Improving GPGPU Concurrency with Elastic Kernels

Sreepathi Pai Matthew J. Thazhuthaveetil R. Govindarajan

Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore, India
sree@hpc.serc.iisc.ernet.in, mjt@serc.iisc.in, govind@serc.iisc.in

Abstract

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 that we used in our work. Current GPUs therefore allow concurrent execution of kernels to improve utilization. In this work, we study concurrent execution of GPU kernels using multiprogram workloads on current NVIDIA Fermi GPUs. On two-program workloads from the Parboil2 benchmark suite we find concurrent execution is often no better than serialized execution. We identify that the lack of control over resource allocation to kernels is 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 concurrency policies that offer significantly better performance and concurrency compared to the current CUDA policy. We evaluate our proposals on real hardware using multiprogrammed workloads constructed from benchmarks in the Parboil 2 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 when compared to the current CUDA concurrency implementation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors)

Keywords GPGPU, CUDA, Concurrent Kernels

1. Introduction

Graphics Processing Units (GPUs) have evolved from hardware-accelerated fixed-function pipelines to multicore data parallel computation engines. Increasingly general purpose in nature, many are now used exclusively for *General Purpose GPU (GPGPU)* code (i.e. non-graphical or computational code), especially as components of some of the world's fastest supercomputers [19].

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 [9] to the NVIDIA Kepler [13], shared memory has trebled, registers have quadrupled, and the number of hardware threads has doubled (Table 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.

Resource	Tesla (1.3)	Fermi (2.0)	Kepler (3.0)
Registers	16384	32768 (2x)	65536 (2x)
Shared Memory	16384	49152 (3x)	49152
Threads	1024	1536 (1.5x)	2048 (1.33x)
Resident Blocks	8	8	16 (2x)

Table 1. Resources per Streaming Multiprocessor across three NVIDIA GPU generations. Multipliers in parentheses indicate change over preceding generation.

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 2). Ironically, we find that the *grid*[1], 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*. By allowing the programmer to create a large number of thread blocks – more than 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. This is possible because 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. Each thread block, in turn, consists of threads which ultimately consume resources and execute on the hardware. 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 illustrates how resource allocation at the thread block level leads to wastage. Each thread block occupies 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

[1] Although these constructs are common to all GPU programming models, we use CUDA terminology in this paper for consistency.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

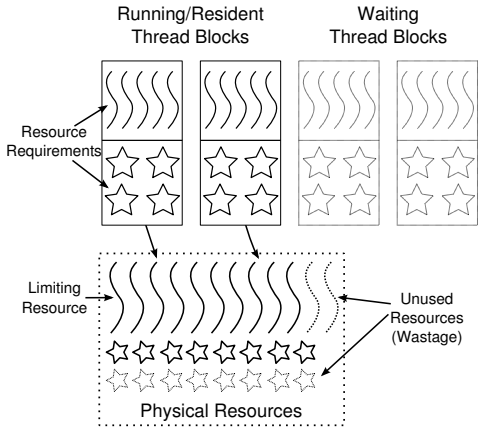

Figure 1. Resource wastage due to allocation at Thread Block granularity.

(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 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 3.1) in the same program. As a consequence, current GPUs have begun supporting concurrent execution of independent 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 2).

Past work [1, 7, 8, 14] has motivated GPU concurrency as a method to improve GPU throughput. Adriaens et al. [1] propose the use of GPU concurrency for mobile GPUs, Ravi et al. [14] look at GPGPU applications in the cloud and Guevara et al. [8] and Gregg et al. [7] 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 [1, 7, 8], or their techniques are not general enough to apply to all kernels [14]. We show in this work that they either underutilize resources significantly or do not perform as well as our policies.

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-serializing Streams* API to work around these limitations in the existing API and hardware.
- 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.
- Finally, we evaluate our proposals on real hardware using multiprogram workloads constructed from the Parboil2 benchmark suite. Averaged across the workloads, our best elastic policy improves system throughput (STP) by 1.21x and average normalized turnaround time (ANTT) by 3.73x over CUDA execution. Our time-slicing technique improves geomean STP by 7.8% and geomean ANTT by 1.55x where applicable. Our policies improve the average number of concurrent kernels by 1.53x over CUDA, and reduce kernel waiting time by up to three orders of magnitude.

