Chimera: Collaborative Preemption for Multitasking on a Shared GPU

Jason Jong Kyu Park
University of Michigan
Ann Arbor, MI
jasonjk@umich.edu

Yongjun Park *
Hongik University
Seoul, Korea
yongjun.park@hongik.ac.kr

Scott Mahlke *
University of Michigan
Ann Arbor, MI
mahlke@umich.edu

Abstract

The demand for multitasking on graphics processing units (GPUs) is constantly increasing as they have become one of the default components on modern computer systems along with traditional processors (CPUs). Preemptive multitasking on CPUs has been primarily supported through context switching. However, the same preemption strategy incurs substantial overhead due to the large context in GPUs. The overhead comes in two dimensions: a preempting kernel suffers from a long preemption latency, and the system throughput is wasted during the switch. Without precise control over the large preemption overhead, multitasking on GPUs has little use for applications with strict latency requirements.

In this paper, we propose *Chimera*, a collaborative preemption approach that can precisely control the overhead for multitasking on GPUs. *Chimera* first introduces streaming multiprocessor (SM) flushing, which can instantly preempt an SM by detecting and exploiting idempotent execution. *Chimera* utilizes flushing collaboratively with two previously proposed preemption techniques for GPUs, namely context switching and draining to minimize throughput overhead while achieving a required preemption latency. Evaluations show that *Chimera* violates the deadline for only 0.2% of preemption requests when a 15μs preemption latency constraint is used. For multi-programmed workloads, *Chimera* can improve the average normalized turnaround time by 5.5x, and system throughput by 12.2%.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management - Multitasking; I.3.1 [*Computer Graphics*]: Hardware Architecture - Graphics processors

Keywords Graphics Processing Unit; Preemptive Multitasking; Context Switch; Idempotence

1. Introduction

Modern computer systems are increasingly adopting graphics processing units (GPUs) to aid traditional processors (CPUs). In these heterogeneous systems, one typically offloads throughput-oriented workloads (or kernels) from CPUs to GPUs. GPUs can accelerate highly data-parallel applications effectively with the help of new programming models, such as OpenCL [14] or CUDA [20], which put an emphasis on thread level parallelism. Threads are distributed among hundreds of processing units on a GPU to obtain high throughput.

These systems often have multiple CPUs that share a single GPU. When multiple CPUs offload data-parallel kernels simultaneously onto a shared GPU, multitasking must be supported. Recently, Nvidia's Kepler architecture [21] introduced the Hyper-Q feature to maintain multiple independent kernel queues to concurrently execute independent kernels on a shared GPU. However, this feature is limited to the kernels *within* a single process. Multi-Process Service (MPS) [22] achieves multitasking with a software solution, but is limited to MPI applications. With current generation GPUs, kernels have to wait until a previously running kernel finishes, if multiple processes are trying to share a GPU.

Traditionally, preemptive multitasking on CPUs has been achieved through context switching, which has a reasonable preemption latency and throughput overhead on CPUs. However, supporting preemptive multitasking on GPUs through context switching can incur a higher overhead compared to CPUs, where the context of an SM can be as large as 256kB of register file and 48kB of on-chip scratch-pad memory [1, 24, 29]. Not only does a kernel have to endure a long preemption latency, the GPU also wastes execution resources while context switching. Although Tanasic et al. [29] has shown that the average normalized turnaround time can still be improved with high context switching overhead, such

* Co-corresponding authors.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.

http://dx.doi.org/10.1145/2694344.2694346

overhead wastes the GPU's computing power and may be ineffective for latency-sensitive applications [3, 12, 13].

To overcome these challenges, we propose *Chimera*, a collaborative preemption approach for GPUs that can precisely control the preemption overhead. *Chimera* can achieve a specified preemption latency while minimizing throughput overhead. Since GPUs consist of multiple SMs, a preemption request can have multiple solutions with diverse overheads by preempting different subsets of SMs with different preemption techniques. Given a preemption request, *Chimera* explores the possible solutions to minimize throughput overhead while conforming to the required preemption latency. *Chimera* achieves the goal by intelligently selecting *which* SMs to preempt and *how* each thread block will be preempted.

Chimera first introduces SM flushing, a GPU-specific preemption technique that is enhanced to exploit the semantics of thread blocks in the GPU programming model and the concept of idempotence to achieve low preemption latency. A kernel is *idempotent* if it generates the same result even if it is restarted in the middle of its execution [6, 8, 15, 17]. *Chimera* further relaxes the idempotence condition to enable flushing for more kernels. We say that a thread block is *idempotent at the time of preemption* if it produces the same result up to preemption point even if it is restarted from the beginning. Thus, the context of a thread block can be safely dropped with the relaxed idempotence condition even if the kernel is non-idempotent. Because non-idempotent execution regions tend to be clustered at the end of execution in GPU kernels, relaxed idempotence is effective for increasing the opportunities for flushing.

With flushing, *Chimera* has three preemption techniques in its toolbox: context switching, draining, and flushing. Context switching [17, 29] stores the context of currently running thread blocks, and preempts an SM with a new kernel. Draining [12, 29] stops issuing new thread blocks to the SM and waits until the SM finishes its currently running thread blocks. Flushing drops the execution of running thread blocks and preempts the SM almost instantly.

These three preemption techniques exhibit different tradeoffs between preemption latency and throughput overhead. Context switching has an almost constant mid-range preemption latency and throughput overhead. Draining has the least throughput overhead, but preemption latency can be long if preemption happens near the beginning of thread block execution. Flushing has almost zero preemption latency, but throughput overhead can be large if preemption occurs near the end of thread block execution.

Chimera recognizes the different tradeoffs of these three preemption techniques and chooses which SMs to preempt and how each thread block will be preempted. *Chimera* estimates the costs of the three preemption techniques for the candidate SMs, and intelligently selects SMs and corresponding preemption technique by comparing the costs.

This paper makes the following contributions:

- We introduce SM flushing, a GPU-specific adaptation of a classic preemption technique that can instantly preempt an SM. We combine the concept of idempotence with the semantics of thread blocks in the GPU programming model to enable efficient SM flushing.
- We show that relaxing the idempotence condition is essential for SM flushing to achieve its promised preemption latency. Detecting the relaxed idempotence condition can be done in software.
- We show that the three available preemption techniques for GPUs, namely context switching, draining, and flushing, make different tradeoffs between preemption latency and throughput overhead. Moreover, these tradeoffs change as a thread block makes execution progress on an SM.
- We propose *Chimera*, a collaborative preemption approach for a shared GPU that achieves a specified preemption latency while minimizing throughput overhead. *Chimera* recognizes tradeoffs of available preemption techniques, and makes an intelligent decision as to which SMs to preempt and how to preempt each thread block.

2. Background and Motivation

This section provides an overview of the GPU programming and execution models, and introduces the three preemption techniques used in *Chimera*. This section also motivates the need for collaboration among the preemption techniques. We will use Nvidia's terminology throughout the paper.

2.1 GPU Programming Model and Execution Model

A GPU program consists of host code, which contains the sequential code sections of the program, and a *kernel*, which has the parallel code sections. The host code is run on the CPU, while the kernel code is offloaded from the CPU to the GPU for acceleration.

The GPU programming model is based on a single instruction multiple thread (SIMT) model to explicitly express the parallelism in the kernel code. In the SIMT model, a programmer only writes a code for the basic unit of execution, called a *thread*. A group of threads called a *thread block* is also specified by the programmer. Within a thread block, the programmer can synchronize threads through an explicit barrier operation. Also, threads within a thread block can access a common, fast, on-chip scratch-pad memory called *shared memory*. The entire kernel is executed by launching *grids* of thread blocks to the GPUs.

