA API calls are embedded in CPU code, the nvcc compiler does not reason about or analyze either GPU memory management or GPU task parallelism. The gcc compiler, too, is completely oblivious to the semantics of these CUDA API calls. Thus, by default, the programmer is then responsible for their correctness and performance. For memory management, this means the programmer has to perform additional book-keeping, a task that becomes harder as programs grow larger.\(^1\) Manually orches-\(^1\)We observe, informally, that the conservativeness of a human programmer approaches that of an
trating concurrent execution of GPU kernels is even harder because CUDA program-
ners can only express dependence information statically. However, whether concurrent
execution will actually occur depends on a number of additional factors (such as avail-
able resources) which are statically unpredictable. In this thesis, by proposing effective
and efficient dynamic solutions to these problems, we postulate that runtimes\textsuperscript{2} – inter-
preted as any component active during execution – are best positioned to effectively
tackle such problems in general and these problems in particular.

1.1 Background

We use NVIDIA GPUs in this thesis. The heterogeneous system we study consists of
a multicore CPU and a discrete GPU as illustrated schematically in Figure 1.1. Al-
though GPUs can be integrated with the CPU (as in the recent AMD Fusion archi-
tecture [Brookwood, 2010]), they do not perform as well as discrete devices due to
bandwidth and power constraints.

The NVIDIA GPUs used in this work contain multiple cores called streaming multi-
processors (SMs) that are connected to GPU-local memory via a high-bandwidth path.
Each SM can execute hundreds of hardware threads at a time multiplexing between
them in hardware. However, depending on the GPU, only 16 to 32 of these threads
execute concurrently.

A GPU program (kernel) is multithreaded with thousands of threads and executes
on the SMs. However, depending on GPU resources, only a fraction of these threads
are actually running on an SM. Threads wait until resources are available for them to
eexecute. A kernel is done when all its threads have finished executing.

\begin{itemize}
\item optimizing compiler as the size of the codebase increases.
\item \textsuperscript{2}This thesis also uses the word runtime to mean execution time. The difference is usually clear from
the context.
\end{itemize}
When running, a kernel can use *shared memory*, a fast per-SM scratchpad-like memory. Each of its thread also occupies registers and thread contexts. These resources are assigned exclusively to a thread when it begins executing and are released only when the thread finishes.

Since GPUs do not run system level code yet, they cannot perform I/O or execute system calls. These tasks are delegated to CPU code. Early GPUs could not allocate memory or execute new kernels from within kernels. However, newer hardware allows kernels to do both, though with significant limitations. In our work, the CPU mostly runs code to control GPU execution, though in the automatic memory management part of this work, the CPU also runs significant computation code that is unsuitable for the GPU.

All the kernels used in this work are written in CUDA or ultimately compile to CUDA. CUDA is NVIDIA’s framework for programming GPUs. It consists of a GPU programming language (a dialect of C++), a toolchain for the dialect, and API libraries and runtimes. The `nvcc` compiler compiles CUDA code for the GPU. The API is differentiated into a high-level CUDA *Runtime* API and a low-level CUDA *Device* API. The latter provides a very low-level pure C library interface to the GPU, while the former emphasizes ease of use. We use both APIs in our work.

In this work, we concern ourselves with two GPU features: memory management and concurrent kernel execution. The remaining sections in this chapter introduce these features and the problems that we tackle. Finer details are relegated to the individual chapter for each problem.

### 1.2 Automatic Memory Management for GPUs in High-Level Languages

Performance obtained on GPUs is often an order of magnitude greater than that obtainable on modern multicore CPUs [Lee et al., 2010]. A key source of GPU performance is the use of local, high-bandwidth *GPU memory* that is distinct from CPU memory. Prior to CUDA 4.0 [NVCCPG], GPU kernels, could only access this memory. All data consumed or produced by them therefore resided in GPU memory. CUDA 4.0 unified CPU and GPU memory into a single 64-bit address space (“unified virtual address”) allowing direct reads and writes of CPU and GPU memory [Schroeder, 2011], but it is of limited applicability and is slow due to the limited bandwidth between the CPU and
GPU. Therefore, programs on the CPU which share data with GPU kernels transfer that data explicitly between CPU and GPU memories in order to communicate. In current programming models for the GPU, such as CUDA [NVCUDA] and OpenCL [KOCL], the programmer must manually perform allocations of GPU memory and transfers of shared data between the CPU and GPU. This manual memory management is tedious, error-prone and a source of performance and portability issues.

Even programmers who do not write CUDA programs but would only like to use GPU libraries are not exempt. NVIDIA supplies highly-optimized CUBLAS and CUFFT GPU libraries that are almost drop-in replacements for the BLAS and FFTW libraries, but these libraries continue to rely on the programmer to manually manage memory allocations and transfers for them. This is because even with highly optimized kernel code, unnecessary data transfers between the CPU and the GPU can impact performance significantly.

When CUDA programs are written by hand, these data transfers are inserted by the programmer. If data transfers are missed out, the CPU (or GPU) can end up accessing stale or older data, leading to correctness errors. Conservatively bracketing each kernel invocation with data transfers of data used by that kernel ensures correctness, but since data transfers are only necessary when the data has changed, these redundant data transfers can degrade performance.

Even when programs are written in higher-level languages that support programming GPUs directly without the use of CUDA or OpenCL, the problem remains. In particular, the partitioned global address space (PGAS) languages X10 [Charles et al., 2005] and Chapel [Chamberlain et al., 2007] already support programming the GPU directly. These languages have built-in support for expressing parallelism, asynchronous communication and execution, and notions of remote and local data. Therefore GPUs can be integrated into these programming languages in a more natural fashion than is possible with CUDA or OpenCL. However, neither X10 nor Chapel fully abstracts CUDA. In X10CUDA, the X10 compiler and runtime for CUDA [Cunningham et al., 2011], the programmer is still required to manually perform GPU memory allocations and transfers. GPU memory management therefore continues to be a source of programming difficulty and potential performance problems even in these languages.

When compilers generate CUDA code, say from parallel loops, these data transfers must also be generated automatically. The OpenMPC system [Lee et al., 2009] compiles OpenMP code to CUDA programs and uses simple data-flow analysis to insert data transfer statements. In a follow-up work, Lee and Eigenmann [2010], the compiler
accepts programmer hints regarding the insertion of data transfers. An alternative approach by Baskaran et al. [2008] uses the polyhedral model to analyze parallel loops and insert relevant data transfer statements. CGCM [Jablin et al., 2011] uses compiler analysis but also maintains minimal runtime state to avoid some classes of redundant transfers. Most, like Lee and Eigenmann [2010], Baskaran et al. [2008], Jablin et al. [2011] insert data transfers by using compiler analysis alone. Some, like Lee et al. [2009], Dathathri et al. [2013], Vasudevan et al. [2013] use programmer-specified directives to aid the compiler, while the work of Jablin et al. [2011] is fully automatic with no information required from the programmer. However, these statically placed data transfers are necessarily conservative, and in all but the simplest of programs lead to excessive data transfers.

Asymmetric DSM (ADSM) [Gelado et al., 2010] is a purely runtime solution that is similar to older user-space DSM libraries [Amza et al., 1996]. ADSM requires the programmer to annotate shared objects. It tracks reads and writes to these shared objects at runtime and can keep shared copies coherent. Similarly, DyManD [Jablin et al., 2012] extends CGCM [Jablin et al., 2011] with an ADSM-like component. However, any runtime solution is limited because unlike the CPU, the GPU does not expose its virtual memory mechanism to user programs. Thus, with only runtime information, a DSM implementation must conservatively assume that the GPU has always written to data. This leads to redundant and excessive data transfers in ADSM and DyManD.

We propose X10+Automatic Memory Management (or X10+AMM) that automates data transfers in X10 programs that use the GPU. Our solution modifies the X10 compiler and runtime to enable data transfers between the CPU and GPU transparently and automatically without any hints from the programmer. Since pure compiler solutions are not optimal due to their conservativeness, and wholly runtime solutions are not performant because GPUs lack support for mechanisms such as trapping page faults, our solution is a hybrid compiler–runtime scheme. Our scheme uses runtime coherence information as a basis for automatic data transfers. This runtime coherence information is maintained by compiler-inserted instrumentation. When the runtime encounters accesses to data that is not current, it initiates data transfers automatically. Compared to runtime-only schemes, our overhead is lower because we use the compiler to instrument only those data accesses that may access stale data. Compared to compiler-only schemes that insert data transfers, our scheme is more accurate since it uses runtime information to initiate data transfers.
<table>
<thead>
<tr>
<th>Resource</th>
<th>Tesla (1.3)</th>
<th>Fermi (2.0)</th>
<th>Kepler (3.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>16384</td>
<td>32768 (2x)</td>
<td>65536 (2x)</td>
</tr>
<tr>
<td>Shared Memory (Bytes)</td>
<td>16384</td>
<td>49152 (3x)</td>
<td>49152</td>
</tr>
<tr>
<td>Threads</td>
<td>1024</td>
<td>1536 (1.5x)</td>
<td>2048 (1.33x)</td>
</tr>
<tr>
<td>Resident Blocks</td>
<td>8</td>
<td>8</td>
<td>16 (2x)</td>
</tr>
</tbody>
</table>

Table 1.1: Resources per Streaming Multiprocessor across three NVIDIA GPU generations. Multipliers in parentheses indicate change over preceding generation.

### 1.3 Elastic Kernels for Improved GPGPU Concurrency

Each new GPU generation incorporates features that expand their ability to tackle larger and more complex computational problems. The architectural resources available to GPU programs also increase with each new GPU generation. We observe that from the NVIDIA Tesla [Lindholm et al., 2008] to the NVIDIA Kepler [NVKWP], shared memory has trebled, registers have quadrupled, and the number of hardware threads has doubled (Table 1.1). GPU programming models like CUDA and OpenCL have been designed to allow older programs to take advantage of these increased resources without any programmer intervention. Programs written in CUDA for the Tesla scale to use the increased resources available on the Kepler because CUDA requires parallelism to be made explicit and performs resource allocation at runtime.

In reality, however, we find that CUDA programs are unable to effectively utilize these additional resources. Programs from the Parboil2 benchmark suite, for example, utilize only 20–70% of resources on average (Section 3.2). Ironically, we find that the grid, a GPU programming construct that was designed to achieve scalability, also leads to under-utilization of those same resources. A grid is the runtime instance of a GPU kernel, and consists of thread blocks. Each thread block, in turn, consists of threads which ultimately consume resources and execute on the hardware. In the CUDA programming model each thread block in the grid is an independent unit of parallelism and can execute independently of other thread blocks in the grid. Therefore, by allowing the programmer to create a large number of thread blocks – more than can run concurrently given the resources available in the hardware – CUDA grids can scale up to newer hardware by simply increasing the number of thread blocks that run concurrently [NVCCPG, Section 1.3].

The number of threads in a thread block and the number of thread blocks in a grid are
decided by the programmer, who divides the work to be performed among the thread blocks. Apart from work distribution and scalability, thread blocks are also used to allocate resources for a grid which leads to under-utilization of resources. Figure 1.2 illustrates how resource allocation at the thread block level leads to wastage. Each thread block consumes a fixed amount of the GPU’s resources – registers, threads and shared memory – which it occupies exclusively until it finishes. A GPU runs as many thread blocks concurrently as possible until limitations due to any of (i) number of registers, (ii) threads, (iii) shared memory or (iv) maximum number of resident blocks is reached. It is possible, and in our experience quite common, for thread blocks to exhaust only one of these resources (the limiting resource), while leaving the others underutilized. For example, on the Fermi, a thread block that uses 16 registers per thread and contains 128 threads per thread block is limited by the maximum number of resident blocks (8) and not by the number of registers or threads. The unused resources obviously cannot accommodate another thread block from the same grid and are therefore wasted.

If the hardware were able to modify grid and thread block sizes, it could choose sizes that would optimize the utilization of hardware resources. Unfortunately, grids are specified at the programming model level and provide the basis for work distribution. Most programmers size grids and thread blocks based on the amount of work available, and not for optimal utilization of hardware resources.

The only way to utilize these wasted resources, then, is to allow concurrent execution of other independent grids whose thread blocks could possibly utilize these wasted resources. Such independent grids may be obtained from other concurrently executing GPGPU programs or from other streams (Section 3.3.1) in the same program. As a consequence, current GPUs have begun supporting concurrent execution of indepen-
dent grids. Policies vary, but in general, after thread blocks from a grid have been dispatched to hardware limits, any leftover resources are distributed to the next independent grid. Under this “LEFTOVER” policy, concurrent execution is not guaranteed – it is still possible for running (or resident) thread blocks from one grid to consume too many resources, preventing other grids from executing concurrently. In practice, we find this is actually a serious problem – 50% of the kernels from the Parboil2 benchmark consume too many resources and prevent concurrent execution of other Parboil2 kernels (Section 3.2).

Past work [Adriaens et al., 2012, Gregg et al., 2012, Guevara et al., 2009, Ravi et al., 2011] has motivated GPU concurrency as a method to improve GPU throughput. Adriaens et al. [2012] propose the use of GPU concurrency for mobile GPUs, Ravi et al. [2011] look at GPGPU applications in the cloud and Guevara et al. [2009] and Gregg et al. [2012] demonstrate throughput improvements due to concurrency. Although these works partition GPU resources among concurrent kernels, the granularity of their techniques is either too coarse, operating at the level of a thread block [Adriaens et al., 2012, Gregg et al., 2012, Guevara et al., 2009], or their techniques are not general enough to apply to all kernels [Ravi et al., 2011]. We show in this work that they either underutilize resources significantly or do not perform as well as our policies.

This work is the first work to look at GPGPU concurrency in detail on real hardware. We find that GPGPU concurrency suffers from a wide variety of serialization bottlenecks: in the CUDA Runtime API to the hardware resource allocation policy. Since lack of control over grid resource usage leads to poor utilization and poor concurrency, we propose Elastic Kernels, a mechanism to control grid resource allocation at runtime. We also replace parts of the CUDA Runtime API with an Elastic-kernel aware API called Non-serializing Streams. We propose several Elastic-kernel-aware resource allocation policies and show that all of them perform better than the default LEFTOVER policy.

1.4 Structural Prediction for Improved Scheduling of Concurrent Kernels

To maintain high throughput, modern GPUs use hardware schedulers to schedule warps and thread blocks. The NVIDIA Fermi, for example, features a two-level hardware scheduler [Glaskowsky, 2009]. The first level, called the Thread Block Scheduler
(TBS), schedules thread blocks from a grid to the GPU cores (*streaming multiprocessors* or SMs). Each SM can accommodate up to 8 thread blocks depending on their resource usage (threads, registers, shared memory). The second level, known as the **Warp Scheduler**, schedules warps on each individual SM. Upto 48 warps can be scheduled onto each of the Fermi’s SMs.

Warp scheduling has received considerable attention recently [Kayiran et al., 2013, Jog et al., 2013, Chen et al., 2013, Gebhart et al., 2011, Narasiman et al., 2011, Meng et al., 2010, Fung et al., 2007]. Thread block scheduling policies, on the other hand, have received little attention. Unlike recent GPUs such as the NVIDIA Fermi and the Kepler [NVKWP] which can execute multiple grids concurrently, older GPUs could only execute a single grid at a time in order of its arrival and hence there was no opportunity to reorder thread blocks across grids. Despite its support for concurrent kernels, however, our microbenchmarking reveals that the TBS on the Fermi issues thread blocks from grids using a FIFO policy – newer grids must wait until all of the thread blocks from older grids have been issued to the SMs.

Past work on concurrent GPU kernel execution [Adriaens et al., 2012, Gregg et al., 2012, Guevara et al., 2009, Ravi et al., 2011] has focused on resource sharing between the co-running kernels to improve throughput. The most recent resource sharing work, i.e. Elastic kernels (Chapter 3), specifically evaluates the Fermi and demonstrates that its FIFO policy serializes kernel execution leading to low throughput and high turnaround times. Essentially, the Fermi’s TBS allocates all resources to the first arriving kernel preventing other kernels from executing concurrently. Elastic kernels allow the runtime to control resources allocated to each concurrently running kernel. By sharing SM resources, serialization is prevented and throughput and turnaround times improved. Since we use actual hardware for evaluation in Chapter 3, no changes could be made to the thread block scheduler. Consequently, the policies presented in that work (and its related work) are resource-sharing policies, not scheduling policies per se.

In this work, we show that the TBS can play a significant role in improving system throughput and turnaround times for concurrent kernel execution. We propose two runtime aware thread block scheduling policies – **Shortest Remaining Time First (SRTF)** and SRTF/Adaptive – for concurrent GPGPU workloads. These policies use estimates of kernel runtime (i.e. execution time) to determine their scheduling decisions when executing concurrent workloads. We propose a novel online runtime predictor for GPU grids to provide these estimates of runtime. To the best of our knowledge, this is the
first work to explore different TBS policies for concurrent workloads on GPUs.

1.5 Contributions of this Thesis

This thesis makes the following contributions:

• In automatic memory management for GPUs:
  – We show that redundant data transfers can be eliminated by maintaining full coherence information for both CPUs and GPUs.
  – We describe compiler analyses, code transformations and runtime support for AMM, a fast and efficient software coherence scheme that performs automatic memory management for GPUs. Designed as a compiler–runtime hybrid scheme, it requires no OS or hardware support (unlike [Saha et al., 2009, Gelado et al., 2010, Jablin et al., 2012]), and is therefore widely applicable to accelerators beyond GPUs. We show that it outperforms all existing proposals and is also highly competitive with manual memory management.

• In resource allocation for concurrent kernels:
  – We identify major inhibitors of GPGPU concurrency in the CUDA execution model where long running kernels and memory transfers act as serialization bottlenecks. We propose a technique to time-slice kernel execution and memory transfers to mitigate this serialization.
  – We identify a lack of mechanisms to control the resource usage of a grid which leads to poor utilization and poor concurrency and therefore propose and describe the use of elastic kernels, a mechanism to control resource allocation for grids during runtime.
  – We propose and study elastic-kernel-aware concurrency policies that perform significantly better than the default LEFTOVER policy, achieving a higher degree of concurrency as well as performance.
  – We find that the current CUDA Streams API and its hardware implementation lead to a high degree of “false serialization”, and we build a replacement Non-serializing Streams API to work around these limitations in the existing API and hardware.

• In scheduling for concurrent kernels:
  – We introduce Structural Runtime Prediction and the Staircase model for online prediction of GPU kernel runtime. This model exploits the uniform structure of grids to predict runtime.
We build an online runtime predictor whose predictions for single-program workloads made after observing only a single thread block evaluated on hardware traces are within 0.48x to 1.08x of actual runtime.

Using this predictor, we implement the Shortest Remaining Time First (SRTF) policy for thread block scheduling which achieves the best system throughput (1.18x better than FIFO) and turnaround time (2.25x better than FIFO) among all policies evaluated. Our implementation of SRTF also bridges 49% of the gap between FIFO and Shortest Job First (SJF), an optimal but unrealizable policy.

To improve fairness of scheduling, we propose SRTF/Adaptive, a resource-sharing and scheduling policy which ensures equitable progress for running kernels while improving system throughput (STP) by 1.16x, average normalized turnaround time (ANTT) by 2.23x and Fairness by 2.95x over FIFO.

### 1.6 Outline

This thesis is organized as follows. Chapter 2 describes our X10+AMM system. We provide additional background for X10 and provide examples of explicit and implicit memory management in X10 GPU programs. We then establish that runtime coherence information is necessary to avoid redundant transfers. Finally, we discuss the design of our compiler analysis and runtime and evaluate it against other recent proposals.

Chapter 3 describes our Elastic Kernels work. We first show that current GPU programs waste a large proportion of GPU resources when running alone. We then show that the current concurrency policy neither improves resource usage nor concurrency. We then establish that concurrency on the GPU is limited fundamentally by kernel execution and memory transfers and also by a number of other implementation issues. We discuss how dynamic control over resource allocation to kernels could solve these fundamental limitations. We also propose an alternate runtime to solve the implementation issues. Finally, we describe the compiler transformations required to convert existing kernels to Elastic kernels, propose several Elastic kernel aware resource-allocation policies and evaluate against proposed state-of-the-art resource allocation policies.

Chapter 4 describes our runtime-aware thread block scheduler that uses Structural Runtime Prediction. This complements our Elastic kernels work in Chapter 3 by demonstrating that in addition to resource allocation, changing the current FIFO scheduling
policy in hardware to a policy like SJF could improve performance even further. We begin by showing that SJF is better than FIFO. Then, we introduce Structural Runtime Prediction and show that it is possible to predict the runtime of a kernel to the degree of accuracy required for making scheduling decisions for the SRTF policy that we implemented. We also detail SRTF/Adaptive, a fairness-aware runtime-aware scheduler. Finally, we detail how such schedulers can be implemented in hardware and evaluate them against resource-allocation policies and the FIFO policy currently implemented in hardware.

Chapter 5 presents a summary of our results and our conclusions. Based on the findings and limitations of this work, we also highlight future avenues for investigation.
Chapter 2

Automatic Memory Management for GPUs

2.1 Introduction

Discrete GPUs have made distributed memory programming ubiquitous. By requiring that all data accessed by a GPU kernel reside in device memory that is largely unaddressable (and inaccessible) from the CPU, discrete GPUs abandoned the shared memory paradigm familiar to most programmers. Since CUDA (and C++) cannot natively express how data is distributed, programmers must manually keep track of data between the CPU and GPU. This is not only tedious, but can lead to both correctness and performance issues. In this chapter, we look at the PGAS language X10 [Charles et al., 2005], a higher-level language, which supports the notion of remote and local data and can also be compiled to the GPU. However, X10CUDA, the dialect of X10 which compiles down to the GPU, still does not automate memory transfers across the CPU and GPU. Even in X10CUDA, the programmer must manually transfer data between the CPU and GPU.

Although there are several proposals that can automate this CPU–GPU memory management [Gelado et al., 2010, Lee and Eigenmann, 2010, Jablin et al., 2011, 2012], our analysis (Section 2.3) shows that all of them introduce redundant transfers. These redundant transfers increase the execution time of programs when compared to the use of hand-tuned manual transfers. Thus, the problem is not only to automate data transfers, but do so in an efficient manner. Our proposal differs from past work in the following ways:

• We present the design of the first full software coherence mechanism for automat-
ing transfers between the CPU and GPU.