This paper is organized as follows. Section 2 describes the motivation for GPGPU concurrency and how current concurrency policies are ineffective. Section 3 identifies other factors that inhibit GPGPU concurrency. Elastic kernels and elastic-kernel aware concurrency policies are presented in Section 4. Our Non-Serializing Streams API is described in Section 5. Section 6 contains the evaluation of our proposals and presents performance results.

2. Motivation

The NVIDIA Fermi is the only GPU available (as of Q2, 2012) 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 2 presents the resources used by the grids of all 18 kernels of the Parboil2 benchmark suite when executed on the NVIDIA Fermi GPU. 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 [3], 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 paper, 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 di-

Program	Kernel	TB	TPB	T%	R%	S%	B%
bfs	BFS_in_GPU	1	512	2	2	2	1
	BFS_multi_blk...	14	512	33	31	26	12
mri-q	ComputePhiMag...	4	512	10	5	0	4
	ComputeQ_GPU	1024	256	83	94	0	62
fft	GPU_FFT_Global	1024	128	67	62	0	100
stencil	block2D_hybrid...	512	256	67	94	17	50
cutcp	cuda_cutoff...	121	128	67	75	69	100
tpacf	gen_hists	201	256	50	70	81	38
histo	histo_final	42	512	100	94	0	38
	histo_intermediates	65	498	100	75	0	38
	histo_main	84	768	100	94	100	25
	histo_prescan	64	512	100	75	25	38
sad	larger_sad_calc_16	99	32	15	33	0	88
	larger_sad_calc_8	99	128	59	66	0	88
	mb_sad_calc	1584	61	33	50	38	100
mm	mysgemmNT	528	128	50	94	6	75
lbm	performStream...	13000	100	58	98	0	88
spmv	spmv_jds_texture	112	192	88	98	0	88
	Average			60	67	20	57

Table 2. 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.

Program	Kernel	CP	KT (ms)	Calls	FPT (%)
bfs	BFS_in_GPU	16	14.41	2	37
	BFS_multi_blk...	15	49.57	1	63
mri-q	ComputePhiMag...	16	0.002	1	0.002
	ComputeQ_GPU	2	44.91	2	99.998
fft	GPU_FFT_Global	0	0.11	8	100
stencil	block2D_hybrid...	2	2.70	100	100
cutcp	cuda_cutoff...	0	2.86	11	100
tpacf	gen_hists	13	840.40	1	100
histo	histo_final	0	0.07	100	6
	histo_intermediates	0	0.18	100	16
	histo_main	0	0.85	100	75
	histo_prescan	0	0.03	100	3
sad	larger_sad_calc_16	15	0.04	1	4
	larger_sad_calc_8	15	0.19	1	19
	mb_sad_calc	0	0.80	1	77
mm	mysgemmNT	2	5.51	1	100
lbm	performStream...	0	23.69	100	100
spmv	spmv_jds_texture	0	0.13	1	100

Table 3. 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.

rected 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. 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 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 [3] 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. [1]. 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. 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.

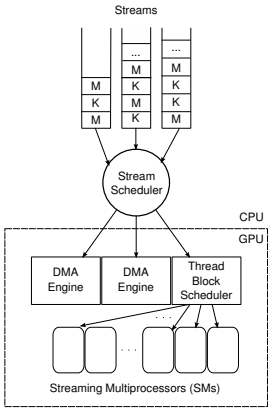

Figure 2. Components that affect concurrency on the Fermi GPU. Streams contain Memory (M) or Kernel (K) commands. A DMA engine performs unidirectional data transfers. With two DMA engines on the Fermi, two memory transfers may run concurrently with kernel execution.

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* [12, 15] 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 2 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. The Fermi Thread 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.

3.2 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 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 [1]. This guarantees that each grid will always have resources to execute and multiple grids can execute across different SMs. However, as noted in Section 2, it does not address the problem of under-utilization of resources. Hence, in this paper, 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 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 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 illustrates this by an example using grids from cutcp, bfs and fft. From Table 2, 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 4.3, we describe several resource-limiting policies using elastic kernels to allow kernels to execute regardless of dispatch order.

