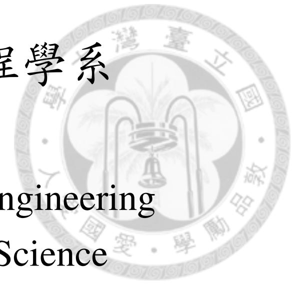


國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

於圖形處理器上支援雙程序並行之快速搶先機制

Enabling Fast Preemption via Dual-Kernel Support on GPUs

薛立維

Li-Wei Shieh

指導教授：楊佳玲 博士

Advisor: Chia-Lin Yang, Ph.D.

中華民國 105 年 8 月

August, 2016

doi:10.6342/NTU201602299

致謝

我很榮幸從308實驗室畢業並完成此論文，碩士兩年的時間過得非常快，日子雖然忙碌卻很充實。感謝我的指導教授—楊佳玲老師，除了研究方向的指引以外，平常對於切入問題以及邏輯推演的訓練都幫助我跳脫既有思考模式的框架，使我更可以做出正確的推論及判斷。感謝每一位口試委員在碩士學位考試過程中的建議以及回饋，才得已讓此論文更加完整；感謝陳坤志老師在論文撰寫中的諸多評論以及校正，使我更能作出清楚的論述並掌握技術性文章寫作的要領。研究的過程中，我很慶幸遇到一群好夥伴，不論是在學術上的討論交流或是日常生活的相互照應，我都從中收穫了很多。這條路上，實驗室的各位給予了我許多前進的動力，我要謝謝博士班學長柏翰的帶領，使我有健全的基礎來作更深入的研究，不論背景知識或是學術研究上都給予了我非常大的幫助；謝謝王立、立展、君濠，一起努力、一起說笑，有你們研究上的協助以及平常的照顧，讓我感覺碩士的生活並不孤單；謝謝實驗室的學弟妹，有你們的鼓勵以及打氣，我才能對於未來更有信心。另外，我還要謝謝我的家人，你們是我碩士生活中最大的依靠，有你們的支持以及祝福我才能夠順利完成碩士學業。謝謝每一位曾經幫助我的人，碩士短暫的兩年我成長了不少，不僅是專業知識或是待人處事上，我都從你們身上學習到我從未有過的體悟。累的時候，一路走來有你們才能使我隨時充滿能量；笑的時候，也因為有你們我才得以分享我的喜悅，由衷得感謝你們每一個人陪我走過這段日子，很辛苦、很踏實，也很深刻。

薛立維　謹上

doi:10.6342/NTU201602299

中文摘要

異質計算提供了一個有效率的方式來增進系統效能。在這些異質系統中，圖形處理器扮演了一個非常重要的角色來協助中央處理器加速應用程式的運行。除了傳統的圖形處理工作，圖形處理器可以有效率地加速具有資料平行化特性的工作，從資料中心、雲端計算甚至手持裝置都可以看到其應用。當加速應用程式的需求逐漸提高，為了滿足不同應用程式之間的服務品質，在圖形處理器上支援搶先機制也顯得日益重要，特別是在那些有硬體限制的手持系統上。然而，圖形處器上龐大的內文會導致傳統的內文轉換引起巨大的搶先代價。一個高優先權的程式必須等待長時間的延遲來搶先正在執行的程式，同時也造成了系統效能的降低。因此，在真正的異質計算思維下支援一個快速搶先的機制是非常關鍵的議題。

近來，透過圖形處理器架構上的修改有許多搶先機制被提出。然而，這些方法沒有考慮到圖形處理器的資源使用率或是許多核心程式資源共享所造成的碎裂問題。因此，一個高優先權程序的執行時間有可能超過其完成的最後期限。為了提供高優先權程序的服務品質，我們運用了雙程序並行的方式來支援快速搶先。這樣的方式除了能夠達到較細部的搶先以外，也簡化了資源碎裂問題。首先，我們提出一個資源分配的政策來避免資源碎裂問題。第二，我們提出了一個選擇犧牲者的方法來最小化造成的搶先代價，並且符合所要求的延遲時間。實驗結果顯示，對於程序執行的期限違反率而言我們的機制可以達到距離完美的搶先機制僅有2%的差距。此外，比起先前別人提出的搶先機制，我們在搶先的過程中平均改善了資源使用率2.93倍。

關鍵字 — 異質計算；圖形處理器；行動系統；搶先；內文轉換

doi:10.6342/NTU201602299

Abstract

Heterogeneous computing has been proven as an efficient way to improve system throughput. In these heterogeneous systems, Graphics Processing Units (GPUs) play an important role for applications acceleration to assist Central Processing Units (CPUs). In addition to traditional graphics workloads, GPUs are able to accelerate data parallel workloads effectively, and have been widely used in data center, cloud computing, and even mobile devices. As the demand of application acceleration increases, a preemption mechanism to fully support Quality of Service (QoS) among different applications is needed, especially for those resource-limited mobile systems. However, traditional context switching incurs tremendous preemption cost due to the large context of GPUs. A high-priority task suffers from a long latency to preempt the running task, and system throughput degrades during the switch time. Therefore, supporting fast GPU preemption is a critical enabling technology to the true heterogeneous computing paradigm.

Recently, many preemption mechanisms were proposed on GPUs with architectural extension. However, these preemption schemes either do not consider GPU resource utilization or the fragmentation problem caused by fine-grained resource sharing among multiple kernels. Consequently, a high-priority tasks may violate its deadline. To meet the QoS of a high-priority task, we introduce a dual-kernel approach to support fast preemption on GPUs. Our approach achieves fine-grained preemption and can simplify the fragmentation problem. First, we proposed an resource allocation policy to avoid fragmentation problem. Second, a victim selection scheme is proposed to minimize the preemption cost while satisfying a required preemption latency. The experimental results show that our approach can reach very close to the ideal preemption scheme within 2% difference in terms of deadline violations. Furthermore, on average we improves GPU resource utilization by 2.93x over prior technique during preemption.

Keywords — Heterogeneous Computing; Graphics Processing Unit; Mobile System; Preemption; Context Switch

doi:10.6342/NTU201602299

Contents

致謝 **i**

中文摘要 **ii**

Abstract **iii**

1 Introduction **1**

2 Background **4**
2.1 GPU Execution Model 4
2.2 GPU Architecture 5
2.3 Prior Preemption Techniques 6

3 Mechanism **7**
3.1 Fast GPU Preemption Mechanism 7
3.2 Dual-Kernel Support with Allocation Alignment 8
3.3 Victim Selection 10
3.3.1 Candidate Victim Sets Determination 10
3.3.2 Preemption Cost Estimation 11
3.3.3 Identifying the Final Victim 12
3.4 Architectural Support 14
3.4.1 Preempting Kernel Allocation Status 14
3.4.2 Preempting Kernel Allocation Positions 15
3.4.3 Candidate Victim Sets 15
3.4.4 Running TB Status 16
3.4.5 Storage Overhead 16

4 Results **18**
4.1 Methodology 18
4.2 Prioritized Task with Deadline 19
4.3 Impact of Preemption 21
4.4 Study of Preemption Latency Constraint 23

5 Related Work **25**
5.1 Multitasking on GPUs 25
5.2 Preemptive Multitasking on GPUs 26
5.3 Fine-Grained GPU Resource Sharing 26

6 Conclusion **28**

doi:10.6342/NTU201602299

Bibliography **29**

doi:10.6342/NTU201602299

List of Figures

1.1 Execution model of GPUs. (a) Exclusive access of a GPU in typical execution model, (b) Spatial resource sharing among SMs, (c) A preemption scheme without multiple kernel support in an SM, and (d) The proposed preemption scheme with dual-kernel support to enable fine-grained preemption. 2

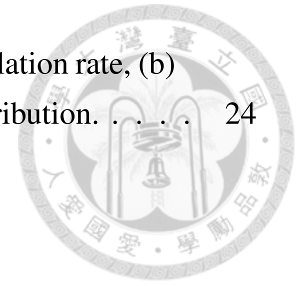
2.1 Baseline GPU architecture overview. 5

3.1 Overview of the proposed preemption framework on GPUs. 8
3.2 An illustration of SM resource allocation. Fragmentation occurs in the case of (a) Naive Resource Allocation, but can be avoided with (b) Resource Allocation Alignment. 9
3.3 Victim candidate sets are determined as shown in flow chart (a), where k is the number of predetermined positions. An example of determining a candidate is shown in (b). 10
3.4 Algorithm 1: Victim Selection 13
3.5 Architectural support for the proposed preemption framework. 14
3.6 An example of preempting kernel allocation status. 15
3.7 An example of preempting kernel allocation positions. 15
3.8 An example of candidate victim sets. 15
3.9 An example of updating candidate victim sets. 16
3.10 An example of TB Progress Status. 16

4.1 The percentage of violations among multiple preemptions with different preemption points. The preemption latency constraint is set to $2us$. 20
4.2 Error rate of draining latency estimation. 20
4.3 Normalized throughput overhead under $2us$ preemption latency constraint when preemption occurs. 22
4.4 Average resource utilization rate in each cycle during the preemption process. 22

doi:10.6342/NTU201602299

4.5 Study of different preemption latency constraints on (a) Violation rate, (b) Normalized Throughput Overhead, and (c) Technique Distribution. 24



doi:10.6342/NTU201602299

List of Tables

3.1 Total storage overhead of the proposed preemption framework. 17

4.1 Simulator configuration parameters. 18

4.2 Benchmarks with different characteristics. 19

doi:10.6342/NTU201602299

Chapter 1

Introduction

With the rise of heterogeneous computing, today's computer systems are increasingly adopting Graphics Processing Units (GPUs) to assist Central Processing Units (CPUs), spanning widely varied environments from mobile systems to cloud computing. In these heterogeneous systems, workloads are offloaded to GPUs for application acceleration. In addition to traditional graphics workloads, GPUs are able to accelerate data parallel workloads (or kernels) with the help of programming models such as CUDA [11] and OpenCL [18], which exploit thread level parallelism to achieve high throughput on GPUs. As the demand of application acceleration increases, a preemption mechanism to fully support QoS (Quality of Service) among different applications is needed, especially for those resource-limited System-on-Chip (SoC) based mobile platforms. However, modern GPUs can have up to 2048 threads concurrently running on a compute unit. With such a large context, traditional context switching adopted in GPUs has high cost in both preemption latency and throughput overhead. Therefore, supporting fast GPU preemption is a critical enabling technology to the true heterogeneous computing paradigm.