Figure 1 illustrates a GPU architecture and its execution model. The top left box shows a kernel with the notion of a GPU programming model. The bottom box depicts the GPU architecture with a memory hierarchy. The top right box represents a GPU execution model with the contexts of running thread blocks. In a GPU, each streaming multiprocessor

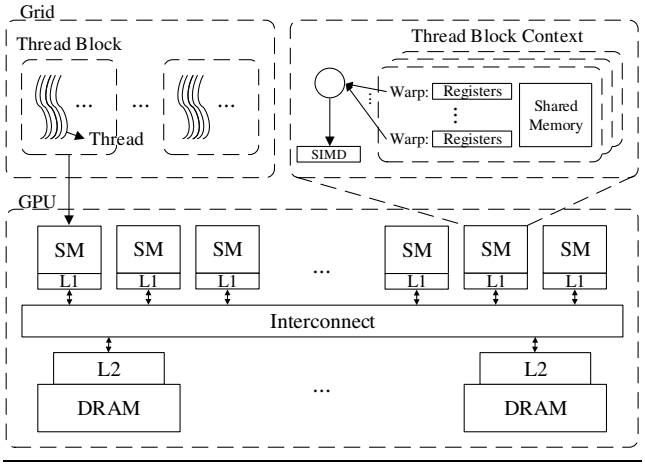

Figure 1. GPU architecture and execution model.

(SM) has a private L1 data cache, a read-only texture cache, and a read-only constant cache. The memory subsystem of the GPU consists of multiple memory partitions. Each memory partition contains a shared L2 cache bank and a memory controller.

The GPU execution model relies on the notion of a thread block. Thread blocks from a kernel can run in arbitrary order because thread block executions are independent from each other. When a kernel is launched, each thread block is scheduled to one of the SMs. Depending on the resource constraints, the number of thread blocks that can run simultaneously on an SM may vary. Also, current generation GPUs only allow thread blocks from the same kernel to be executed on the same SM. When a thread block is dispatched to an SM, the thread block is split into groups of 32 threads called *warps*, where threads within a warp operate on a single common instruction. Each warp has its own register contents, and shares the state of a shared memory if they are in the same thread block.

SMs do not share any states among themselves. Spatial multitasking [1] exploits this property to allow GPUs to run multiple kernels on different subsets of SMs. Preemptive multitasking can also exploit the same property by preempting only a subset of SMs to yield to a new kernel. Starvation can also be avoided by scheduling at least one SM to each available kernel.

2.2 Prior Preemption Techniques

Supporting preemptive multitasking incurs overheads in terms of latency and throughput. For example, context switching for preemption in CPUs involves saving context of the currently running process/thread, running the operating system (OS) scheduler to choose the next process/thread to run, and loading context of the selected process/thread. Preempting a process/thread experiences increased response time due to preemption latency. Also, system throughput is degraded because no progress is made during context switching. The overhead of context switching is proportional to the size of the context.

Modern GPUs can have up to 2048 threads concurrently running on a single SM [21]. Because each thread accesses its own registers, the context size for a SM can grow quickly. Moreover, each SM has its own on-chip scratch-pad memory, which is shared by the threads within a thread block. For modern GPUs, the context of a single SM can be as large as 256kB of register file and 48kB of shared memory [1, 24, 29]. With such a large context, preempting with context switching has high overhead in both preemption latency and wasted throughput.

To avoid throughput overhead of context switching, SM draining [29] has been proposed, which exploits the GPU execution model. Because thread block executions are independent from each other, a thread block does not have to remember its context when it finishes execution. When an SM is preempted with draining, new thread blocks are no longer issued to that SM. When the SM finishes all the running thread blocks, the SM is preempted and can be assigned to another kernel. As the SM is continuously making progress during preemption, throughput overhead of draining is much less than that of context switching.

Draining, however, does not solve the preemption latency problem. Because the preemption latency of draining is dependent on the remaining execution time of thread blocks in the SM, it can be much higher than that of context switching.

2.3 SM Flushing

To enable low preemption latency, we introduce SM flushing, which further exploits the independence of thread block execution in the GPU execution model. Flushing drops an execution of a thread block without context saving and re-executes the dropped thread block from the beginning on another SM. Because thread block executions are independent, other thread blocks do not notice whether a thread block has been rerun from the beginning. Flushing reduces the preemption latency to almost zero. However, certain conditions have to be met to ensure the correctness of the thread block execution that was dropped and rerun from the beginning.

A GPU kernel is *idempotent* if it produces the same result regardless of the number of times it is executed [6, 8, 15, 17]. Because there is no interaction between thread blocks, idempotence conditions for a GPU kernel are much simpler than those in general CPU applications. To be idempotent, a kernel should not have any 1) atomic operations, and 2) overwrites to a global memory location that is read in the kernel. In the benchmarks we studied, 12 out of 27 kernels were found to be idempotent. The idempotence conditions are listed in Table 2. Without enabling flushing in all the kernels, flushing loses its effectiveness because it cannot preempt non-idempotent kernels. We discuss the details of relaxing the idempotence conditions, and implementation of flushing in Section 3.4.

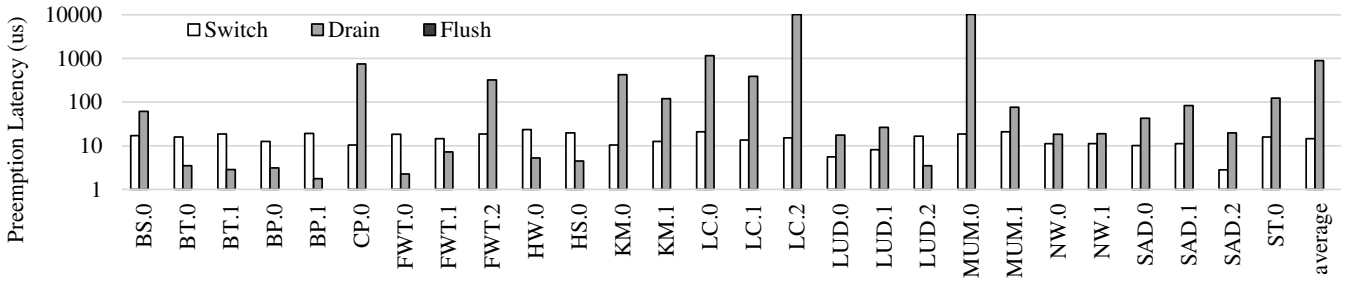

Figure 2. Estimated preemption latency for each preemption technique. For draining, a uniform random distribution on the preemption point across thread block execution is assumed. For flushing, zero preemption latency is assumed.

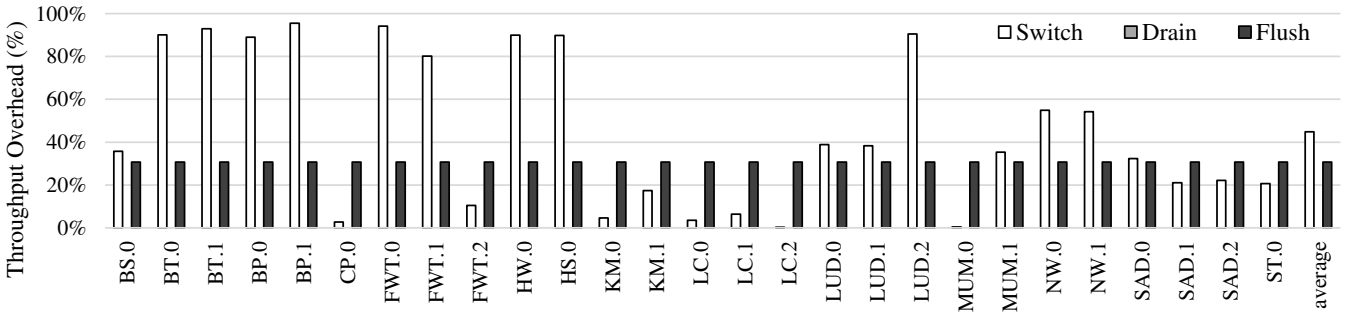

Figure 3. Estimated throughput overhead for each preemption technique when thread blocks running on an SM are assumed to be in sync. For flushing, a uniform random distribution on the preemption point across thread block execution is assumed.