- Designed as a compiler–runtime hybrid scheme, it requires no OS or hardware support (unlike Saha et al. [2009], Gelado et al. [2010], Jablin et al. [2012]), and is therefore potentially applicable to accelerators other than GPUs that also use distributed memory architectures.

- Importantly, our design treats the GPU and CPU as peers, by keeping full coherence state for both, allowing it to avoid redundant transfers. Hence it outperforms all existing proposals and is also highly competitive with manual memory management.

We integrate our automatic memory management system into X10CUDA, making the following specific contributions:

- We describe compiler analyses, code transformations and runtime support for AMM, a fast and efficient software coherence scheme that performs automatic memory management for GPUs. We implement the scheme as X10CUDA with Automatic Memory Management (X10CUDA+AMM).

- On programs from the Rodinia Benchmark suite, X10CUDA+ AMM obtains a geometric mean speedup of 1.06x over the original programs that use hand-tuned programmer-specified memory management.

- X10CUDA+AMM is 1.29x faster than CGCM [Jablin et al., 2011], a recent automatic memory management proposal.

- Comparison with ADSM [Gelado et al., 2010], a runtime only system, reveals that it is 1.32x slower on average than the original programs. It also transfers 2.2x more data on average than AMM.

- Evaluation of OpenMPC [Lee and Eigenmann, 2010], a compiler-only system, shows that it is 3.72x slower on average than the original programs and transfers 13.3x more data on average than AMM.

The rest of this chapter is organized as follows. We compare manual memory management in X10CUDA to automatic memory management in X10CUDA+AMM in Section 2.2. We then investigate a naive compiler-based automatic memory management system in Section 2.3 and demonstrate that it performs poorly because of large number of redundant transfers. Section 2.4 presents an overview of our scheme, while the compiler analyses, code transformations and runtime support are described in Section 2.5. We describe our port of CGCM to X10CUDA, resulting in X10CUDA+CGCM in Section 2.6. The results of our evaluation of X10CUDA+CGCM and X10CUDA+AMM are presented in Section 2.7. Other relevant related work is discussed in Section 2.8.
We present our conclusions in Section 2.9.

## 2.2 Background

Programs for NVIDIA GPUs are written in a C++ dialect called CUDA. CUDA code is structured as *kernels* which execute atomically on the GPU. A kernel’s data must reside in GPU memory, which is a distinct address space from that of the CPU. The CUDA API provides functions which a programmer can use to create copies of shared data on the GPU and keep them updated.

X10CUDA [Cunningham, 2010] provides an X10 dialect of CUDA. Kernels are written as X10 Closures and are translated by the X10CUDA compiler to CUDA code. A GPU is exposed to the programmer as an X10 *Place* which is an X10 construct used to distribute data and schedule computations. Data is transferred using the X10 *Rail* type which is a C array-like type whose contents must reside entirely in a single Place. The Rail type provides methods to create Rails in GPU Places and to transfer data between a CPU-side Rail and a GPU-side Rail. We refer the interested reader to Saraswat et al. [2010] for a complete reference to the X10 language, though X10CUDA only supports a limited subset of X10 [X10GPU].

Listing 2.1 shows an excerpt from the Rodinia [Che et al., 2009] *srad* benchmark program that we have ported to X10CUDA with the original hand-tuned memory management statements suitably translated, but intact. First, `Rail.makeRemote` is used to create GPU-side copies of CPU-side arrays (akin to CUDA’s `cudaMalloc()`). In *srad*, `Jcuda` is the GPU copy of `J` and must therefore be kept in sync with it. There is no CPU-side equivalent for `E_c` because it is private to the GPU and communicates intermediate data between the two kernels (`srad_cuda_1` and `srad_cuda_2`). It appears in CPU code because GPU memory must be allocated by the CPU as kernels lack the ability to allocate memory.

Next, the programmer inserts the necessary memory transfers. The statements `J.copyTo` and `J.copyFrom` copy to and from the GPU respectively (akin to CUDA’s `cudaMemcpy()`). The `srad_cuda_1` kernel reads `Jcuda` and writes `E_c`. Therefore, the programmer updates `Jcuda` on the GPU using the `copyTo` call on `J`. The `srad_cuda_2` kernel reads from `E_c` and writes to `Jcuda`, so the `copyFrom` call is used to update the CPU-side version `J`. No calls are made to transfer the `E_c` array since it is never read or written to on the CPU.

Listing 2.2 shows the same *srad* program without explicit memory management. In
var J: Rail[Float]! = Rail.make[Float](...);

val J_cuda = Rail.makeRemote[Float](gpu, ...);
val E_c = Rail.makeRemote[Float](gpu, ...)
...
for (iter = 0; iter < niter; iter++){
    for (...) {
        for (...) {
            // read J
        }
    }
}

finish J.copyTo(..., J_cuda, ...);

// Run GPU kernels
srad_cuda_1(gpu, E_c, ..., J_cuda, ...);
srad_cuda_2(gpu, E_c, ..., J_cuda, ...);

finish J.copyFrom(..., J_cuda, ...);

Listing 2.1: Code excerpt from Rodinia’s srad ported to X10CUDA with manual memory management

var J: Rail[Float]! = Rail.make[Float](...);
val E_c = Rail.make[Float](...)
...
for (iter = 0; iter < niter; iter++){
    for (...) {
        for (...) {
            // read J
        }
    }
}

// Run GPU kernels
srad_cuda_1(E_c, ..., J, ...);
srad_cuda_2(E_c, ..., J, ...);
}

Listing 2.2: Rodinia’s srad in X10CUDA with implicit memory management

18
Figure 2.1: Total data transferred by a naive automatic memory transfer scheme normalized to that transferred by explicitly-specified transfers

this implicit version, no copies of Rails on the GPU are created by the programmer. Nor are memory transfers explicitly specified. The compiler and runtime automatically create copies on the GPU and keep them updated by performing memory transfers when required. This version of srad is therefore much easier to write and involves considerably less book-keeping by the programmer. By removing explicit GPU-specific memory transfers from the source code, we have also increased the portability of the code.

This increase in convenience, productivity and portability cannot come at the expense of performance. In well-written hand-tuned programs, memory transfers are usually not the primary determinants of performance and occupy only a small fraction of execution time. Inefficient automatic memory management can unfortunately change this, as we shall see in the next section.

2.3 The Need for Efficient Memory Management

Figure 2.1 shows the total data transferred between the CPU and GPU by a naive automatic scheme for eight applications from the Rodinia [Che et al., 2009] benchmark suite written in implicit X10CUDA. The results are normalized to the total data transferred by the original X10CUDA program where transfers are specified manually by the programmer.

This naive automatic scheme transfers the read set into the GPU when a kernel is invoked, and copies the write set to the CPU immediately after the kernel ends\(^1\). It is easy to implement in a compiler and has been used in an initial version of Chapel [Sidel-\(^1\)In both the manual and naive versions, the granularity of transfer is an entire Rail (array).]
Figure 2.2: Breakup of data transferred by the naive automatic memory manager

Forik et al., 2011] for the GPU and with minor variations in the OpenMP to GPU compiler [Lee et al., 2009]. The scheme keeps the data always coherent in CPU memory, but it performs a large number of redundant transfers that cause it to transfer 1.5x to 255x more data than manually-specified transfers. We classify these redundant data transfers into three main categories described below and plot this information in Figure 2.2.

**Transfers of non-stale data** On kernel invocation, the naive scheme often transfers data which is not stale on the GPU. This happens, for example, when the CPU acts as a read-only consumer of data produced by the GPU kernel. The original copy of such data is in GPU memory and does not need to be transferred back. The naive scheme does not track the state of shared data making these transfers hard to avoid. Consider also the case of the streamcluster benchmark which uses a randomized CPU algorithm that only occasionally writes to shared data between kernel invocations. Thus, for some invocations, the GPU will have a non-stale copy of the data since the CPU has not written to it, but for others, it will have to update its copy of the data from the CPU. A pure compiler-only scheme would end up transferring data redundantly every time. The redundant transfers in this case are referred to as “not stale” in Figure 2.2. A scheme can prevent non-stale redundant transfers by only transferring data that has changed.

**Eager transfers of data** The naive scheme transfers data eagerly back to the CPU immediately after the kernel finishes. This can lead to redundant transfers, especially in loops that invoke kernels. Consider the following extract from the main loop of the lud benchmark, where all the functions invoked are GPU kernels:

```c
for (...) {
    lud_diagonal_gpu(m, ...)
```
lud_perimeter_gpu(m, ...);
lud_internal_gpu(m, ...);
}
lud_diagonal_gpu(m, ...);

As there is no CPU-side code that reads or writes shared data inside the loop, it is sufficient to transfer the matrix m once at the beginning and once at the end of the loop. In this case, even the transfer at the end of the loop can be postponed to after the lud_diagonal_gpu outside the loop has executed. The naive scheme fails to utilize the opportunity presented by such loops to reduce the number of transfers since it always transfers data before kernel invocation and immediately after the kernel ends. Such loops are not the only source of such redundant transfers. Calls to GPU kernels with no intervening CPU-side reads or writes, like the calls to srad_cuda_1 and srad_cuda_2 in Listing 2.2, also cause redundant transfers in schemes that transfer data eagerly\(^2\). A scheme can avoid eager transfers by only initiating transfers when it knows the data will be read.

**Transfers of private GPU-only data** The naive scheme also transfers private GPU-only data between the CPU and GPU. This is data that is resident on the GPU and never read or written by the CPU. It is used by GPU kernels to communicate intermediate data between themselves (e.g., E_c in Listing 2.2) or across multiple invocations of the same kernel. Since a GPU kernel cannot allocate GPU memory on its own, the CPU must perform the allocation on its behalf and pass the kernel the pointer to the allocated memory. In an automatic scheme, a private GPU array would end up being represented by a CPU array that is never read or written by CPU code, but it would still feature in the read and write sets of GPU kernels. Not knowing that the CPU never reads or writes this array, the naive memory manager would keep this array coherent in CPU memory with transfers that are redundant. In srad, of the six same-sized arrays allocated on the GPU, five are private to the GPU and are only used for communication between the two kernels. Avoiding redundant transfers of GPU-only private data requires identification of GPU-only private data, but a scheme can also take advantage of the fact that the CPU does not read or write GPU-only private data.

So, of the total data transferred by the naive memory manager, only 12% is useful in that the transfer updates a stale copy of the data. Private data constitutes 14%, eager

\(^2\)In srad’s case, these eager transfers are actually of private GPU-only data, and therefore accounted as “private” in Figure 2.2.
transfers about 2.5%, while over 70% of data transferred simply overwrites a non-stale copy (Figure 2.2).

We conclude that an automatic memory management scheme can eliminate redundant data transfers by: (i) not doing eager data transfers, (ii) ensuring that data is only transferred from a non-stale copy to a stale copy, (iii) transferring data only if it will change the destination copy. In practice, the third requirement can be costly to implement, so we assume data that was written to has its value changed as well.

Table 2.1 evaluates existing automatic memory management schemes against the first two requirements. These schemes span from pure compiler-only schemes (OpenMPC) to compiler–runtime schemes (CGCM, DyManD) and fully runtime schemes (ADSM). We see that all current systems fail to eliminate redundant transfers. For example, ADSM does not track changes on the GPU, so it always copies back data from the GPU leading to non-stale transfers. Similarly, compiler-based OpenMPC and CGCM are conservative and transfer modified data eagerly from the GPU back to the CPU even when there is no CPU reader. Some of DyManD’s transfers, too, are redundant non-stale transfers, because it tracks ownership of data but not whether data has changed or not. It also exhibits eager transfers, because although, like ADSM, it only transfers data back to the CPU on a read or a write, on occasion it brings back all GPU data to the CPU leading to eager transfers. We describe and evaluate these schemes quantitatively in Section 2.7.

To address these inefficiencies, we propose AMM, a compiler-assisted runtime coherence scheme for automatic GPU memory management. Our scheme reduces redundant data transfers by (i) tracking data staleness, (ii) identifying and preventing transfers of private GPU-only data, and (iii) transferring data in a lazy fashion.

Table 2.1: A comparison of the ability of automatic memory management systems to eliminate redundant transfers. Legend: P = Partial, C2G = CPU-to-GPU, G2C = GPU-to-CPU

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Only Non-stale to Stale</th>
<th>Prevents Eager</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C2G</td>
<td>G2C</td>
</tr>
<tr>
<td>Naive</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>ADSM [Gelado et al., 2010] (Lazy)</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>ADSM [Gelado et al., 2010] (Rolling)</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>OpenMPC [Lee and Eigenmann, 2010]</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>CGCM [Jablin et al., 2011]</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DyManD [Jablin et al., 2012]</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>AMM</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
2.4 Overview of Automatic Memory Management using Software Coherence

Since the CPU and GPU have distinct address spaces, programmers are forced to maintain copies of shared data in each address space. Without a hardware coherence mechanism, these copies must be kept coherent by software, usually by programmers manually performing data transfers. An automatic software coherence mechanism is therefore a natural alternative to manual GPU memory management. Such a mechanism would detect accesses to shared data and automatically schedule data transfers whenever accesses to stale data are detected. Thus, instead of statically inserting memory transfers to keep data coherent, we insert coherence checks to ensure that data accessed is not stale. These checks cause the runtime to trigger data transfers dynamically as required.

In this section, we present an overview of how GPU memory is automatically managed using our software coherence scheme in our implementation of X10CUDA+AMM. We discuss what our scheme keeps coherent, how it checks the coherence status of data on the CPU and GPU, and how it initiates data transfers. We then apply our scheme to the srad benchmark and demonstrate how it eliminates redundant transfers. The details of compiler analyses, transformations and runtime support required to implement our software coherence scheme outlined here are discussed in Section 2.5. For simplicity of presentation, we assume a system with one GPU in the text, however, our scheme is trivially extensible to multiple GPUs.

2.4.1 Implementing Coherence for Rails

The primary data type that must be kept coherent in an X10CUDA program is the Rail. X10CUDA+AMM kernels also accept primitive types, but these are passed by value. In general, CUDA kernels can accept arbitrary pointer parameters, e.g., to structures in GPU memory. However, X10CUDA does not yet support passing structures to its kernels. Nevertheless, our description of implementing coherence for Rails can be generalized easily to other non-primitive types.

We choose to implement software coherence at the granularity of an entire Rail. This granularity is also used by the manual versions of almost all of the benchmarks we tested. In these benchmarks, GPU kernels are written expecting Rails to be transferred in their entirety before and after kernel execution. We speculate that this is the result of
the large parallelism available on the GPU, the atomic nature of kernel executions and the inability to transfer data mid-kernel to the GPU.

We associate with each Rail state information indicating its coherence status ("stale", "not stale") on each device (CPU, GPU) as well as addresses of the buffers on each device if allocated. This state is maintained by the runtime as described in Section 2.5.3. All Rails start out as not stale on all devices until the first write. Then the device that performed the write owns the copy of the Rail, and the other copies are marked as stale.

2.4.2 Maintaining Coherence on the GPU

A GPU kernel cannot transfer data while executing, so inserting coherence checks into kernels would be futile – we cannot update stale GPU copies mid-kernel execution. Therefore, X10CUDA+AMM must check and update all the Rails in a kernel’s read set before it begins execution.

When a kernel is invoked, checks inserted by our compiler cause our coherence mechanism to update all the stale Rails in the kernel’s read set from their CPU copies. If this is the first time a particular Rail is being transferred to the GPU, our coherence mechanism also allocates memory on the GPU for it before the transfer.

The coherence status of all the Rails written to by the kernel must also be updated on the CPU. Compiler-inserted calls to our coherence mechanism mark the CPU copy of each Rail in the kernel’s write set as stale.

To prevent eager transfers, our approach does not transfer the non-stale GPU copy of a Rail back to the CPU immediately after the kernel finishes. Transfer to the CPU of a Rail updated on the GPU is initiated only when a compiler-inserted check on the CPU detects a stale access to that Rail.

2.4.3 Maintaining Coherence on the CPU

Unlike GPU code, where a kernel’s entry and exit points are convenient points to insert coherence checks, there are no clearly defined boundaries in CPU code where checks to maintain coherence can be inserted.

Our coherence mechanism therefore tracks all CPU reads and writes to every Rail in the program. Each read or write of a Rail in CPU code is preceded by a coherence check inserted by our X10CUDA+AMM compiler. If the check detects a stale Rail, it updates the Rail from its GPU copy and sets its coherence status to not stale, so redundant transfers are avoided.
Figure 2.3: srad runtime control flow. The labels on the edges denote changes in coherence status for the Rails $J$ and $E_c$ on a per-device (CPU/GPU) basis. $J$ is never stale on the GPU after the first iteration of the loop, a fact used by the runtime to eliminate redundant transfers of $J$ to the GPU.

Since checking every read or write would be inefficient, we develop a compiler analysis that conservatively identifies a set of reads and writes that need to have checks inserted before them (Section 2.5.2). Only these reads or writes trigger coherence checks. To prevent overhead from redundant execution of checks at runtime, we also optimize the placement of these checks (Section 2.5.2).

### 2.4.4 Putting it all together

We demonstrate how our software coherence scheme triggers data transfers and avoids redundant transfers in the srad benchmark, whose high-level description was given in Section 2.2.

Figure 2.3 shows the flow of control during srad’s execution. We annotate the edges with changes in coherence status for the two Rails $J$ and $E_c$. The initial part of srad reads an image and initializes $J$. It then executes a loop (Listing 2.2) that consists of a CPU-side reduction of $J$ and consecutive calls to the two CUDA kernels srad_cuda_1 and srad_cuda_2.
Inside the loop, $J$ and $E_c$ experience changes in their coherence status on CPU and GPU due to the code execution on these devices. Data transfers are only triggered based on the coherence status of the Rails. These transfers occur at kernel boundaries (for CPU to GPU transfers) and before the actual read/write (for GPU to CPU transfers). The elimination of redundant transfers happens as follows.

**Elimination of non-stale data transfers** The Rail $J$ is initialized by the CPU causing its GPU copy to be marked as stale. The first GPU kernel, `srad_cuda_1` reads $J$, so in the first iteration of the loop our coherence mechanism transfers it to the GPU. The second GPU kernel, `srad_cuda_2`, writes to $J$ causing the CPU copy to be marked as stale. When the reduce node inside the loop attempts to read $J$, the compiler-inserted check before the read of $J$ operation detects the stale access, and a transfer is initiated to update the CPU copy of $J$. There is no CPU code in the loop that writes to $J$, so the copy on the GPU is never stale. Hence, in subsequent iterations, transfers guarded by the coherence checks avoid the transfer and no transfers of $J$ to the GPU occur in later iterations of the loop.

Although the redundant transfer in this example can be eliminated by purely compiler-based approaches (e.g., using loop peeling transformation), there are situations (e.g. in `streamcluster` as explained in Section 2.3) where runtime techniques are indeed needed to avoid some redundant data transfers.

**Elimination of private GPU-only data transfers** In `srad`, $E_c$ is private GPU-only data. It starts out as not stale on the GPU and CPU, and since it is never written by the CPU, the first invocation of `srad_cuda_1` causes a GPU memory allocation for it but no transfer takes place since it is not stale on the GPU. This GPU kernel writes to $E_c$ so the CPU-held copy of $E_c$ is marked as stale. However, no CPU code ever reads $E_c$ hence there are no stale accesses to it and hence no transfers back to the CPU are triggered by the coherence mechanism. Thus, transfers of private GPU-only data are completely avoided.

**Elimination of eager transfers** Since the software coherence mechanism only transfers data when there are stale accesses, there are no eager transfers. If the `srad` loop consisted only of kernel calls and the reduction code was eliminated, no reads or writes from the CPU would occur, and data would not be transferred back to the CPU during execution of the loop.
As shown above, our software coherence scheme automatically manages shared data between the GPU and CPU and avoids redundant transfers.

2.4.5 Coherence Checks versus Data Transfers

Earlier compiler-based approaches [Lee et al., 2009, Chamberlain et al., 2007, Jablin et al., 2011, Lee and Eigenmann, 2010] insert data transfers statically and use increasingly sophisticated analyses and transformations to avoid redundant transfers. However, the conservativeness inherent in a compiler-only approach cannot avoid all redundant transfers. Our approach of using the compiler to only insert coherence checks while utilizing runtime coherence state to eliminate redundant transfers allows us to avoid this fundamental limitation that plagues earlier work. Compiler conservativeness (e.g. in disambiguating aliases) can only result in redundant coherence checks in our approach, not redundant transfers.

2.5 X10CUDA+AMM

Our X10CUDA+AMM compiler is based on the publicly available X10CUDA compiler [Cunningham, 2010]. Through it, we extend the X10CUDA compiler to insert coherence checks in CPU code and optimize their placement. These coherence checks then invoke our runtime coherence mechanism which maintains coherence status for all Rails and performs dynamic data transfers as required. The X10CUDA runtime uses the asynchronous CUDA API to overlap computation with communication, we plan to use this to perform ahead-of-time transfers to the GPU in the future.

2.5.1 X10CUDA+AMM Programs

Programs written for X10CUDA+AMM differ from those written for X10CUDA in several ways by not requiring explicit memory management. Table 2.2 compares X10CUDA and X10CUDA+AMM languages features used for GPU programming.

To summarize, X10CUDA+AMM programs do not to allocate GPU Rails and transfer them between the CPU and GPU manually. If a GPU kernel requires the use of private data, the programmer simply allocates the private data as a standard CPU-side Rail. Even the arguments passed to GPU kernels refer to the original CPU Rails treating GPU kernels as if they were standard CPU functions. This results in programs that look like Listing 2.2, i.e., smaller, simpler and more portable than their X10CUDA versions.
\[
\begin{array}{|c|c|c|c|}
\hline
\textbf{X10CUDA} & \textbf{E.g.} & \textbf{Usage description} & \textbf{X10CUDA+AMM} \\
\hline
\texttt{makeRemote} & \_c & \text{GPU copy of shared data} & \text{not needed} \\
\texttt{E} & \_c & \text{GPU private data} & \text{Rail.make} \\
\texttt{copyFrom}, & \texttt{copyTo} & \text{Manual transfers} & \text{not needed} \\
\texttt{Remote} & \texttt{GPU Rail} & \text{arguments to kernel functions} & \text{Original (J)} \\
\hline
\end{array}
\]

Table 2.2: Differences between X10CUDA and X10CUDA+AMM language features for GPU programs. Examples are from Listing 2.1.