In practical, the execution engine of GPU is composed of multiple Streaming Multiprocessors (SMs), and each SM includes large numbers of memory resources, such as registers and shared memory. In typical GPU execution models, a task (application) can be split into multiple kernels, and each task is served by GPU exclusively. A kernel consists of multiple thread blocks (TBs), which are dispatched to SMs for kernel execution, and a TB is a basic unit of task dispatch. NVIDIA's Hyper-Q [12] technology makes GPUs able to execute independent kernels from multiple independent kernel queues concurrently. However, this feature is limited to the kernels within a single application. Figure 1.1(a) illustrates an example. There are two tasks, T_0 and T_1, each of them is composed of two independent kernels, respectively. Independent kernels within T_0 can share a GPU, but kernels from T_1 have to wait until T_0 finishes. To solve the aforementioned problem, Multi-Process Service (MPS) [13] is introduced with a software solution, allowing ker-

doi:10.6342/NTU201602299

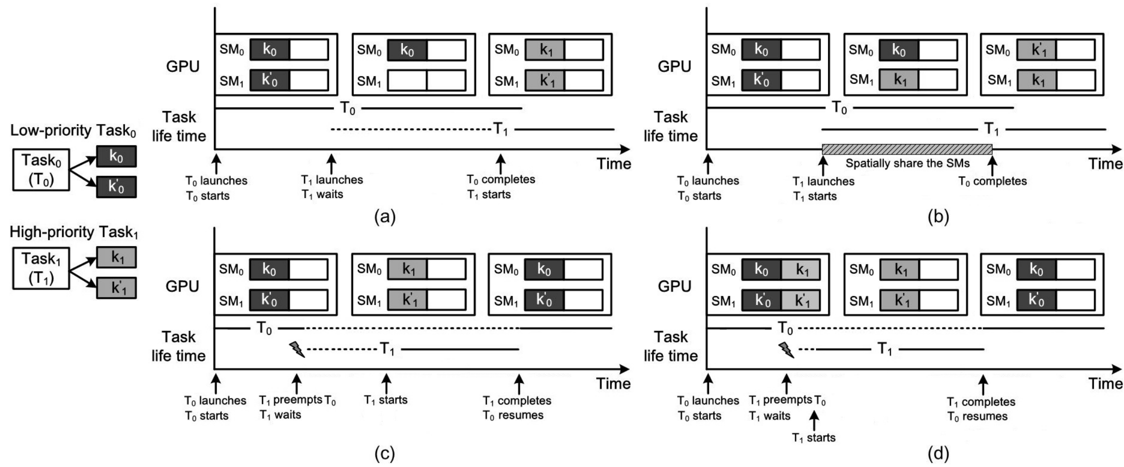
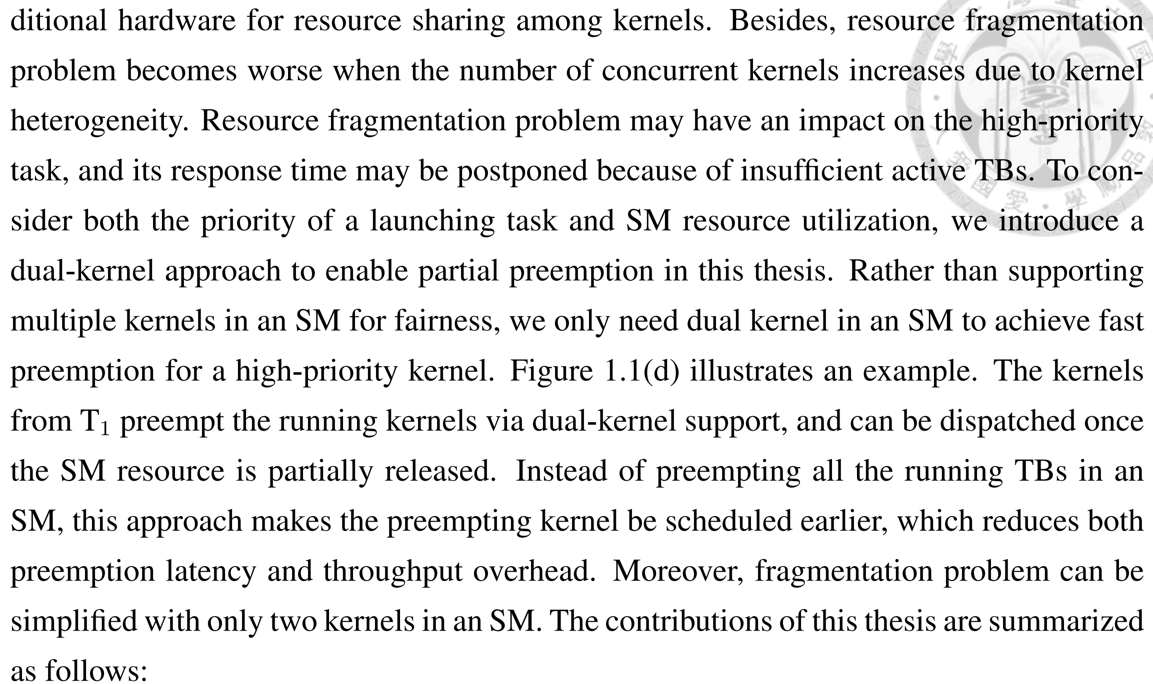


Figure 1.1: Execution model of GPUs. (a) Exclusive access of a GPU in typical execution model, (b) Spatial resource sharing among SMs, (c) A preemption scheme without multiple kernel support in an SM, and (d) The proposed preemption scheme with dual-kernel support to enable fine-grained preemption.

nels from different applications to co-execute on the same GPU, but still has no control of either kernel scheduling or resource management. As shown in Figure 1.1(b), k_0 from T_0 and k_1 from T_1 can spatially share SMs, but k'_1 still cannot be dispatched to SM_0 until k_0 finishes. Recently, many preemption schemes were proposed with architectural extension [20, 16], allowing the new-coming high-priority kernel to preempt the running kernel on an SM, rather than waiting for its completion. Figure 1.1(c) illustrates an example. The kernels from the high-priority task T_1 preempts the running kernel T_0 on both SM_0 and SM_1, respectively. T_1 gains control of GPU resources after preempting T_0, and T_0 resumes its execution after T_1 finishes. Although these approaches can make the high-priority task start its execution earlier, they still suffer from the problem of the incurred preemption latency and resource utilization degradation because their preemption granularity is an entire SM.

To further increase SM resource utilization, Simultaneous Multikernel (SMK) [22, 23] is proposed to make multiple kernels share all resource in an SM. By exploiting heterogeneity of different kernels, SMK can fully utilize the resources within an SM. In addition, partial context switching in an SM is proposed for achieving fairness among multiple kernels. Hence, the victims are thread blocks (TBs) from the kernel which occupies the most of the resources. However, for a high-priority task with deadline, we have to allocate GPU resources for the task as soon as possible to meet its deadline. Therefore, identifying appropriate preemption victims to satisfy the given preemption latency constraint is crucial. On the other hand, SMK needs complex resource allocation policy and ad-

doi:10.6342/NTU201602299



ditional hardware for resource sharing among kernels. Besides, resource fragmentation problem becomes worse when the number of concurrent kernels increases due to kernel heterogeneity. Resource fragmentation problem may have an impact on the high-priority task, and its response time may be postponed because of insufficient active TBs. To consider both the priority of a launching task and SM resource utilization, we introduce a dual-kernel approach to enable partial preemption in this thesis. Rather than supporting multiple kernels in an SM for fairness, we only need dual kernel in an SM to achieve fast preemption for a high-priority kernel. Figure 1.1(d) illustrates an example. The kernels from T_1 preempt the running kernels via dual-kernel support, and can be dispatched once the SM resource is partially released. Instead of preempting all the running TBs in an SM, this approach makes the preempting kernel be scheduled earlier, which reduces both preemption latency and throughput overhead. Moreover, fragmentation problem can be simplified with only two kernels in an SM. The contributions of this thesis are summarized as follows:

1) We introduce a lightweight dual-kernel approach to support fine-grained preemption, and propose a resource allocation policy to avoid resource fragmentation problem in an SM.

2) We propose a victim selection scheme and a computing model for preemption cost estimation to achieve fast preemption based on dual-kernel support.

3) The proposed scheme achieves fine-grained preemption to avoid resource utilization degradation during preemption, and can identify the victims with minimal preemption cost that satisfies a required preemption latency constraint.

doi:10.6342/NTU201602299

Chapter 2

Background

This section provides a brief background of modern GPU execution model and architecture that are necessary for understanding our proposed scheme. We model the baseline architecture according to NVIDIA Fermi GTX480 architecture [10], and we use NVIDIA's CUDA [11] terminology in this thesis.

2.1 GPU Execution Model

Typically, a GPU program consists of two parts: 1) host code, and 2) kernel code. Host code is executed on the CPU to process I/O operations, including input data preparation for the GPU, parallel workloads offload, and output data transfer from GPU to CPU. On the other hand, kernel code is executed on the GPU, exploiting thread-level parallelism for program acceleration. The host CPU offloads GPU kernels by kernel launch operation, which passes and stores kernel parameters to GPU global memory. The kernel launch overhead is in the order of microseconds (*e.g.*, around 5 microseconds [8]).

Programmers write code for the execution of one thread, and multiple threads execute the same code on the GPU. Threads are grouped into thread blocks (TBs), which are dispatched to Streaming Multiprocessors (SMs) for kernel execution by block scheduler. The total number of threads and the number of threads in a TB are specified by programmers. Within a TB, threads can be synchronized with barrier operations, and communicate through a on-chip scratchpad memory, called shared memory. Before dispatching a TB to an SM, the following conditions should be checked due to hardware limitations [11]:

1) The maximum number of TBs and threads, which can be concurrent running within an SM, is fixed.

2) The required memory resources must to be consecutive (*i.e.*, registers and shared memory).

doi:10.6342/NTU201602299

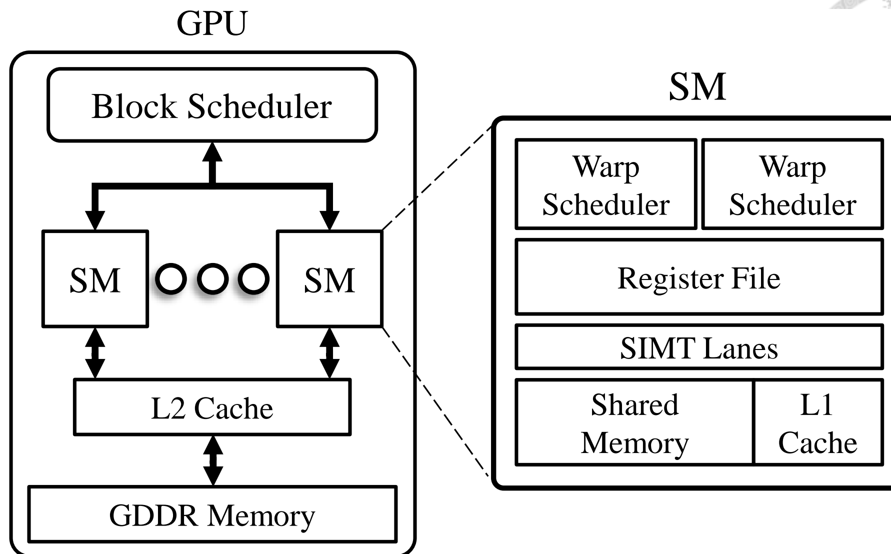


Figure 2.1: Baseline GPU architecture overview.

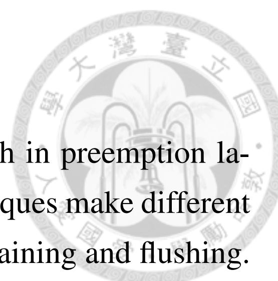
If all the mentioned requirements are met, a TB can be dispatched to the SM. As a basic unit of task dispatch, there is no shared state between TBs but within a TB. Therefore, the occupied SM resources by a TB can be released when all the threads in the TB are finished.

2.2 GPU Architecture

Figure 2.1 shows an overview of our baseline GPU architecture, which is composed of a scheduling unit, multiple SMs and memory units. Block scheduler is responsible for dispatching TBs to SMs, and managing the resource allocation between kernels. SM is main execution unit with many CUDA cores (*i.e.*, Streaming Processors (SPs) or SIMT lanes, including ALUs and FPUs). Each SM has its own L1 cache, and all SMs share a L2 cache. When a TB is dispatched to an SM, the threads within a TB are partitioned into several warps. A warp consists of 32 threads, which is the SIMT width of the GPU. Warp scheduler issues warps to SIMT lanes for threads execution, and each thread executes the same instruction within a warp. SMs can switch warps without overhead because their context are the same. Other warps can be scheduled to fill the SIMT lanes up while some of the threads are stalled due to long latency operations or memory accesses. As a result, the long latency can be hidden by warp scheduling if the number of warps (or thread) are sufficient, improving the resource utilization and having no impact on throughput.

doi:10.6342/NTU201602299

2.3 Prior Preemption Techniques



Supporting preemption schemes on GPUs incurs additional cost both in preemption latency and throughput overhead. However, different preemption techniques make different trade-offs. Prior preemption techniques include context switching, draining and flushing. GPU context switching is like conventional CPU context switching, which involves saving the context of current running task/threads to temporal memory space, scheduling the preemting task/threads and loading the context of the preempted task/threads to restart its execution.