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

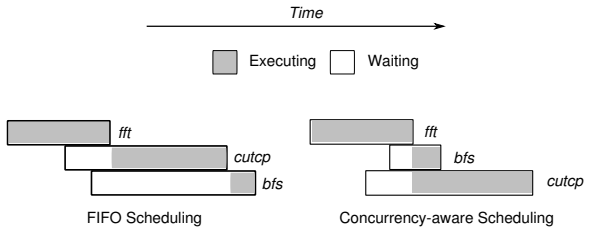

Figure 3. Execution of kernels from bfs, cutcp and fft with a FIFO scheduler and a Concurrency-aware scheduler. Note: Timeline is not to scale.

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 [1]) 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 4.4.

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 4 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.

3.6 Serialization in the CUDA API

The CUDA API contains several functions that *implicitly synchronize* commands from different streams [12], i.e. they act as

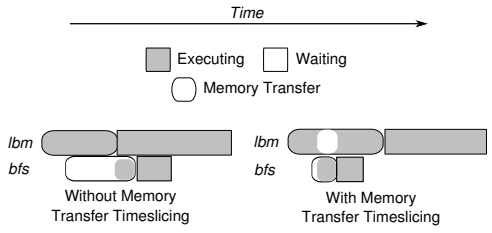

Figure 4. Memory transfers can stall progress across programs, but time-slicing them can reduce waiting times. Note: Timeline is not to scale.

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 [12], 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.

3.7 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" [15] 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" [12].

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 5 describes our NSS implementation.

4. Elastic Kernels

Elastic kernels decouple physical hardware resource allocation for thread blocks from logical program-level grid and thread block identity. In Section 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 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 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 4.4.

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 [17, 18] work which executes CUDA grids efficiently on multicore CPUs. They developed a technique called *iterative wrapping* (similar to *chunked iteration* [16]) 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.

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 original number of blocks and threads.[2] 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.

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 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.

Although the code of Listing 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

[2] This 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.

```
// 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 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.

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.

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. For thread-level resources likes threads and registers, either grid dimensions or thread block dimensions can be modified.

Given resource constraints ($Limits$), Algorithm 1 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 [11], 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 ($ChangeInThreads$). 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.

Algorithm 1 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
1: procedure ReduceBlocks(RLimit, PerBlockUsage)
2:     // Blocks refers to the value in GetPhyGrid
3:     CurUsage ← Blocks * PerBlockUsage
4:     if CurUsage > RLimit then
5:         Deficit ← CurUsage − RLimit
6:         ReduceBlocks ← ⌈Deficit/PerBlockUsage⌉
7:         Blocks ← Blocks − ReduceBlocks
8:     end if
9: end procedure
```

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.

4.3.1 MEDIAN

The MEDIAN policy uses profile-based information to reserve resources for a hypothetical *median* kernel. Using the data in Table 2 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.

4.3.2 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.

4.3.3 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. [1]. 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 2.

4.3.4 QUEUEMOLD

The QUEUEMOLD policy is based on the GetAffinityByMolding algorithm of Ravi et al. [14]. 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.

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.

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 3.7, 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).

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–K* and *K–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–K* and *K–M* dependency checks. However, NSS must enforce these dependencies.

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

To enforce *M–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–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–M* and *K–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 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.

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 [13] 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.

6. Evaluation

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 [12], 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

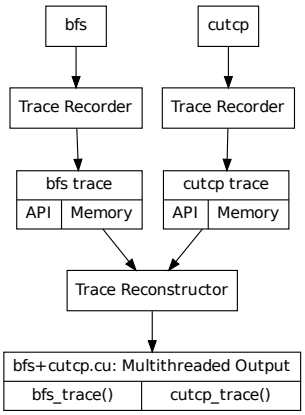

Figure 5. Construction example of a multithreaded concurrent CUDA workload from traces of individual programs bfs and cutcp.

original Parboil2 programs. Figure 5 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. 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.

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 nss replaces the use of CUDA Streams API by our Non-Serializing Streams implementation (Section 5) to work around the false serialization encountered by the cuda variant. Finally, the fourth variant elastic builds on nss and allows the resources assigned to a grid to be varied by the policies of Section 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 4.1 and to provide a fair comparison.

There are a total of 55 two-program workloads, of which we evaluate 54. The bfs+lbm workload was not evaluated because it experiences CUDA launch timeouts. The implementation of 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 bfs is not affected, the execution with 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 [6]. 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 Tullsen's methodology [20] – 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 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 [10] 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 [4] to record API calls on the CPU side.

In our initial experiments, CUDA memory allocation functions limit achievable STP to that of nss (Section 3.6). To mitigate this, we built a custom GPU memory allocator based on the CPU memory allocator `jemalloc` 3.2.0 [5] 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 `cudaMemset` with a custom, non-serializing implementation in all variants.