2.4 Tradeoff

Context switching, draining, and flushing make different tradeoffs between preemption latency and throughput overhead. Figure 2 shows the estimated preemption latency for each preemption technique. In the figure, the y-axis shows preemption latency on a logarithmic scale, and the x-axis shows the kernels in the benchmarks. If multiple kernels are launched in a benchmark, they are differentiated with numbers after the benchmark name and a dot. All the labels and numbers for benchmarks and kernels are listed in Table 2. To estimate the preemption latency of context switching, an SM is assumed to have only its share of global memory bandwidth to save its context. Context size can be calculated from the kernel's resource usage even before the kernel launch. The same method was used in [29] to project the estimated preemption latency for context switching. To estimate the preemption latency for draining, the average time to execute a thread block is first measured through simulation. Assuming uniform random distribution on the preemption point across the execution of a thread block, the preemption latency for draining can be calculated. The preemption latency for flushing is assumed to be zero.

In Figure 2, context switching shows a relatively constant response time in the order of 10 μs, while draining exhibits diverse response time ranging from 0.8 μs to 10212.8 μs. On average, context switching, draining, and flushing require 14.5 μs, 830.4 μs, and 0 μs, respectively, to preempt an SM. They are equivalent to an order of 10,000, 500,000, and 0 cycles, respectively, in current generation GPUs.

Figure 3 shows the estimated throughput overhead for each preemption technique. In the figure, the y-axis shows the percentage of throughput overhead for each preemption technique compared to the throughput without preemption. Thread blocks running on an SM are assumed to be in sync. The throughput overhead of context switching is twice the preemption latency divided by the thread block execution time, where the preemption latency is doubled because throughput overhead comes both from context saving and context loading. SM draining is assumed to have zero throughput overhead because it continuously does useful work until the thread block finishes. In reality, thread blocks can be out of sync, which will cause draining to incur some throughput overhead. To estimate the throughput overhead of flushing, a uniform random distribution on the preemption point across the execution of a thread block is again assumed. The throughput overhead of flushing is independent of the kernel, and is constant across all the benchmarks. Overall, context switching, draining, and flushing have throughput overhead of 47.7%, 0%, and 30.7%, respectively.

2.5 Motivation

Different tradeoffs from the three preemption techniques encourage using different preemption techniques for each kernel. In fact, different tradeoffs can be further exploited by

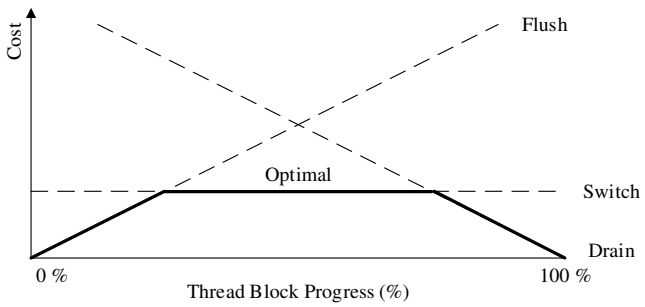

Figure 4. Theoretical cost of each preemption technique when preempting a thread block at a given amount of execution progress. Context switching has constant cost across the execution, draining has lower cost as a thread block is near the end of execution, and flushing has lower cost as a thread block is closer to the beginning of execution.

using different preemption techniques within one preemption request. Because a preemption request would typically want multiple SMs at the same time, each SM can be preempted with a different preemption technique. Moreover, each thread block in the SM can be preempted with a different preemption technique.

Figure 4 depicts the theoretical cost of each preemption technique if a thread block at given progress is preempted. The cost can be thought of as an aggregate measure of preemption latency and throughput overhead. The cost of context switching is dependent on the context size and the available bandwidth for an SM, which is almost constant across thread block execution. The cost of draining, which is primarily preemption latency, is dependent on the remaining execution time of a thread block. It decreases toward the end of the thread block progress. The cost of flushing, on the other hand, is primarily throughput overhead, which is dependent on the work thrown away by flushing. More work is thrown away as the thread block progresses; hence, the cost increases accordingly.

The figure motivates to preempt with flushing if a thread block is in the early stage of execution, with context switching if a thread block is in the middle stage of execution, and with draining if a thread block is near the end of execution. The exact points at which to switch the preemption decision is based on the cost estimation of each preemption technique.

3. Architecture

Chimera is a collaborative preemption with three individual techniques: context switching, draining, and flushing. Context switching is implemented with a software trap routine. Draining is performed by adding logic in a thread block scheduler that stops issuing new thread blocks. Flushing requires reset logic in SMs, which clears all the states and in-flight instructions in the SM. For context switching and flushing, an SM has to send the stopped thread blocks' IDs

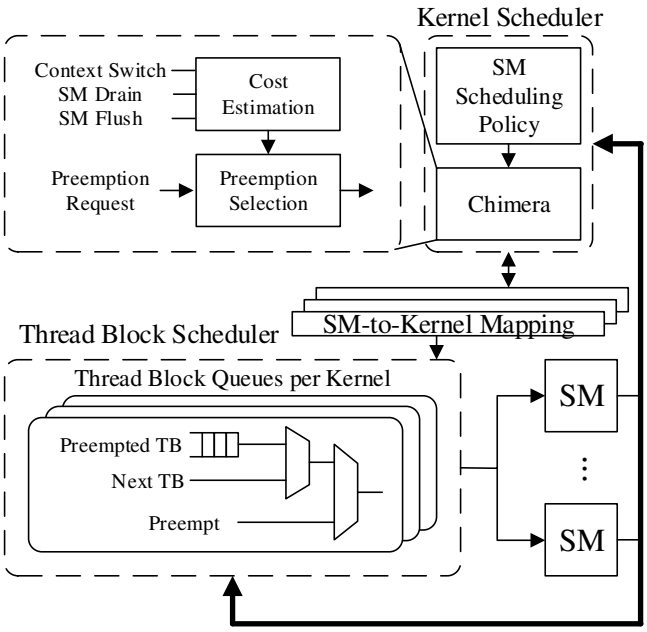

Figure 5. GPU scheduler with preemptive multitasking. The scheduler is two-level: the kernel scheduler assigns SMs to each kernel that may involve preemption decisions, and the thread block scheduler executes the decision by dispatching or preempting thread blocks from each SM. SMs can feedback the schedulers when an event that can change the scheduling decision occurs.

back to the thread block scheduler so that they can be reissued to the other SMs.

Chimera decides which SMs to preempt and which preemption technique to use for each thread block in the SMs, given the number of SMs to preempt. *Chimera* makes the decision based on the upper bound for preemption latency given by the preempting application or kernel. *Chimera* first estimates preemption latency and throughput overhead for each thread block in an SM when it is preempted with each preemption technique. *Chimera* chooses preemption techniques with the least throughput overhead that satisfy the given preemption latency for an SM, which can give the total cost of preemption for each SM. With the calculated costs, *Chimera* selects SMs which can minimize throughput overhead while meeting the required preemption latency.

3.1 GPU Scheduler with Preemptive Multitasking

Figure 5 illustrates the GPU scheduler with preemptive multitasking when multiple GPU kernels are running concurrently. The scheduler is a two-level scheduler: the kernel scheduler assigns kernel to each SM, which may involve preemption decisions, and the thread block scheduler carries out the decision. The thread block scheduler dispatches a new thread block to an SM, or preempts an SM with the given preemption techniques based on the decisions from the kernel scheduler. The kernel scheduler is a part of an operating

system that manages a GPU device, while the thread block scheduler is a hardware module in a GPU, which is an extension of GigaThread engine in Fermi [19] with a preemption support.

An SM partitioning policy in the kernel scheduler tells how many SMs each kernel will run on. The policy is orthogonal to the preemption decisions. It may be dependent on a characteristic of a kernel [1] or a priority of a kernel [29]. *Chimera* in the kernel scheduler achieves an SM partitioning policy by making preemption decisions. The kernel scheduler communicates to the thread block scheduler through SM-to-kernel mapping information which contains per-SM information about which kernel to schedule, whether preemption is necessary or not, and which preemption technique to use. The thread block scheduler always prefers to schedule the preempted thread blocks first so that the size of the preempted thread block queue can be limited.