However, Modern GPUs can have up to 2048 threads concurrently running on an SM. Each thread accesses its own registers, and the threads within a TB share their own on-chip scratchpad memory. With such a large context, traditional context switching adopted in GPUs may cause high cost in preemption latency. Furthermore, no progress is done during context switching and GPU throughput is degraded significantly. To overcome the throughput overhead caused by context switching, Tanasic et al. [20] propose draining technique. Draining keeps executing the running TBs and stops issuing TBs from the running kernel. Therefore, throughput overhead can be reduced because the running TBs still make progress during preemption, and the preempting kernel can be scheduled once the running TBs are finished. Although draining has less throughput overhead when compared to context switching, the preemption latency may be longer due to long running TBs.

To enable low latency preemption, Park et al. [16] introduce flushing, and further utilize flushing with the two mentioned techniques collaboratively to reduce preemption cost based on the progress of a TB. Context switching has almost constant cost across the execution, while draining has lower cost when a TB is near the end of its execution. Flushing drops the execution of the running TBs immediately without saving any context. The dropped TBs resume their execution from the beginning afterwards. Although flushing can achieve extremely low preemption latency (*i.e.*, almost zero latency), the wasted throughput can grow quickly if the progress of the preempted TBs are nearly at the end of its execution. Hence, flushing has lower cost when a TB is near the beginning of its execution.

doi:10.6342/NTU201602299

Chapter 3

Mechanism

3.1 Fast GPU Preemption Mechanism

To provide a fast preemption mechanism on GPUs, we first propose dual kernel execution in an SM. With dual-kernel support, we allow the preempting kernel to co-execute with the running kernel. Once the required resources are partially released, the new TBs can be dispatched. Instead of preempting all the running TBs in an SM, this approach makes the preempting kernel be scheduled earlier, which reduces both preemption latency and throughput overhead. However, resource fragmentation may occur while two different kernels are concurrent running on the same SM if we allocate SM resource for the preempting kernel naively. Second, determine which TBs to preempt and which preemption technique to use are crucial. The preemption decisions are critical to minimize throughput overhead while meeting the preemption latency constraint given by the preempting kernel or application.

Figure 3.1 shows the overview of our proposed preemption framework on GPUs. Once a high-priority task is launched, the GPU driver will deliver the preemption request from the host (*i.e.*, OS or kernel scheduler) to the GPU device. At first, we restrict the allocation positions for the preempting kernel to avoid resource fragmentation within an SM. The predetermined positions are stored in block scheduler. Next, block scheduler determines candidate victim sets based on the resource usage and the allocation status of the running and preempting kernel. As mentioned in Chapter 2, Section 2.1, the kernel launch overhead is in the order of microseconds, hence the above latency can be hidden. Afterwards, the SM passes the required information to block scheduler for preemption cost estimation (*i.e.*, executed cycles/instructions per running TB). At last, block scheduler makes preemption decisions based on the preemption cost of each candidate to identify the least throughput overhead one that satisfies the given preemption latency. The algorithm preempts a subset of the running TBs once at a time for each SM, which are itera-

doi:10.6342/NTU201602299

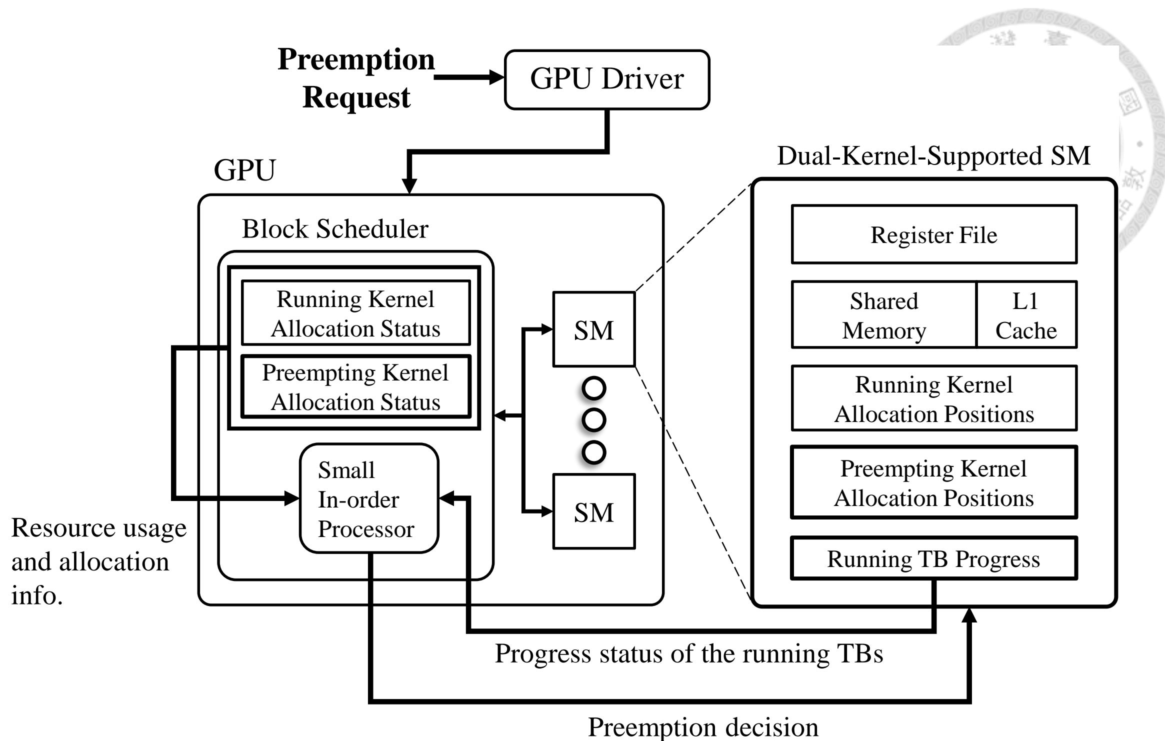


Figure 3.1: Overview of the proposed preemption framework on GPUs.

tively performed until the preempting kernel gains control of the GPU. Since we require fine-grained and frequent decisions, we can leverage a small, in-order, programmable processor in today's GPU [14].

3.2 Dual-Kernel Support with Allocation Alignment

We add very little hardware to enable dual kernel execution in an SM, including preempting kernel information and the metadata for its execution. Before dispatching a new TB to an SM, we have to ensure that the required SM resource (*i.e.*, registers and shared memory) for the new TB are sufficient [11]. Moreover, the allocated resource must be consecutive because a TB is a basic unit of dispatch. However, resource fragmentation may occur in an SM if we do not allocate SM resource for the preempting kernel carefully. As shown in Figure 3.2, there are four TBs (TB_0~TB_3) running on the SM in the beginning, then TB_1 and TB_2 are leaving. We assume a new TB can be dispatched if any two consecutive running TBs are preempted. Hence, in the case of naive allocation (Figure 3.2(a)), a new TB can be dispatched to the position of TB_1 and TB_2. However, we cannot dispatch another new TB when TB_0 and TB_3 both leave the SM. Even the released resources are sufficient, but we still need consecutive resource allocation for a new TB. In this case, only one new TB runs on the SM, but actually the maximum number can be two. The fragmentation problem may lead to insufficient active TBs (or threads) in an

doi:10.6342/NTU201602299

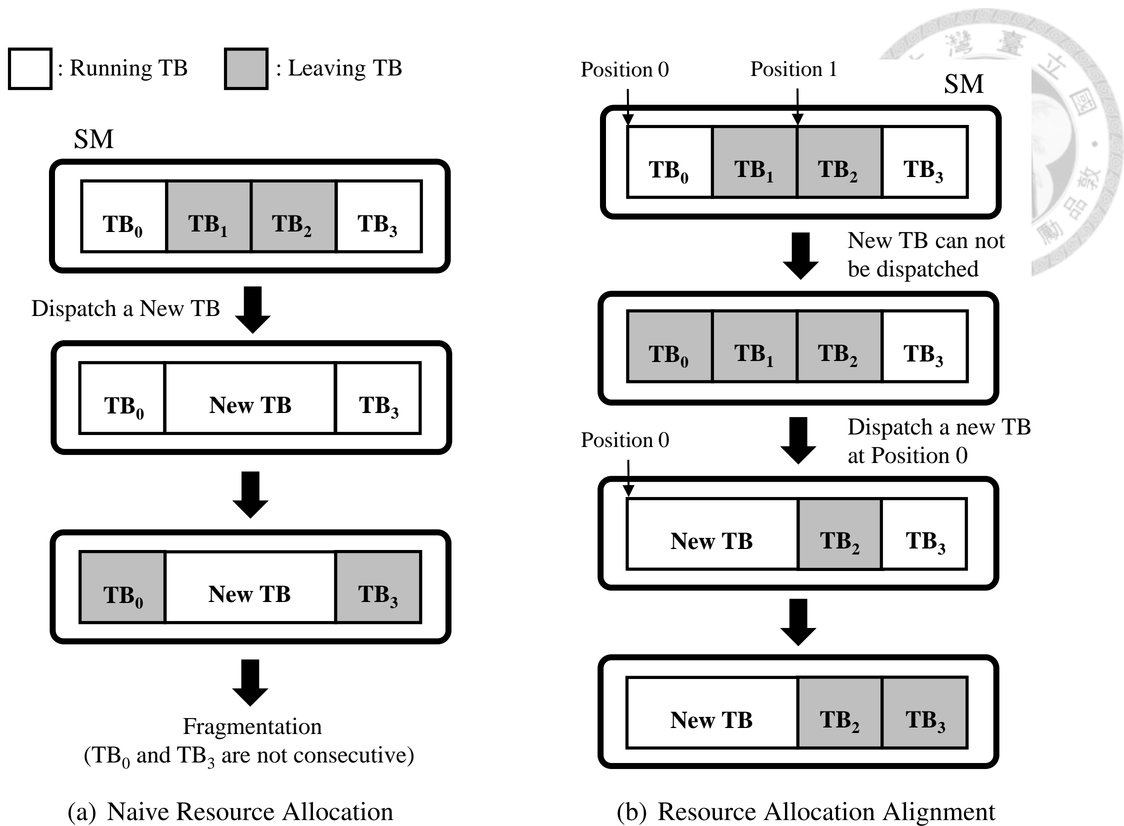
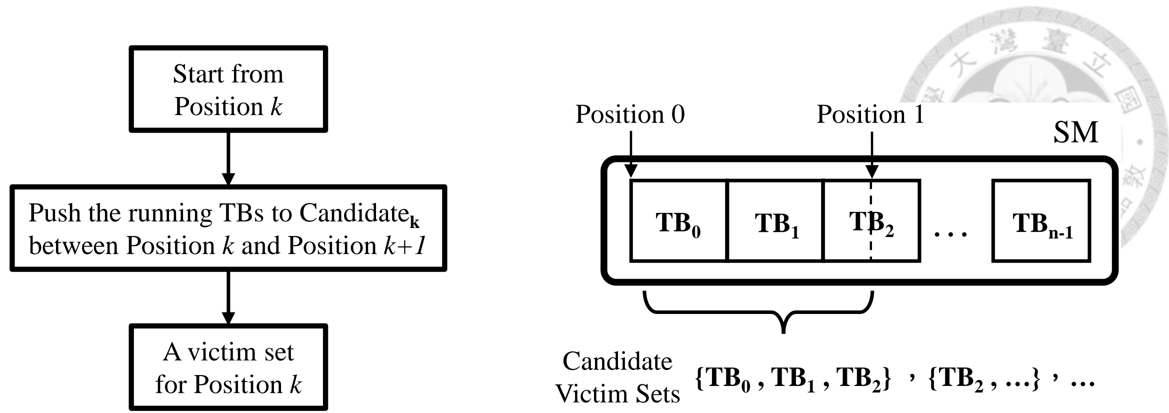


Figure 3.2: An illustration of SM resource allocation. Fragmentation occurs in the case of (a) Naive Resource Allocation, but can be avoided with (b) Resource Allocation Alignment.

