ABSTRACT

LIN, ZHEN. Architectural Support for Efficient GPU Multiprogramming. (Under the direction of Dr. Huiyang Zhou).

Graphics processing units (GPUs) have become the most prevalent accelerator in high-performance computing. Since more and more applications are leveraging GPUs to accelerate their performance, the concurrent applications can potentially share the same GPUs. Therefore, there is an increasing demand to enable efficient GPU multiprogramming. In this paper, we study the GPU multiprogramming from two main aspects. First, we study efficient context switching techniques to enable kernel preemption on GPUs. Second, we study how to co-run multiple kernels on the same GPU so that the GPU resources can be optimally utilized. Specifically, this paper includes the following contents.

First, this paper presents an efficient support to enable fast context switching on GPUs. Context switching is a key technique enabling preemption and time-multiplexing for CPUs. However, for GPUs, it is challenging to support context switching due to the huge amount of architectural states to be swapped during context switching. The architectural state of GPUs includes registers, shared memory, single-instruction multiple-thread stacks and barrier states. Recent works present thread-block-level preemption on GPUs to avoid context switching overhead. However, because the execution time of a thread block (TB) is highly dependent on the kernel program. The response time of preemption cannot be guaranteed and some TB-level preemption techniques cannot be applied to all kernel functions. In this paper, we propose three complementary ways to reduce and compress the architectural states to achieve lightweight context switching on GPUs. Experiments show that our approaches can reduce the register context size by 91.5% on average. Based on lightweight context switching, we enable fast instruction-level preemption on GPUs with compiler and hardware co-design. With our proposed schemes, the preemption latency is reduced by 59.7% on average compared to the naive approach.

Second, we leverage the lightweight context switching approach to improve the performance of the single kernel execution. GPUs leverage massive thread-level parallelism (TLP) to achieve high
computation throughput and hide long memory latency. In our work, we characterize the kernels that are limited by their TLP levels. To improve TLP for such applications efficiently, we propose to use a fast context switching approach. When a warp/TB is stalled by a long latency operation, the context of the warp/TB is spilled to spare on-chip resource so that a new warp/TB can be launched. The switched-out warp/TB is switched back when another warp/TB is completed or switched out. With this fine-grain fast context switching, higher TLP can be supported without increasing the sizes of critical resources like the register file. Our experiment shows that the performance can be improved by up to 47% and a geometric mean of 22% for a set of applications.

Third, in the scenario of multiple kernels co-running on the same GPU, we propose to coordinate partition the TB and bandwidth resources for concurrent kernel execution (CKE). Contemporary GPUs support multiple kernels to run concurrently on the same streaming multiprocessors (SMs). Recent studies have demonstrated that CKE improves both resource utilization and computational throughput. Most of the prior works focus on partitioning the GPU resources at the TB level or the warp scheduler level to improve CKE. However, significant performance slowdown and unfairness are observed when latency-sensitive kernels co-run with bandwidth-intensive ones. The reason is that bandwidth over-subscription from bandwidth-intensive kernels leads to much aggravated memory access latency, which is highly detrimental to latency-sensitive kernels. In our work, we observe that such problems cannot be sufficiently solved by managing TB combinations alone. Then, we propose a coordinated approach for TB combination and bandwidth partitioning. Our approach partitions both bandwidth resources coordinately along with selecting proper TB combinations. The key objective is to allocate more TB resources for latency-sensitive kernels and more bandwidth resources to bandwidth-intensive kernels. Compared with two state-of-the-art CKE optimization schemes, SMK [Wan16] and [Xu16], our approach improves the average harmonic speedup by 78% and 39%, respectively. Even compared to the best possible CTA combinations, which are obtained from an exhaustive search among all possible CTA combinations, our approach improves the harmonic speedup by up to 51% and 11% on average.
Architectural Support for Efficient GPU Multiprogramming

by

Zhen Lin

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2019

APPROVED BY:

Dr. Gregory Byrd

Dr. Xipeng Shen

Dr. James Tuck

Dr. Huiyang Zhou
Chair of Advisory Committee
DEDICATION

To my Ph.D. advisor, Dr. Huiyang Zhou.
BIOGRAPHY

The author was born in Shijiazhuang, a northern city in China. He received his bachelor degree of Electronic Engineering from Yanshan University in 2010, and his master degree of Computer Science from Beihang University in 2014. He started his Ph.D. program in North Carolina State University in 2014. Since then, he has worked on several research projects under the supervision of Prof. Huiyang Zhou. His research mainly focused on GPU architecture and low-level software design. He had internships as a GPU software engineer at Apple Inc. and GPU architecture engineer at AMD Inc.
ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my advisor, Dr. Huiyang Zhou. Dr. Zhou has been a great advisor through my Ph.D. career. I am thankful for his encouragement, guidance, as well as his critical challenges. Because without them, this dissertation would not even be possible. Also, I would like to thank Dr. Huiyang Zhou, Dr. Gregory Byrd, Dr. Xipeng Shen, and Dr. James Tuck for serving on my dissertation committee. Thank Dr. Yan Solihin for his insightful discussions.