### 2.5.2 Compiler Analyses and Transformations

To determine which Rails must be kept coherent across devices, the compiler conservatively identifies the read and write sets of each GPU kernel using a standard def–use analysis. It then inserts before each call to a kernel, coherence checks for each Rails in the kernel’s read and write set. At runtime, before the kernel is invoked, these checks make the GPU copies coherent by transferring them as necessary.

The procedure to insert coherence checks in CPU code relies on compiler analysis to determine which reads or writes in the source code are likely to access stale data. The checks are then placed just before these reads or writes. Since this placement might result in redundant checks executed at runtime, our compiler optimizes their placement. The analysis to insert these checks and the optimization of their placement are discussed in detail in the following sections.

**Inserting Coherence Checks in CPU code**

As previously discussed in Section 2.4.3, CPU code lacks the clearly defined boundaries of a GPU kernel where coherence checks can be inserted. The coherence mechanism must therefore check every read or write on the CPU to ensure that no stale data is read or written. We implement this check in the form of two methods, \texttt{check_read()} (Section 2.5.3) and \texttt{check_write()} (Section 2.5.3), that are invoked on a Rail object before a read and a write respectively. If the data in the Rail is stale, these methods trigger memory transfers and ensure that data available to subsequent reads or writes is updated. They also update the Rail’s coherence status to not stale, so that redundant transfers are prevented.

Our compiler could insert calls to these functions before every statement that read
or wrote to a Rail, but that would be inefficient because of the overheads involved. Not all of these calls are necessary, though. A Rail’s coherence status on the CPU can only change when a GPU kernel that reads or writes it is invoked. If it can be determined that a static reference (read or write) would always access coherent data the check can be eliminated. For example, if a static reference to a Rail is preceded by another reference to the same Rail and no GPU kernel invocations occur in between these two references, the second reference is guaranteed to always access coherent data, since the Rail would have been made coherent by the first reference\(^3\).

Our analysis identifies such references so that the compiler can eliminate coherence checks whenever possible. Specifically, for every statement, our approach ultimately calculate two sets: \(FIRST\_READ\) and \(FIRST\_WRITTEN\) that contain Rails for which that statement is the first reader or writer respectively along some path from entry or the previous GPU kernel call. Our compiler then inserts `check_read()` and `check_write()` calls for these Rails before the statement. Note that there may be one first read or first write for a Rail after each GPU kernel call.

The data flow analysis to calculate \(FIRST\_READ\) and \(FIRST\_WRITTEN\) (Figure 2.4) begins by initializing three sets for each statement: the `read` set which contains all Rails read in that statement, the `write` set which contains all Rails written to in that statement, and the `kill` set which contains Rails which may have gone stale during execution of that statement. The `kill` set for a kernel invocation, for example, contains all the Rails read or written to by that kernel.

The dataflow equations in Figure 2.4 are iterated over the control-flow graph. This produces the set of Rails already read \((IN\_read(s))\) and written \((IN\_write(s))\) along all

\[
\begin{align*}
OUT\_WRITE(ENTRY) &= \emptyset \\
IN\_WRITE(s) &= \cap_{p \in pred(s)} OUT\_WRITE(p) \\
OUT\_WRITE(s) &= (IN\_WRITE(s) \cup write) - kill_s \\
FIRST\_WRITTEN(s) &= OUT\_WRITE(s) - IN\_WRITE(s) \\
\end{align*}
\]

\[
\begin{align*}
OUT\_READ(ENTRY) &= \emptyset \\
IN\_READ(s) &= \cap_{p \in pred(s)} OUT\_READ(p) \\
OUT\_READ(s) &= (IN\_READ(s) \cup read) - kill_s \\
FIRST\_READ(s) &= (OUT\_READ(s) - IN\_READ(s)) - IN\_WRITE(s) \\
\end{align*}
\]

Figure 2.4: Data flow equations to determine placement of coherence checks

\(^3\)Note that the coherence is maintained at the granularity of the whole Rail.
paths from ENTRY to statement $s$. Intuitively, these sets contain Rails that have reads or writes respectively along all paths from ENTRY to $s$. If $R$ is a Rail in one of these sets, and if $s$ reads or writes $R$, we do not need to invoke `check_read()` or `check_write()` on $R$ because a prior read or write would have already done so – a coherent copy of $R$ is available on all paths to $s$. It is, therefore, the difference of the $OUT$ and $IN$ sets for reads and writes that gives us the Rails first read or written to in this statement.

Once the dataflow analysis has converged, calls to `check_read()` and `check_write()` are inserted for Rails for each statement’s $FIRST\_READ(s)$ and $FIRST\_WRITTEN(s)$ sets respectively.

### Optimizing Placement of Coherence Checks

The analysis of the previous section inserts `check_read()` and `check_write()` as close to a read or write as possible. However, this can result in sub-optimal placements. Consider this example:

```plaintext
for (i = ...) {
    R.CPU.check_read();
    ... = R(i)
}
```

In this code, `R.CPU.check_read()` will check if $R$ is coherent in every iteration of the loop. However, the loop in this example contains no calls to GPU kernels, and hence $R$ cannot go stale during execution of this loop. If $R$ was stale on entry to the loop, the first dynamic instance of `R.CPU.check_read` will result in a transfer, but the subsequent dynamic instances from the loop will be redundant.

We can hoist this call out of the loop, resulting in:

```plaintext
R.CPU.check_read();
for (i = ...) {
    ... = R(i)
}
```

Our compiler implements a pass that moves calls to `check_write()` or `check_read()` for a Rail $R$ out of a loop if the following conditions hold: (i) The reference to $R$ is loop invariant, (ii) No statement in that loop causes $R$ to possibly go stale on the CPU (directly or indirectly calls a GPU kernel).
var J: Rail[Float]! = Rail.make[Float](...);
val E_c = Rail.make[Float](...)
...
for (iter=0; iter<niter; iter++){
    J.CPU.check_read()
    for (...) {
        for (...) {
            // read J
        }
    }
}

// Run GPU kernels
E_c.GPU.check_write()
J.GPU.check_read()
srad_cuda_1(E_c, ..., J, ...);
E_c.GPU.check_read()
J.GPU.check_write()
srad_cuda_2(E_c, ..., J, ...);
}
Listing 2.3: Listing 2.2 after insertion and of placement optimization of checks
This pass is repeated until no calls are moved. Listing 2.3 is the result of compiling Listing 2.2 by our compiler. Checks have been inserted at the appropriate points, and the check for $J$ has been hoisted out of the doubly-nested reduction loop.

2.5.3 Runtime Support

The X10CUDA+AMM runtime is responsible for tracking the coherence state of a Rail during program execution. It also initiates data transfers dynamically across the CPU and GPU as required.

Modifications to the Rail type

In X10CUDA+AMM, the Rail type is extended to maintain state that is used by the coherence mechanism. Each Rail object contains a coherence status for each device (CPU, GPU) which marks it as “stale” or “not stale” on that device.

This new state also includes the locations of buffers allocated for this Rail on each device. Buffers on the GPU are only allocated the first time data is transferred to it.

We also add the check_read() and check_write() methods to the Rail type. These methods maintain the coherence status for the Rail, and also initiate transfers as necessary. The Rail is always coherent after these methods are invoked.

The check_read() method

The check_read() method (Algorithm 1) signals the coherence mechanism that the CPU or GPU will read the Rail’s data in the code to follow.

This method updates the coherence status of the Rail on the device (CPU or GPU) by marking it not stale on that device, after updating the stale Rail by transferring from another non-stale copy if necessary.

In our implementation, we use a highly optimized version of this method to implement checks in CPU code to lower overheads.

The check_write() method

The check_write() method (Algorithm 2) signals the coherence mechanism that the CPU or GPU will write to the Rail.

The check_write() method first performs the equivalent of a check_read(), updating a stale Rail from a non-stale copy. The coherence status of the Rail on the
Algorithm 1 Pseudocode for the Rail’s `check_read()` method

```plaintext
if device == GPU then
    if GPU.data_ptr == NULL then
        GPU.data_ptr = cuMemAlloc(Rail.length)
    end if
    if GPU.stale then
        copy(destination = GPU.data_ptr, source = CPU.data_ptr)
        GPU.stale ← False
    end if
else
    if CPU.stale then
        copy(destination = CPU.data_ptr, source = GPU.data_ptr)
        CPU.stale ← False
    end if
end if
```

Device is then set to not stale while all other copies of the Rail are marked as stale.

Again, like `check_read()`, our implementation uses an optimized check to lower overheads. Also, in the case of some X10 library functions that completely overwrite a Rail, our implementation skips the call to `check_read()`, saving on the transfer of data that would be overwritten anyway.

Algorithm 2 Pseudocode for the Rail’s `check_write()` method

```plaintext
if device == GPU then
    if GPU.data_ptr == NULL then
        GPU.data_ptr = cuMemAlloc(Rail.length)
    end if
    if GPU.stale then
        copy(destination = GPU.data_ptr, source = CPU.data_ptr)
        GPU.stale ← False
    end if
else
    if CPU.stale then
        copy(destination = CPU.data_ptr, source = GPU.data_ptr)
        CPU.stale ← False
    end if
end if
```

GPU.stale ← True
2.6 CPU–GPU Communications Manager (CGCM) for X10CUDA

The CPU–GPU Communications Manager (CGCM) [Jablin et al., 2011] is a system that automates memory allocations and transfers of data between the CPU and GPU. In that work, CGCM is coupled to an auto-parallelization system that produces GPU code from the programs written in C/C++. In order to compare this state-of-the-art system with X10CUDA+AMM, we port CGCM to X10CUDA, resulting in X10CUDA+CGCM.

Transfers in CGCM are handled by map, unmap and release calls placed into the code by the compiler. The compiler brackets each kernel invocation site with these calls for every Rail in the kernel’s read and write set. Simplistically, a map and its corresponding unmap transfer data to and from the GPU, whereas map and release manipulate a runtime reference count to prevent some redundant transfers.

To decrease redundant transfers, the CGCM compiler performs map promotion, which hoists the above function calls out of loops and function bodies. Map promotion does not succeed if there are CPU references or modifications to the Rails inside the loop/function.

To improve the applicability of map promotion, CGCM performs glue kernel identification whereby CPU code that interferes with map promotion is converted to a single-threaded GPU function where possible. We perform map promotions and glue kernel identifications faithfully for all our benchmarks as well.

Listing 2.5 shows the map, unmap and release function calls that would be inserted by the CGCM compiler for the program in Listing 2.2. \( \text{E}_c \), being a GPU-private variable has no CPU-side references so map promotion succeeds for it, while the reference to \( J \) prevents its map promotion. However, after creating a glue kernel for the CPU-side reduction loop, we are able to perform map promotion for \( J \) as well.

2.7 Evaluation

Our objective is to evaluate the AMM scheme and its impact on program execution time. We compare the performance of our AMM implementation against (i) X10CUDA implementation which uses programmer inserted (manual) memory management, and (ii) X10CUDA+CGCM in Sections 2.7.2–2.7.4. In Section 2.7.5, we contrast AMM against the compiler-only OpenMPC and the runtime-only ADSM schemes. In Section 2.7.6, we quantify the overheads of our implementation.
var J: Rail[Float]! = Rail.make[Float](...);
val E_c = Rail.make[Float](...)

... E_c.map();
for (iter=0; iter < niter; iter++){
    for (...) {
        for (...) {
            // read J
        }
    }
}

// Run GPU kernels
J.map();
srad_cuda_1(E_c, ..., J, ...);
J.unmap();
J.release();
J.map();
srad_cuda_2(E_c, ..., J, ...);
J.unmap();
J.release();
}
E_c.unmap();
E_c.release();

Listing 2.4: Listing 2.2 with CGCM runtime calls as would be inserted by the CGCM compiler

var J: Rail[Float]! = Rail.make[Float](...);
val E_c = Rail.make[Float](...)

...
E_c.map();
J.map();

for (iter=0; iter < niter; iter++){
    J.map();
    srad_glue_kernel_1(...);
    J.release();

    // Run GPU kernels
    ...
}
J.unmap()
...

Listing 2.5: srad after glue kernel identification, insertion of CGCM runtime calls and
map promotion. The glue kernel srad_glue_kernel_1 implements the reduction loop
(lines 7–11 in Figure 2.4) on the GPU
Table 2.3: Rodinia benchmarks used in this study, descriptions adapted from the Rodinia documentation. Legend: KL=Kernel launches, A=Allocations, G=Gets to CPU, P=Puts to GPU

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input Size</th>
<th>KL</th>
<th>A</th>
<th>G+P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpropagation (backprop)</td>
<td>Trains neural network</td>
<td>65536</td>
<td>2</td>
<td>6</td>
<td>3+5</td>
</tr>
<tr>
<td>Breadth-First-Search (bfs)</td>
<td>Classic graph algorithm</td>
<td>1M</td>
<td>24</td>
<td>7</td>
<td>13+18</td>
</tr>
<tr>
<td>Hotspot (hotspot)</td>
<td>Simulates processor temperature</td>
<td>512x512</td>
<td>1</td>
<td>3</td>
<td>1+2</td>
</tr>
<tr>
<td>Kmeans (kmeans)</td>
<td>Clustering Algorithm</td>
<td>494020</td>
<td>38</td>
<td>4</td>
<td>37+75</td>
</tr>
<tr>
<td>LU Decomposition (lud)</td>
<td>Linear algebra</td>
<td>2048x2048</td>
<td>46</td>
<td>1</td>
<td>1+1</td>
</tr>
<tr>
<td>Needleman–Wunsch (nw)</td>
<td>Aligns DNA sequences</td>
<td>2048x2048</td>
<td>255</td>
<td>3</td>
<td>1+2</td>
</tr>
<tr>
<td>SRAD (srad)</td>
<td>Reduces noise in images</td>
<td>2048x2048</td>
<td>4</td>
<td>6</td>
<td>2+2</td>
</tr>
<tr>
<td>StreamCluster (streamcluster)</td>
<td>Streaming clustering algorithm</td>
<td>65536</td>
<td>1611</td>
<td>5</td>
<td>3222+3224</td>
</tr>
</tbody>
</table>

2.7.1 Methodology

The benchmarks used in the evaluation (Table 2.3) are taken from the Rodinia Benchmark suite (v1) [Che et al., 2009]. The original benchmarks are available as C++ and CUDA programs and contain manually parallelized kernels and explicit hand-tuned memory allocations and transfers. We port these benchmarks to X10CUDA as faithfully as possible, maintaining the same memory allocations and transfers as in the C++ version. These programs are referred to as the X10CUDA (manual) versions [X10BMK].

X10CUDA does not support passing arrays of structures to the GPU, so we convert programs that use arrays of structures extensively (backprop, bfs, streamcluster) to use structures of arrays instead. The version of X10CUDA used in our work also does not support GPU textures or constant memory, so we modify the kmeans benchmark which uses these features to work without them.

The heartwall, leucocyte, cfd and mummergpu benchmarks are not included because they make extensive use of C/C++ native libraries, GPU SIMD types and structures as kernel arguments, none of which are supported by the base X10CUDA compiler.

We run hotspot for 360 iterations and srad for 100 iterations as described in the Rodinia paper [Che et al., 2009].

We rewrite each X10CUDA version of the benchmark for X10CUDA+AMM using Table 2.2 as a guide. In this rewritten version, all data transfers are implicit. We convert GPU-only private Rails to CPU-side Rails.

To obtain the X10CUDA+CGCM versions, we take the implicit version of the benchmark produced for X10CUDA+AMM and perform Map Insertion, Map Promotion and Glue Kernel Identification manually. Map promotion hoists the map calls for Rails out of all loops in all programs except kmeans and streamcluster. In hotspot, the main loop limits map promotion because the Rodinia source code refers to Rails used
by the GPU kernels in a loop-dependent fashion. In srad, a glue kernel for the CPU-side reduction improves the applicability of map promotion. No such opportunities are found in kmeans or streamcluster.

Lastly, the naive version of each benchmark uses the same source code as the X10CUDA+AMM but has transfers inserted statically by the compiler. Note that the manual version of each benchmark acts as the baseline for all our experiments.

All of the resulting versions are compiled using the same base X10 compiler, which invokes gcc 4.4.5 and CUDA 2.3 compilers with -O3 optimizations. The benchmarks are run on a system equipped with an NVIDIA Tesla C1060 with 4GB RAM. The CPUs are two quad-core Intel Xeon E5405 2GHz with 16GB RAM. In all our experiments the CPU code and the GPU code are executed in a single X10 Place each, the first uses two CPU cores (the default) and the second uses a single GPU. Each program was run 10 times, and the runtime averaged across those ten runs.

We report the normalized execution time of X10CUDA+AMM, X10CUDA+CGCM and X10CUDA+Naive relative to the runtime of the manual version of the program which has hand-tuned memory transfers as in the original Rodinia source code. Normalized execution times are aggregated using geometric means (geomean).

### 2.7.2 Performance of Individual Schemes

Figure 2.5 shows the normalized execution times of the implicit X10CUDA+Naive, X10CUDA+CGCM and X10CUDA+AMM versions over their manual version.

First, for reasons discussed in Section 2.3, the X10CUDA+Naive implementation of memory management performs very poorly for all programs, slowing down execution time by 3.14x (geomean).

Second, X10CUDA+AMM is able to perform as well as the manual version except for nw and backprop where it is within 4%. In two benchmarks, srad and streamcluster, AMM outperforms the manual version of the benchmarks by 34% and 18% respectively. This is due to the reduction in redundant transfers as explained later in Section 2.7.3.

While X10CUDA+CGCM achieves a higher performance in srad (improvement of 44% primarily due to the glue kernel optimization which reduces the overall data transferred) and comparable performance in bfs, lud and nw, it suffers significant slowdowns in hotspot, kmeans and streamcluster – 91%, 26% and 205% respectively – compared to the manual versions.
Overall, X10CUDA+CGCM is slower than manual by 22%. X10CUDA+AMM, on the other hand, is 6% faster than manual. X10CUDA+AMM is 1.29x faster than X10CUDA+CGCM and 3.33x faster than X10CUDA+Naive.

Next, we compare the amount of data transferred by these schemes to understand their speedups.

### 2.7.3 Data Transferred by X10CUDA+AMM

From Figure 2.6, other than in streamcluster, X10CUDA+AMM does not transfer more data than the equivalent manual version. Further, for three of the benchmarks – backprop, srad, kmeans, it actually transfers less data than the manual version.

Examination of the source code for backprop, srad, kmeans reveals they have redundant transfers in the original Rodinia source code that transfer non-stale data to the GPU. X10CUDA+AMM detects and eliminates these transfers. In srad’s case, for example, as we noted in Section 2.4.4, the J.copyTo inside the loop (Listing 2.1) is redundant after the first iteration.

The case of streamcluster is more complicated. The excess data transferred is not due to X10CUDA+AMM transferring non-stale or private data or even through eager data transfers. It turns out that streamcluster is the only benchmark that performs partial transfers of a Rail in the manual version. Since X10CUDA+AMM always transfers
back an entire Rail, the automatic version of streamcluster transfers back more data (3.91x) for one Rail (work_mem.h), than the manual version. However, there are other Rails in streamcluster that are transferred from the CPU to the GPU. These Rails (p_assign, p_weight and p_cost) are only modified occasionally by the CPU inside a randomized algorithm. The manual version of streamcluster transfers these Rails on every kernel invocation, but as X10CUDA+ AMM tracks the coherence state of each Rail, they are only transferred if they have been modified by the CPU. In this case, X10CUDA+AMM transfers only 9.5GB instead of 11.5GB transferred by the manual version. Overall, X10CUDA+AMM transfers 1.37x more data than manual in streamcluster. However, the automatic version of streamcluster is 20% faster than the manual version. Overall X10CUDA+AMM transfers on average 0.9x the data transferred by manual.

## 2.7.4 Data Transferred by X10CUDA+CGCM

Figure 2.6 shows that for six of the eight programs, X10CUDA+ CGCM transfers more data than the equivalent programmer-tuned manual version. The amount of data transferred is as high as 240x for hotspot, 9.5x for streamcluster and 13.9x for kmeans. In lud, the amount of data transferred under X10CUDA+CGCM is equal to that transferred under manual because the map promotion successfully hoists the map outside the main loop, and then outside the function. In srad, the glue kernel leads to all code being run
on the GPU therefore transferring only 0.06x of the data transferred by manual.

To examine why X10CUDA+CGCM transfers more data for the remaining six programs we use the analysis of Section 2.3 to plot in Figure 2.7, a breakup of the data transferred by X10CUDA+CGCM. As described in Section 2.6, X10CUDA+CGCM does not track CPU reads or writes at all. This causes it to transfer non-stale data redundantly, prevents it from identifying private GPU-only data and also causes eager transfers. Note that GPU-only private data has no CPU reads or writes, so map promotion does succeed in reducing the amount of private data transferred. However, X10CUDA+CGCM still ends up transferring GPU-only private data at least once.

Loops in which map promotion failed end up contributing to both non-stale redundant transfers and eager transfers. In kmeans and streamcluster, where opportunities to perform map promotion were not found, X10CUDA+CGCM performs equivalent to the naive scheme.

To summarize, X10CUDA+CGCM ends up transferring 2.9x more data (geomean) than manual and only 14% of the total data transferred by X10CUDA+CGCM turns out to be useful. Compared to this, all the data transferred by our X10CUDA+AMM is useful.

### 2.7.5 Comparison with Cetus+OpenMPC, ADSM and DyManD

To contrast CGCM and AMM with a compiler-only approach, we present data transfer information for benchmarks compiled by the OpenMPC compiler [Lee and Eigenmann,
which translates OpenMP programs to CUDA. It inserts data transfers statically by performing interprocedural analysis to identify GPU-resident and CPU-live variables. We were only able to compile six Rodinia benchmarks with Cetus 1.3. Figure 2.6 shows that the OpenMPC compiler transfers from 1.22x to 256x the data transferred by the manual versions for these six benchmarks, even when the benchmarks were compiled with the highest level of memory transfer optimizations as supported by OpenMPC. Overall, OpenMPC transfers 12x (geomean) more data.