SM, causing low SM resource utilization and throughput. Moreover, the response time of the task may be postponed.

In order to avoid this problem, we propose allocation alignment for the new TBs from the preempting kernel. We allocate SM resource at the predetermined positions that are aligned to the resource usage per TB of the preempting kernel, as shown in Figure 3.2(b). The allocation positions for the preempting kernel are shown with the arrows. We do not dispatch a new TB in the beginning, because no sufficient and consecutive resource at Position 0 or Position 1. However, we allocate the required resource for a new TB at Position 0 while TB_0 leaves the SM. Similarly, we can allocate the resource for another new TB at Position 1 while TB_3 leaves. In this case, we can make two new TBs run on the SM (*i.e.*, the case of Figure 3.2(b)) rather than one. By restricting the allocation positions, SM resource are allocated consecutively in a steady state to avoid resource fragmentation.

doi:10.6342/NTU201602299



(a) Flow chart of candidate victim sets determination

(b) An example of candidate victim sets determination

Figure 3.3: Victim candidate sets are determined as shown in flow chart (a), where k is the number of predetermined positions. An example of determining a candidate is shown in (b).

3.3 Victim Selection

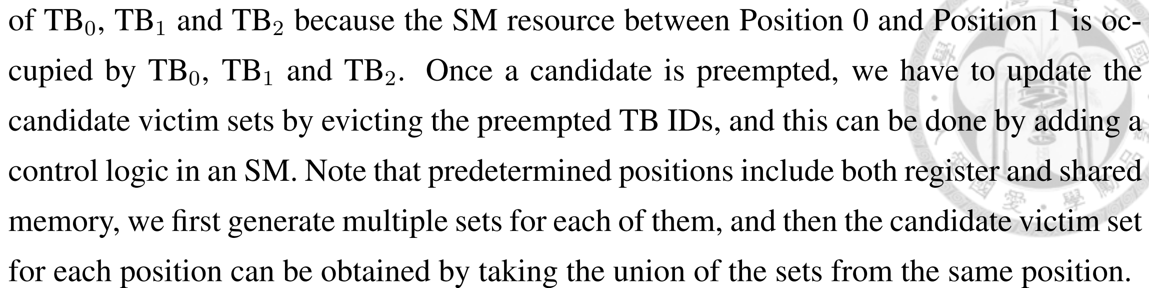
With dual-kernel support, we can achieve fine-grained preemption by partially preempting the running TBs in an SM. However, we still have to further determine appropriate victims to release the required resource for a new TB, and appropriate preemption techniques for the victims to achieve fast preemption. The proposed algorithm helps to determine multiple candidate victim sets in the beginning, and estimates the preemption cost for each candidate. At last, we identify the final victim to minimize throughput overhead while meeting the preemption latency constraint.

3.3.1 Candidate Victim Sets Determination

A candidate victim set contains a set of running TBs that could release the required resource for a new TB while they are preempted. Besides, the required resource for a new TB must be consecutive, a candidate should be composed of consecutive TBs. However, we cannot determine candidate victim sets naively through dividing the resource usage of a new TB by a running TB, because sometimes it is not divisible. Since the allocation positions for the preempting kernel are predetermined, we can further determine candidate victim sets based on these positions. Note that candidate victim sets are determined only once in an SM for a single preemption request.

Figure 3.3(a) illustrates a flow chart of candidate victim set determination. The candidates are determined based on the predetermined positions. The running TBs between any of two consecutive predetermined positions will be pushed into the candidate for the former position. In the example of Figure 3.3(b), a victim set for Position 0 is composed

doi:10.6342/NTU201602299



of TB_0, TB_1 and TB_2 because the SM resource between Position 0 and Position 1 is occupied by TB_0, TB_1 and TB_2. Once a candidate is preempted, we have to update the candidate victim sets by evicting the preempted TB IDs, and this can be done by adding a control logic in an SM. Note that predetermined positions include both register and shared memory, we first generate multiple sets for each of them, and then the candidate victim set for each position can be obtained by taking the union of the sets from the same position.

3.3.2 Preemption Cost Estimation

To minimize preemption cost, we have to further determine the final victim among candidate victim sets. We first estimate the preemption cost of each TB in a candidate with respect to different preemption techniques, then the minimal cost of the candidate can be calculated. The cost is interpreted as an aggregate of preemption latency and throughput overhead. In this thesis, we define throughput overhead as wasted instructions that is incurred by preemption.

As mentioned before, similar to the approach in [16], we integrate three preemption techniques (*i.e.*, context switching, draining, and flushing) for our preemption scheme. For flushing, the preemption latency can be neglected because running TBs are dropped immediately, and the throughput overhead is equal to the number of executed instructions of the preempted TBs. For context switching, due to the fixed context size of each TB in a kernel, the SM resource requirement can be calculated before a kernel launches. Therefore, the preemption latency can be estimated by dividing the context size based on an SM's share of global memory bandwidth. We assume global memory bandwidth is fairly shared among all SMs. To estimate the throughput overhead of context switching, the average instruction-per-cycle (IPC) of the preempted TBs in the SM is multiplied by the double of its preemption latency. Note that preemption latency is doubled because we have to consider the overhead incurred by both saving and restoring the context. We accumulate the executed instructions of each running TB in cycles by adding very little hardware in an SM. Hence, the average cycle-per-instruction (CPI) or IPC can be calculated. Note that we accumulate executed instructions for each TB at warp granularity instead of thread granularity to minimize the variation of instruction counts introduced by branch divergences.

For draining, the draining latency for a TB is defined as the remaining time for the GPU to finish it, which is the product of the remaining instructions and the CPI of executing those instructions. The number of remaining instructions can be simply obtained by subtracting the executed instructions from the average instructions of a TB in the preempted kernel. On the other hand, the CPI of executing remaining instructions cannot be obtained the same way as the number of remaining instructions, as CPIs of the TBs have

doi:10.6342/NTU201602299

higher variation. CPIs may vary across time for the same TB because of indirect memory accesses or branch divergences. Therefore, the past CPIs may not be representative for future CPIs even in the same TB. CPIs can also be data dependent, as some TBs access data regions with better locality than the others, leading to lower CPI. Hence, representing a TB's CPI with the average CPI of other finished TBs may not be accurate. In order to predict the future CPIs more accurately, we choose the way that has lower variation (*i.e.*, use past CPIs of running TBs as prediction over the CPIs of finished TBs if the standard deviation of the former is lower than the latter, and vice versa). In [16], draining incurs some throughput overhead because the number of active threads within an SM is decreasing during preemption, the latency cannot be hidden as long as the active threads are insufficient. However, in our scheme, the throughput overhead of draining is assumed zero because we can smoothly dispatch new TBs once there are enough SM resources. Note that if the cost cannot be estimated due to the insufficiency of gleaned statistics, we conservatively apply the maximum value as the estimated cost.

3.3.3 Identifying the Final Victim

Figure 3.4 illustrates the procedure of how the final victim is determined. Once the preemption cost is calculated for each TB, the preemption technique is also determined to minimize the cost while meeting the preemption latency constraint (line 1-14). If a TB cannot meet the preemption latency constraint by using any one of the three preemption techniques, we select the one with the minimal preemption latency. Once the preemption technique is determined for each TB, we then calculate the cost of each candidate (line 15-20). The cost of a candidate can be calculated as shown in the formulas below:

$$Preemption_Latency(Candidate_k) = \max_{i=0...n-1}(Preemption_Latency(TB_i)) \quad (3.1)$$

and

$$Throughput_Overhead(Candidate_k) = \sum_{i=0}^{n-1} Throughput_Overhead(TB_i) \quad (3.2)$$

, where *n* is number of TBs in candidate *k*. The preemption latency of a candidate is bounded by the last leaving TB, and the throughput overhead is the summation of the overhead caused by preempting each TBs in the candidate. Last, we select the least throughput overhead one while meeting the preemption latency constraint to be the final victim (line 21-26). Again, if all the candidates fail to meet the preemption latency constraint, we select the one with the minimal preemption latency.

The time complexity of Figure 3.4 is *O(NlogN+KlogK)*, where *N* is the number of TBs among all candidates, and *K* is the number of candidates. The first term is from the

doi:10.6342/NTU201602299

first for loop (line 1-14), estimation for N TBs takes $O(TN)=O(N)$, where T is the number of preemption techniques which is at most 3, and sorting takes $O(TNlogTN)=O(NlogN)$. The second term is from the remaining instructions (line 15-26), summing up the overhead for all TBs among K candidates takes $O(N)$ and sorting takes $O(KlogK)$. Note that N and K are fixed numbers (*e.g.*, 8 is the maximum number of TBs on an SM for NVIDIA Fermi GTX480 [10], also the maximum number of victim candidates is at most 8, but in general these two number are less than the maximum). Since the complexity of the algorithm is almost constant, this overhead is nearly negligible.

Algorithm 1: Victim Selection

Input: *k* victim candidates, *n TBs* and *t* techniques
Output: Preemption decision with selected candidate

```
1   for each TB in the SM
2       cost_estimation(TB)
3       for each preemption technique t
4           if meet_latency then
5               set technique t
6           end if
7       end for
8       sort_overhead(TB) with set techniques
9       if no technique meet_latency then
10          Technique[TB] = minimal latency technique
11      else
12          Technique[TB] = least overhead technique
13      end if
14  end for
15  for each victim candidate VC
16      if meet_latency for all TBs in VC then
17          set candidate VC
18      end if
19      Sum up the overhead for VC
20  end for
21  sort_overhead(VC) with set candidates
22  if no candidate meet_latency then
23      FinalVictim = minimal latency candidate
24  else
25      FinalVictim = least overhead candidate
26  end if
```

Figure 3.4: Algorithm 1: Victim Selection

doi:10.6342/NTU201602299

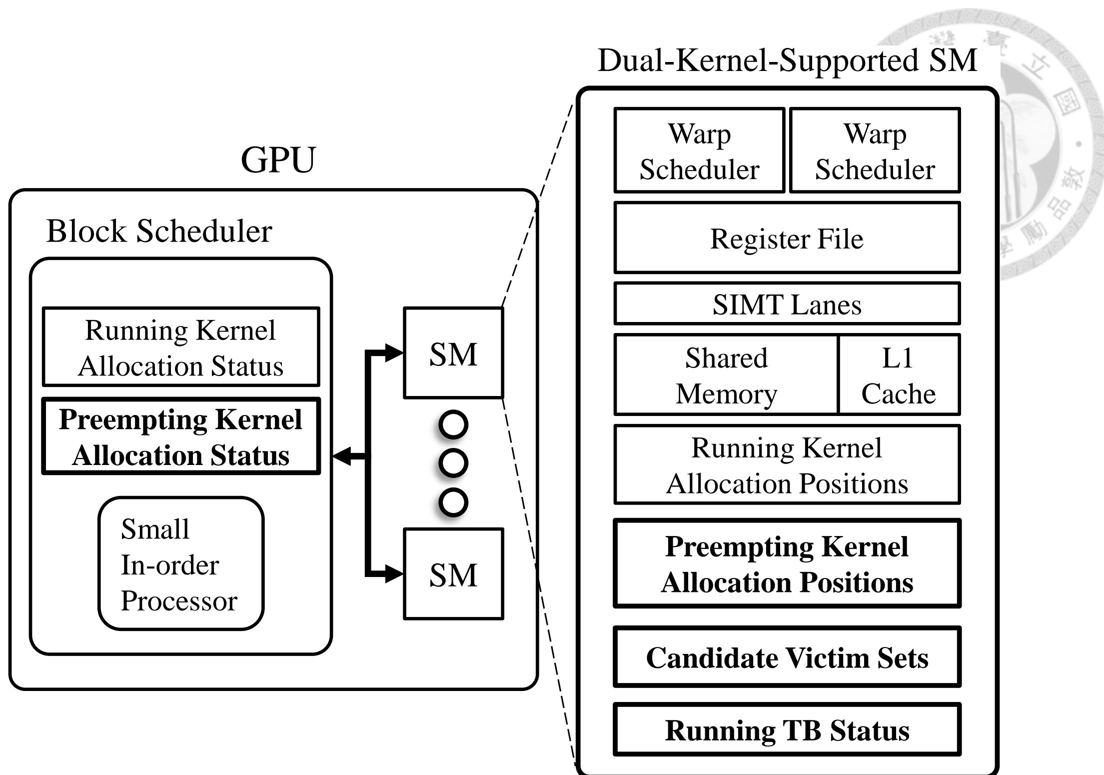


Figure 3.5: Architectural support for the proposed preemption framework.