Chimera consists of two parts: estimating costs of preemption for each technique, and selecting SMs to preempt with corresponding preempting techniques. To estimate the costs, *Chimera* gathers statistics for SMs. The statistics are measured using hardware and reported directly to *Chimera*. Based on these statistics, *Chimera* estimates preemption latency in cycles, and throughput in the number of instructions rather than IPC. *Chimera* can directly compare the estimated cost of each preemption technique because they are calculated in the same units.

3.2 Cost Estimation

To distinguish different tradeoffs between different preemption techniques, *Chimera* has to estimate the cost of each preemption technique precisely for each SM. First, *Chimera* measures the total number of executed instructions for each thread block to determine the progress of each thread block. Note that instructions are counted not in thread granularity, but in warp granularity so that control divergence in a warp has minimal effect on the total executed instructions. Second, *Chimera* also measures the progress of each thread block in cycles. *Chimera* can calculate the average instructions-per-cycle (IPC) or cycles-per-instruction (CPI) of a thread block with these two statistics.

We estimate the preemption latency of context switching using the same method as detailed in Section 2.4. To estimate the throughput overhead of context switching, the average IPC of preempted kernel on a single SM is multiplied by twice the preemption latency of context switching. Note that preemption latency is doubled because throughput overhead not only comes from context saving, but also from context loading. The preemption latency of draining is estimated by multiplying the remaining instructions to execute in a thread block by the average CPI of the preempted kernel. We avoid using the average execution cycles per thread block directly because it has much larger variance compared to the average executed instructions, leading to less accurate estimations. The throughput overhead of draining is estimated by summing the difference between the number of executed instructions for each thread block and the maximum number of executed instructions among them. Flushing is always assumed to have zero preemption latency. The total number of executed instructions for thread blocks in the SM is used to estimate the throughput overhead of flushing. When the cost cannot be estimated due to the lack of gathered statistics, we conservatively use the maximum value as the estimated cost to avoid selecting affected techniques.

Algorithm 1 Preemption Selection

```
    Input: LatLimit, Kernel, NumPreempts ▷ From SM Scheduling Policy
    Output: SM_Preemptions[1..NumPreempts]
 1: for each SM in Kernel do
 2:     for each TB in the SM do
 3:         for each Preemption Technique do
 4:             TBCosts.push(EstimateCost(TB, Technique))
 5:         end for
 6:     end for
 7:     TBSorted = SortByThroughputOverhead(TBCosts)
 8:     while !TBSorted.empty() do
 9:         TBCandidate = TBSorted.pop()
10:         if meets_latency(TBCandidate) and TBCandidate.TB not in
    SingleSMCost then
11:             SingleSMCost.add(TBCandidate)
12:         end if
13:     end while
14:     for each TB not in SingleSMCost do
15:         SingleSMCost.add(EstimateCost(TB, Switch))
16:     end for
17:     SMCosts.push(SingleSMCost)
18: end for
19: SMSorted = SortByThroughputOverhead(SMCosts)
20: for i = 1 to NumPreempts do
21:     while !SMSorted.empty() do
22:         SMCandidate = SMSorted.pop()
23:         if meets_latency(SMCandidate) then
24:             SM_Preemptions[i] = SMCandidate
25:             break
26:         end if
27:     end while
28: end for
29: return SM_Preemptions[1..NumPreempts]
```

3.3 Preemption Selection

Chimera is a collaborative preemption that achieves low overhead multitasking through multiple preemption techniques with different overheads. SM scheduling policy, which is independent of these decisions, provides *Chimera* a preemption latency constraint, a kernel to preempt, and the number of SMs to preempt. Given the inputs, *Chimera* generates combinations of which SM to preempt and how to preempt, while satisfying the latency constraint.

Algorithm 1 illustrates how *Chimera* selects a subset of SMs and techniques to preempt. The algorithm starts by estimating the cost of each preemption technique for the thread blocks in an SM (line 2-6). The costs for the thread blocks are sorted by throughput overhead (at line 7), and a preemption technique for a particular thread block is selected if the preemption latency constraint is met and it is not al-

ready selected with another preemption technique (line 8-13). If a thread block cannot meet the constraint with any preemption technique, *Chimera* performs context switching for the thread block (line 14-16). Now, *Chimera* knows the preemption costs for each SM that is running the given kernel. It sorts all the costs by throughput overhead (at line 19). From the list of sorted candidates, *Chimera* finalizes the preemption selection (line 20-28). When finalizing the decision, *Chimera* checks whether the candidate satisfies the preemption latency constraint (at line 23). Because only one candidate exists for an SM, *Chimera* does not have to check whether the candidate SM is already selected.

The time complexity of algorithm 1 is $O(NTlogT + NlogN)$, where N is the number of SMs that a kernel to preempt is occupying, and T is the number of available thread blocks in an SM. The first term comes from the first loop (line 1-18), where preemption techniques are selected for particular thread blocks. Two loops (line 2-6 and line 8-13) take the linear time complexity of $O(PT)$, where P is the number of preemption techniques. In *Chimera*, P is a maximum of 3. The third loop (line 14-16) only takes the linear time complexity of $O(T)$. Therefore, sorting (at line 7) defines the time complexity with $O(PTlogPT) = O(TlogT)$. Since the outer loop runs N times, the entire loop (line 1-18) takes $O(NTlogT)$. The second term, which is $O(NlogN)$, comes from sorting for SMs (at line 19). The last loop (line 20-28) only takes the linear time complexity of $O(N)$. In general, N is in the order of 10 for current GPU generations. Furthermore, N will be reduced as more kernels run concurrently on the GPU because each kernel is likely to occupy a lower number of SMs. Also, T is a fixed number (maximum of 16 in Kepler [21]), but is typically less than the maximum (8 is the largest number of thread blocks per SM for simulated benchmarks in Table 2). Thus, the impact of the selection algorithm in *Chimera* is negligible to the preemption latency.

3.4 SM Flushing

SM flushing can be effective if it can preempt all kernels, whether they are idempotent or not. Flushing may still violate the required preemption latency if it cannot preempt an SM due to non-idempotence. Implementation of flushing is fairly straightforward as an SM already has a circuit that resets or clears itself.

We relax the idempotence condition by looking at thread blocks individually with the notion of time. A GPU thread block is *idempotent at a given time* if it neither 1) has executed any atomic operations yet, nor 2) has overwritten a global memory location that is read by the thread block. Because atomic operations or global memory overwrites tend to be performed at the end of a thread block execution, a thread block can remain idempotent for the most of its execution time even if the kernel itself is non-idempotent.

With the relaxed idempotence condition, SMs have to notify the GPU scheduler when thread blocks have progressed beyond the non-idempotent point. The notification is implemented in software by inserting a store instruction in front of atomic operations or global overwrite operations. The store is made to a pre-defined address, which is non-cacheable. SMs will prepend their ID to the store so that the store address is unique for each SM. As SMs are in-order cores, these inserted stores are guaranteed to take place before the atomic or global overwrite operations. The GPU scheduler looks at these pre-defined addresses to check whether each SM can be preempted with flushing.

As atomic operations are separate hardware instructions, they are trivial to find. Global overwrite operations are found by compiler analysis to distinguish between global writes and global overwrites. While pointer alias analysis is undecidable [10] in general, pointers are used in a more restricted fashion in GPU kernels, which allows the compiler to find global overwrites precisely in most cases.

System	Parameters
SM	30 SMs, 1400 MHz, 8 SIMT width 32768 registers per SM 8 maximum thread blocks per SM 48 kB shared memory
Memory Subsystem	6 memory partitions 177.4 GB/s bandwidth

Table 1. System configuration

4. Results

We use the GPGPU-Sim v3.2.2 [2] to evaluate *Chimera*. GPGPU-Sim only models the GPU, while the host code runs natively on CPUs. GPGPU-Sim does not model the overhead of data transfers between the CPU and GPU as well. We model a Fermi [19] architecture with 30 SMs. All the system configuration parameters are summarized in Table 1. We implement context switching by halting an SM for the estimated context switch time instead of using software trap routine. The result for context switching will be rather optimistic in the sense that the memory bandwidth consumed by context switching will affect other SMs to slow down in reality and vice versa, whereas our implementation does not account for the effect.