We also evaluate ADSM [Gelado et al., 2010] which is a fully runtime scheme, but requires the programmer to annotate shared data. It uses OS-level memory protection mechanisms to intercept reads and writes to this shared data. We use the libgmac 0.0.20 [Gelado et al.] implementation and evaluate both Lazy and Rolling schemes. We find that Lazy is faster and transfers less data as compared to the Rolling scheme. ADSM handles partial data transfers and tracks coherence information on the CPU so it is able to eliminate redundant transfers in srad and streamcluster. However, since it does not track GPU coherence information, it transfers non-stale data from the GPU, for example in kmeans. We observe that data structure initialization can confuse ADSM’s Rolling scheme leading to repeated redundant transfers. In the extreme case exhibited by bfs, this transfers $2.94 \times 10^6$ times more data. Compared to manual, on average ADSM transfers 2x more data for Lazy, and 9x (1.4x without bfs) more data for Rolling.

These excess data transfers lead to loss of performance, as seen in Figure 2.8 which shows the performance of the ADSM schemes normalized to Rodinia CUDA code. ADSM is faster than manual only for srad, and is upto 2.2x slower for kmeans. Overall, ADSM Lazy is 32% (geomean) slower than the manual Rodinia code and Rolling is 4x slower (39% without bfs). Similarly, OpenMPC’s excess data transfers cause it to never be faster than manual for any program. Overall it is 3.72x (geomean) slower than manual.

DyManD [Jablin et al., 2012] extends CGCM with a memory optimization framework similar to ADSM. However its state machine is considerably less detailed than that of ADSM – tracking only ownership but not coherence state. We therefore believe that its performance will be similar to that of ADSM.

### 2.7.6 Overheads of redundant check_read() and check_write()

Table 2.4 shows the number of check_read() and check_write() calls that are inserted by our compiler into each benchmark. The OoL entry in the table corresponds to the sum of check_read() and check_write() calls that end up outside loops
Figure 2.8: Normalized execution time for OpenMPC and ADSM relative to the Rodinia CUDA manual benchmarks

![Normalized Execution Time Graph]

Table 2.4: Number of coherence checks inserted and hoisted per benchmark. OoL refers to checks that have been placed outside-of-loops.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>check_read()</th>
<th>check_write()</th>
<th>OoL</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop</td>
<td>24</td>
<td>15</td>
<td>29</td>
</tr>
<tr>
<td>bfs</td>
<td>3</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>lud</td>
<td>9</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>nw</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>srad</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>hotspot</td>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>kmeans</td>
<td>13</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>streamcluster</td>
<td>53</td>
<td>37</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 2.5: Number of dynamic check_write() and check_read() executions compared to number of dynamic reads and writes per benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>reads + writes</th>
<th>check_write</th>
<th>check_read</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop</td>
<td>18,088,926</td>
<td>2,228,377</td>
<td>2,228,345</td>
</tr>
<tr>
<td>bfs</td>
<td>859,964</td>
<td>65</td>
<td>12</td>
</tr>
<tr>
<td>hotspot</td>
<td>1,835,384</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>kmeans</td>
<td>1,975,962,573</td>
<td>6487</td>
<td>6516</td>
</tr>
<tr>
<td>lud</td>
<td>65,537</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>nw</td>
<td>16,822,812</td>
<td>264</td>
<td>2</td>
</tr>
<tr>
<td>srad</td>
<td>14,221,329</td>
<td>2</td>
<td>103</td>
</tr>
<tr>
<td>streamcluster</td>
<td>4,733,235,158</td>
<td>21,228,272</td>
<td>26,905,978</td>
</tr>
</tbody>
</table>

42
after the optimization procedure described in Section 2.5.2. The calls that are not hoisted are in loops that contain GPU kernel calls and/or that have loop-dependent references to Rails. In addition, in our current implementation, we do not hoist checks that are inside conditional blocks of code, so as to prevent redundant transfers, even if those conditional blocks lie inside a loop.

Table 2.5 shows the number of actual dynamic calls that are executed during a run of each benchmark. The absolute number of dynamic `check_read()` and `check_write()` calls are low when compared to the actual number of reads and writes. Only in backprop do they form a significant fraction (12.3%) of the number of reads and writes. This is because of a loop that iterates over the outermost index of a 2D Rail in the innermost loop, thus creating a loop dependence that prevents the hoisting of the `check_read()` and `check_write()` calls. For all other benchmarks, the fraction of checks to reads and writes is less than 0.01% with the exception of streamcluster for which the checks fraction is 1%.

From this we estimate the runtime overhead for execution of these checks. We measure the cycles taken by `check_read()` and `check_write()` using the RDTSC instruction [Paoloni, 2010]. On average, these methods take 8–9 cycles per call\(^4\) when no data is transferred. The overhead in execution time is very low (the maximum of 2.46% is experienced by backprop) with the average overhead being around 0.35% of runtime.

## 2.8 Related Work

Software caching schemes have been used on the Cell accelerator [Eichenberger et al., 2005] to communicate data between the CPU and the Cell’s SPUs. However, unlike SPUs, GPUs cannot initiate memory transfers, so it is not possible to realize software caches on current GPUs.

The OpenMPC compiler translates OpenMP code to GPU kernels [Lee and Eigenmann, 2010] and inserts transfers by using an interprocedural analysis to identify GPU-resident and CPU-live variables at compile-time. Our scheme identifies possibly stale reads or writes instead and inserts coherence checks, not transfers. OpenMPC is evaluated in Section 2.7.5.

\(^4\)Our implementation is very efficient. In an experiment where all reads and writes were subject to coherence checks, we observed noticeable slowdowns only in kmeans (19%) and streamcluster (8%).
Saha et al. [2009] describe a programming model for C on heterogeneous x86 platforms which provides a logical shared address space between the CPU and an accelerator modeled on the Larrabee [Seiler et al., 2008]. Coherence is maintained by the CPU and the accelerator by intercepting reads and writes using the operating system’s memory protection mechanisms. However, this requires the accelerator to support virtual memory and protection to support its coherence mechanisms, something current GPUs lack.

We have compared and evaluated our work against CGCM, ADSM and OpenMPC in this work.

2.9 Conclusions

In this chapter, we have presented the AMM system which is a hybrid compiler-assisted runtime coherence scheme to automatically manage all aspects of CPU–GPU communication including allocations and memory transfers. Our design was motivated by the observation that not all redundant transfers can be completely eliminated at compile time. Our system can be used both by a programmer writing GPU kernels manually and as part of a parallelizing compiler. It significantly eases the task of programming the GPU by eliminating a potential source of errors and performance problems.

We have incorporated AMM into the X10CUDA compiler and runtime, and evaluated it on a set of benchmark programs from the Rodinia suite. The AMM system achieves comparable performance to programmer-inserted manual memory management. In fact it achieves a speedup of 1.06x over manual by eliminating some redundant data transfers. Compared to a state-of-the-art memory management system, CGCM [Jablin et al., 2011], our AMM system is faster by 1.29x. Moreover, compared to other existing runtime-only and compiler-only proposals, it also transfers 2.2x to 13.3x less data on average.
Chapter 3

Elastic Kernels for Improved GPGPU Concurrency

3.1 Introduction

Graphics Processing Units (GPUs) have evolved from fixed-function hardware pipelines to multicore data-parallel computation engines. As many are now components of some of the world’s fastest supercomputers [TOP500] and used exclusively for General Purpose GPU (GPGPU) code (i.e. non-graphical or computational code), they are increasingly being designed to have CPU-like features. For instance, the Fermi GPU [Glaskowsky, 2009] became the first GPU to have the ability to execute multiple kernels concurrently. As discussed earlier in Section 1.3, this ability may have been strongly motivated by the wastage of resources occurring when allocating resources to fixed-size grids. In this chapter, we examine concurrency on the Fermi GPU and find it flawed – in general, it neither reduces resource wastage nor promotes concurrency. We find several impediments to concurrency and propose that the hardware and runtime support elastic kernels whose resource allocation is controlled by the runtime or hardware. We show that elastic kernels can alleviate all of the issues we raise.

Specifically, we make the following contributions:

• We identify major inhibitors of GPGPU concurrency in the CUDA execution model where long running kernels and memory transfers act as serialization bottlenecks. We propose a technique to time-slice kernel execution and memory transfers to mitigate this serialization.

• We find that the current CUDA Streams API and its hardware implementation lead to a high degree of “false serialization”, and we build a replacement Non-
**3.2 Motivation**

The NVIDIA Fermi is the first GPU available with the software and hardware ability to run multiple GPU grids concurrently. Therefore, we use it in our examination of GPGPU concurrency mechanisms and policies.

Table 3.1 presents the resources used by the grids of all 18 kernels of the Parboil2 benchmark suite when executed on the NVIDIA Fermi GPU (Table 1.1). Observe that no grid from any kernel utilizes 100% of all resources. The vast majority of grids exhaust only a single resource. On the Fermi, this is either registers or resident blocks. Overall, over 40% of threads and blocks, 30% of registers and 80% of shared memory are not used on average. When we examine the benchmarks in the Rodinia 2 suite [Che
Table 3.1: Resource usage of Parboil2 kernels on the Fermi GPU. Legend: TB=Thread Blocks, TPB=Threads per Thread Block, T=Threads used, R=Registers used, S=Shared Memory used, B=Thread Blocks used. All usage is expressed as percentage of total GPU resources.

et al., 2009], we arrive at similar conclusions: 35% of threads, 47% of registers, 88% of shared memory and 52% of blocks are not utilized on average.

Since these wasted resources cannot be utilized by thread blocks from the same grid, the Fermi GPU supports concurrent execution of up to 16 grids. Independent grids must be identified and the programs modified in order to convey dependency information to CUDA. None of the programs in the Parboil2 or Rodinia 2 benchmarks currently executes multiple grids in parallel; support for GPU concurrency is relatively new. So, in this work, we use multiprogrammed workloads which are a convenient source of independent grids. Therefore, in this chapter, the term “GPU concurrency” exclusively refers to the concurrent execution of independent grids from multiprogrammed workloads.

The Fermi’s concurrency policy is not publicly documented. Our experiments using microbenchmarks of concurrently executing synthetic kernels suggest that the Fermi uses the LEFTOVER policy. Under this policy, a grid begins concurrent execution only if there are enough resources to allow execution of at least one of its thread blocks. This is a fairly conservative policy and seems directed at improving resource utilization.
whenever possible. It is a poor policy for concurrency, however, because it cannot guarantee that two independent grids will always execute concurrently, i.e. form a *concurrent pair*. If a grid consumes all thread blocks, for example, no other independent grid can execute concurrently with it. Thus, for concurrent execution under this policy, the programmer must ensure that each independent grid will not consume too many resources. Table 3.2 shows that 9 of the 18 Parboil2 kernels do not form concurrent pairs at all (i.e. zero concurrent pairs) under this policy. Similarly, in the Rodinia 2 [Che et al., 2009] benchmark suite, 22 kernels (of the total 38) do not form concurrent pairs under this policy. The LEFTOVER policy also makes concurrent execution of grids a function of GPU resources, which keeps changing over GPU generations.

An alternative policy, *spatial partitioning*, is suggested by Adriaens et al. [2012]. Under this policy, the streaming multiprocessors (SM) of a GPU are partitioned among thread blocks of concurrently executing grids. While independent pairs of grids can therefore always execute concurrently, albeit with fewer resources, their work does not consider the wastage due to the GPU’s resource allocation mechanisms. Each partition wastes the same percentage of resources as when the grids were executing alone. Thus, this policy does not improve GPU resource utilization.

In this work, we show that the grid’s resource allocation considerations can be separated from the programming model considerations by the use of what we term “elastic kernels”. Since elastic kernels provide finer control over GPU resources, we can design concurrency policies that improve concurrency *and* resource utilization. These policies can then attempt to achieve both high system throughput and low turnaround time when compared to current policies that only try to achieve maximum concurrency or maximum resource utilization.

### 3.3 Limiters of GPU Concurrency

The performance benefits of GPU concurrency are governed by Amdahl’s Law – speedup is limited by the extent of serialization. Serialization due to dependences demanded by program semantics is unavoidable, but serialization which arises as artifacts of the GPU execution model decreases potential GPU concurrency. In this section, we identify a number of serialization factors, all of which must be tackled to reduce the extent of serialization. Before we describe these causes of serialization, we first review how concurrency is expressed in GPU programs.
<table>
<thead>
<tr>
<th>Program</th>
<th>Kernel</th>
<th>CP</th>
<th>KT (ms)</th>
<th>Calls</th>
<th>FPT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs</td>
<td>BFS_in_GPU</td>
<td>16</td>
<td>14.41</td>
<td>2</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>BFS_multi_blk...</td>
<td>15</td>
<td>49.57</td>
<td>1</td>
<td>63</td>
</tr>
<tr>
<td>mri-q</td>
<td>ComputePhiMag...</td>
<td>16</td>
<td>0.002</td>
<td>1</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>ComputeQ_GPU</td>
<td>2</td>
<td>44.91</td>
<td>2</td>
<td>99.998</td>
</tr>
<tr>
<td>fft</td>
<td>GPU_FFT_Global</td>
<td>0</td>
<td>0.11</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td>stencil</td>
<td>block2D_hybrid...</td>
<td>2</td>
<td>2.70</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>cutcp</td>
<td>cuda_cutoff...</td>
<td>0</td>
<td>2.86</td>
<td>11</td>
<td>100</td>
</tr>
<tr>
<td>tpacf</td>
<td>gen_hists</td>
<td>13</td>
<td>840.40</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>histo</td>
<td>histo_final</td>
<td>0</td>
<td>0.07</td>
<td>100</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>histo_intermediates</td>
<td>0</td>
<td>0.18</td>
<td>100</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>histo_main</td>
<td>0</td>
<td>0.85</td>
<td>100</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>histo_prescan</td>
<td>0</td>
<td>0.03</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>sad</td>
<td>larger_sad_calc_16</td>
<td>15</td>
<td>0.04</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>larger_sad_calc_8</td>
<td>15</td>
<td>0.19</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>mb_sad_calc</td>
<td>0</td>
<td>0.80</td>
<td>1</td>
<td>77</td>
</tr>
<tr>
<td>mm</td>
<td>mysgemmnNT</td>
<td>2</td>
<td>5.51</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>lbm</td>
<td>performStream...</td>
<td>0</td>
<td>23.69</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>spmv</td>
<td>spmv_jds_texture</td>
<td>0</td>
<td>0.13</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 3.2: Concurrent Execution possible on Fermi (LEFTOVER policy) for Parboil2 kernels. Legend: CP=Concurrent Pairs with kernel starting first (0–17), KT=Average Kernel Time, FPT=Fraction of GPU Program Time.

### 3.3.1 Task-level Concurrency in GPU programs

The tasks in a GPU program can be divided broadly into memory operations (memory allocations, memory transfers, memsets, etc.) and kernel executions. Task-level concurrency in a GPU program therefore involves exploiting concurrency between memory operations and kernels. CUDA uses a mechanism called Streams [NVCCPG, Rennich, 2012] to expose task-level concurrency in a GPGPU program. Each stream is a queue-like structure into which the CPU program inserts commands. Each command is either a memory operation or a kernel execution. Commands placed in the same stream execute in order, but those from different streams can execute concurrently subject to resource availability.

Figure 3.1 presents a simplified view of the various components that play a role in the concurrent execution of GPU tasks: (i) Streams, which contain commands to be sent to the GPU, (ii) the Stream Scheduler, which dispatches commands from each stream to the hardware units, sending memory transfer commands to one of the two DMA engines depending on the direction of the transfer, and sending kernel execution commands to the Thread Block Scheduler, (iii) The Thread Block Scheduler which instantiates a grid for the kernel as specified by the programmer and then dispatches thread blocks from the grid to the Streaming Multiprocessors as described in Section 1.3. The Fermi Thread
Figure 3.1: Components that affect concurrency on the Fermi GPU. Streams contain Memory (M) or Kernel (K) commands. A DMA engine performs unidirectional data transfers. With two DMA engines on the Fermi, two memory transfers may run concurrently with kernel execution.

Block Scheduler can dispatch thread blocks from up to 16 concurrently executing grids.

In the following sections, we describe how each component of the above GPU execution model can contribute to unnecessary serialization.

### 3.3.2 Serialization due to Lack of Resources

A GPU consists of a number of Stream Multiprocessors (SM). Each SM has a fixed amount of resources in terms of thread blocks, registers, threads and shared memory. For a kernel to begin execution on an SM, resources for the execution of at least one of its thread block must be available. Under the current \textsc{leftover} policy, two grids cannot execute concurrently if the one scheduled first consumes too many resources to allow the other to begin execution. Since the resources consumed are specified by the programmer, the Thread Block Scheduler is forced to serialize the execution of such grids.

One way to guarantee concurrent execution and prevent serialization is to partition the SMs among the different grids [Adriaens et al., 2012]. This guarantees that each grid will always have resources to execute and multiple grids can execute across different SMs. However, as noted in Section 3.2, it does not address the problem of under-
utilization of resources. Hence, in this chapter, we explore an alternative technique that seeks instead to control the amount of resources occupied by a grid to allow concurrent execution. This will also allow multiple grids to share the same SM. In Section 3.4, we show that support for kernels which we term elastic kernels can allow control of SM resources by permitting modification of their grid and thread block dimensions. This allows us to design resource allocation policies that can guarantee resources for grids to varying degrees and thus allow concurrent execution, preventing serialization due to lack of resources.

3.3.3 Serialization due to Inter-stream Scheduling

The LEFTOVER concurrency policy renders the concurrency relation non-commutative, making the order in which grids are dispatched to the Thread Block Scheduler important. Figure 3.2 illustrates this by an example using grids from cutcp, bfs and fft. From Table 3.1, we know that the grids from cutcp and fft consume 100% of resident blocks. Thus, grids starting after them will not have resident blocks or threads to run and will have to wait. However, neither grid from bfs consumes all of the resources, so it is possible for other grids to execute with them. Assume now that GPU_FFT_Global from fft is already running. Now, the grids from cutcp and bfs arrive, in that order, and must wait for GPU_FFT_Global to complete. After GPU_FFT_Global finishes execution, the scheduler must decide which grid from cutcp or bfs must execute first. A FIFO scheduler would execute the kernel from cutcp first, thus serializing bfs’s grid, while a concurrency-aware scheduler would first dispatch bfs, leading to concurrent execution with cutcp. Thus, to avoid serialization, we either need to reorder grids to maximize concurrency or we need to ensure that order of arrival does not matter. The latter can be achieved by controlling resource allocation so that no one grid blocks the execution of another. In Section 3.4.3, we describe several resource-limiting policies using elastic kernels to allow kernels to execute regardless of dispatch order.

3.3.4 Serialization due to Kernel Execution

A long-running grid can serialize the execution of grids from other programs if it consumes too many resources, preventing other grids from executing concurrently with it. Even if it allows concurrent execution, the resources it occupies will be unavailable to other kernels for the duration of its execution, possibly slowing them down. The ability to time slice such long running kernels can help improve not only turnaround time, but
also throughput and resource utilization. Grid execution is not pre-empted in current CUDA implementations. The large amount of state involved (also noted by Adriaens et al. [2012]) make pre-emption prohibitively expensive. However, this is true only if we want the ability to stop and restore a grid at any arbitrary point of execution. Instead, we propose to use the discrete nature of grid execution to identify points where little or no state will need to be preserved. Completion of a thread block is an example of one such point. The thread block is the unit of state in a grid and there is no state to save after it completes. Therefore the thread block scheduler can switch from kernel $K_1$ to $K_2$ by simply: (i) halting the dispatch of ready thread blocks from $K_1$’s grid, allowing current thread blocks to complete, and (ii) starting the dispatch of ready thread blocks from $K_2$’s grid. 

While this still means that we cannot pre-empt a running block, in practice this scheme enables concurrency, and requires no additional state to be preserved. The Thread Block Scheduler already maintains state on which blocks have completed and therefore can restart a grid where it left off. We describe our implementation for time-slicing of GPU grids in Section 3.4.4.

### 3.3.5 Serialization due to Memory Transfers

Current NVIDIA GPUs have two DMA engines. One performs memory transfers to the GPU and the other performs memory transfers from the GPU. Thus, the GPU can sustain two memory transfers at the same time. Further, memory transfers can execute in parallel with kernel execution. However, since only one transfer can be active in a given direction, memory transfers can cause serialization similar to that caused by long-running grids. A large memory transfer in one program can stall progress in other concurrently executing programs as they wait for the DMA engine to become free.
Increasing the number of DMA engines is not a solution, since PCIe bandwidth is the actual limiting factor. But increasing the number of memory transfers that can be active and time-slicing between them can reduce waiting time. Figure 3.3 uses lbm and bfs to illustrate the problem and how timeslicing memory transfers can help. Initially, a 100MB memory transfer in lbm prevents execution of bfs’s memory transfer, which in turn causes bfs’s kernels to serialize behind lbm’s kernel. By timeslicing the large memory transfer, bfs’s memory transfer finishes early, and its kernel can execute in parallel with lbm’s memory transfer. In our implementation, we break up a long-running memory transfer into smaller chunks and interleave them with chunks of other active memory transfers, thus achieving memory transfer timeslicing.

### 3.3.6 Serialization in the CUDA API

The CUDA API contains several functions that implicitly synchronize commands from different streams [NVCCPG], i.e. they act as program-wide barriers for GPGPU functionality. For example, device memory allocation functions `cudaMalloc` and `cudaFree` will wait for all currently executing commands to complete before executing, and will prevent later commands from beginning execution until they complete. The effects of this barrier-like behaviour can be severe – if a `cudaFree` is issued when the GPU is executing a kernel, it will wait as long as the kernel takes to finish execution while stalling progress across the rest of the program. The `cudaMemset` function is another source of serialization, but one that seems unnecessarily so since it can be associated with a stream [NVCCPG], unlike `cudaFree` which is global in nature. It is difficult for us to comment on the difficulty of implementing non-serializing API functions without concrete low-level details of the driver and hardware, but we observe in our experiments that serializing functions can lower throughput drastically.
3.3.7 Serialization in the Implementation

Finally, we note a source of serialization that is specific to the implementation on the Fermi. To maintain ordering of commands in a single stream, the Fermi uses “signals” [Rennich, 2012] between the kernel and memory transfer hardware queues. These signals act as barriers and prevent Kernel–Memory Transfer, Memory Transfer–Kernel, Memory Transfer–Memory Transfer, and Kernel–Kernel dependencies (abbreviated as $K$–$M$, $M$–$K$, $M$–$M$ and $K$–$K$ respectively) from being violated in a stream. Unfortunately, in the current implementation, these signals act as a barrier between commands of all streams. Thus, a $K$–$M$ (or $M$–$K$) dependency in one program imposes a barrier between commands from all concurrently executing GPU programs. The CUDA Programming Guide calls this an “implicit synchronization caused due to a dependency check” [NVCCPG].