3.4 Architectural Support

Figure 3.5 shows the architectural support for our proposed preemption frame work on GPUs. We add very little hardware in GPU and SMs (the highlighted part with bold frame) to enable dual-kernel execution in an SM. The added hardware keeps the information that is required for both the execution of the preempting kernel and preemption decision making. Preempting kernel allocation status records the mapping of TB-to-SM and resource allocation status for the preempting kernel. Preempting kernel allocation positions record the allocated positions of the register/shared memory for the TBs from the preempting kernel. Candidate victim sets contain multiple sets of running TBs' ID. Running TB status accumulates the executed instructions of each running TB in cycles, and use a bit to indicate if a TB can be flushed or not. Once a preemption request arrives, the processor in block scheduler first initializes candidate victim sets for each SM, and then gather the required information from SMs for the preemption decision making.

3.4.1 Preempting Kernel Allocation Status

Block scheduler records preempting kernel allocation status and the mapping of TB-to-SM, as shown in Figure 3.6. Block scheduler updates the allocation status once a TB is dispatched, similarly, once a TB from the preempting kernel finishes, SM reports its

doi:10.6342/NTU201602299

SM ID	Position	Status
SM M	0	True (Issued)
	1	False (Finished)
	…	…
	N	True (Issued)

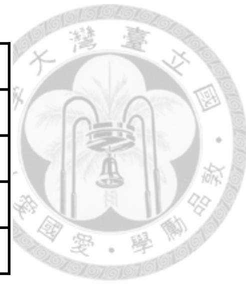


Figure 3.6: An example of preempting kernel allocation status.

	Position 0	Position 1	…	Position N
Register	0	4096	…	65536
Sh. Mem	0	2048	…	49152

Figure 3.7: An example of preempting kernel allocation positions.

kernel info and allocated position to block scheduler, then block scheduler updates the preempting kernel's allocation status.

3.4.2 Preempting Kernel Allocation Positions

Preempting kernel allocation positions record the allocated position for the TBs from the preempting kernel, including the predetermined addresses of register and shared memory, as shown in Figure 3.7.

3.4.3 Candidate Victim Sets

Candidate victim sets contain multiple sets of running TBs' ID, as shown in Figure 3.8. In the y-axis, it represents the number of candidates. While in the x-axis, it represents a bitset for each candidate, and a bit stands for a allocated position of the running TB in an SM. If the bit is set, it means the corresponding TB of the position belongs to the candidate.

	Bitsets
Candidate 0	11100000
Candidate 1	00111000
…	…
Candidate K	00000011

Figure 3.8: An example of candidate victim sets.

doi:10.6342/NTU201602299

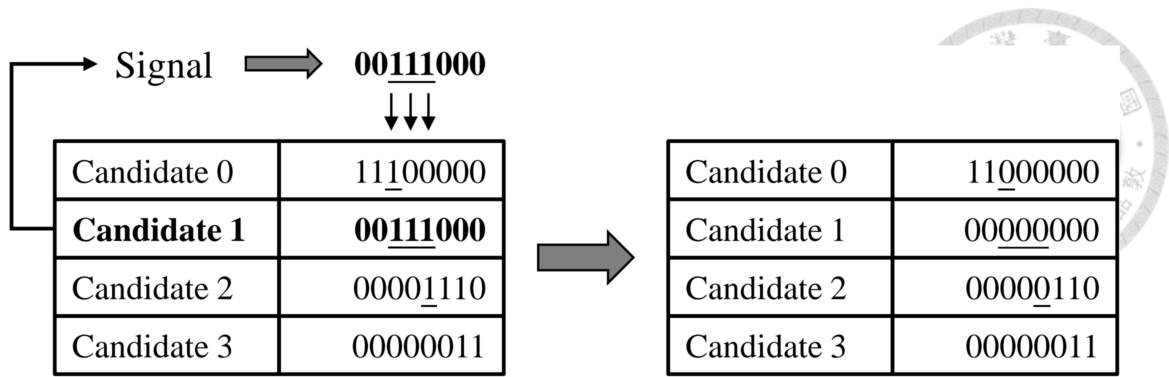


Figure 3.9: An example of updating candidate victim sets.

TB ID	Executed Instructions	Issue Cycle	Flush
0	486	2000	True
1	20	2474	False
…	…	…	…
N	574	1862	True

Figure 3.10: An example of TB Progress Status.

During the preemption process, we have to update the candidate victim sets once a candidate are preempted, and this can be done by adding a control logic in an SM. When a candidate is preempted, the added logic receives a signal and evicts the same positions of the bits from each candidate if the bits are set in the preempted candidate, as shown in Figure 3.8. We assume the preempted candidate is Candidate 1 in this case, and the updated victim candidate sets after preemption are shown on the right of Figure 3.9.

3.4.4 Running TB Status

Once a preemption request arrives, SMs report the status of each running TB to block scheduler for preemption cost estimation. As shown in Figure 3.10, the required information includes the executed instructions, issue cycle and a flush bit to check if a TB can be flushed or not.

3.4.5 Storage Overhead

Total storage overhead is shown in Table 3.1. In linux operating system, default maximum value of pid is 32768, which can be represented with 15bits. The maximum number that can be concurrent running on current generation GPUs is 32, which can be represented with 5bits. Since the minimum number of threads in a thread block is 32 and the

doi:10.6342/NTU201602299

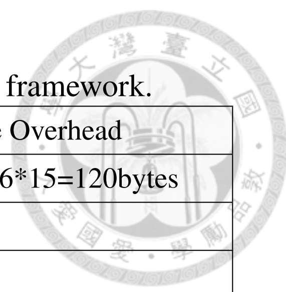


Table 3.1: Total storage overhead of the proposed preemption framework.

Components	Contents	Storage Overhead
Preempting Kernel Allocation Status	TB-to-SM Mapping	4bits*16*15=120bytes
Preempting Kernel Allocation Positions	Process ID	15bits
	Kernel ID	5bits
	Register	16bits*64=128bytes
	Shared Memory	16bits*64=128bytes
Candidate Victim Sets	Thread Block ID	4bits*16*16=128bytes
Running TB Status	Instructions	4bytes*16=64bytes
	Issue Cycle	4bytes*16=64bytes
	Flush Bit	1bit*16=2bytes
Total		≈0.5kB per SM + 0.12kB

maximum number of thread blocks in an SM is 2048 in current generation of GPUs, the predetermined allocation positions can be represented with 16bits*64=128bytes for 65536 registers and 48kB shared memory, respectively. Candidate victim list can be composed up to 16 candidates, and the maximum thread blocks in each candidate can be 16, which can be represented with 4bits*16*16=128bytes. The executed instructions and issue cycle for a TB requires 4byte respectively. Hence, we need 4bytes*2*16+16bits=130bytes to record the status of each TB in an SM. To sum up, the total storage cost is around 0.5kB per SM, which is less than 0.2% of the capacity of an SM in current generation GPUs.

doi:10.6342/NTU201602299

Chapter 4

Results

4.1 Methodology

To evaluate the proposed preemption mechanism, we use a cycle-accurate GPU simulator, GPGPU-Sim [2] in this thesis. We use default configuration to model a Fermi architecture [10], which is similar to NVIDIA GTX480. The configuration parameters are listed in Table 4.1. The default warp scheduling policy is Greedy-Then-Oldest (GTO) policy [17]. For the implementation of context switching, the latency depends on the current memory loading of GPUs, and the consumed bandwidth will affect other SMs. In this thesis, We implement context switching by stalling TBs for estimated context switch time to simplify the problem, and the memory bandwidth is assumed fairly shared among each SM.

We use two popular benchmark suites from Rodinia [3] and Parboil [19] in this thesis. To evaluate the effectiveness of our proposed scheme, we select benchmarks with various characteristics (*i.e.*, threads number, memory resource usage, and idempotence). The selected benchmarks and kernel attributes are listed in Table 4.2. A kernel is idempotent if it can be executed multiple times without changing the results beyond the initial execution [6, 9]. Therefore, an idempotent kernel should not have any atomic or global memory overwrite operations [16]. If we drop the execution of a non-idempotent kernel by flushing, the correctness of its execution cannot be ensured.

Although the Flushing mechanism can achieve low preemption latency, it is not eligi-

Table 4.1: Simulator configuration parameters.

# of SMs	15	SIMT width	32
SM Frequency	1.4GHz	Max # CTAs / SM	8
Memory Channels	6	Max # Threads / SM	1536
Memory Bandwidth	177.4GB/s	# Registers / SM	32768
Default Warp Scheduler	GTO	Shared Memory / SM	48kB

doi:10.6342/NTU201602299

Table 4.2: Benchmarks with different characteristics.

Benchmark	**Kernel**	**# Regs / TB**	**Sh. M. / TB (B)**	**# Threads / TB**	**TBs / SM**	**Idempotent**
pathfinder	dynproc	4096	2048	256	6	Yes
hotspot	calculate_temp	9216	3072	256	3	Yes
streamcluster	Kernel_compute_cost	6144	0	512	3	Yes
dwt2d	c_CopySrcToComponents	3584	3096	64	8	Yes
	Fdwt53Kernel	3072	768	256	6	Yes
kmeans	invert_mapping	3072	3072	256	6	Yes
	kmeansPoint	3072	3072	256	6	Yes
stencil	block2D_hybrid_coarsen_x	3584	0	128	8	Yes
srad	srad_cuda_1	5120	6144	256	6	Yes
	srad_cuda_2	5120	5120	256	6	No
b+tree	findRangeK	5120	0	256	6	No
	findK	4096	0	256	6	No
nw	needle_cuda_shared_1	1536	2180	32	8	No
	needle_cuda_shared_2	1536	2180	32	8	No
bfs	Kernel	8192	0	512	3	No
	Kernel2	6144	0	512	3	No
backprop	bpnn_layerforward_cuda	3072	1088	256	6	No
	bpnn_adjust_weights_cuda	6144	0	256	5	No
heartwall	kernel	8192	11872	256	4	No

ble with non-idempotent kernel due to the strict restrictions. To increase the opportunities for flushing, the relaxed idempotent condition is introduced in [16]. A TB can be regarded as idempotent if the mentioned operation (*i.e.*, atomic or global memory overwrites) have not been executed yet. Therefore, the execution of a TB can be safely dropped with the relaxed idempotence condition even if the kernel is non-idempotent. The detection of these operations can be done by compiler analysis. In this thesis, we follow the same idea and apply relaxed idempotent condition on flushing during simulations.

4.2 Prioritized Task with Deadline

We use a synthetic benchmark to mimic a high priority task with a hard deadline, and the GPGPU benchmarks from Table 4.2 are the tasks to be preempted. The deadline is preemption latency constraint plus the execution time of the preempting task (*i.e.*, the synthetic benchmark in this thesis). The synthetic benchmark uses 4096 registers, 2048 bytes shared memory, and 256 threads per TB with real world benchmark behavior.