For all the preemption techniques, we use the same SM scheduling policy, which is similar to the mix of **Smart Even** and **Rounds** in spatial multitasking [1]. SMs are distributed evenly across the kernels except when the kernel requires less SMs than the even split. A kernel can request less SMs than the even split for two reasons: if a kernel is size-bound, where the grid size or the number of thread blocks for the kernel cannot fully occupy its portion of SMs at its launch, or if the remaining number of thread blocks is insufficient to fully occupy the given number of SMs near the end of execution.

We use a wide range of GPGPU applications from Nvidia Computing SDK [18], Rodinia [4], and Parboil [28] bench-

Benchmark (Label) [Input]	Source	Kernel (Label)	Average Drain Time	Context /TB	TBs /SM	Switching Time	Idempotent
BlackScholes (BS) [4M Options]	Nvidia SDK [18]	BlackScholesGPU (0)	60.9 μs	24 kB	4	17.0 μs	Yes
B+ Tree (BT) [1M Nodes]	Rodinia [4]	findRangeK (0)	3.5 μs	46 kB	2	15.9 μs	No
		findK (1)	2.8 μs	36 kB	3	18.7 μs	No
Back Propagation (BP) [128K Nodes]	Rodinia [4]	bpnn_layerforward (0)	3.1 μs	12 kB	6	12.5 μs	No
		bpnn_adjust_weights (1)	1.8 μs	22 kB	5	19.0 μs	No
Coulombic Potential (CP) [2K Atoms on 256x256 Grid]	Parboil [28]	cenergy (0)	746.9 μs	7 kB	8	10.4 μs	No
Fast Walsh Transform (FWT) [8M]	Nvidia SDK [18]	fwtBatch2Kernel (0)	2.3 μs	21 kB	5	18.2 μs	No
		fwtBatch1Kernel (1)	7.2 μs	28 kB	3	14.5 μs	No
		modulateKernel (2)	321.8 μs	18 kB	6	18.7 μs	No
Heart Wall Tracking (HW) [656x744 Pixels/Frame]	Rodinia [4]	kernel (0)	5.2 μs	67 kB	2	23.4 μs	No
HotSpot (HS) [1024x1024 Data Points]	Rodinia [4]	calculate_temp (0)	4.5 μs	38 kB	3	19.7 μs	Yes
Kmeans (KM) [0.5M Data Points, 34 Features]	Rodinia [4]	invert_mapping (0)	424.3 μs	10 kB	6	10.4 μs	Yes
		kmeansPoint (1)	118.8 μs	12 kB	6	12.5 μs	Yes
Leukocyte Tracking (LC) [640x480 Pixels/Frame]	Rodinia [4]	GICOV_kernel (0)	1162.0 μs	17 kB	7	20.9 μs	Yes
		dilate_kernel (1)	391.7 μs	9 kB	8	13.5 μs	Yes
		IMGVF_kernel (2)	10173.2 μs	87 kB	1	15.2 μs	No
LU Decomposition (LUD) [512x512 Data Points]	Rodinia [4]	lud_diagonal (0)	17.4 μs	4 kB	8	5.6 μs	No
		lud_perimeter (1)	26.2 μs	5 kB	8	8.1 μs	No
		lud_internal (2)	3.5 μs	16 kB	6	16.6 μs	No
MUMmer (MUM) [50000 25-character Queries]	Rodinia [4]	mummergpuKernel (0)	10212.8 μs	18 kB	6	18.7 μs	Yes
		printKernel (1)	76.4 μs	24 kB	5	20.8 μs	Yes
Needleman-Wunsch (NW) [4096x4096 Data Points]	Rodinia [4]	needle_cuda_shared_1 (0)	18.2 μs	8 kB	8	11.1 μs	No
		needle_cuda_shared_2 (1)	18.7 μs	8 kB	8	11.1 μs	No
SAD [1920x1072 Pixels]	Parboil [28]	mb_sad_calc (0)	42.3 μs	7 kB	8	10.1 μs	Yes
		larger_sad_calc_8 (1)	82.9 μs	8 kB	8	11.1 μs	Yes
		larger_sad_calc_16 (2)	19.7 μs	2 kB	8	2.8 μs	Yes
Stencil (ST) [512x512x64 Grid]	Parboil [28]	block2D_hybrid_coarsen_x (0)	122.3 μs	11 kB	8	15.9 μs	Yes

Table 2. Benchmark Specification

mark suite. Table 2 lists all the evaluated benchmarks, their inputs, and their kernels with their characteristics. We show the estimated average drain time, the size of the context for one thread block, the maximal number of concurrent thread blocks per SM, the estimated context switch time, and idempotence of the kernel. Note that kernel idempotence can be relaxed in flushing. We selected a subset of benchmarks from each benchmark suite so that they show diverse characteristics in terms of the context switching time, draining time, and the idempotence condition.

4.1 Periodic Task with Deadline

We first analyze each GPGPU benchmark, when it is concurrently run with a synthetic GPU benchmark, which mimics a periodic, real-time task that has a hard deadline. The synthetic GPU benchmark is launched every 1ms, preempting half of the SMs, and executed for 200μs. The deadline for the synthetic benchmark is the execution time plus the required preemption latency. The synthetic benchmark is killed if it misses the deadline. The simulation is run until a GPGPU benchmark executes 1 billion instructions or finishes its execution.

Figure 6 illustrates the percentage of preemptions that violate the deadline of the synthetic benchmark when the preemption latency constraint is set to 15μs. On average, context switching, draining, flushing, and *Chimera* miss the deadline for 56.0%, 61.3%, 7.3%, and 0.2% of preemptions, respectively. Flushing, despite its zero preemption latency, violates the deadline for BT and FWT because these benchmarks have non-idempotent kernels with short thread block execution time. In such cases, flushing is more likely to miss the deadline because thread blocks have higher chances to be in a non-idempotent region even with the relaxed idempotence condition. On the other hand, *Chimera* misses the dealine in 0.2% cases only. These misses are primarily due to the incorrect estimation of draining latency. However, the error is in the range of few hundred cycles ($< 1\mu$s), and can be avoided by providing a headroom to preemption latency constraint against the deadline.

Figure 7 shows the overhead on throughput for each preemption technique in the same scenario. If the deadline of the synthetic benchmark is missed, we ignore the throughput additionally gained by running GPGPU benchmark more during that period so that the measured overhead is fair among the preemption techniques. Also, we neglect the throughput of the synthetic benchmark on purpose, to isolate throughput overhead of each preemption technique. Overall, context switching, draining, flushing, and *Chimera* have

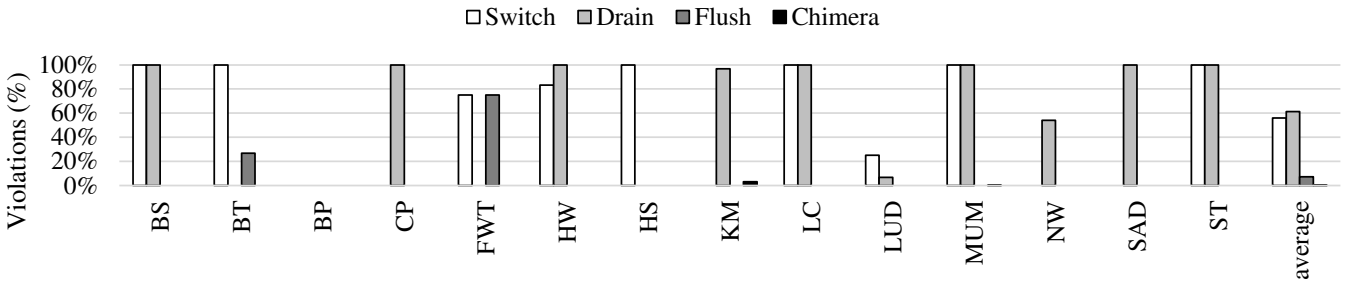

Figure 6. The percentage of preemptions that violate the deadline of a periodic, real-time task when GPGPU benchmarks are run together. The preemption latency constraint is 15 μs.