This false serialization can severely restrict the ability of programs to exploit concurrency. For our evaluation, therefore, we developed a replacement for the default CUDA streams implementation that does not suffer from this problem. Our “Non-Serializing Streams” (NSS) implementation provides CUDA-like Streams behaviour but uses alternate methods that do not induce serialization to enforce intra-stream dependencies. Section 3.5 describes our NSS implementation.

3.4 Elastic Kernels

Elastic kernels decouple physical hardware resource allocation for thread blocks from logical program-level grid and thread block identity. In Section 3.4.1, we first describe how an ordinary CUDA kernel can be transformed into an elastic kernel by source-to-source transformations. These transformations are necessary because we currently implement elastic kernels in software. The resulting elastic kernel can run using physical grid and thread block dimensions that are different from programmer specified grid and thread block dimensions. Then, the algorithm presented in Section 3.4.2 uses this ability to control the resource usage of an elastic kernel. Given resource constraints, the algorithm computes physical grid and thread block dimensions such that the elastic kernel’s grid will satisfy those constraints. Next, in Section 3.4.3, we describe several elastic-kernel aware concurrency policies which improve concurrency by controlling the resources allocated to an elastic kernel. Finally, we describe how elastic kernels are used to implement time-slicing of kernels in Section 3.4.4.
3.4.1 Elastic Kernel Transformations

Currently, the hardware maps logical thread blocks and threads to physical thread blocks and threads using a 1:1 logical-to-physical mapping scheme. If we could implement an \( N : 1 \) logical-to-physical mapping scheme while preserving CUDA semantics, we can achieve fine-grained resource allocation for GPU grids.

The idea of an \( N : 1 \) mapping for CUDA grids has been explored by Stratton et al. in their MCUDA [Stratton et al., 2008, 2010] work which executes CUDA grids efficiently on multicore CPUs. They developed a technique called iterative wrapping (similar to loop chunking [Shirako et al., 2009]) to efficiently run the large number of CUDA threads on the comparatively fewer threads available on the CPU. Iterative wrapping executes \( N \) CUDA thread blocks using a single CPU thread by inserting an enclosing iterative loop. However, it does not change the number of thread blocks or threads. Our work uses a similar \( N : 1 \) mapping scheme, but performs a GPU-to-GPU transformation and also allows modification of the number of threads and thread blocks. Changing the number of threads or thread blocks will change thread identities which, in turn, will affect any work distribution based on identity being used by the kernel. Therefore, our transformations also preserve the original identity of each thread block and thread. Essentially, the elastic kernel uses whatever physical grid it was launched with to execute the original logical grid.

Our \( N : 1 \) mapping scheme takes a 2D logical grid and a 3D logical thread block and executes them using a 1D grid and 1D thread blocks. Listing 3.1 shows the code for a physical thread block that implements our mapping scheme. This code is placed around the original kernel code and implements a general transformation from physical grid identities to logical grid identities while looping over the original kernel code. To preserve program semantics with respect to identities, we also replace any references to physical dimension variables (gridDim, blockDim) and physical identity variables (blockIdx, threadIdx) in the original kernel code with their logical equivalents (gd, bd, bi and ti respectively). The code is now an elastic kernel that can execute the original logical grid faithfully with any number of physical thread blocks as well as any physical thread block dimension.

We note that some kernels (e.g. histo_main_kernel in histo) can run with changed thread blocks or threads even without our transformations. To identify such kernels, we performed a test where the number of blocks and threads was varied for each kernel and their results compared to that obtained by the kernel running with the
// find global thread id in physical grid
// note: physical grid and thread blocks are 1D
int tid = threadIdx.x + blockIdx.x * blockDim.x;

// iterate over threads in logical grid
for (int gtid = tid;
     gtid < (gdX * gdY * bdX * bdY * bdZ);
     gtid += blockDim.x * gridDim.x)
{
    // linearized identities in logical grid
    int block_id = gtid / (bdX * bdY * bdZ);
    int thread_id = gtid % (bdX * bdY * bdZ);

    // logical block identities
    int biX = block_id % gdX;
    int biY = block_id / gdX;

    // logical thread identities
    int tiX = (thread_id % (bdX * bdY)) % (bdX);
    int tiY = (thread_id % (bdX * bdY * bdZ)) / (bdX * bdY);
    int tiZ = thread_id / (bdX * bdY * bdZ);

    // original kernel code follows
    // ...
}

Listing 3.1: Code in each physical thread block to execute logical thread blocks according to our mapping scheme. The values $gdX$, $gdY$, $bdX$, $bdY$, and $bdZ$ are logical grid and thread block dimensions as set by the programmer.
original number of blocks and threads. Only 4 kernels of the 18 among the Parboil2 benchmarks passed the test. These kernels can run with different grid and thread block dimensions because they compute a global thread identifier and use it as a basis for work distribution. While we do not need to transform such kernels, in this work we apply our transformations to all kernels.

Although the code of Listing 3.1 allows us to change thread block dimensions, we do not do so for kernels that use shared memory or synchronization instructions. In general, doing so could split a logical thread block across two physical thread blocks. This would violate CUDA semantics for shared memory accesses and behaviour of synchronization instructions. Essentially, shared memory accesses are tied to physical thread blocks and not logical thread blocks and such a split would lead to incorrect execution. Similarly, it is not currently possible to synchronize threads across different physical thread blocks without using slow global synchronization primitives. Therefore, in this work, we do not apply thread block resizing to kernels that use shared memory or synchronization instructions, i.e. 9 of the 18 kernels in the Parboil2 benchmarks.

Our software implementation suffers from two performance issues. Firstly, we are forced to use ordinary variables to store logical identities and dimensions. This increases register usage of the kernel and can cause a potential drop in throughput because the number of threads that can be resident could reduce. Dedicated hardware registers for logical identities and dimensions would solve this problem. The second performance issue arises from the use of the division and modulus operations in our transformed kernels which are not supported natively by the Fermi hardware. Their use increases the runtime of elastic kernels compared to the original kernels. Note that a hardware implementation of elastic kernels would not have these issues and would also be completely transparent to the programmer.

3.4.2 Resource Control with Elastic Kernels

In this section, we present an algorithm to limit physical resource usage for grids of elastic kernels. To control physical resource usage, we manipulate the elastic kernel’s physical grid and thread block dimensions. By changing the number of physical thread blocks, we can control the utilization of threads and registers. For thread block-level resources like shared memory and resident blocks, grid dimensions must be modified.

---

1This test only identifies kernels which cannot handle different grid and thread block dimensions. We still need to examine the code for those that pass to verify that the kernels can indeed display elastic kernel behaviour.
For thread-level resources likes threads and registers, either grid dimensions or thread block dimensions can be modified.

Algorithm 3 Algorithm GETPHYGRID

1: function GETPHYGRID(Kernel, Blocks, Threads, Limits)
2:   Usage ← BLOCKUSAGE(Kernel)
3:   MaxResident ← Usage.SMBlocks * GPU.SMCount
4:   Blocks ← MIN(Blocks, MaxResident, Limits.Blocks)
5:   if THREADSPERBLOCKCANCHANGE(Kernel) then
6:     Incr ← ⌊(ChangeInThreads/Blocks)⌋
7:     Threads ← Threads + Incr
8:   end if
9:   REDUCEBLOCKS(Limits.ShMem, Usage.ShMem)
10:  REDUCEBLOCKS(Limits.Threads, Usage.Threads)
11:  REDUCEBLOCKS(Limits.Registers, Usage.Registers)
12:  return (Blocks, Threads)
13: end function

procedure REDUCEBLOCKS(RLimit, PerBlockUsage)

1: // Blocks refers to the value in GETPHYGRID
2:  CurUsage ← Blocks * PerBlockUsage
3:  if CurUsage > RLimit then
4:    Deficit ← CurUsage − RLimit
5:    ReduceBlocks ← ⌈Deficit/PerBlockUsage⌉
6:    Blocks ← Blocks − ReduceBlocks
7:  end if
8: end procedure

Given resource constraints (Limits), Algorithm 3 determines the physical grid and thread block dimensions for an elastic kernel that satisfies those constraints. The input to the algorithm is the number of logical thread blocks, the number of logical threads per thread block and resource constraints on the four resources – resident thread blocks, shared memory, threads and registers. The algorithm first obtains the current resource usage of each thread block using BLOCKUSAGE, a routine derived from the CUDA Occupancy Calculator [NVOCC], which calculates the number of registers, threads, shared memory and thread blocks that the kernel currently occupies. Then, the algorithm sets the number of physical thread blocks to the maximum number of concurrent thread blocks that can be accommodated on the GPU (i.e. MaxResident) since all blocks in excess of MaxResident have to wait to execute and, on the NVIDIA Fermi,
also prevent blocks of later concurrent grids from beginning execution. The algorithm then reduces the number of thread blocks further to meet constraints on the maximum number of resident blocks. As thread blocks and threads are coupled, reducing thread blocks will also reduce the total number of threads \( \text{ChangeInThreads} \). If the kernel supports thread block resizing, we compensate by increasing the number of threads per thread block. Finally, for each resource constraint on shared memory, threads, or registers, the algorithm reduces the number of blocks to satisfy the constraint. The number of blocks and threads computed by this algorithm are the physical grid and thread block dimensions respectively and can be used to run the elastic kernel under the specified resource constraints.

3.4.3 Elastic Kernel Aware Concurrency Policies

With mechanisms to elasticize a CUDA kernel and control its resource usage available, we now present policies that impose resource constraints on elastic kernels in order to improve concurrency. These policies are implemented at the Stream Scheduler level and apply their resource constraints during the launch of a kernel.

**MEDIAN**

The MEDIAN policy uses profile-based information to reserve resources for a hypothetical median kernel. Using the data in Table 3.1 about the programs in Parboil2, we compute this median kernel’s resources to be 224 threads, 6144 registers and 256 bytes of shared memory per thread block. These numbers are the medians for those resources among the Parboil2 kernels. The MEDIAN policy limits the resource usage of actual kernels to ensure that each streaming multiprocessor will have resources leftover to run one block of this median kernel.

**MPMAX**

The multiprogram maximum or MPMAX policy also uses profile information to reserve resources. For each program, the MPMAX policy constructs a largest kernel based on the maximum resource usage of its kernels. This largest kernel need not correspond to an actual kernel of the program – it may have the register usage of one kernel and the shared memory usage of another. Then, for each program, it computes the othersLargest kernel by repeating a similar procedure using the largest kernels of other concurrently executing programs. Thus, the othersLargest kernel is different for each
of the programs executing concurrently. Then, like MEDIAN, the resources of each actual kernel in a program are restricted so that one thread block of the othersLargest kernel will always have enough resources to run.

EQUAL

The EQUAL policy partitions GPU resources equally among all concurrently running programs. For two program workloads, for example, it limits each grid’s resource usage to 50% of GPU resources. It is based on the Even SM partitioning heuristic of Adriaens et al. [2012]. It is thus a form of spatial partitioning though our implementation does not prevent multiple grids from executing concurrently on the same SM. Thus, this policy dedicates resources for each concurrently executing grid, but without incurring the per-SM wastage noted in Section 3.2.

QUEUEMOLD

The QUEUEMOLD policy is based on the GetAffinityByMolding algorithm of Ravi et al. [2011]. This policy examines all kernels waiting to be launched and modifies the resources requested by them if they make excessive use of: (i) shared memory or (ii) threads. In our implementation, the policy examines kernels waiting in the NSS scheduler queues. If the sum total of shared memory of a newly arrived kernel and a kernel waiting in the queue exceeds the total shared memory available, the number of thread blocks of the waiting kernel is reduced. Similarly, if the thread usage of a waiting kernel exceeds 512 threads, the number of thread blocks (or the number of threads per block) is reduced. The original implementation also uses a notion of affinity in order to distribute kernels across multiple GPUs. Since we do not use multiple GPUs in our evaluation, we do not use any affinity values. Note that this policy only modifies the resources used by a kernel if another kernel arrives in the queue before it is launched.

3.4.4 Implementing Timeslicing of Grid Execution

We implement timeslicing using elastic kernels. By default, elastic kernels iterate over all the thread blocks of the original grid in a single launch. We modify them to accept offset and limit parameters, so as to restrict execution to only a certain range of thread blocks of the original grid. As each range completes, we simply relaunch the kernel with the next range. The per-invocation ranges for each kernel are currently chosen offline.
to ensure that the runtime of each range is nearly equal to 1ms whenever possible. The thread block scheduler would, of course, be able to do this online.

3.5 Non-Serializing Streams Implementation

To avoid false serialization on the NVIDIA Fermi due to dependency checks inserted by the CUDA Streams implementation as described in Section 3.3.7, we develop a CUDA-like streams implementation called “Non-Serializing Streams” (NSS) that does not introduce false serialization. NSS is also used to implement our elastic policies, and can timeslice kernel execution and memory transfers. It also reorders items in the queues so as to avoid the issues caused by inter-stream scheduling (Section 3.3.3).

NSS prevents false serialization by avoiding any action that would insert or perform a dependency check in a CUDA stream. In general, submitting each command in the NSS stream to a different CUDA stream prevents dependency checks from being inserted since no dependency checks are inserted between commands of different CUDA streams. However, this does not mean that we need a new CUDA stream for each command. Since we are only concerned with the serializing behaviour of the $M\rightarrow K$ and $K\rightarrow M$ dependency checks, we only need two CUDA streams. One of these is used for all memory operations, and the other is used for kernel execution. The use of different and exclusive streams for memory and kernel commands prevents CUDA from inserting $M\rightarrow K$ and $K\rightarrow M$ dependency checks. However, NSS must enforce these dependencies.

To enforce $M\rightarrow K$ dependencies in NSS, we note that in the current API, memory transfers that involve pageable memory are synchronous. So a memory transfer is nearly complete when the function call returns. NSS can thus enforce $M\rightarrow K$ dependencies by simply waiting for the cudaMemcpy call to complete. This does not present a performance limitation in our studies because none of the applications in Parboil2 use asynchronous memory transfers.

Enforcing $K\rightarrow M$ and $K\rightarrow K$ dependencies is harder since kernel launches are always asynchronous and CUDA provides no way to check if a kernel has completed without introducing a dependency check. We therefore improvise a kernel completion notification mechanism as follows. On every exit path of a kernel, we add the code snippet in Listing 3.2. This code counts the thread blocks that have finished executing so far in the GPU-side blocks\_done\_d variable. Once all thread blocks have finished, the CPU/GPU shared (host-memory mapped) blocks\_done\_h is set, alerting a polling routine on the CPU of the completion of this kernel.
if ( threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0 ) {

    int blocks_done = atomicAdd( blocks_done_d , 1 );

    if ( blocks_done == ( gridDim.x * gridDim.y − 1 ) ) {
        blocks_done_d = 0;
        blocks_done_h = 1;
    }
}

Listing 3.2: Kernel completion notification code that is inserted onto every exit path of a kernel to notify the CPU that the kernel has completed.

Like our elastic kernel transformations, the addition of this code also changes the original kernel. Firstly, the number of registers used by the kernel may increase. This affects the number of resident threads and throughput. Secondly, atomic operations have low throughput, causing a performance loss when compared to the original kernel.

The NVIDIA Kepler K20 based on the GK110 GPU is advertised to contain a Hyper-Q feature [NVKWP] that provides one hardware queue per CUDA stream and prevents this form of serialization. So, on the GK110, NSS may be able to use hardware dependency checks without incurring the overheads of false serialization.

3.6 Evaluation

3.6.1 Workload Construction

We evaluate our policies using multiprogrammed workloads since all of the Parboil2 benchmarks are single-threaded serial CUDA programs. However, since CUDA does not execute grids from different programs concurrently [NVCCPG], we cannot obtain a multiprogrammed workload execution by simply running the programs together. To work around this limitation, we study multithreaded CUDA programs whose individual threads execute traces of the original Parboil2 programs. Figure 3.4 illustrates the procedure we use to construct such a multithreaded “multiprogrammed” workload. We run each program in Parboil2 individually and obtain traces of its CUDA API calls using ltrace. Simultaneously, we obtain the memory and argument values passed to those API calls using a library interposer we have developed for the CUDA Runtime API.
Figure 3.4: Construction example of a multithreaded concurrent CUDA workload from traces of individual programs bfs and cutcp.

These traces are input to our trace reconstructor which outputs a C program that replays the traced API calls. To obtain an $n$-program workload, we reconstruct the $n$ original programs as separate threads of a new $n$-threaded program. Each thread represents an individual program of the workload and consists of three distinct parts: (i) an initialization part to load the CUDA modules, functions and textures, (ii) a “replay” part to re-execute the program’s kernel launches and memory transfers using values from the stored trace, and (iii) a cleanup part to release any resources obtained. Since the original programs do not use streams, our trace reconstructor also translates the original non-stream CUDA API calls (e.g. cudaMemcpy) to their functionally equivalent CUDA Streams API (e.g. cudaMemcpyAsync) or the corresponding NSS replacement to obtain concurrent execution. Thus, we essentially obtain a re-execution of the CUDA portions of the original programs as a concurrent workload.

### 3.6.2 Methodology

We evaluate workloads consisting of two programs. Each workload has four variants that are all generated from the same trace. The first variant single uses CUDA API calls and runs each trace of the workload one after the other, so that each trace executes alone with the full resources of the GPU at its disposal. The second variant cuda uses CUDA
API calls and runs each trace concurrently with a separate CUDA stream for each trace, establishing the baseline for CUDA concurrency. The third variant \texttt{nss} replaces the use of CUDA Streams API by our Non-Serializing Streams implementation (Section 3.5) to work around the false serialization encountered by the \texttt{cuda} variant. Finally, the fourth variant \texttt{elastic} builds on \texttt{nss} and allows the resources assigned to a grid to be varied by the policies of Section 3.4.3. This variant also has kernel timeslicing enabled. All variants execute the same kernel source code to mitigate the performance issues of our software implementation as described in Section 3.4.1 and to provide a fair comparison.

There are a total of 55 two-program workloads, of which we evaluate 54. The \texttt{bfs+lbm} workload was not evaluated because it experiences CUDA launch timeouts. The implementation of \texttt{bfs} in the Parboil2 benchmarks uses a global atomic barrier which requires that all its thread blocks be resident concurrently. However, CUDA does not guarantee that all thread blocks will be resident concurrently. While the standalone execution of \texttt{bfs} is not affected, the execution with \texttt{lbm} causes the global atomic barrier to deadlock.

The workloads are run on a Fermi-equipped NVIDIA Tesla C2070 with CUDA driver 295.41 and CUDA runtime 4.2, the latest available at the time of writing. The host machine is a quad-core Intel Xeon W3550 CPU with 16GB of RAM and runs the 64-bit version of Debian Linux 6.0.

We report the performance of each workload using the System Throughput (STP) and the Average Normalized Turnaround Time (ANTT) metrics [Eyerman and Eeckhout, 2008]. In our results, we substitute runtime for cycles in the equations for STP and ANTT. We record the runtime for each individual program as the time spent in the replay portion of the thread but subtract all time spent in performing trace I/O, which is an artifact of our workload construction technique. To account for interleaving effects, we use Tuck and Tulsen’s methodology [Tuck and Tulsen, 2003] – the replay part runs until all of the programs in the workload have been replayed at least 7 times. The last runtime of each program in the workload is discarded to avoid counting non-overlapping executions. For each program in the workload, the execution times of its replays are averaged and used in the equations for STP and ANTT as the multiprogram mode time. The \texttt{single} variant provides the single-program mode execution time for a program.

We use the log files generated by the NVIDIA Compute Command Line Profiler [NVCPROF] to compute other metrics such as utilization and waiting time. Utilization is estimated from the start and end times of kernels because the profiler does not report starting and
ending times of each individual thread block. We define waiting time as the time from API call (e.g. cuLaunchKernel) to actual execution as reported in the profiler log. This measures waiting time in the GPU’s hardware queues and does not contain time spent in NSS queues (which is accounted for in the total program time). Since the CPU and GPU use different clocks, we use the TIMESTAMPFACTOR value stored in the log by the CUDA Profiler to correlate the two timestamps. We use the Linux User Space Tracing Toolkit [Desnoyers et al.] to record API calls on the CPU side.

In our initial experiments, CUDA memory allocation functions limit achievable STP to that of NSS (Section 3.3.6). To mitigate this, we built a custom GPU memory allocator based on the CPU memory allocator jemalloc 3.2.0 [Evans, 2006] and use it for all the variants. Although our custom allocator uses the CUDA allocation functions internally, by allocating memory in bulk and recycling allocations, it calls them less frequently than the original program. Another possibility would be to ignore cudaMalloc and cudaFree from the second and first run onwards respectively. In such experiments, the average STP improves by about 5% and the ANTT improves by 10% over those reported in our evaluation. We also replace cudaMemcpy with a custom, non-serializing implementation in all variants.

3.6.3 Analyzing Runtime Behaviour of Kernels in Concurrent Workloads

Kernel execution dominates the runtime of workloads. In concurrent workloads, a kernel’s execution time is affected by the presence of other concurrently executing kernels. To better understand the effect of other concurrent kernels on execution time, we classify each instance of a kernel into one of four overlap categories (Figure 3.5). For each kernel instance we identify its set of co-runners, i.e. other kernels whose instances overlap with it in time. Four overlap categories can then be defined as:

- **EXCLUSIVE**: If the set of co-runners is empty, the kernel instance is classified as EXCLUSIVE, i.e. it ran alone.

- **SHARED/FULL**: If the set of co-runners is not empty, but the kernel instance being classified started before all co-runners, we classify it as SHARED/FULL. In this case, the instance started alone, but will share some of its runtime with concurrently executing kernels.

- **SHARED/RESTRICTED**: If the kernel instance started after any of its co-runners then it will run with restricted resources. If none of the earlier co-runners termi-
nated before this kernel, we classify it as **SHARED/RESTRICTED**.

- **SHARED/PARTIAL**: If, however, some earlier co-runner terminated before this kernel did, possibly making its resources available to this kernel, we classify the instance as **SHARED/PARTIAL**.