Figure 4.1 illustrates the percentage of violations among multiple preemption requests under $2us$ preemption latency constrain. The preemption time is assumed an uniform tim-

doi:10.6342/NTU201602299

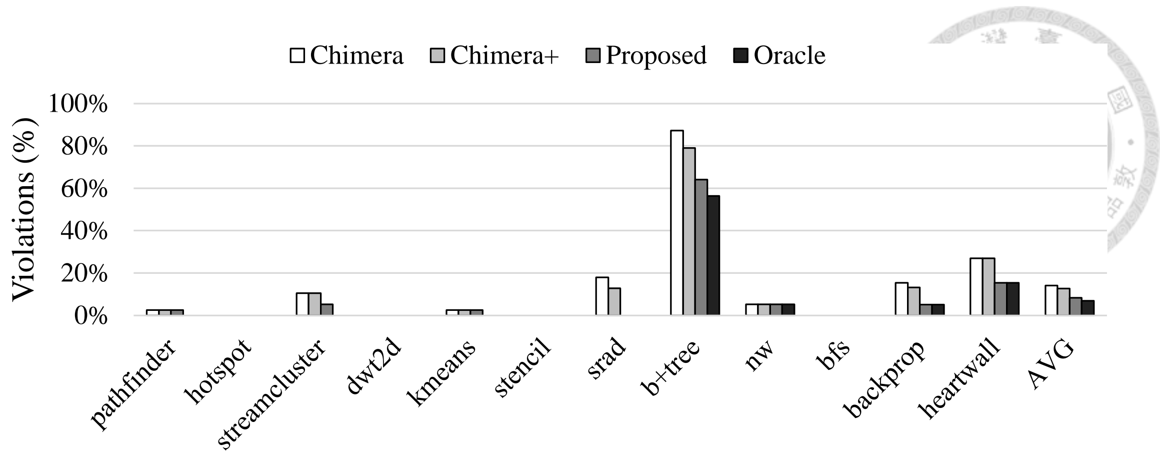


Figure 4.1: The percentage of violations among multiple preemptions with different preemption points. The preemption latency constraint is set to $2us$.

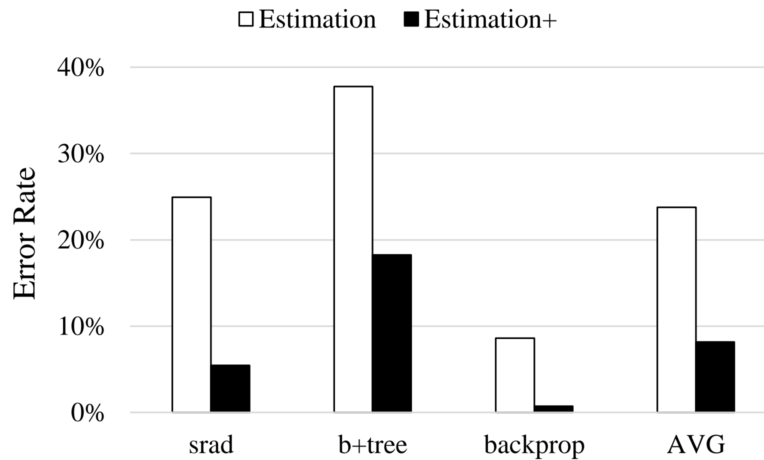
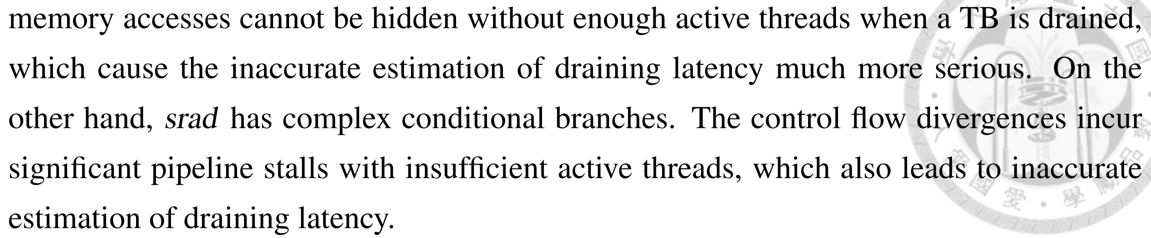


Figure 4.2: Error rate of draining latency estimation.

ing distribution across the execution of the GPGPU benchmarks. To clarify the effectiveness of our proposed scheme, we also compare the experimental results to Chimera [16] and Oracle. The oracle scheme is an ideal preemption mechanism, which can preempt the kernels with completed information of preemption latency from off-line profiling. As shown in Figure 4.1, the deadline violation rates by applying Chimera, the proposed scheme, and Oracle are 14.0%, 8.4% and 6.9% of preemptions respectively. Obviously, the proposed scheme can make the violation rate very close to the idea result of Oracle.

The deadline violations are primarily caused from the inaccurate estimation of draining latency and the strict restrictions of flushing. The result of *b+tree* shows the highest violation rate because *b+tree* is an idempotent kernel with short TB execution time. Because there are over 60% TBs are belong to non-idempotent, flushing is not eligible to achieve low preemption latency at most of the time. The other reason is that *b+tree* is a tree-based traversal algorithm with many indirect memory accesses. The long latency

doi:10.6342/NTU201602299



memory accesses cannot be hidden without enough active threads when a TB is drained, which cause the inaccurate estimation of draining latency much more serious. On the other hand, *srad* has complex conditional branches. The control flow divergences incur significant pipeline stalls with insufficient active threads, which also leads to inaccurate estimation of draining latency.

Figure 4.2 shows the error rate of draining latency estimation for Chimera and our scheme, respectively. We demonstrate the results of the three benchmarks which have violation reduction after using our estimation method, we can see the average error rate reduction is ≈15.6% when compared to Chimera. With dual-kernel support, the proposed victim selection scheme can achieve low preemption latency through fine-grained preemption. Moreover, we estimate draining latency based on program behavior to achieve higher accuracy, which leads to less violations than Chimera. The reasons mentioned above lead to the most violations reduction in *b+tree* and *srad*. Even when the preemption latency constraint is extremely low as in the case of 2*us* (In GK110 [12], context switch time can take up to 44*us* for an SM [20]), we still achieve low violation rate that is very close to oracle with ≈1.5% difference.

4.3 Impact of Preemption

When preemption occurs, GPUs suffer from utilization degradation, which may cause significant throughput overhead. To demonstrate the effectiveness of our proposed scheme, we also evaluate the throughput overhead and the resource utilization in each cycle during the preemption process. In this thesis, we define the throughput overhead as the wasted instructions, which are incurred by the three preemption mechanisms as mentioned earlier. For fair comparison, we normalize the throughput overhead to the result of flushing the preempted TBs.

Figure 4.3 shows normalized throughput overhead when the preemption latency constraint is set to 2*us*. Chimera+ stands for the preemption scheme of applying our draining latency estimation model on Chimera [16]. In *srad* and *b+tree*, most of the cases show that the inaccurate draining latency estimations in Chimera can be improved through our proposed estimation model, as shown in Figure 4.1 and Figure 4.2. Therefore, in these cases, Chimera+ utilize less draining to reduce the wasted instructions, which results in higher throughput overhead when compared to Chimera. On the other hand, we can see the incurred overhead by our proposed scheme is no more than Chimera under most of the cases due to dual-kernel support. Overall, the proposed scheme reduces throughput overhead by 25.3% compared to Chimera. In Figure 4.3, our scheme can achieve the largest throughput overhead reduction in *bfs* because the execution time of its TB is

doi:10.6342/NTU201602299

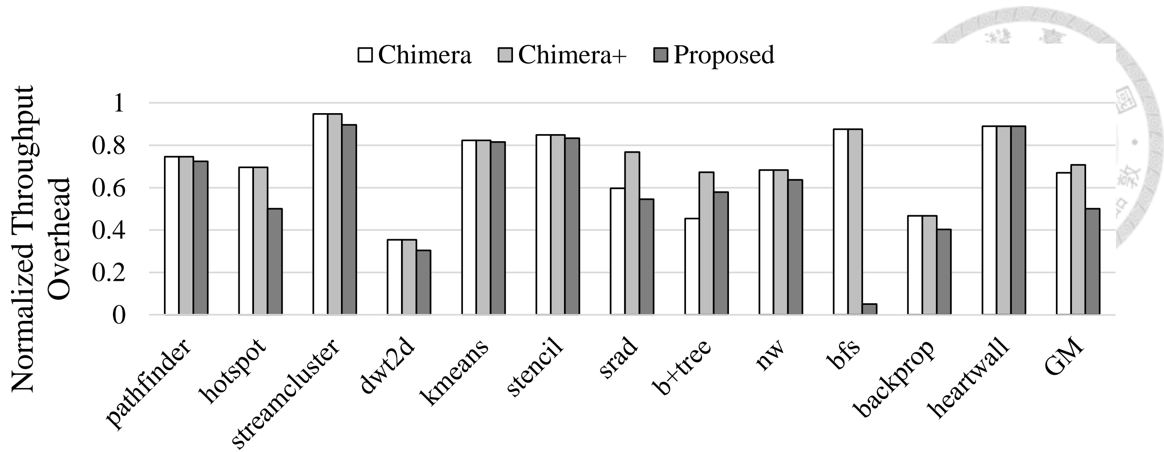


Figure 4.3: Normalized throughput overhead under $2us$ preemption latency constraint when preemption occurs.

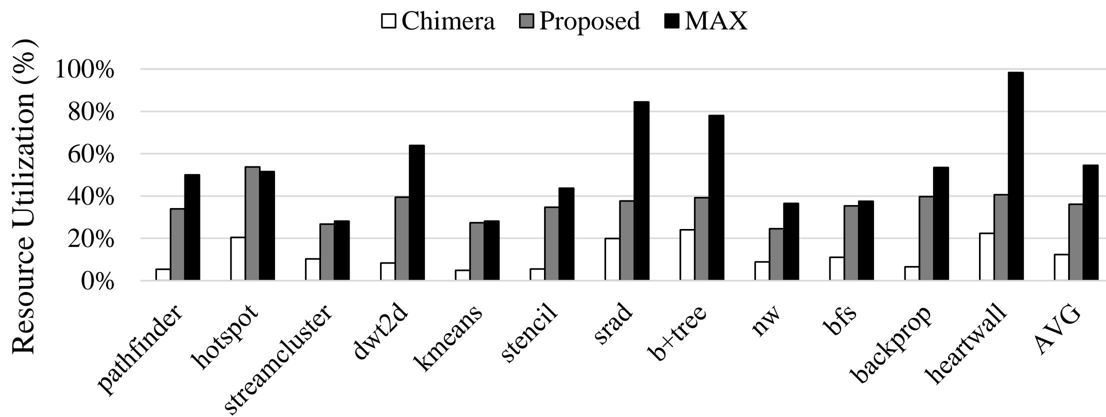


Figure 4.4: Average resource utilization rate in each cycle during the preemption process.

very short. Therefore, most of the TBs can be drained within the latency constraint, which causes very little wasted instructions. Compared with Chimera, we achieve more throughput overhead reduction among all benchmarks except *b+tree*. Chimera makes decisions without considering its program behavior, which cause inaccurate estimations of draining latency as explained in Section 4.2. However, we make decisions more accurately and use much less draining in this case, showing 23.1% violations reduction in Figure 4.1, but at the expense of additional 12.5% overhead in Figure 4.3.