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 6). 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

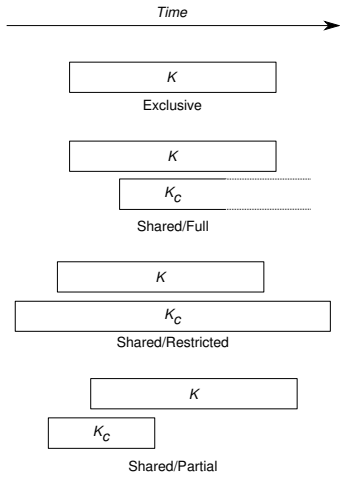

Figure 6. Overlap Categories: Categorization of a kernel instance K based on its overlap with other kernels.

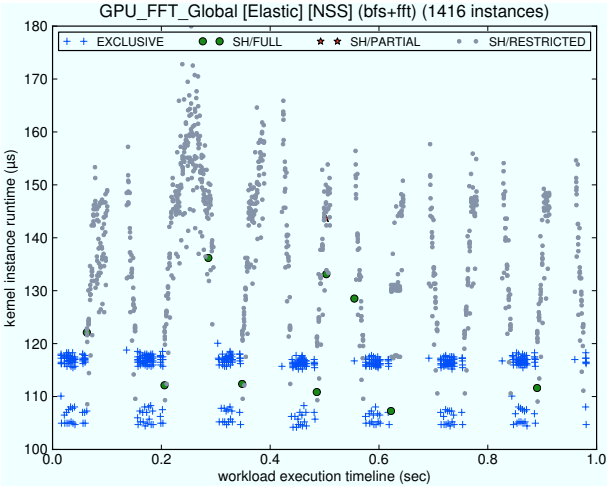

Figure 7. Runtimes of `GPU_FFT_Global` instances from one run of the bfs+fft workload under the MEDIAN elastic policy with each instance categorized into one of the four overlap categories.

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 terminated 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 7 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 because both the kernels of bfs take much longer to complete (see Table 3). A few SHARED/PARTIAL and 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.

Variant(+Policy)	STP		ANTT	
	geomean	best	geomean	worst
cuda	1.03	1.24	6.71	194.93
nss	1.18	1.81	3.05	69.07
elastic+mpmax	1.25	2.08	1.80	19.23
elastic+median	1.24	1.95	1.86	16.83
elastic+equal	1.18	1.83	1.91	30.70
elastic+queuemold	1.18	1.79	2.08	17.94

Table 4. Overall STP and ANTT results for two-program workloads. Note: ANTT is a lower-is-better metric.

Variant(+Policy)	Improvement		MTSpeedup
	STP	ANTT	
cuda	1.00	1.00	1.00
nss	1.15	2.20	1.20
elastic+mpmax	1.21	3.73	1.28
elastic+median	1.20	3.62	1.24
elastic+equal	1.14	3.51	1.15
elastic+queuemold	1.15	3.22	1.17

Table 5. Improvements in STP and ANTT and Multithreaded Speedup (MTSpeedup) for two-program workloads over the cuda variant. For MTSpeedup, a workload's execution time is taken as the maximum execution time of the programs in that workload.

6.4 Results

We evaluate all possible two-programmed workloads (except for bfs+lbm as noted) of the 11 programs from Parboil2 (Table 2) with the elastic policies MPMAX, MEDIAN, EQUAL and QUEUEMOLD.

6.4.1 Overall Results

Table 4 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 5 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 5) 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 polices 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 multithreaded speedup for four-program workloads also increases to

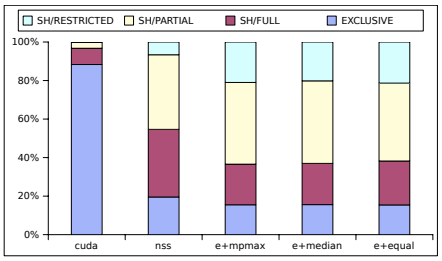

Figure 8. Fraction of execution time of kernels spent in different overlap categories per variant and per policy.

1.47 as compared to cuda. Due to lack of space, we do not elaborate on our results for four-program workloads in this paper.