We expect the runtime for an **EXCLUSIVE** kernel instance in a concurrent workload to be similar to when the kernel runs alone. For all of the **SHARED** categories, however, we expect the runtime to vary depending on the degree of overlap with other kernels. Additionally, for **SHARED/PARTIAL** and **SHARED/RESTRICTED**, we expect the runtime to further vary based on the resources allocated to the kernel by the GPU.

To illustrate this categorization, Figure 3.6 portrays the execution of the `GPU_FFT_Global` kernel from the `fft` benchmark during its execution as part of the `bfs+fft` two-program workload. In the figure, the $y$-axis shows the runtime of each `GPU_FFT_Global` kernel instance, while the $x$-axis denotes the start time of the corresponding instance along the workload execution timeline. Using information from CUDA profiler log files, each instance has been categorized into one of the four overlap categories, indicated by different markers in the figure. Initially, during transfers of data by `bfs`, `GPU_FFT_Global` kernel instances run in **EXCLUSIVE** mode. When the kernels of `bfs` begin execution, the instances of `GPU_FFT_Global` transition to running mostly in **SHARED/RESTRICTED** mode.
Figure 3.6: Runtimes of GPU_FFT_Global instances from one run of the bfs+fft workload under the MEDIAN elastic policy with each instance categorized into one of the four overlap categories. (Partially occluded SH/PARTIAL at approx. $x = 0.5, y = 144$)

mode because both the kernels of bfs take much longer to complete (see Table 3.2). A SHARED/PARTIAL and a few SHARED/FULL instances can be observed during the transitions from one bfs kernel to another. The runtimes of each GPU_FFT_Global instance demonstrate the trends we have described in the previous paragraph.

## 3.6.4 Results

We evaluate all possible two-programmed workloads (except for bfs+lbm as noted) of the 11 programs from Parboil2 (Table 3.1) with the elastic policies MP MAX, MEDIAN, EQUAL and QUEUEMOLD.

### Overall Results

Table 3.3 shows the average STP and ANTT across the workloads. All elastic policies improve throughput and turnaround time compared to cuda and nss. The elastic policies have the best STP values compared to all the variants. Similarly, their worst ANTT values are considerably lower than both cuda and nss, indicating that they have better turnaround times. Table 3.4 shows that compared to cuda, on average the elastic policy
MPMAX improves system throughput by 1.21x and turnaround time by 3.73x. Viewing the workload as a multithreaded workload, on average the elastic policy MPMAX obtains a 1.28x speedup (Table 3.4) over cuda. The nss variant also improves on cuda’s STP by 15%, and its ANTT is significantly better by 2.20x. On average, although the elastic policies EQUAL and QUEUEMOLD perform better than cuda, they do not do as well as our elastic policies, for reasons explained in the following sections.

In experiments with four-program workloads (excluding bfs), we find that the average STP value is 1.21 for the elastic policy MEDIAN. The average ANTT for four-program workloads is 6.61 for MPMAX, which is 8.9x better than that of cuda. The multi-threaded speedup for four-program workloads also increases to 1.47 as compared to cuda. Due to lack of space, we do not elaborate on our results for four-program workloads in this chapter.

**Effect of Elastic Policies on Kernel Runtime Behaviour**

Figure 3.7 shows the breakup of execution timing across kernel using the categories of Section 3.6.3. While the cuda kernels largely run in EXCLUSIVE mode about 90% of the time, the elastic policies tend to spread execution over the four categories.
Although EXCLUSIVE mode implies high performance on a per-kernel instance basis, in the case of cuda, this comes at the cost of waiting time. Figure 3.8 shows the distribution of average waiting time for a kernel for each workload under each variant. Kernel waiting times under cuda are higher than those under any of the other variants. The mean waiting times under cuda (130.5ms) and nss (11.21ms) are significantly higher than those for the elastic policies – MPMAX (0.53ms), MEDIAN (0.96ms), EQUAL (0.51ms) and QUEUEMOLD (1.57ms). The cuda kernels thus have to wait up to three orders of magnitude more time to execute compared to any of the elastic policies. This delay can largely be attributed to false serialization introduced by the CUDA implementation. The nss variant shows a 11.6x reduction in average kernel waiting time by eliminating false serialization, but still suffers a mean waiting time of 11.21ms due to lack of resources. The elastic policies reduce waiting times even further by preventing serialization due to lack of resources.

**Effect of Elastic Policies on STP and ANTT**

Figure 3.9 shows the distribution of STP values for all workloads. The elastic policy MPMAX has the highest STP values for nearly 40% of the workloads. For the remaining 60%, it behaves similarly to elastic policy MEDIAN. Both these policies offer significantly better performance than non-resource aware nss and cuda. The performance of the third elastic policy EQUAL is not quite as straightforward. For about 30% of the workloads, it performs poorly, even worse than nss. It shadows but is lower than MEDIAN and MPMAX for about 35% of the workloads. Then, for the remaining 35% of the
workloads, its performance is like that of MPMAX. Finally, the elastic QUEUEMOLD exhibits performance that is only marginally better than nss.

The performance variation in STP exhibited by these policies is determined by their resource-limiting decisions. Both the MEDIAN and EQUAL elastic policies set aside a fixed amount of GPU resources regardless of the workload. By design, the MEDIAN policy does not take away too many resources from a running kernel in about half of the workloads. (This will be borne out later in the next section while examining utilization.) For the remainder, the reserved resources are not enough to improve concurrency, causing the drop in performance. The EQUAL elastic policy, takes away too many resources in the worst performing 30% of the workloads. At the other end, the reserved half of resources is adequate to provide the rapid execution in the best performing 35% of the workloads. Our evaluation of the performance of QUEUEMOLD shows that it gets few opportunities to reduce the resources of individual kernels; given the wide disparity in kernel execution times (Table 3.2), the NSS queues rarely contain two or more kernels. Therefore, under QUEUEMOLD, most of the kernels execute with full resources like in nss. The elastic policy MPMAX differs from EQUAL and MEDIAN in that the GPU resources reserved vary per program and per workload. Thus, MPMAX avoids overcommitting resources and achieves a balance that delivers good concurrency and good performance. We conclude that for two program workloads, a policy that adapts to co-executing programs delivers the best performance.

Figure 3.10 shows the distribution of ANTT values across workloads. The elastic
Figure 3.9: System throughput for each two-program workload, higher is better.

policies other than QUEUEMOLD display similar behaviour. The QUEUEMOLD shows lower ANTT values than nss but 60% of the values are clearly higher than the other elastic policies. Section 3.6.4 will show that the kernel timeslicing implemented for all elastic policies leads to reduction in ANTT compared to nss. For cuda the high degree of serialization leads to very high values of ANTT. This is corroborated by Figure 3.8 which shows that average waiting time for kernel execution is much higher for these variants as compared to the elastic variants, indicating a high degree of serialization.

Effects of Policies on Utilization

Figure 3.11 shows the average number of concurrent kernels for each variant and the different elastic policies. All elastic policies except for QUEUEMOLD behave similarly, with the average number of concurrent kernels ranging from 1.23 for MP\textsc{max} to 1.22 for \textsc{equal}. This is 1.53x better than cuda, which averages 0.81 concurrently running kernels and also 1.34x better than the nss variant which averages 0.92 concurrently running kernels.

However, a higher average number of concurrent kernels alone does not necessarily translate into higher utilization of GPU resources. The average number of threads utilized (Figure 3.12) is high even for nss and cuda because these variants do not limit resource usage. However, the \textsc{equal} policy, which enforces large fixed limits, shows much lower thread utilization. For performance, thread utilization as well as number of concurrent kernels should be high.
Effects of Time-slicing

In all our experiments so far, timeslicing of kernels has been enabled for long running grids from the following benchmarks (in order of grid running time): mm, lbm, mri-q and tpacf, i.e. 33 of the 54 workloads. To quantify the impact of timeslicing, we performed additional measurements with kernel timeslicing turned off. For the timeslicing-enabled subset of workloads, time-slicing of kernels with a timeslice value of approximately 1ms produces an overall improvement of 7.8% in geomean STP and improves geomean ANTT by 1.55x over elastic policies that do not have timeslicing enabled (evaluated using the MP\textsc{max} elastic policy). For all 54 workloads, time-slicing of kernels produces improvements of 3.3% geomean STP and 1.28x improvements in geomean ANTT over elastic policies that do not perform timeslicing.

Next we evaluate the effect of memory transfer slicing with a 4MB chunk size. Our experiments reveal no significant effect on geomean STP or geomean ANTT. The 4MB memory chunk size used means that only transfers from the following programs will be time-sliced (size of transfers in parentheses): mm (4.1M), tpacf (4.7M), bfs (7.6M, 45M), stencil (64M) and lbm (105M), i.e. 40 pairs. However, over 97% of dynamic transfers in our workloads are less than 1MB in size. Hence, memory transfers do not seem to be a significant serialization bottleneck in practice yet.
Figure 3.11: Average number of running kernels for each two-program workload.

Figure 3.12: Average utilization of threads for each two-program workload.
3.7 Related Work

To the best of our knowledge, our work is the first to examine an actual implementation of GPGPU concurrency and to identify and address issues of poor resource utilization and poor concurrency. We have also identified many serialization factors in the CUDA execution model, the CUDA API and hardware implementation that inhibit GPGPU concurrency and have proposed solutions for all of them. We now list works that have examined GPU concurrency and resource allocation for concurrent execution of GPGPU kernels.

Guevara et al. [2009] present the first work on GPU concurrency that predates the NVIDIA Fermi GPU. The GPUs they use do not support hardware concurrency, so they resort to combining the source of two kernels into a single kernel at compile-time to execute them concurrently, a technique they call “thread interleaving”. Their technique only merges kernels together and does not change resource allocation for each kernel.

Gregg et al. [2012] introduce the KernelMerge runtime framework to investigate GPGPU concurrency for OpenCL programs. This framework uses a technique similar to thread interleaving to merge kernels into a single kernel, but does so at runtime. However, each thread block of this merged kernel can choose to execute thread blocks of any of the merged kernels under the control of a scheduler. Thus, different scheduling algorithms can achieve different partitioning of resources at the thread block level.

Wang et al. [2010] propose the use of kernel fusion to achieve power efficiency on the GPU. Again, they use a technique similar to thread interleaving to merge kernels. All of these works demonstrate significant improvements in throughput and power efficiency through the use of GPU concurrency. Also, their merging technique does not require hardware support for concurrency. However, merging kernels into one large kernel leads to wastage of resources because GPU resources cannot be reclaimed until both component kernels finish. In our work, we have used hardware support available on the NVIDIA Fermi to achieve concurrency.

Wang et al. [2011] propose context funneling to execute kernels from different GPU contexts concurrently. Their technique allows kernels from different operating system threads (which before CUDA 4 used different contexts) or kernels from different programs to execute concurrently.

Adriaens et al. [2012] propose that GPU streaming multiprocessors be spatially partitioned for GPU concurrency. They partition the set of streaming multiprocessors (SMs) among concurrently executing programs using different SM partitioning heuris-
tics and evaluate their policies using the GPGPU-Sim [Bakhoda et al., 2009] simulator. In this work, we base our EQUAL policy on their Even SM spatial partitioning heuristic which distributes SMs evenly among concurrent applications. We have evaluated EQUAL in this work and shown that it significantly underutilizes resources compared to our policies.

The work closest to our own is the work by Ravi et al. [2011] which uses GPU concurrency to improve GPU throughput for applications in the cloud. A key feature of their work is the ability to change the resources assigned to a kernel by varying grid and thread block dimensions, a technique they call molding. However, they only claim to support molding for kernels which are already written to run with any number of threads. As we have shown, there are only 4 such kernels among the 18 in the Parboil2 benchmark suite. Our work proposes a transformation (Section 3.4.1) to convert any kernel to an elastic kernel. We base our QUEUEMOLD policy on their GetAffinityBy-Molding resource allocation algorithm and evaluate it in our work. We find that given the wide disparity in execution times of GPU kernels (Table 3.2), this policy rarely finds the opportunity to limit resources of kernels and does not perform as well as our policies.

3.8 Conclusion

In this work, we looked at concurrent execution of GPGPU workloads. We showed that the current grid programming model of the GPU leads to wastage that can be reduced by concurrent execution of GPGPU workloads. However, we found that the current implementation of concurrency on the GPU suffers from a wide variety of serialization issues that prevent concurrent execution of GPGPU workloads. To the best of our knowledge, this is the first work that raises these issues. Prominent among these issues are serialization due to lack of resources, and serialization due to exclusive execution of long-running kernels and memory transfers. To tackle serialization due to lack of resources, we proposed elastic kernels, a mechanism that allows fine grain control over the amount of resources allocated to a GPU kernel. We used this ability to build elastic kernel-aware concurrency policies that significantly improve concurrency for GPGPU workloads. To tackle serialization due to long-running kernels, we also presented a simple and effective technique to timeslice kernel execution using elastic kernels. We have also identified several other implementation issues in the CUDA hardware and API that inhibit GPGPU concurrency, and have suggested solutions for all of them.
Our proposals improve average system throughput (STP) by 1.21x and average normalized turnaround time (ANTT) by 3.73x for two-program workloads compared to CUDA on real hardware. They also increase the number of concurrent kernels by 1.53x and reduce waiting times for kernels by three orders of magnitude. Our policies also achieve higher throughput, lower turnaround times and better resource utilization when compared to a static partitioning scheme and a runtime resource allocation scheme. Finally, our proposal for time-slicing of kernels improves STP by 7.8% and ANTT by 1.55x for programs with long running kernels.
Chapter 4

Structural Prediction for Improved Scheduling of Concurrent Kernels

4.1 Introduction

Elastic kernels established that runtime control of resource allocation to running kernels is necessary to improve concurrency. However, since our implementation ran on existing hardware, each concurrent kernel was still executed as per the FIFO policy of the NVIDIA thread block scheduler (TBS) which is responsible for dispatching thread blocks to each individual SM. Provided that resources are available, thread blocks from any running kernel can be executed on any SM. Further, this can be done in any order since CUDA explicitly allows reordering of thread blocks [NVCCPG]. However, the current TBS only dispatches thread blocks from kernels in the order of their arrival, i.e. in FIFO order. For example, if two kernels $K_1$ and $K_2$ ran concurrently, but arrived in that order, then under the FIFO TBS policy, none of $K_2$’s thread blocks would be scheduled until all of $K_1$’s thread blocks had been scheduled. As we shall show in Section 4.2, this can lead to poor performance.

For any 2-program workload, a FIFO decision matches either that of Shortest Job First (SJF) or that of Longest Job First (LJF). The match is based purely on chance and depends entirely on the order in which the kernels arrived. To improve scheduling, say by implementing Shortest Remaining Time First, the TBS would require an estimate of kernel runtime. We find that it is possible to exploit the structure of the grid of a GPU kernel to predict its runtime. Since the grid is uniform, online profiling of the first few thread blocks allows us to obtain the runtime of the entire kernel.

We make the following specific contributions:
• We introduce Structural Runtime Prediction and the Staircase model for online prediction of GPU kernel runtime. This model exploits the uniform structure of grids to predict runtime.

• We build an online runtime predictor whose predictions evaluated on hardware traces are within 0.48x to 1.08x of actual runtime for single-program workloads after observing only a single thread block.

• Using this predictor, we implement the Shortest Remaining Time First (SRTF) policy for thread block scheduling which achieves the best system throughput (1.18x better than FIFO) and turnaround time (2.25x better than FIFO) among all policies evaluated. Our implementation of SRTF also bridges 49% of the gap between FIFO and Shortest Job First (SJF), an optimal but unrealizable policy.

• To improve fairness of scheduling, we propose SRTF/Adaptive, a resource-sharing and scheduling policy which ensures equitable progress for running kernels while improving STP by 1.16x, ANTT by 2.23x and Fairness by 2.95x over FIFO.

This chapter is organized as follows. Section 4.2 motivates the need for better thread block schedulers and online predictors. Section 4.3 introduces Structural Runtime Prediction and the Staircase model for prediction. In Section 4.4 we describe the construction of an online predictor. Section 4.5 describes the scheduler and scheduling policies that we evaluate. Section 4.6 evaluates our scheduler. Section 4.7 lists related work. We conclude in Section 4.8.

4.2 Motivation

To evaluate the performance of the First-in First-out (FIFO) policy on concurrent kernels, we simulate the scheduling of 28 two-program workloads from the ERCBench suite [Chang et al., 2010]. For a two-program workload, FIFO’s schedule is the same as either of Shortest Job First (SJF) or Longest Job First (LJF) depending on the order of arrival of the kernels. Note that in our evaluation (Section 4.6), there are 56 two-program workloads possible and the subset chosen arbitrarily here consists of workloads $A + B$ such that the names of benchmarks $A$ and $B$ are in alphabetical order. In each $A + B$ workload tested, benchmark $A$’s kernel launches before that of benchmark $B$.

Figure 4.1 presents the system throughput (STP, as defined in Eyerman and Eeckhout [2008]) under FIFO scheduling for each of the 28 two-program workloads. For comparison, the figure also shows the system throughput achieved by the SJF and LJF policies. The geomean STPs are: SJF, 1.82; FIFO, 1.58; LJF, 1.16. We observe that for
17 of the 28 workloads, FIFO achieves the same STP as SJF, that for 8 workloads its STP is the same as LJF, and for the 3 remaining workloads, the STP does not differ for SJF and LJF. Since FIFO is oblivious to kernel characteristics, these results are entirely an artefact of arrival order of the 2 kernels in each of the workloads we chose. In this case, as kernels were launched in alphabetical order of benchmark name, in 17 of the pairs the shorter kernel started before the longer kernel.

Since the NVIDIA Fermi uses a FIFO policy, its throughput for concurrent kernels is also governed solely by the order in which the kernels were launched. For the experimental scenario described above, the Fermi would lose 15% in system throughput on average. In the worst case, shorter kernels will arrive while a longer kernel is already executing, so FIFO would end up scheduling like LJF and the Fermi would lose 57% on average for these workloads. FIFO is also non-preemptive, so execution of shorter kernels can end up being serialized behind those of larger kernels. This serialization at the GPU level can lead to slowdowns of the CPU part of the program as well. For example, TimeGraph [Kato et al., 2011] demonstrates that GPU programs with high OS priorities can suffer from priority inversion when the GPU is monopolized by a long-running kernel from a lower-priority program.

Recently proposed resource reservation policies [Adriaens et al., 2012, Gregg et al., 2012, Guevara et al., 2009, Ravi et al., 2011] and Elastic kernels (Chapter 3) partially address the problem by reserving resources for every kernel that is running while con-
tinuing to use FIFO. While this prevents serialization by guaranteeing access to the GPU, our evaluation of a state-of-the-art reservation policy will show (Section 4.6) that policies other than FIFO can lead to better performance. In particular, since thread blocks can reordered without violating CUDA semantics, a TBS can make the following decisions if a new grid arrives while an old grid is executing:

1. Do nothing: Continue executing thread blocks from the currently running grid.
2. Run with available resources: Issue all thread blocks from the currently running grid and if resources are leftover, attempt to schedule thread blocks from any concurrently running grid. As resources on an SM are allocated at the granularity of an entire thread block, some grids may underutilize resources potentially permitting their use by concurrently running grids.
3. Adjust grid resources: Vary the number of thread blocks or the number of threads in a thread block as in Elastic kernels, in order to distribute a SM’s resources between concurrently executing grids.
4. Pre-empt running grid: Pause scheduling of thread blocks from the current grid, while allowing thread blocks from other grids to be scheduled in order to prevent serialization of short kernels or enforce OS priorities.

The first item in the list above describes FIFO execution. Past resource-sharing policies can be described by the second and third items. In this work, we primarily focus on the fourth item, i.e. switching between grids, but also describe a dynamic grid resource adjustment policy.

To implement an SJF-like scheduling policy, a TBS requires knowledge of the runtimes of currently executing and the newly arrived grids. There are two main techniques that could be used to obtain runtimes in advance. Offline models such as those from Sim et al. [2012], Baghsorkhi et al. [2010] or Kothapalli et al. [2009] could be used to predict runtimes. Alternatively, a historical database of runtimes could be maintained per kernel as in Luk et al. [2009], Jia et al. [2012], Gregg et al. [2011], Diamos and Yalamanchili [2008] or Belviranli et al. [2013] and used to predict runtimes.

The primary disadvantage of offline models is that they are built for specific GPUs and require profile information for every kernel that may run. This is impractical in general. Predictors that use historical databases fare better since they use profile information, but they are unable to make predictions until they have seen enough complete runs. Crucially, since none of these predictors handles concurrent kernels at all, they cannot be used when scheduling thread blocks of concurrent kernels.

Ideally, an online predictor that is both aware of concurrent kernels and that can
predict runtime for all kernels in advance – either at kernel launch or after a few thread blocks have finished executing – is needed. Such a predictor could be used by the TBS to make scheduling decisions. Therefore, this work develops: (i) an online runtime predictor for GPU kernels, and (ii) thread block scheduler policies that use this predictor.

4.3 Structural Runtime Prediction

We introduce the principle of Structural Prediction on which our online predictor is based. Structural prediction essentially treats the execution of a grid’s $N$ thread blocks (all of which have the same code) as $N$ repeated executions of the same program. So by profiling the first few thread blocks of a grid, we can predict the behaviour of the remaining thread blocks. In this work, we observe the runtime of a thread block and use it to predict the runtime of the whole kernel. We call this technique Structural Runtime Prediction.

4.3.1 The Staircase Model

CUDA kernels are massively multithreaded programs. They execute as grids, with each grid consisting of a programmer-specified number of thread blocks (or co-operative thread arrays/CTAs). Each thread block, in turn, consists of programmer-specified number of threads. Although the code for each thread is the same, each individual thread can execute different code paths based on thread identity or data values.

NVIDIA GPU hardware is organized as a number of streaming multiprocessors (SMs). Since programmers can create up to $2^{32} - 1$ thread blocks, the CUDA programming model specifies a many-to-one mapping from thread blocks to SMs. Essentially, thread blocks are assigned to SMs and executed in batches. The size of a batch is determined algorithmically and can vary for each kernel and GPU pair as well as with invocation parameters such as size of thread blocks or shared memory used.