As shown in Figure 4.4, MAX represents the maximum resource utilization rate for each GPGPU benchmark while there are no preemption occurs. We can obtain higher resource utilization rate than MAX in *hotspot* because the resources usage of *hotspot* and the synthetic benchmark are complementary. Therefore, the synthetic benchmark can be directly dispatched to SMs at the beginning of preemption. By leveraging the property of concurrent kernel execution in an SM, we can keep GPU busy rather than idle while

doi:10.6342/NTU201602299

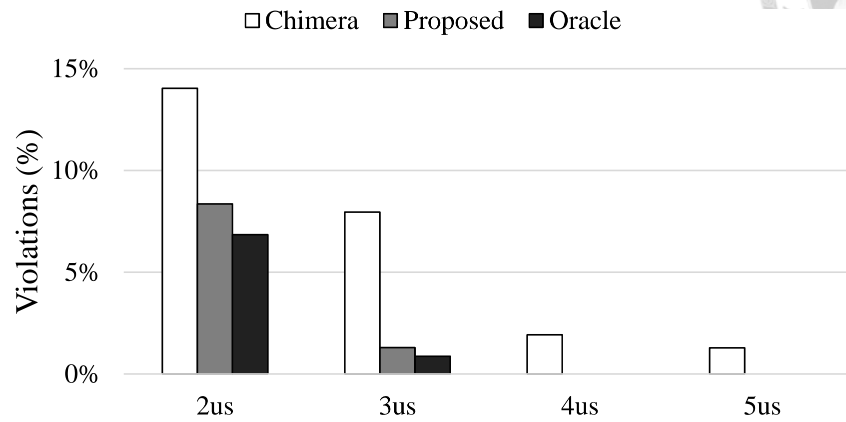
maintain resource utilization rate during the preemption process. On average we improve the resource utilization by 2.93x compared to Chimera.

4.4 Study of Preemption Latency Constraint

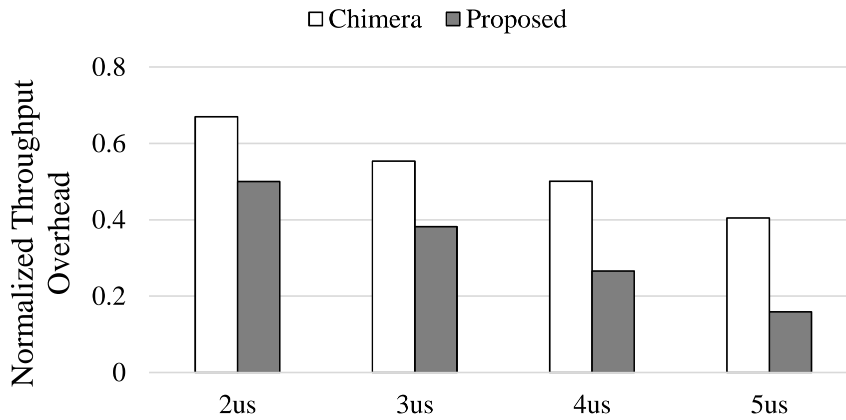
To demonstrate the impact of preemption latency constraint, we further study the violation rate, throughput overhead and strategy distribution when the preemption latency constrain is set from 2*us* to 5*us*. In Figure 4.5(a), Chimera [16] and our scheme shows higher violation rate at strict constraint. The reasons are that both two approaches rely on flushing to achieve low preemption latency, but flushing cannot be applied under all circumstances. However, we can still benefit from the property of concurrent kernel execution in an SM and start the execution of the preempting kernel earlier. As the preemption latency constraint increases, the proposed preemption scheme can benefit more from this property as well because of the increasing slack time. On average, Chimera violates the deadline for 14.03%, 7.95%, 1.93%, and 1.28% when constraint is varied from 2*us* to 5*us*, while our scheme are 8.36%, 1.30%, 0%, and 0%, respectively, which are very close to the results of Oracle within 2% difference.

As Figure 4.5(b) illustrates, Chimera causes the normalized throughput overhead for 67%, 55.3%, 50.1% and 40.5% when constraint is varied from 2*us* to 5*us*, while our scheme are 50.1%, 38.2%, 26.6%, and 15.9%, respectively. When the preemption latency constraint is tight, most of the TBs are forced to be flushed in both two approaches to meet deadline, which results in higher throughput overhead. In this case, the decisions of the proposed scheme are almost the same as Chimera's, as shown in Figure 4.5(c). However, the benefit of our scheme is clarified when the preemption latency constraint becomes loose. Because we leverage the property of dual-kernel execution in an SM and utilize draining to reduced wasted instructions, much throughput overhead reduction is demonstrated.

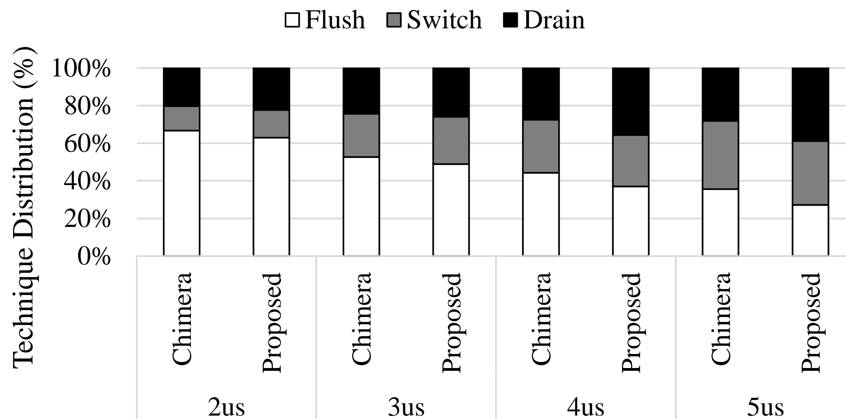
doi:10.6342/NTU201602299



(a) Violation rate



(b) Normalized Throughput Overhead



(c) Technique Distribution

Figure 4.5: Study of different preemption latency constraints on (a) Violation rate, (b) Normalized Throughput Overhead, and (c) Technique Distribution.

doi:10.6342/NTU201602299

Chapter 5

Related Work

As GPUs are increasingly becoming important components of modern computer systems, GPU sharing among multiple tasks is receiving a lot of attention from the research community. First, we introduce prior techniques and several works on supporting multitasking on GPUs. Next, we list several works on enabling preemptive multitasking on GPUs. At last, we present recent studies on fine-grained GPU resource sharing.

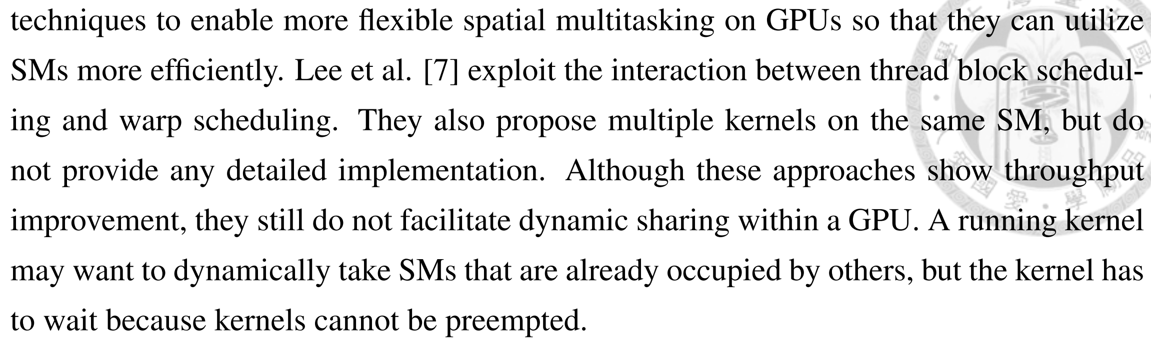
5.1 Multitasking on GPUs

Supporting kernel concurrency relies on the programmers that define CUDA stream, and each CUDA stream contains a sequence of kernels with dependencies [11]. Hyper-Q [12] technology further improves the concurrency by providing several hardware queues for CUDA streams. Therefore, the kernels from different hardware queues can be processed concurrently on the same GPU. However, the above mentioned techniques are restricted to a single application.

Multitasking on GPUs are intitially supported by providing an illusion of single process to a GPU or using cooperative multitasking. MPS [13] is a software solution that can make kernels for different applications share the same GPU by inserting kernels into a single MPS server process, and context funneling [21] merges GPU contexts of multiple processes into a shared GPU context so that they can run concurrently on a single GPU. KernelMerge [4] makes GPUs only see a single kernel rather than individual independent kernels. Ino et al. [5] propose a cooperative multitasking method for concurrent execution of scientific and graphics applications on GPUs.

Many works are also proposed to increase GPU throughput when multitasking is enabled. Spatial multitasking [1] is proposed to allow multiple kernels to simultaneously share the same GPU. Different kernels can occupy different subset of the SMs by partitioning GPU resource. Pai et al. [15] and Wu et al. [24] propose code transformation

doi:10.6342/NTU201602299



techniques to enable more flexible spatial multitasking on GPUs so that they can utilize SMs more efficiently. Lee et al. [7] exploit the interaction between thread block scheduling and warp scheduling. They also propose multiple kernels on the same SM, but do not provide any detailed implementation. Although these approaches show throughput improvement, they still do not facilitate dynamic sharing within a GPU. A running kernel may want to dynamically take SMs that are already occupied by others, but the kernel has to wait because kernels cannot be preempted.

5.2 Preemptive Multitasking on GPUs

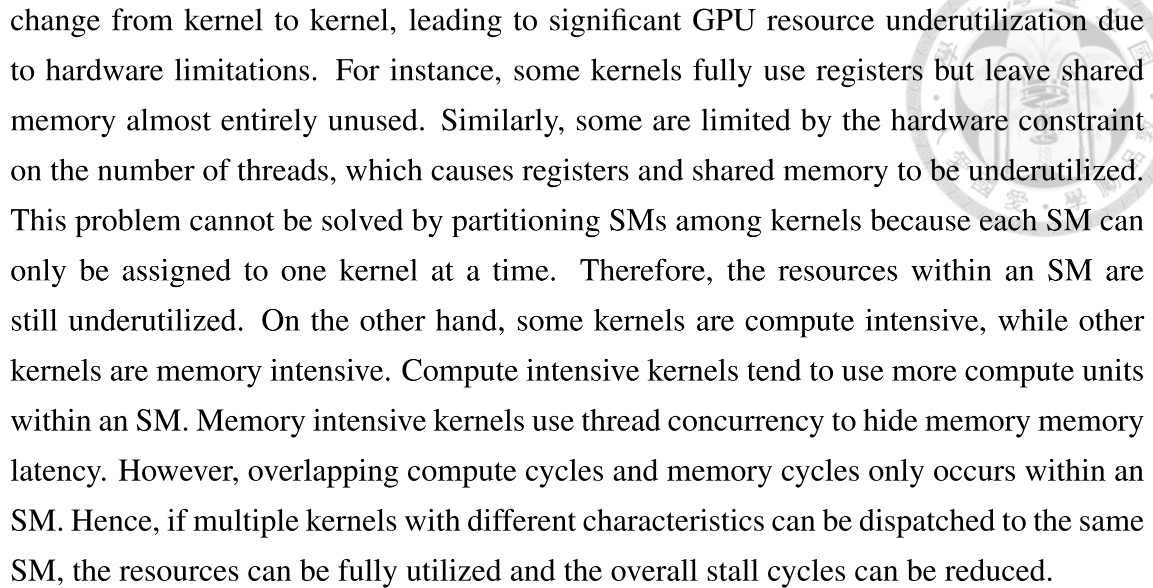
Recently, many preemption mechanisms have been proposed with architectural extensions for preemptive multitasking on GPUs. Tanasic et al. [20] implement context switching on GPUs, which swap the context of an SM to a temporal memory space, allowing another kernel to execute on the preempted SM, but this may incur tremendous overhead due to the large context of numerous concurrently executing threads in the SM. They also propose draining, which stops dispatching TBs to an SM and wait until the SM finishes the running TBs [20]. Therefore, throughput overhead can be reduced because the SM is still in progress during preemption.

In addition to context switching and draining, Chimera [16] introduces flushing to achieve low preemption latency by dropping the execution of the running TBs immediately. However, certain conditions have to be met to ensure the correctness of the TB execution that was dropped and restart from the beginning (*i.e.*, the idempotence of the kernels [6, 9]). To increase the opportunities for flushing, Chimera also proposes the relaxed idempotent condition to detect specific operations through compiler support. Chimera is a collaborative preemption mechanism that utilizes flushing with context switching and draining collaboratively to further reduce both preemption latency and thoughpur overhead within an SM. Because thread block executions are independent from each other, different TBs can be preempted by different preemption techniques. However, the preemption granularity of these approaches is an entire SM. A preempting kernel cannot be dispatched to an SM until all the running TBs are preempted, as a result, the preemption latency is bounded by the last leaving TB.