6.4.2 Effect of Elastic Policies on Kernel Runtime Behaviour

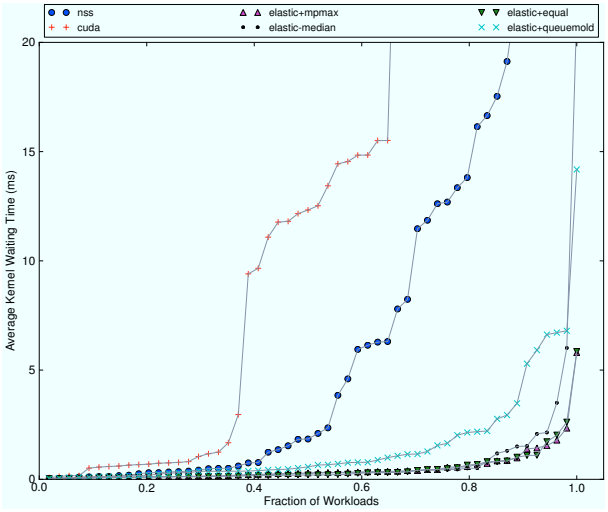

Figure 9. Average Kernel Waiting Time for two-program workloads. Lower is better.

Figure 8 shows the breakup of execution timing across kernel using the categories of Section 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 9 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.

6.4.3 Effect of Elastic Policies on STP and ANTT

Figure 10 shows the distribution of STP values for all workloads. The elastic policy MPMAX has the highest STP values for nearly

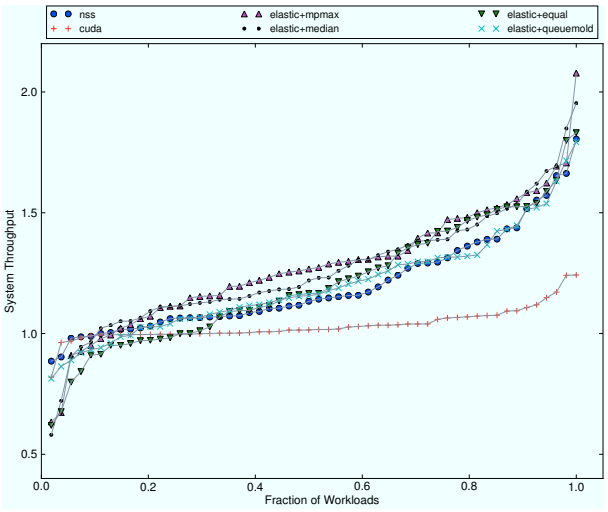

Figure 10. System throughput for each two-program workload, higher is better.

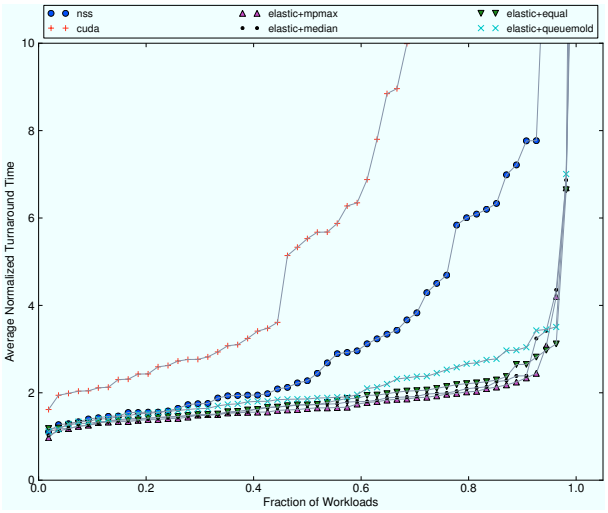

Figure 11. Average Normalized Turnaround Time for each two-program workload, lower is better.

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

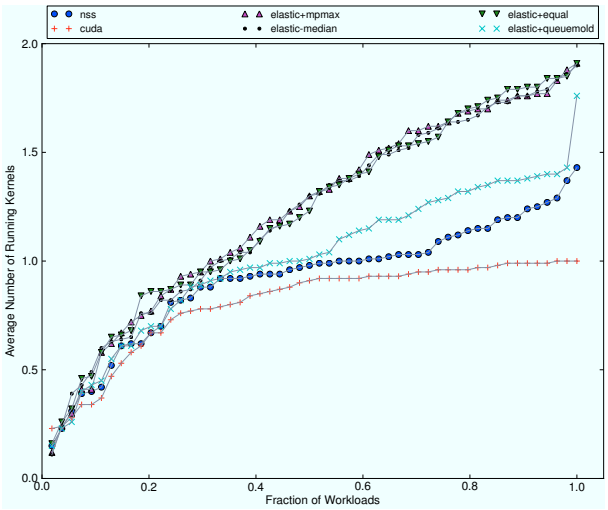

Figure 12. Average number of Running Kernels for each two-program workload.