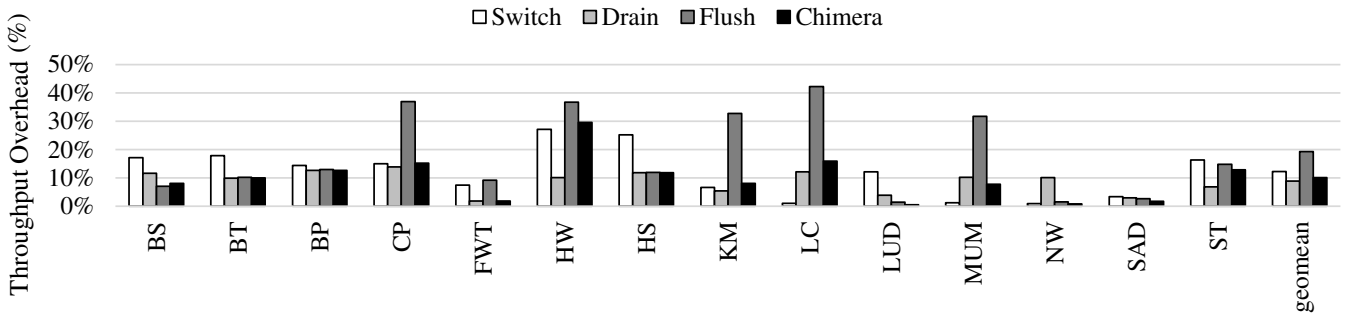

Figure 7. Throughput overhead of each preemption technique when GPGPU benchmarks are run with a periodic, real-time task. The preemption latency constraint is 15 μs. Effective throughput is used to avoid giving unfair advantage to the preemption techniques that frequently miss the deadline.

throughput overhead of 12.2%, 8.9%, 19.3%, and 10.1%, respectively. Draining does not have zero throughput overhead as explained in Section 2.4 because the assumption that thread blocks are running in sync is not true in practice. However, it still achieves the minimum throughput overhead compared to switching, and flushing. *Chimera* shows similar throughput overhead with significantly fewer deadline misses. FWT, LUD, and NW show least throughput overhead for *Chimera* because they include kernels that either have short execution time (SMs will be quickly freed) or occupy less than half of all available SMs from the beginning. In such case, preemption can be performed with low overhead since a new kernel can be launched to the idle SMs.

As shown by the figures, *Chimera* can almost always meet the preemption latency constraint while individual preemption technique cannot. *Chimera* achieves this goal while maintaining the low throughput overhead of draining. In fact, *Chimera* can have lower throughput overhead than individual preemption technique as in LUD by collaboratively utilizing all the techniques.

4.2 Impact of Preemption Latency Constraint

Chimera is a collaborative preemption with controlled overhead. Figure 8 demonstrates the impact of preemption latency constraint when it is varied from 5μs to 20μs. We use the same multi-programmed workloads as in Section 4.1.

Figure 8 (a) shows the percentage of preemptions that *Chimera* violates the deadline when the preemption latency constraint is varied from 5μs to 20μs. When the preemption latency is 5μs, 10μs, 15μs, and 20μs, violations happen for 2.00%, 1.08%, 0.24%, and 0.00% of preemptions, respectively. As explained with Figure 6, flushing may fail to meet the deadline if a kernel is non-idempotent, and has short thread block execution time. Since flushing is what *Chimera* relies on to achieve low preemption latency, *Chimera* also suffers from the same problem when the preemption latency constraint is extremely low as in the case of 5μs.

Figure 8 (b) shows throughput overhead of *Chimera* when the preemption latency constraint is varied from 5μs to 20μs. Again, we use effective throughput to measure throughput overhead to avoid giving unfair advantage to the preemption techniques that miss the deadline. *Chimera* have 16.5%, 12.2%, 10.0%, and 9.0% throughput overhead when the preemption latency constraint is 5μs, 10μs, 15μs, and 20μs, respectively. As shown in the figure, *Chimera* can reduce more throughput overhead when the preemption latency constraint is increased. If only one preemption technique is utilized, the loose deadline cannot be exploited.

Figure 8 (c) shows distribution of each preemption technique used in *Chimera* when the preemption latency constraint is varied from 5μs to 20μs. As the preemption latency constraint is reduced, *Chimera* exploits flushing more because flushing is the only preemption technique that pro-

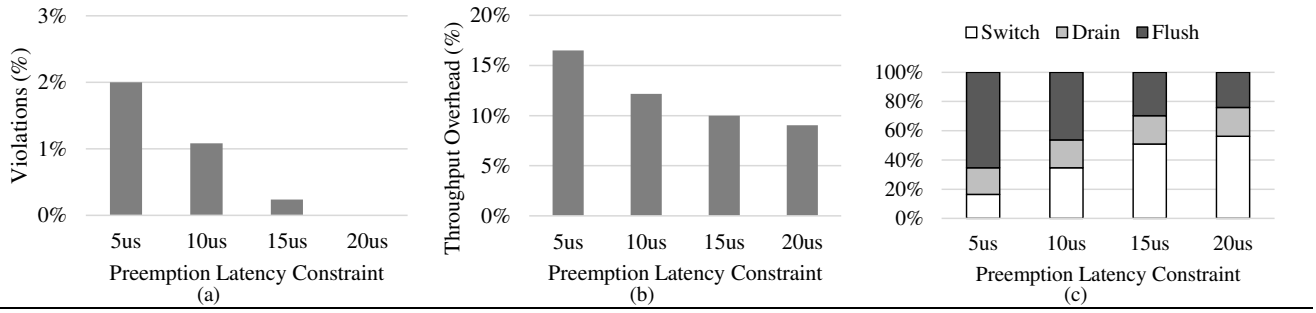

Figure 8. Impact of varying preemption latency constraint on (a) the percentage of deadline violations, (b) throughput overhead, and (c) distribution of each preemption technique used in *Chimera*.

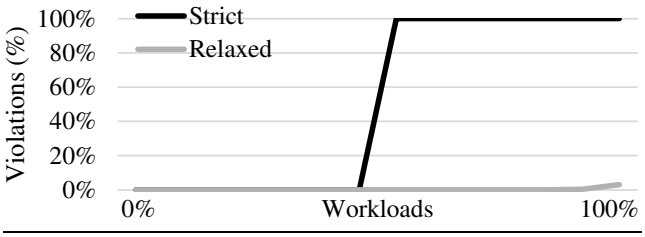

Figure 9. The percentage of preemptions that violate 15μs preemption latency constraint when SM flushing uses strict or relaxed idempotence condition.

vides low preemption latency at the expense of throughput overhead. About 19% of preemptions constantly utilize draining because thread blocks near the end of their execution always exist even for low preemption latency constraint. Context switching has constant preemption latency regardless of the constraint hence its utilization quickly drops as the preemption latency constraint is reduced.

4.3 Relaxed Idempotence Condition in SM Flushing

Figure 9 illustrates the effectiveness of relaxing the idempotence condition for flushing. We refer to the original idempotence condition as strict, and the relaxed idempotence condition as relaxed. The percentage of preemptions that violate 15μs preemption latency constraint is shown for all the workloads used in Section 4.1. On average, flushing violates the deadline for 50.0%, and 0.2% of the total preemptions with strict and relaxed idempotence condition, respectively.