5.3 Fine-Grained GPU Resource Sharing

Wang et al. [22, 23] first discuss the resource sharing granularity within a GPU, and then propose a fine-grained resource sharing approach, Simultaneous Multikernel (SMK) to maximize GPU throughput. They observe that resource usage and runtime behavior

doi:10.6342/NTU201602299



change from kernel to kernel, leading to significant GPU resource underutilization due to hardware limitations. For instance, some kernels fully use registers but leave shared memory almost entirely unused. Similarly, some are limited by the hardware constraint on the number of threads, which causes registers and shared memory to be underutilized. This problem cannot be solved by partitioning SMs among kernels because each SM can only be assigned to one kernel at a time. Therefore, the resources within an SM are still underutilized. On the other hand, some kernels are compute intensive, while other kernels are memory intensive. Compute intensive kernels tend to use more compute units within an SM. Memory intensive kernels use thread concurrency to hide memory memory latency. However, overlapping compute cycles and memory cycles only occurs within an SM. Hence, if multiple kernels with different characteristics can be dispatched to the same SM, the resources can be fully utilized and the overall stall cycles can be reduced.

In order to improve GPU resource utilization effectively, SMK allows multiple kernels to share the resources within an SM by exploiting the heterogeneity of different kernels. Besides, partial context switching and fair resource allocation policies are also proposed to improve system throughput while maintaining fairness. Partial context switching only swaps out just enough TBs in an SM to make enough room for a new TB from the preempting kenel, which reduces context switching overhead because the other TBs are still in progress. However, increasing concurrent kernels requires additional hardware overhead and the incurred resource fragmentation problem within an SM may be worse.

doi:10.6342/NTU201602299

Chapter 6

Conclusion

In this thesis, we present a fast preemption mechanism via dual-kernel support on GPUs, which makes a high-priority task able to finish before the deadline. We extend GPU architecture to enable dual-kernel execution, allowing two concurrent kernels to run on the same SM. This approach can help the proposed scheme to make fine-grained preemption decisions, and the preemption cost can be further reduced by partially preempting the running TBs. In addition, it also simplifies resource fragmentation problem due to at most two different kernels in an SM, and the problem can be avoided through our proposed resource allocation policy. With dual-kernel support, the incurred throughput overhead can be minimized while meeting the given preemption latency constraint through our victim selection algorithm and preemption cost estimation model. Evaluations show that the proposed preemption scheme can reach very close to the ideal preemption scheme within 2% difference in terms of deadline violations. Furthermore, on average we improves GPU resource utilization by 2.93x over prior technique during preemption.

doi:10.6342/NTU201602299

Bibliography

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The Case for GPGPU Spatial Multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

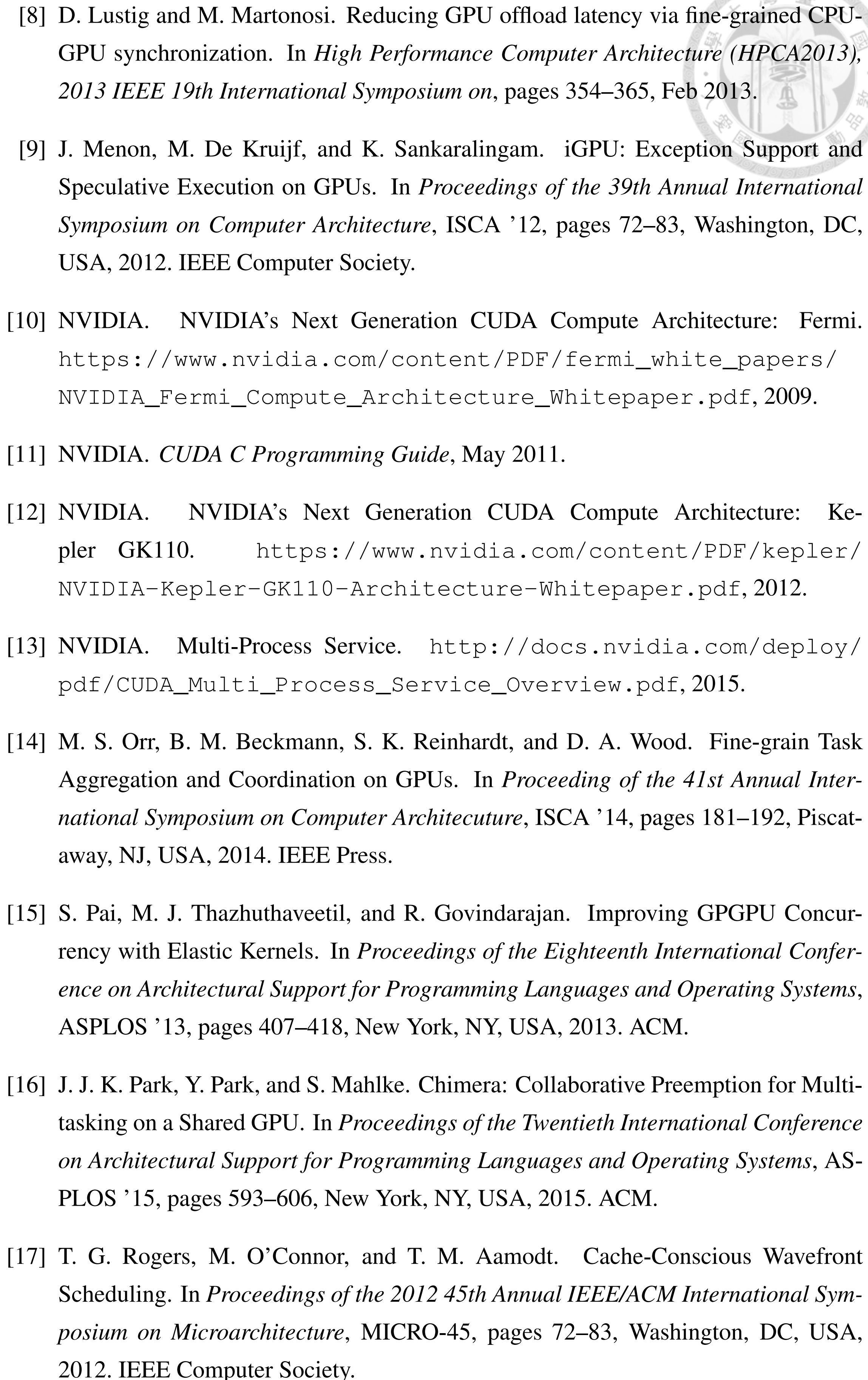
[4] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-Grained Resource Sharing for Concurrent GPGPU Kernels. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX.

[5] F. Ino, A. Ogita, K. Oita, and K. Hagihara. Cooperative Multitasking for GPU-Accelerated Grid Systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 774–779, May 2010.

[6] S. W. Kim, C.-l. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 2–11, New York, NY, USA, 2001. ACM.

[7] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 260–271, Feb 2014.

doi:10.6342/NTU201602299



[8] D. Lustig and M. Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 354–365, Feb 2013.

[9] J. Menon, M. De Kruijf, and K. Sankaralingam. iGPU: Exception Support and Speculative Execution on GPUs. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

[10] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

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

[12] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012.

[13] NVIDIA. Multi-Process Service. `http://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`, 2015.

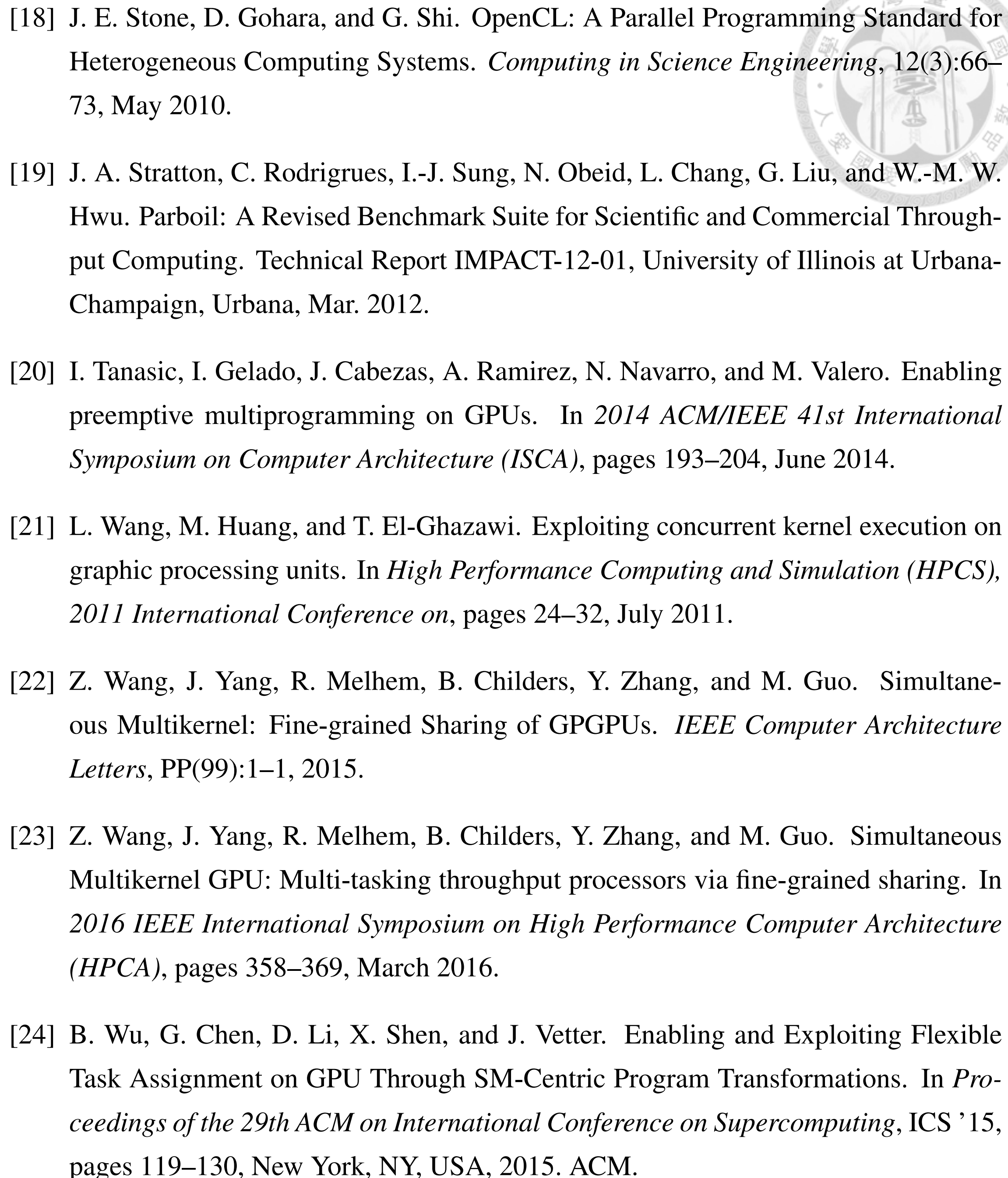
[14] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood. Fine-grain Task Aggregation and Coordination on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 181–192, Piscataway, NJ, USA, 2014. IEEE Press.

[15] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, New York, NY, USA, 2013. ACM.

[16] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 593–606, New York, NY, USA, 2015. ACM.

[17] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

doi:10.6342/NTU201602299



[18] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[19] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[20] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 193–204, June 2014.

[21] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 24–32, July 2011.

[22] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous Multikernel: Fine-grained Sharing of GPGPUs. *IEEE Computer Architecture Letters*, PP(99):1–1, 2015.

[23] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369, March 2016.

[24] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and Exploiting Flexible Task Assignment on GPU Through SM-Centric Program Transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 119–130, New York, NY, USA, 2015. ACM.

doi:10.6342/NTU201602299