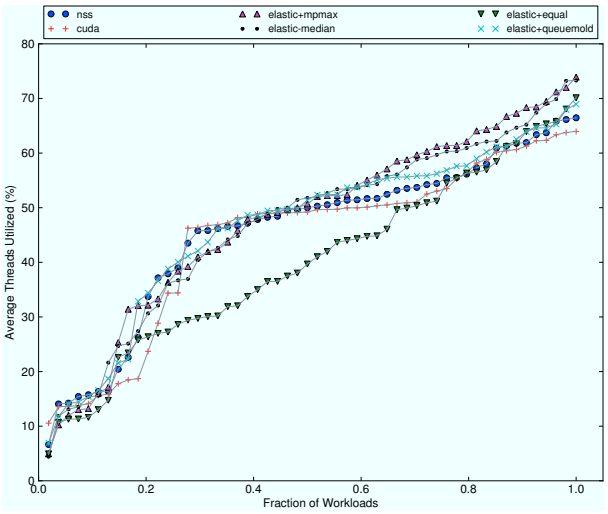

Figure 13. Average utilization of threads for each two-program workload.

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), 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 11 shows the distribution of ANTT values across workloads. The elastic 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 6.4.5 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 9 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.

6.4.4 Effects of Policies on Utilization

Figure 12 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 MPMAX to 1.22 for 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 13) is high even for nss and cuda because these variants do not limit resource usage. However, the 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.

6.4.5 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 MPMAX 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.

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. [8] 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. [7] 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.

G. Wang et al. [21] 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.

L. Wang et al. [22] 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. [1] 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 heuristics and evaluate their policies using the GPGPU-Sim [2] simulator. In this paper, 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 paper and shown that it significantly underutilizes resources compared to our policies.

The work closest to our own is the work by Ravi et al. [14] 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 4.1) to convert any kernel to an elastic kernel. We base our QUEUEMOLD policy on their GetAffinityByMolding resource allocation algorithm and evaluate it in our paper. We find that given the wide disparity in execution times of GPU kernels (Table 3), this policy rarely finds the opportunity to limit resources of kernels and does not perform as well as our policies.

8. Conclusion

In this paper, 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.

Acknowledgments

We thank Sanjiv Satoor and Dibyapran Sanyal of NVIDIA for assistance with the CUDA Profiler. We thank our anonymous reviewers and our shepherd, Rodric Rabbah, for their feedback which has significantly improved this work. We acknowledge partial funding from Microsoft Corporation towards this work.

References

[1] J. Adriaens et al. The case for GPGPU spatial multitasking. In *HPCA*, 2012.

[2] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[3] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[4] M. Desnoyers et al. LTTng-UST User Space Tracer.

[5] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD. In *BSDcan*, 2006.

[6] S. Eyerman and L. Eeckhout. System-level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3), 2008.

[7] C. Gregg et al. Fine-grained resource sharing for concurrent GPGPU kernels. In *HotPar*, 2012.

[8] M. Guevara et al. Enabling task parallelism in the CUDA scheduler. In *Workshop on Programming Models for Emerging Architectures (PMEA)*, 2009.

[9] E. Lindholm et al. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[10] NVIDIA. Compute Command Line Profiler: User Guide.

[11] NVIDIA. CUDA Occupancy Calculator.

[12] NVIDIA. NVIDIA CUDA C Programming Guide (version 4.2).

[13] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110.

[14] V. T. Ravi et al. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *HPDC*, 2011.

[15] S. Rennich. CUDA C/C++ Streams and Concurrency.

[16] J. Shirako et al. Chunking parallel loops in the presence of synchronization. In *ICS*, 2009.

[17] J. A. Stratton et al. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *CGO*, 2010.

[18] J. A. Stratton, S. S. Stone, and W. mei W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *LCPC*, 2008.

[19] TOP500.org. The Top 500.

[20] N. Tuck and D. M. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *PACT*, 2003.

[21] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, 2010.

[22] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *HPCS*, 2011.