For example, when a grid is launched on the NVIDIA Fermi, its thread blocks are mapped to one of the Fermi’s many streaming multiprocessors. Each SM has a finite number of resources (registers, threads, shared memory, block contexts) that are allocated at thread block granularity. A SM accommodates as many thread blocks of the grid as possible until one of these resources runs out. The maximum number of thread blocks of a grid that can be accommodated on an SM is called the maximum residency of that grid. Thread blocks that cannot be accommodated wait in queue until resources
Figure 4.2: Staircase Model Execution of $N$ thread blocks on a single SM, with maximum residency $R = 4$ and each block taking $t$ time to execute. Here, $N = 3R$.

are available, which happens only when a running thread block finishes. Then, the Fermi thread block scheduler picks a thread block from the queue and schedules it in place of the most recently finished thread block. The grid has finished executing when all its thread blocks are done [NVCCPG, Section 1.3].

Figure 4.2 illustrates this model execution of a grid on a single SM. This grid has a maximum residency of $R = 4$ and each thread block executes in $t$ time. The total number of thread blocks is $N = 3R$, and the total time $T$ is $3t$. Further, the batch size is equal to $R$ on a single SM. So from the figure, the total time for execution of all $N$ blocks assigned to a single SM is therefore a simple linear (step) function:

$$T = ([N/R]) t$$

(4.1)

To generalize to multiple SMs, we assume an even distribution of $B$ thread blocks across $N_{SM}$ SMs, so $N = B/N_{SM}$. The maximum residency $R$ can be determined at grid launch time using resource usage per thread block which in turn can be calculated using formulae like those in the NVIDIA Occupancy Calculator [NVOCC]. Then, to predict runtime, equation 4.1 only needs the value of $t$. This could be obtained by sampling, possibly as soon as a single thread block finishes execution. However, the equation also suggests a fundamental limitation to this method. It is of little use for grids that execute fewer than $R$ blocks per SM since the result would not be obtained
Figure 4.3: Execution of SGEMM’s thread blocks on one SM. Blocks are ordered by finishing time. Black squares represent start times of each thread block, dark blue circles denote ending time. Green line is linear fit to all the end timings. Red line is prediction from equation 4.1, with \( t \) being the duration of the first block to finish.

in a timely manner. Note that if the grid uses global synchronization primitives, it must use \( R \) or less blocks per SM and therefore our technique does not apply.

### 4.3.2 Staircase Model Evaluation

To evaluate the Staircase model, we instrument all kernels in the Parboil2 [Stratton et al., 2012] and the ERCBench [Chang et al., 2010] suites to record the start and end time of each thread block and the SM it was executed on. We run these instrumented kernels on a Fermi-based NVIDIA Tesla C2070. The host machine is a quad-core Intel Xeon W3550 CPU with 16GB of RAM and runs the 64-bit version of Debian Linux 6.0 with CUDA driver 295.41 and CUDA runtime 4.2.

Figure 4.3 plots the end times of the thread blocks of the Parboil2 SGEMM kernel from a single SM. This instance of SGEMM execution closely resembles the Staircase model execution of Figure 4.2. Also shown is the linear fit to these end times using least-squares linear regression, as well as the runtime value \( (T) \) predicted using Equation 4.1 with \( t \) set to the duration of the first finishing block. The linear fit overestimates the actual finish time by 4.8% while the staircase model prediction underestimates it by 6.04%.

We now obtain predictions for the other kernels using both least-squares linear re-
Figure 4.4: Boxplots of Predictions from Linear Regression and Staircase Models for ERCBench and Parboil2 benchmarks normalized to actual runtime.

Regression as well as equation 4.1. Predictions are obtained for every invocation of a kernel and on every SM. Since the Fermi has 14 SMs, and some kernels are invoked multiple times, we obtain 4508 predictions for Parboil2 kernels and 112 predictions for the kernels in ERCBench. Figure 4.4 is a boxplot of predictions normalized to the actual runtime obtained using both linear regression and Equation 4.1 for ERCBench and Parboil2. Outliers (lying beyond the 1.5 inter-quartile range) are also plotted. Linear regression results in normalized predictions between 0.98x to 1.10x of actual runtime for ERCBench and 0.95x to 1.12x for Parboil2. This strongly supports our hypothesis that GPU kernel runtime is a linear function.

Unlike the models constructed by linear regression which are built using the end times of all thread blocks, predictions from Equation 4.1 only use the duration of the first thread block. These predictions normalized to actual runtime lie between 0.54x to 1.18x for ERCBench and 0.39x to 1.49x for Parboil2. If we exclude outliers, normalized predictions are between 0.66x and 1.18x for ERCBench and 0.6x and 1.2x for Parboil2. We investigate the major causes for this inaccuracy in the following sections.
Figure 4.5: SGEMM execution as recorded on a different SM from that of Figure 4.3. Start times of all subsequent blocks are altered by the end times of the first 5 blocks. Again, black square represent start times of a thread block, dark blue circles represent ending times, the green line depicts the linear fit, and the red line depicts the value of equation 4.1.

4.3.3 Non-Staircase Model Behaviour

Figure 4.5 presents a case where prediction using Equation 4.1 underestimates the total runtime. In the figure, from the same execution as Figure 4.3 but from a different SM, the first $R$ blocks each end at different times. As a result, the starting times of subsequent blocks are staggered. While the runtime continues to be linear, direct application of Equation 4.1 to such executions lead to gross underestimates. We observe such staggered executions commonly on hardware, but they were entirely absent in the simulator that we used.

4.3.4 Systematic Variations in Duration per Thread Block

The duration of each thread block, $t$, can vary during execution due to both random errors and systematic factors. In this section, we look at the factors that systematically affect $t$ and therefore need to be accounted for in prediction mechanism.
Fig. 4.6: Boxplots of thread block durations ($t$) normalized to their average for a kernel. The maximum value for render is 4.

**Differing Work per Thread Block**

Although the CUDA Programming Guide [NVCCPG] recommends that work per thread block be uniform for best performance, it does not require it, and thus $t$ can be non-uniform across thread blocks. For example, if the work done by thread blocks in the kernel differs based on the value of their inputs, then each thread block could take a different amount of time to run.

We use the data from Section 4.3.2 to examine the distribution of thread block durations. Figure 4.6 shows the boxplot of thread block durations ($t$) normalized to their average for kernels in the ECRBench suite. Observe that values of $t$ for the majority of thread blocks are within 0.95x to 1.1x of the average except in the case of RayTrace’s render kernel. This is expected since render’s thread blocks perform differing amounts of work. But even in this case, we can see 50% of its thread blocks are within 0.75x to 1x of the mean. Furthermore, despite the magnitude of deviation, the linear regression model for RayTrace predicts with a maximum error of 9%, while even Equation 4.1 has a maximum error of 18%. For Parboil2 (not shown here), the major long-running kernels tend to have uniform thread block durations, but the smaller kernels can exhibit non-uniformity – cutcp’s thread block times vary from 0.4x to 1.37x of the average. We conclude that the majority of kernels demonstrate the tendency to perform nearly uniform amount of work per thread block. For those kernels that do
not, our predictor implementation (Section 4.4) uses the actual runtime as feedback to correct any drift.

**Differing SM Behaviour**

Figures 4.3 and 4.5 demonstrate that individual SMs can vary in their behaviour for the same kernel during the same run. Some GPU programs, such as those studied by Liu et al. [2009] and Samadi et al. [2012], exhibit load imbalance across SMs when sizes of their inputs are varied. To obtain reliable predictions for these programs we implement per-SM predictors.

**Effect of Residency**

Although each kernel has a fixed maximum residency $R$, non-availability of resources during concurrent kernel execution might limit the number of resident thread blocks. Therefore, we investigate the effects of residency on $t$. To separate out the effects of co-runners, these experiments are run on hardware with each kernel running alone at different residencies. The next section considers the effects of co-runners. Figure 4.7 shows the variation in $t$ as residency is varied for a kernel. The values of $t$ are smallest

---

Figure 4.7: Average thread block duration at various residencies normalized to average thread block duration at residency 1. AES-e, AES-d and render have a maximum residency of 6 blocks and all other kernels have maximum residency of 8 thread blocks.
when residency is 1 and increase with residency. However, as residency is increased, Figure 4.8 shows that total runtime decreases and ultimately saturates. Thus, increases in $t$ are offset by the increase in throughput due to increased residency.

The actual rate of increase in $t$ for a kernel as the residency increases is a non-linear function and depends on both the kernel and the GPU it is running on. For example, SHA1 has a maximum residency of 8 thread blocks. However, at 64 threads to each thread block, there are only 16 warps at maximum residency. Therefore, on the Fermi which can issue two instructions per clock, SHA1 is unable to supply enough instructions to saturate issue at low residencies ($< 4$). However, once its residency increases beyond 4, it supplies at least two instructions per cycle, but its performance is now limited by two other factors: shared memory bandwidth, and its limited ILP due to long dependent chains of consecutive instructions. We leave the detailed modeling of these interactions to future work. In our predictor, we simply resample $t$ whenever residency changes.

**Effects of Co-runners**

The effect of co-runners on $t$ was studied by running thread blocks from different kernels together. Unlike the data presented up to this point in this chapter, the results in this section are necessarily from the simulator (Section 4.6.2).
Figure 4.9 shows the effect on mb_sad_calc’s average thread block durations at different residencies. In this experiment, mb_sad_calc runs along with 256 threads of another kernel. From the figure, we observe that the effect on average thread block duration varies depending on the co-running kernel. The 256 threads of SHA1 result in a greater change in the thread block duration of mb_sad_calc than any other kernel.

Figure 4.10 shows the effect on the thread block duration of mb_sad_calc as the number of threads of co-running NLM2 are varied. The duration per thread block for mb_sad_calc varies from approximately 16000 cycles when alone to nearly 28000 cycles when running with seven thread blocks of NLM2.

Clearly, runtime is affected by both the partitioning of resources and the identity of the co-running kernels. In our predictor, therefore, we resample \( t \) whenever the set of co-runners or their residencies changes.

### 4.3.5 Summary

The runtime of a GPU kernel running alone at a fixed residency is a linear function. While the models obtained by linear regression are accurate, they must be built using data from complete runs as using a limited number of thread blocks affects accuracy. We found that linear models built using the end times of the first \( R \) thread blocks were not
Figure 4.10: Average thread block durations from simulator for SAD mb_sad_calc kernel (61 threads per block) when sharing the GPU with NLM2 as residencies are varied for both kernels.

very accurate. Linear models that used the first $2R$ thread blocks fared better, predicting between $0.8x$ to $2x$ of actual runtime for the ERCBench kernels. However, in terms of runtime, $2R$ blocks represent 7% to 65% of total ERCBench kernel runtimes (median 18%), thus compromising on timeliness.

Runtimes of concurrently running kernels, on the other hand, are at best only piecewise linear, necessitating frequent resampling of $t$. Equation 4.1, which only requires the duration of a single thread block is therefore a better choice to predict concurrent kernel runtime as it can be extended easily to deal with them (Section 4.4). In fact, we find that for the scheduling policies we evaluate, a rough but early prediction is just as useful as an accurate oracle-supplied prediction (Section 4.6.4).

### 4.4 The Simple Slicing Predictor

The Simple Slicing (SS) runtime predictor is an online, concurrent-kernel aware predictor whose design is based on equation 4.1 while taking Sections 4.3.3–4.3.5 into consideration. Its prediction of runtime is an estimate of how much time a kernel would take to complete if it was running from now (i.e. the time at which the prediction is made) to completion, under the current conditions ($t$, residency and co-runners).

To accommodate changes in $t$ as the kernel executes (Section 4.3.4–4.3.4), we split
the execution of a kernel into multiple slices. Each slice is demarcated by any of the events that cause changes in $t$. In our current design, kernel launches and kernel endings mark the boundaries of slices for all running kernels. We assume that $t$ remains constant within a slice enabling the predictor to predict timings for blocks in that slice. Our predictions assume that the last thread block to execute is contained in the current slice since slice boundaries cannot be predicted in advance. Finally, as each SM can vary in behaviour, our predictor predicts runtimes for each kernel on a per-SM basis.

### 4.4.1 Predictor State

Table 4.1 details the state used by the Simple Slicing predictor that is maintained on each SM on a per-kernel basis. State updates (except for the prediction itself) are independent of the predictor and occur on any of the following four events: launch of a kernel onto an SM (ONLAUNCH), start of a thread block (ONBLOCKSTART), end of a thread block, (ONBLOCKEND) and finally end of a kernel (ONKERNELEND). Algorithms for state updates are detailed in Algorithm 4.

When a kernel is launched, we initialize all its per SM counters to zero. Then, $Resident\_Blocks$ is initialized to $R$, the maximum number of blocks that can reside at a time on an SM when running alone. Finally, we initialize $Total\_Blocks$ to the number of thread blocks we expect to execute on that SM. With current schedulers, this is only an estimate. Current schedulers (e.g., Fermi) dynamically assign thread blocks to SMs. Depending on when each thread block terminates, the number of thread blocks executed per SM can vary. $Total\_Blocks$ can, therefore, be less than or more
Algorithm 4 Functions to update per-kernel state on each SM. Kernel.Residency is the maximum number of resident blocks for Kernel; Kernel.Blocks is the total number of thread blocks; $N_{SM}$ is the number of SMs; $clock()$ is the current clock cycle; blkindex is the block identifier on the SM (0–7).

1: `function` ONLAUNCH(Kernel)
2: `Resident.Blocks` ← Kernel.Residency
3: `Total.Blocks` ← $\lceil$ Kernel.Blocks/$N_{SM}$ $\rceil$
4: `Reslice` ← true
5: `end function`
6: `function` ONKERNELEND(Kernel)
7: `Reslice` ← true
8: `end function`
9: `function` ONBLOCKSTART(Kernel, blkindex)
10: `Active.Blocks` + +
11: Block.Start[blkindex] ← $clock()$
12: `end function`
13: `function` ONBLOCKEND(Kernel, blkindex)
14: `Active.Blocks` − −
15: `Done.Blocks` + +
16: $t$ ← $clock() - Block.Start[blkindex]$
17: `end function`

than the actual number of blocks that execute on an SM. We currently assume uniform distribution and hence set it to $\lceil$ Kernel.Blocks/$N_{SM}$ $\rceil$ where $N_{SM}$ is the number of SMs..

In ONBLOCKSTART and ONBLOCKEND, a number of book-keeping variables Start_Cycle, Active_Blocks and Done_Blocks are updated. Active_Kernel_Cycles tracks the actual number of cycles the kernel has been running and is incremented on every cycle that it has a warp running on an SM. Pred_Cycles contains the runtime prediction for the kernel and is calculated by the predictor as described in the next section.

4.4.2 Prediction

We use the following equation to predict the runtime for a kernel:

$$Pred_{\text{Cycles}} = Active_{\text{Kernel}_{\text{Cycles}} + \frac{(Total_{\text{Blocks}} - Done_{\text{Blocks}}) \times t}{Resident_{\text{Blocks}}}} \quad (4.2)$$

The prediction is calculated at the end of the handler of ONBLOCKEND, and uses
the duration of the first thread block of a slice as the value for $t$. The actual runtime of a kernel so far, $Active\_Kernel\_Cycles$, is used to correct predictor drift. The latter part of the equation is essentially Equation 4.1 though we choose not to use the ceiling function to correct for effects of the staggered endings of the initial blocks. $Reslice$ is set to $false$ after every prediction.

### 4.4.3 Predictor Accuracy

We evaluate the predictor’s accuracy by comparing the predicted runtime to actual runtime. We measure accuracy only for the last slice.

We obtain predictions for evaluation by running traces of actual program runs through the predictor. Each trace of a program contains the start and end times for every thread block as well as the SM it ran on. We group the traces as: (i) single-gpu – traces from runs of single applications on the GPU, (ii) single-sim – traces from runs of single applications on the simulator, (iii) mpmax – traces from two-program workloads executed on the simulator using the MPMax scheme described later in Section 4.5.2. The single-gpu and single-sim traces feature only a single slice whereas mpmax features at least two slices.

We evaluate equation 4.2 in a slice-aware mode as well as a slice-unaware mode,
where the prediction is only made once, at the beginning of the kernel. Figure 4.11 presents the results. For all the groups, the simple slicing predictor is accurate to within 2x of the actual runtime for the majority of programs. For single-gpu, the predictions are between 0.48x to 1.08x of actual runtime. Since equation 4.2 is not step function, single-sim predictions are less accurate than those for the hardware. For mp-max, the simple slicing predictor corrects the underestimates made by the slice-unaware predictor, and the majority of its predictions are between 0.5x and 2x of runtime. We emphasize that these errors do not limit our scheduling policies as our overall evaluation will show.

4.5 Thread Block Scheduling

We next describe the four scheduling policies that we evaluate in this work. First, we describe two of our policies, SRTF and SRTF/Adaptive, both of which use estimates of runtime provided by the Simple Slicing predictor to guide their scheduling decisions. We then describe the policies FIFO and MPMax against which we compare, neither of which use runtime estimates. Other than FIFO, all the policies described in this section are concurrent-kernel aware.

4.5.1 Runtime Aware Policies

Shortest Remaining Time First (SRTF)

Our implementation of the Shortest Remaining Time First scheduling policy is best understood through the (logical) queue system depicted in Figure 4.12. When a grid (say $K1$) is launched by `cudaLaunch`, it is added to the GPU-wide kernel queue. If
Figure 4.13: SRTF versus SRTF/Adaptive. Note that \( N_2^1 + N_2^2 = N_2 \).

no other kernel is running, it is moved directly to the running queue and starts executing on the SM. In this case, \( K_1 \)’s runtime prediction will be obtained passively when its first thread block finishes.

If a new grid (say \( K_2 \)) arrives while \( K_1 \) is running, we must actively obtain a prediction to decide if it must replace \( K_1 \) on the SMs. To do this with minimum disruption and delay, \( K_2 \) is first added to a single SM’s sampling queue. On that SM, \( K_2 \)’s thread blocks receive priority over those of \( K_1 \) (which is in the SM’s running queue) and it commences a sampling run where it deploys a fixed number of thread blocks to the SM. In the course of the sampling run, \( K_2 \)’s thread blocks may execute concurrently with \( K_1 \)’s thread blocks. After these sample blocks finish, \( K_2 \) is moved to the running queue since the predictor now has enough information to make an initial prediction.

Once in the running queue, thread blocks from \( K_1 \) or \( K_2 \) are dispatched to the SM depending on which of \( K_1 \) or \( K_2 \) is predicted to have shorter runtime. SRTF does not share GPU resources between kernels. Finally, when a grid finishes execution, it is removed from both the kernel and running queues.

SRTF/Adaptive

The SRTF scheduling policy does not share resources among concurrently executing kernels. While this leads to good performance, it can be unfair. In the example of SRTF scheduling depicted in Figure 4.13, kernel \( K_2 \) is delayed by \( T_1 \), the time taken by kernel \( K_1 \), leading to a slowdown of \((T_1 + T_2)/T_2\). In the extreme case, if \( T_1 \) is only slightly smaller than \( T_2 \), then \( K_2 \) experiences nearly a 2x slowdown. By sharing resources between the two kernels, as in the SRTF/Adaptive part of the same figure, it is possible to ensure equitable progress for both kernels. The slowdown in shared execution for \( K_1 \) and \( K_2 \) is then \( TS_1/T_1 \) and \((TS_1 + TS_2)/T_2 \) respectively. In this example, sharing also leads to an overall speedup, but this is not guaranteed in general.

Our proposal for resource sharing, SRTF/Adaptive, shares GPU resources among concurrently executing kernels, but only if it detects that the current SRTF schedule is
unfair. To do this, it calculates the slowdown for running kernels in non-sharing mode as above. If the difference between the smallest slowdown and the largest slowdown exceeds a threshold (we use 0.5), SRTF/Adaptive switches to sharing mode in which the maximum residency of kernel is limited with the rest turned over to co-running kernels.

Once in sharing mode, SRTF/Adaptive continues to monitor slowdowns for each program. Calculation of shared runtimes (as in Figure 4.13) is slightly more involved. The value of $TS_1$ is simply the runtime for $K_1$. Calculating $TS_2$ requires knowing $N_{21}$ and $N_{22}$, the number of blocks of $K_2$ that execute in shared and exclusive mode respectively. $N_{21}$ is obtained by solving for $N$ using Equation 4.1 with $R$ equal to the shared residency and $T = TS_1$ (i.e. $N = TR/t$). The value of $N_{22}$ is then $N - N_{21}$. This is an iterative procedure over all running kernels and is hence bounded by the limited number of concurrent kernels (8 on the NVIDIA Fermi). The exclusive runtime (i.e. $T_1$ or $T_2$) is the prediction from the exclusive part of a run. If there was no exclusive part, current predictions of runtime are used instead.

Based on an exhaustive study of all possible residency assignments for sharing, we find that the best fairness and performance (as defined in this chapter) for the workloads in this chapter are obtained when the SM is fully occupied and the co-running kernels run nearly equal number of thread blocks. Therefore, we chose a fixed limit of 3 for the fastest kernel. An alternative implementation that distributed resources equally among all running kernels initially and then redistributed resources as necessary was too slow to achieve fair execution and good performance.

4.5.2 Runtime Unaware Policies

FIFO (Baseline)

The FIFO Thread Block Scheduler is based on the NVIDIA Fermi Thread Block Scheduler. It schedules thread blocks from running kernels in the order of their arrival. Only when all the thread blocks of a kernel have been dispatched to the streaming multiprocessors for execution are blocks from the next kernel scheduled.

Just-in-Time MPMax

Just-in-time MPMax is a resource-allocation policy, not a scheduling policy per se. It is based on the the best-performing MPMax policy from Chapter 3. In this policy, each executing kernel sets aside resources for a hypothetical “MPMax” kernel, a composite constructed from co-running kernels. For example, under this policy, when two kernels
Table 4.2: Grid Characteristics of ERCBench Kernels. TPB = Threads Per Block, R = Maximum Residency

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel</th>
<th>R</th>
<th>TPB</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-d</td>
<td>aesDecrypt128</td>
<td>6</td>
<td>256</td>
<td>1429</td>
</tr>
<tr>
<td>AES-e</td>
<td>aesEncrypt128</td>
<td>6</td>
<td>256</td>
<td>1429</td>
</tr>
<tr>
<td>ImageDenoising-nlm2</td>
<td>NLM2</td>
<td>8</td>
<td>64</td>
<td>4096</td>
</tr>
<tr>
<td>JPEG-d</td>
<td>CUDA...IDCT</td>
<td>8</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>JPEG-e</td>
<td>CUDA..DCT</td>
<td>8</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>RayTracing</td>
<td>render(^1)</td>
<td>5</td>
<td>128</td>
<td>2048</td>
</tr>
<tr>
<td>SAD</td>
<td>mb_sad_calc</td>
<td>8</td>
<td>61</td>
<td>1584</td>
</tr>
<tr>
<td>SHA1</td>
<td>sha1_kernel_direct</td>
<td>8</td>
<td>64</td>
<td>1539</td>
</tr>
</tbody>
</table>

\(^1\)The instrumented version of render used in Figure 4.7 uses one fewer register allowing it to have six resident blocks. This is an artefact of the CUDA compiler.