When strict idempotence condition is used, the kernel idempotence decides whether an SM can be preempted with flushing or not. With relaxed idempotence condition, thread blocks in such kernel can still be preempted with flushing if they have not reached non-idempotent region. Without the relaxed idempotence condition, flushing cannot achieve its promised low preemption latency because non-idempotent kernels cannot be preempted. The violations for strict idempotence condition will be the same regardless of the preemption latency constraint. The results show that it is mandatory for flushing to have relaxed idempotence condition to provide instant preemption.

4.4 Case Study

In this section, we further investigate *Chimera* when a combination of GPGPU benchmarks without hard deadline is concurrently executed. Each multi-programmed workload is a combination of LUD with one of the benchmarks in Table 2. LUD is chosen because it launches multiple kernels that require different number of SMs, which results in numerous preemption requests. GPGPU benchmarks do not have hard deadline hence we chose preemption latency constraint to be 30μs, which is the maximum possible preemption latency for context switching in our configuration.

Each simulation runs until *all* benchmarks either execute 1 billion instructions or finish its execution. Among the benchmarks, FWT, HW, KM, LC, MUM, SAD, and ST run more than 1 billion instructions. When one benchmark finishes earlier than the others, it is restarted from the beginning to prohibit the last remaining benchmark from running on its own. The reported results are gathered only for the first 1 billion instructions or first execution whichever is reached first. All the benchmarks are started simultaneously at the beginning. This is a typical setting for simulating multi-programmed workloads [1, 25, 29–31]. For the baseline, we use non-preemptive scheduling, where each kernel has to wait until previous kernel has finished its execution. Kernels are launched following first-come, first-serve (FCFS) policy.

To compare the performance of preemption techniques, we use the metrics suggested by Eyerman et al. [7]. Average normalized turnaround time (ANTT) quantifies the user-perceived slowdown due to multitasking using Equation 1, where N denotes the number of kernels, CPI_i^{multi} is the CPI when a kernel is executed in the multi-programmed workload, and CPI_i^{single} is the CPI when the kernel is executed on its own. System throughput (STP) measures the progress of the system under multitasking using Equation 2, where parameters are the same as in ANTT.

$$ANTT = \frac{1}{N}\sum_{i=1}^{N}\frac{CPI_i^{multi}}{CPI_i^{single}} \quad (1)$$

$$STP = \sum_{i=1}^{N}\frac{CPI_i^{single}}{CPI_i^{multi}} \quad (2)$$

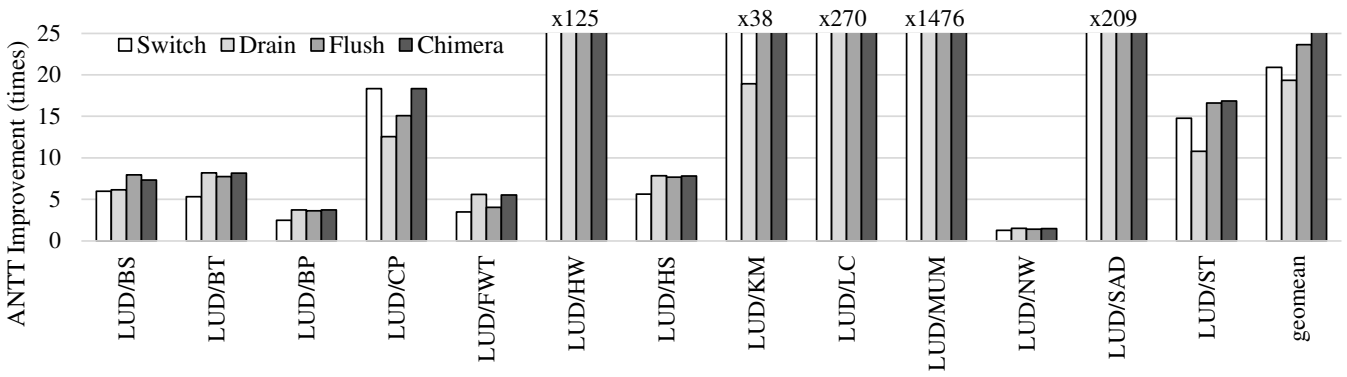

Figure 10. ANTT improvement over the non-preemptive FCFS when LUD is concurrently executed with another benchmark.

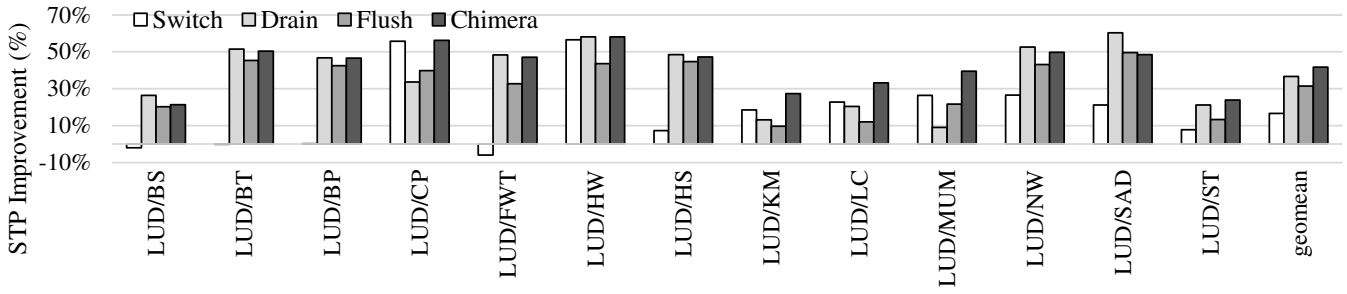

Figure 11. STP improvement over the non-preemptive FCFS when LUD is concurrently executed with another benchmark.

Figure 10 presents the ANTT improvement of each preemption technique over the non-preemptive FCFS. On average, context switch, draining, flushing, and *Chimera* improves the ANTT by 20.9x, 19.3x, 23.6x, and 25.4x, respectively. HW, KM, LC, MUM, and SAD have a kernel, whose execution time is extremely long. Preemptive multitasking improves ANTT drastically over non-preemptive FCFS because non-preemptive FCFS has to wait until these kernels finish their execution. Among single preemption techniques, flushing has the most ANTT improvement as it has the least preemption latency. *Chimera* can improve the ANTT more than flushing because it can preempt non-idempotent thread blocks using other preemption techniques.

Figure 11 shows the STP improvement of each preemption technique over the non-preemptive FCFS. Overall, context switch, draining, flushing, and *Chimera* improves the STP by 16.5%, 36.6%, 31.4%, and 41.7%, respectively. Since LUD does not occupy all the available SMs, STP is significantly improved despite the throughput overhead of preemption techniques. Here, spatial multitasking is effectively improving STP. In LUD/SAD, *Chimera* improves STP much less than the top single preemption technique, which is draining. *Chimera* shows such behavior when the cost estimation is not accurate enough for context switching, and draining. Cost estimation is not precise if thread blocks have large variations in the execution time or in the CPI. Again, *Chimera* achieves the most STP improvement over any single preemption technique because *Chimera* can choose alternative low overhead preemption technique if one preemption technique incurs larger overhead compared to the others while a single preemption technique is forced to endure the overhead. We see larger difference in STP compared to throughput overhead in the Section 4.1 because preemptions occur more frequently than 1ms interval in the simulated multi-programmed workloads.

Chimera can improve ANTT and STP for GPGPU benchmarks without hard deadlines as it utilizes adequate preemption technique when preemption request occurs. When all the combinations of GPGPU benchmarks are used, *Chimera* improves ANTT and STP by 5.5x, and 12.2%, respectively, on average. Other combinations of GPGPU benchmarks have smaller number of preemption requests, which results in smaller improvement compared to LUD combinations.

5. Related Work

Multitasking on GPUs is receiving a lot of attention from the research community as GPUs are becoming common in modern computer systems. First, we list recent works on enabling multitasking on GPUs. Next, we present previous studies on reducing the overhead of context switching on CPUs. Lastly, we discuss prior research that exploit the notion of idempotence.