\(K_1\) and \(K_2\) execute together, \(K_1\) will set aside resources for one thread block (per SM) of \(K_2\) and vice versa. Our adaptation differs from the original MPMax in two significant aspects: i) the resources set aside by each kernel are calculated on the basis of the kernels actually running on the GPU at that instant and ii) when concurrent kernel execution ceases, kernels reoccupy the resources they had set aside. When scheduling, thread blocks are issued from a kernel until its MPMax limit is reached, after which the next kernel in FIFO order gets to issue thread blocks.

### 4.6 Evaluation

We evaluate the execution of concurrent kernels under the four TBS policies discussed in Section 4.5. Since existing benchmarks lack concurrent kernels, our evaluation uses 2-program workloads. The primary metrics reported are system throughput (STP), average normalized turnaround time (ANTT) [Eyerman and Eeckhout, 2008] and the StrictF metric for fairness [Vandierendonck and Seznec, 2011]. StrictF is defined as the ratio of minimum slowdown to maximum slowdown, with a value of 1 indicating high fairness.

### 4.6.1 Benchmarks

Our 2-program workloads are composed of 8 kernels from 8 ERCBench [Chang et al., 2010] benchmarks. The BitonicSort, DVC and RSA benchmarks are not used in our evaluation. BitonicSort uses a single thread block configuration, DVC does not compile on our system, while RSA uses `cudaHostMalloc` which is currently not supported.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel</th>
<th>Simulator (cycles)</th>
<th>Simulator (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Runtime</td>
<td>Mean t</td>
</tr>
<tr>
<td>AES-d</td>
<td>aesDecrypt128</td>
<td>234154</td>
<td>14529</td>
</tr>
<tr>
<td>AES-e</td>
<td>aesEncrypt128</td>
<td>226335</td>
<td>14031</td>
</tr>
<tr>
<td>ImageDenoising-nlm2</td>
<td>NLM2</td>
<td>692686</td>
<td>19873</td>
</tr>
<tr>
<td>JPEG-d</td>
<td>CUDA..IDCT</td>
<td>24853</td>
<td>5238</td>
</tr>
<tr>
<td>JPEG-e</td>
<td>CUDA...DCT</td>
<td>25383</td>
<td>5367</td>
</tr>
<tr>
<td>RayTracing</td>
<td>render</td>
<td>416563</td>
<td>15167</td>
</tr>
<tr>
<td>SAD</td>
<td>mb_sad_calc</td>
<td>441297</td>
<td>32332</td>
</tr>
<tr>
<td>SHA1</td>
<td>sha1_kernel...</td>
<td>22224223</td>
<td>1708531</td>
</tr>
</tbody>
</table>

Table 4.3: Runtimes for ERCBench Kernels on the simulator. \( \%\text{RSD} = 100 \times \text{Std. Dev}(t)/\text{Mean} t \)

<table>
<thead>
<tr>
<th>Number of SMs</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources per SM</td>
<td>1536 threads, 32768 registers, 48KB shared memory, Maximum 8 Thread Blocks, Maximum 48 warps</td>
</tr>
<tr>
<td>Threads per warp</td>
<td>32</td>
</tr>
<tr>
<td>Warp scheduler</td>
<td>Loose Round Robin</td>
</tr>
</tbody>
</table>

Table 4.4: Simulator Configuration

by the simulator. Tables 4.2 and 4.3 highlight the grid characteristics and runtimes for these kernels which will be used to interpret our results in the following sections.

### 4.6.2 Simulator

We modify the GPGPU-Sim simulator (3.2.0) [Bakhoda et al., 2009] for our experiments. Our modifications extend the simulator to execute multiple kernels concurrently (the released version only runs a single kernel on an SM at a time) but leave the actual cycle-accurate simulator for each thread unchanged. We also add a functional thread block scheduler and predictors whose behaviour is as described in Sections 4.4 and 4.5. The simulated GPU configuration, as listed in Table 4.4, is the GTX 480 configuration supplied with GPGPU-Sim.

### 4.6.3 Methodology

Multiprogrammed workloads cannot be run directly on the GPU or on the simulator. Therefore, to achieve concurrent execution of kernels, we use the techniques described
Table 4.5: Geomean STP, ANTT and Fairness for various scheduling policies. Note that ANTT is a lower-is-better metric.

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>STP</th>
<th>ANTT</th>
<th>Fairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>1.35</td>
<td>3.66</td>
<td>0.19</td>
</tr>
<tr>
<td>MPMAX</td>
<td>1.37</td>
<td>2.15</td>
<td>0.36</td>
</tr>
<tr>
<td>SRTF</td>
<td>1.59</td>
<td>1.63</td>
<td>0.52</td>
</tr>
<tr>
<td>SRTF/ADAPTIVE</td>
<td>1.51</td>
<td>1.64</td>
<td>0.56</td>
</tr>
<tr>
<td>SJF</td>
<td>1.82</td>
<td>1.13</td>
<td>0.80</td>
</tr>
</tbody>
</table>

in Section 3.6.1 to construct multithreaded workloads with each thread executing the individual CUDA programs. As GPGPU-sim performs cycle-accurate simulation only for GPU kernels and memory transfers are only functionally emulated [Lustig and Martonosi, 2013] with CPU code running at full speed, we only record timings for the simulated kernels even though we run the entire workload to completion.

We use all possible 28 2-program workloads from the ERCBench suite. Since the order of kernel arrivals affects a scheduler significantly, we simulate and present results for both orders of arrival making for a total of 56 2-program workloads. Our primary results evaluate kernel arrivals that are staggered by upto 100 cycles, thus the two kernels start nearly together. We also present results for different arrival offsets where the second kernel arrives after 25% and 50% of the first kernel has finished executing.

### 4.6.4 Results

#### Overall Results

Table 4.5 summarizes the results of our evaluation. SJF shows the best STP, ANTT and fairness values, but is unrealizable. Next to SJF, the SRTF policy has the highest STP and the lowest ANTT among all scheduling policies. It also has the second best fairness value among the realizable policies that we evaluated. Compared to our baseline FIFO, SRTF improves STP by 1.18x, ANTT by 2.25x and Fairness by 2.74x. SRTF also outperforms MPMAX by 1.16x (STP) and 1.3x (ANTT). The ADAPTIVE policy is the fairest among all the realizable policies studied, with a 2.95x fairness improvement over FIFO. It is also the second-best policy with its STP being 1.12x better and ANTT being 2.23x better than baseline FIFO. However, since ADAPTIVE achieves fairness by sharing resources between concurrently executing kernels, its STP is lower by 5% than that of SRTF. FIFO is the least fair policy.
Figure 4.14: System Throughput (STP) for various scheduling policies

System Throughput

Figure 4.14 plots the system throughput for all 56 workloads for all policies. SRTF outperforms other non-SJF schedulers in nearly all of the workloads. However, as Table 4.5 shows, there is a gap of 12.64% between SRTF and SJF. Unlike SJF which schedules kernels even before they run, SRTF must learn the runtimes of concurrently executing kernels to determine which is the shorter kernel. In our implementation, this is done through a sample execution as described in Section 4.5. Since running thread blocks cannot be pre-empted, sample execution must wait until resources are available for sampling. This leads to two possible scenarios. In the first scenario, the kernel currently executing has the shortest runtime, and hence the sampling procedure disrupts its execution when compared to SJF. In the second scenario, the latest kernel to arrive has the shorter runtime and so the time it waits for sampling to begin delays its execution as compared to SJF.

The effect of sampling on performance is largely determined by the relative thread block durations of each concurrently executing kernel. Consider the RayTracing+JPEG-d pair in which JPEG-d is launched as the second kernel. JPEG-d’s kernel is very small, with average thread block duration of 5238 cycles (Table 4.3). RayTracing’s thread blocks take on average 15168 cycles to execute. Arriving second, JPEG-d may
wait up to 15168 cycles to begin sampling. Once sampling is done, it may have to wait another 15168 cycles on the SMs that were not participating in the sampling and which would have continued executing thread blocks from RayTracing. So, for a kernel that takes a total of about 25000 cycles when running alone, in this example JPEG-d has already slowed down by 2x.²

To quantify the effects of sampling on performance, we conducted an experiment where we omitted the sampling phase. In this zero-sampling variant of SRTF, we provided the runtimes to the SRTF scheduler directly as in SJF. For our workloads, this improved STP by 3% to 1.64 and ANTT by 22% to 1.33. The remaining performance gap is therefore the time spent waiting for existing thread blocks to finish. Since the zero sampling experiment also provided accurate runtimes to the SRTF, the results also show that SRTF is very tolerant of errors in the simple slicing predictor.

On average, MPMAX and FIFO have almost the same throughput. The detailed results show that MPMAX outperforms FIFO for slightly more than 50% of the workloads. This is because of our experimental setup which evaluates all possible 2-program workloads. For half of the workloads, FIFO schedules just as SJF would.

**Average Normalized Turnaround Time**

Figure 4.15 plots the ANTT for each of the 56 workloads. Although the ANTT values are nearly indistinguishable for about 35% of the benchmarks, SRTF and ADAPTIVE are the only realizable policies to have the lowest ANTT values for all but two of the workloads. At 30.95 and 37.77, the highest ANTT values for SRTF/ADAPTIVE and SRTF (not shown in the figure) are well below the highest ANTT value (FIFO with 425.45) but are still higher than MPMAX whose highest ANTT value is 10.08. These maximum values are for SHA1+JPEG-d (the next highest value is for SHA1+JPEG-e) and are the result of JPEG-e having to wait 1.7M cycles for SHA1’s thread blocks to finish on the SMs not participating in the sampling. Since MPMAX reserves runtime resources on all SMs for concurrently executing kernels, JPEG-e does not experience any such delay.

**Fairness**

Figure 4.16 plots StrictF, the fairness metric we use, for all of the workloads. SRTF/ADAPTIVE, our fairness-oriented policy, executes 35 of the 56 workloads in sharing

---

²This is still better than FIFO where the slowdown in this case would be 17.76x.
Figure 4.15: Average Normalized Turnaround Time (ANTT) for various scheduling policies. ANTT is lower-is-better metric.

Figure 4.16: Fairness (StrictF) for various scheduling policies.
Table 4.6: Geomean STP, ANTT and Fairness for various scheduling policies when second kernel arrives at 25% and 50% of first kernel’s runtime. ANTT is a lower-is-better metric.

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>25%</th>
<th></th>
<th>50%</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>STP</td>
<td>ANTT</td>
<td>Fair.</td>
<td>STP</td>
</tr>
<tr>
<td>FIFO</td>
<td>1.44</td>
<td>2.74</td>
<td>0.27</td>
<td>1.48</td>
</tr>
<tr>
<td>MPMAX</td>
<td>1.45</td>
<td>2.05</td>
<td>0.38</td>
<td>1.49</td>
</tr>
<tr>
<td>SRTF</td>
<td>1.62</td>
<td>1.60</td>
<td>0.53</td>
<td>1.63</td>
</tr>
<tr>
<td>SRTF/ADAPTIVE</td>
<td>1.56</td>
<td>1.65</td>
<td>0.56</td>
<td>1.59</td>
</tr>
</tbody>
</table>

While MPMAX achieves higher fairness than FIFO for 80% of the workloads, for 20% of the workloads, however, sharing leads to a loss in performance that is relatively greater for the smaller component of the workload as compared to running with full resources.

4.6.5 Sensitivity to Arrival Time

To evaluate sensitivity to arrival time, we simulate workloads where the second kernel arrives after the first kernel has executed a fixed number of cycles. Table 4.6 presents the results when the second kernel arrives at 25% and 50% of the runtime of the first kernel. SRTF continues to perform well across all metrics for both 25% and 50% arrival offsets. From 25% to 50%, however, the gaps between the different policies shrink, a consequence of the fact that as the kernels start farther apart in time, there is less opportunity for the scheduler to perform.

4.7 Related Work

To the best of our knowledge, this is the first work to explore thread block scheduling policies for concurrent kernels on GPUs. In this section, we list closely related work.

TimeGraph [Kato et al., 2011] schedules OpenGL programs at the OS-level. It enforces policies by limiting access to the GPU at the device driver level using OS-level priorities. Once a GPU kernel is launched, however TimeGraph has no control over it. TimeGraph’s policies do not support concurrent kernels. Our work differs by
being able to schedule concurrent kernels, as well by being implemented at the thread block scheduler level.

Runtime prediction of GPU kernel execution time for scheduling across heterogeneous CPU/GPU systems has been explored in several works [Belviranli et al., 2013, Jia et al., 2012, Gregg et al., 2011, Luk et al., 2009, Diamos and Yalamanchili, 2008]. Although we do not explore scheduling a kernel across CPU and GPU, our online predictor predicts runtime without requiring a historical database and is also aware of concurrent kernels.

Scheduling of concurrent GPU kernels is potentially similar to co-scheduling of programs on SMTs and CMPs. A representative work is that of Chen and John [2011], who profile running applications using hardware profilers and then use an analytical model to predict performance and compute resource partitions on an CMP.

4.8 Conclusion

We presented a novel online runtime predictor for GPU kernels that exploited the structure of kernel grids to obtain predictions. Our predictor observed thread block durations and predicted the total runtime for the kernel. On benchmarks from ERCBench, the predictions for single programs on hardware were within 0.48x to 1.08x of actual execution time using only the duration of a single thread block.

We used the predictor to build a thread block scheduler with runtime aware policies. The two runtime aware policies that we evaluated, SRTF and SRTF/Adaptive, were superior to the other realizable policies in terms of system throughput, turnaround time and fairness. Compared to FIFO, our implementation of the SRTF policy improved STP by 1.18x and ANTT by 2.25x. SRTF also outperformed MPMax, a state-of-the-art resource allocation policy, with improvements of 1.16x in STP and 1.3x in ANTT. Our SRTF/Adaptive policy achieved the highest fairness among all the realizable policies, 2.95x better than FIFO. Finally, SRTF bridged 49% of the gap between FIFO and SJF, approaching to within 12.64% of the throughput of SJF.
CUDA was introduced in 2006. It has been a tremendous success and has inspired several standards such as OpenCL. However, seven years and nearly as many major revisions later, CUDA still does not provide any assistance to the programmer to tackle the problems identified in this thesis. Even now, the toolchain for GPU programming basically treats it as yet another architectural target which means that several problems that emerge only in the interaction of the CPU and GPU are simply abandoned to the programmer. In fact, it is not clear exactly how or who is responsible for tackling these problems. In this thesis, we have advocated that runtimes are best positioned to tackle these problems by demonstrating that they (and only they) possess information required for good solutions. We have demonstrated several runtime solutions that advance the state-of-the-art for each problem.

We advise that future GPGPU systems should make it easier for the programmer to control the behaviour of the CPU or GPU runtime. This could be done, for example, by exposing low-level device-specific primitives directly to the programmer instead of layering them with a device-specific runtime. Whole-system über runtimes could then be constructed in a scalable and composable fashion. For example, as of November 2013, OpenCL 2.0 [Khronos OpenCL Working Group, 2013] has abandoned explicit data transfer mechanisms for shared virtual memory instead choosing to adopt primitives that declare coherence intent – read, write or overwrite. Presumably, suitable mechanisms in the runtime (mostly hardware) will keep data coherent once intent has been clarified. No such primitives exist for resource allocation and scheduling policies yet. Our work has highlighted that control over these policies are important to performance, so we may yet see primitives appear in the future.

We conclude with a summary of our results and avenues for future work.
5.1 Summary of Results

In AMM, we demonstrated that it is necessary for an automatic memory management scheme to use runtime coherence if it must avoid redundant transfers in the general case. This led to us developing a compiler–runtime hybrid scheme for maintaining runtime coherence. We showed that the AMM system achieves comparable performance to programmer-inserted manual memory management. In fact it achieves a speedup of 1.06x over manual by eliminating some redundant data transfers. Compared to a state-of-the-art memory management system, CGCM [Jablin et al., 2011], our AMM system is faster by 1.29x. Moreover, compared to other existing runtime-only and compiler-only proposals, it also transfers 2.2x to 13.3x less data on average.

Although concurrent kernels were born out of a necessity to minimize resource wastage, we showed that current policy for concurrent kernels neither minimized resource wastage, nor improved concurrency. We showed that the fixed grid model of CUDA did not scale well across generations, particularly since resource allocation was tied to the grid, and hence ultimately controlled by the programmer. Without hardware control over resource allocation, this led to poor resource utilization and poor concurrency. Therefore, we proposed elastic kernels which decouple physical resource allocation from logical programming model considerations. By doing so, we allowed the runtime to make resource allocation decisions depending on the hardware. At the same time, we showed the GPU concurrency is limited by a number of serialization factors that prevent concurrency. One fundamental factor was kernel execution can prevent concurrency when it occupies too many resources. Using elastic kernels, we developed a number of resource allocation policies that mitigate the effects of these serialization factors.

Elastic kernels and our resource allocation policies improve average system throughput (STP) by 1.21x and average normalized turnaround time (ANTT) by 3.73x for two-program workloads compared to CUDA on real hardware. They also increase the number of concurrent kernels by 1.53x and reduce waiting times for kernels by three orders of magnitude. Our policies also achieve higher throughput, lower turnaround times and better resource utilization when compared to a static partitioning scheme and a runtime resource allocation scheme. Finally, our proposal for time-slicing of kernels improves STP by 7.8% and ANTT by 1.55x for programs with long running kernels.

With the GPU finally executing concurrent kernels, we showed that current FIFO thread block scheduling policies is sub-optimal. Compared to shortest-job first (SJF),
a provably optimal policy, FIFO had system throughput that was lower by 15% on average. Since SJF requires estimates of runtime, we proposed Structural Runtime prediction. By using the CUDA programming model’s requirement that every GPU program be expressed as a grid–thread block–thread, we developed a runtime prediction scheme that would exploit the resultant uniformity of structure.

We showed that it is possible to predict within 0.48x to 1.06x of a kernel’s runtime by observing only a single thread block. Using the predictor, we built a thread block scheduler for concurrent kernels that supported the Shortest Remaining Time First (SRTF) policy. Compared to existing FIFO hardware scheduling, the SRTF policy improved the system throughput on average by 1.18x and turnaround time by 2.25x. We also demonstrated a fairness-aware scheduler, SRTF/Adaptive, which is 2.95x more fair on average compared to FIFO. Our implementation of SRTF bridged 49% of the gap between FIFO and SJF, approaching to within 12.5% of SJF system throughput.

It is possible to build a system that combined AMM and one of Elastic Kernels or our SRTF scheduler. The latter two attack the same problem and represent two differing approaches to improve GPGPU concurrency. The tradeoffs between Elastic Kernels and SRTF have been outlined in Section 4.6. Although a lack of benchmarks prevents us from comparing Elastic Kernels and our SRTF scheduler exhaustively, we speculate that multi-threaded programs would benefit from Elastic Kernels while multi-programmed workloads would benefit from our SRTF scheduler. AMM, on the other hand, is completely orthogonal to both and thus can be used in combination with either. For single-threaded programs running alone, we do not expect such a system to either improve or worsen performance compared to current systems. However, for programs that share the GPU or deploy concurrent kernels, we expect the performance to be inline with those reported in Sections 3.6 or 4.6 depending on the chosen implementation.

5.2 Future Work

AMM, as presented, is limited to single-threaded programs. Extending AMM to multi-threaded programs might require mutual exclusion primitives to implicitly execute coherence checks as well. Other automatic memory management systems offer two features that may be desirable: sub-array granularity data transfers and pointer translation. The former will require changes to the compiler analysis, but for performance reasons is perhaps best implemented in hardware. For example, current integrated GPUs targeted by OpenCL 2.0 use AMM-like primitives instead of data transfers and use hard-
ware mechanisms to maintain coherence between the integrated GPU and the CPU. Pointer translation is orthogonal to data transfers and is perhaps best implemented by co-operation between the address translation mechanisms of the GPU and CPU.

AMM currently requires that the entire program be instrumented with checks. Modifying AMM so that only subparts of the program need to be instrumented may lead to gains in performance as well as increased compatibility with libraries that cannot be instrumented.

Elastic kernels were used in this work to improve concurrent scheduling. However, thread block expansion can also be used to improve single-kernel performance by dynamically optimizing a grid’s resource usage. A key stumbling block to this remains the inapplicability of thread block expansion to kernels that use shared memory or barriers. Current CUDA programs do not expose enough semantic information about their shared memory usage preventing it from being sized automatically. Techniques that overcome this, say by annotating CUDA source code or by static analysis, are needed to extend the applicability of elastic kernels to such kernels.

Our current thread block scheduling policies emphasize throughput and turnaround time. However, the general ability to schedule thread blocks might potentially benefit from being more tightly integrated into the operating system scheduler.

A significant performance limiter in our TBS design arises from not being able to pre-empt running thread blocks. If existing thread blocks could be pre-empted, sampling as well as new kernel execution can begin earlier.

In this work, we used structural prediction to only predict the runtime of a kernel. Other properties of CUDA programs (e.g., branching behaviour, memory access patterns, etc.) could potentially benefit from the general technique.


D. Cunningham. *X10 2.1 CUDA*, 2010. URL http://docs.codehaus.org/display/XTENLANG/X10+2.1+CUDA.


I. Gelado et al. GMAC: Global Memory for Accelerators (version 0.0.20). Software Library. URL http://code.google.com/p/adsm/.

P. N. Glaskowsky. NVIDIA’s Fermi: The first complete GPU computing architecture. Whitepaper, 2009.


TOP500. TOP500 supercomputer sites. URL http://www.top500.org/.

TOP500List. TOP500 supercomputer sites: June 2013. URL http://www.top500.org/lists/2013/06/.

TOP500No1. Top500 supercomputer sites: Top #1 systems. URL http://www.top500.org/featured/top-systems/.


X10BMK. X10 code for all benchmarks used in the AMM evaluation. Website. URL http://hpc.serc.iisc.ernet.in/~sree/src/x10/.