Multitasking on GPUs: First attempts on GPU multitasking have been made on top of current GPUs by providing an illusion of single process to GPU or using cooperative multitasking. Context funneling [32] merges GPU con-

texts of multiple processes into a shared GPU context so that they can run concurrently on a single GPU. KernelMerge [9] makes GPUs only see a single scheduler kernel instead of individual independent kernels. Ino et al. [11] used cooperative multitasking to allow the concurrent execution of scientific and graphics applications on GPUs.

Some of the recent works study the scheduling policy when multitasking is enabled. Elastic kernels [24] transform kernels to enable fine grain control over the resource usage of kernels so that they can utilize SMs more efficiently. They study their scheme with multitasking by timeslicing a kernel to launch only a range of thread blocks at a time. Lee et al. [16] studies the interaction between thread block scheduling and warp scheduling. They also propose to run multiple kernels on the same SM, but do not present any detailed implementation.

Several works have paid attention to the independence of thread block execution. RGEM [12] splits memory transfers to GPU into smaller chunks so that they can be preempted at the chunk boundary. PKM [3] partitions a kernel into subkernels, where each subkernel executes a subset of thread blocks. SM draining [29] stops issuing a thread block and waits until all the running thread blocks are finished. Independence of thread block execution is a unique property of GPUs and creates opportunities for efficient preemption specific to GPUs. *Chimera* also utilizes the independence of thread block execution to enable SM flushing.

Spatial multitasking [1] observes that kernels may not fully occupy all the available SMs and shows that kernels can run on different subset of SMs. However, spatial multitasking still requires preemption if one kernel wants to dynamically take SM that is already running another kernel. Tanasic et al. [29] implement context switching to show that it still improves ANTT, however, they do not solve the problems of long preemption latency and large throughput overhead.

Chimera can control the overhead in preemptive multitasking with collaborative preemption. *Chimera* is the only solution so far that can meet a given preemption latency with minimized throughput overhead.

Context switching: Reducing the overhead of context switching has been researched for CPUs as well. One approach is finding fast context switch points, where there are only few live registers so that the amount of context to be stored can be reduced [27, 33]. But they either achieve small amount of gain or require code specific to each switch point for context switching. Another approach is to mark registers with additional bits to annotate whether corresponding register should be stored during the context switch [23]. With tens of thousands of registers in GPUs, the extra storage overhead is not acceptable. ASTI [26] statically sets context switching points during compile time, thus needs to know which applications will be running concurrently in advance.

All of these studies are limited in their applicability, and cannot be directly used in GPUs for low overhead context switching. In *Chimera*, context switching collaborates with preemption techniques that are specialized for GPUs to achieve low overhead preemption.

Idempotence: Idempotence has been primarily exploited to reduce the overhead of checkpointing in hardware. Reference idempotency [15] optimizes speculative execution by not tracking idempotent references, thus reduces speculative storage. Idempotent processor [6] and iGPU [17] implement low overhead exception support for CPUs and GPUs, respectively. They reconstruct a consistent program state for precise exception by re-executing from the beginning of idempotent region to the point of exception. Relax [5] and Encore [8] recover from soft errors with low overhead by selectively rerunning the idempotent regions rather than checkpointing all the states.

Chimera shares the idea of idempotence and is the first solution to try the notion of idempotence to eliminate preemption latency on GPUs with flushing. Moreover, flushing can be implemented with minimal overhead because the flush logic already exists, and relaxed idempotence condition is detected in software.

6. Conclusion

In this paper, we presented *Chimera*, a collaborative preemption approach on a shared GPU, that enables multitasking with controlled overhead. *Chimera* utilizes two GPU-specific preemption techniques called draining and flushing on top of traditional context switching. Draining exploits the independence of thread block execution to allow low throughput overhead preemption. Flushing brings the concept of idempotent execution to preemption, which can be combined with the independence of thread block execution to enable low preemption latency. By intelligently selecting a subset of SMs to be preempted as well as the preemption techniques for thread blocks, *Chimera* can meet a given preemption latency constraint with minimal throughput overhead. Evaluations show that *Chimera* violates a 15μs preemption latency constraint for only 0.2% of the preemption requests. For multi-programmed workloads, *Chimera* can improve the average normalized turnaround time by 5.5x, which can go up to 25.4x when a large number of preemption requests occur. System throughput can be improved by 12.2%, which can go up to 41.7% when a large number of preemption requests exist. *Chimera* demonstrates that preemptive multitasking on a shared GPU requires a different strategy from a traditional CPU, but is practical to implement.

Acknowledgments

We would like to thank the anonymous reviewers as well as Daya S. Khudia, Shruti Padmanabha, and Ankit Sethia for their valuable comments and feedbacks. This work is supported by the National Science Foundation under grants CNS-0964478 and CCF-1438996.

References

[1] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for GPGPU spatial multitasking. In *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012.

[2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[3] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 287–296, 2012.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009.

[5] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 497–508, June 2010.

[6] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *Proc. of the 44th Annual International Symposium on Microarchitecture*, pages 140–151, 2011.

[7] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[8] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott Mahlke, and David August. Encore: Low-cost, fine-grained transient fault recovery. In *Proc. of the 44th Annual International Symposium on Microarchitecture*, pages 398–409, 2011.

[9] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*. USENIX, 2012.

[10] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, June 2001.

[11] Fumihiko Ino, Akihiro Ogita, Kentaro Oita, and Kenichi Hagihara. Cooperative multitasking for GPU-accelerated grid systems. *Concurrency and Computation: Practice and Experience*, 24(1):96–107, 2012.

[12] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66, 2011.

[13] Shinpei Kato, Karthik Lakshmanan, Ragunathan (Raj) Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *2011 USENIX Annaul Technical Conference (USENIX ATC'11)*, pages 17–30, 2011.

[14] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010.

[15] Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T.N. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. In *Proc. of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 2–11, 2001.

[16] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proc. of the 20th International Symposium on High-Performance Computer Architecture*, pages 260–271, 2014.

[17] Jaikrishnan Menon, Marc de Kruijf, and Karthikeyan Sankaralingam. iGPU: Exception support and speculative execution on GPUs. In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 72–83, 2012.

[18] NVIDIA. GPU Computing SDK. http://developer.nvidia.com/gpu-computing-sdk.

[19] NVIDIA. Fermi: Nvidias next generation CUDA compute architecture, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[20] NVIDIA. *CUDA C Programming Guide*, May 2011.

[21] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012. www.nvidia.com/content/PDF/NVIDIA_Kepler_GK110_Architecture_Whitepaper.pdf.

[22] NVIDIA. Sharing a GPU between MPI processes: Multi-process service (MPS) overview, 2014. http://docs.nvidia.com/deploy/mps/index.html.

[23] Andris Padegs, Brian B. Moore, Ronald M. Smith, and Werner Buchholz. The IBM system/370 vector architecture: Design considerations. *IEEE Transactions on Computers*, 37(5):509–520, 1988.

[24] Sreepathi Pai, Matthew J. Thazhuthaveetil, and Ramaswamy Govindarajan. Improving GPGPU concurrency with elastic kernels. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 407–418, March 2013.

[25] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance runtime mechanism to partition shared caches. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 423–432, 2006.

[26] Siddhartha Shivshankar, Sunil Vangara, and Alexander G. Dean. Balancing register pressure and context-switching delays in ASTI systems. In *Proc. of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 286–294, September 2005.

[27] Jeffrey S. Snyder, David B. Whalley, and Theodore P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, 1995.

[28] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and

Wen mei Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, March 2012.

[29] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on GPUs. In *Proc. of the 41st Annual International Symposium on Computer Architecture*, pages 193–204, 2014.

[30] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 26–34, 2003.

[31] Javier Vera, Francisco J. Cazorla, Alex Pajuelo, Oliverio J. Santana, Enrique Fernandez, and Mateo Valero. FAME: Fairly measuring multithreaded architectures. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 305–316, 2007.

[32] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *2011 International Conference on High Performance Computing and Simulation*, pages 24–32, 2011.

[33] Xiangrong Zhou and Peter Petrov. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proc. of the 43rd Design Automation Conference*, pages 352–357, 2006.