I would also like to thank the colleague students in our CAS group, Chao Li, Hongwen Dai, Hussein Elnawawy, Utkarsh Mathur, Mohammad Alshboul, Xiangyang Guo, and Qi Jia for their contribution and advice in my research. Also, many thanks to Michael Mantor from AMD, Lars Nyland and Jin Wang from NVIDIA for their valuable comments.

I am especially thankful for my wife, Wenshuang Hao, for her love and company during my studying. I would like to thank my mother, Hong Lyu and my father, Xihai Lin, for their love and support.
# TABLE OF CONTENTS

**LIST OF TABLES** ........................................................................................................... viii  
**LIST OF FIGURES** ......................................................................................................... ix  

## Chapter 1  INTRODUCTION .............................................................................................. 1  

## Chapter 2  Enabling Efficient Preemption for GPUs with Lightweight Context Switching  4  
  2.1  Introduction ............................................................................................................. 4  
  2.2  Background and Motivation ................................................................................... 7  
    2.2.1  Baseline Architecture ...................................................................................... 7  
    2.2.2  Prior Preemption Techniques for GPUs ............................................................ 8  
  2.3  Efficient Context Switching ..................................................................................... 9  
    2.3.1  In-Place Context Switching .............................................................................. 10  
    2.3.2  Architectural State Reduction ............................................................................ 12  
    2.3.3  Architectural State Compression ....................................................................... 14  
  2.4  Context Switching for Preemption ......................................................................... 19  
    2.4.1  Selective Preemption ......................................................................................... 19  
    2.4.2  Context Format .................................................................................................. 20  
    2.4.3  Preemption Pipeline ......................................................................................... 21  
  2.5  Experiments ............................................................................................................ 22  
    2.5.1  Methodology ..................................................................................................... 22  
    2.5.2  Spilling Latency ............................................................................................... 24  
    2.5.3  Preemption Latency ......................................................................................... 25  
    2.5.4  Worst Case Preemption Latency ....................................................................... 26  
    2.5.5  Impact of In-Place Context Switching ............................................................... 27  
  2.6  Related Work .......................................................................................................... 27  
  2.7  Conclusions ............................................................................................................. 29  

## Chapter 3  GPU Performance vs. Thread-Level Parallelism: Scalability Analysis and A Novel Way to Improve TLP  30  
  3.1  Introduction ............................................................................................................. 30  
  3.2  Background ............................................................................................................. 33  
    3.2.1  GPU Architecture ............................................................................................ 33  
    3.2.2  Crossbar Interconnect Network ....................................................................... 34  
  3.3  Methodology .......................................................................................................... 35  
    3.3.1  Simulation Infrastructure .................................................................................. 35  
    3.3.2  Benchmarks ..................................................................................................... 36  
  3.4  Dynamic Resource Utilization .............................................................................. 37  
    3.4.1  Experiment Description .................................................................................... 37  
    3.4.2  Overall Results ............................................................................................... 38  
    3.4.3  Scalability Analysis ......................................................................................... 40  
  3.5  Motivation for High TLP ....................................................................................... 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1</td>
<td>Which Benchmarks Benefit From Increased TLP</td>
<td>41</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Virtual Thread</td>
<td>42</td>
</tr>
<tr>
<td>3.6</td>
<td>Improving TLP with Fast Context Switching</td>
<td>43</td>
</tr>
<tr>
<td>3.6.1</td>
<td>General Idea</td>
<td>43</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Latency Hiding with Two-Level Context Switching</td>
<td>44</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Deadlock Avoidance</td>
<td>45</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Determining the Number of Extra TBs</td>
<td>46</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Warp States</td>
<td>48</td>
</tr>
<tr>
<td>3.6.6</td>
<td>GPU Duet Architecture</td>
<td>49</td>
</tr>
<tr>
<td>3.6.7</td>
<td>Context Switching Latency</td>
<td>52</td>
</tr>
<tr>
<td>3.6.8</td>
<td>Hardware Overhead Comparing to Virtual Thread</td>
<td>53</td>
</tr>
<tr>
<td>3.7</td>
<td>Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Impact of Maximum Extra TBs</td>
<td>53</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Comparing with Virtual Thread</td>
<td>55</td>
</tr>
<tr>
<td>3.7.3</td>
<td>TLP Improvement</td>
<td>56</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Impact of Context Switching Latency</td>
<td>57</td>
</tr>
<tr>
<td>3.8</td>
<td>Related Works</td>
<td>57</td>
</tr>
<tr>
<td>3.9</td>
<td>Conclusions</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Background on GPU Architecture</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>Methodology</td>
<td>64</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Simulation Specifications</td>
<td>64</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Benchmark Categorization</td>
<td>65</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Evaluation Metrics</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>Limitations of CTA Management</td>
<td>68</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Effects of Bursty Memory Requests</td>
<td>68</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Prioritization of Memory Requests</td>
<td>71</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Coarse Granularity of TLP Control</td>
<td>72</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Motivation of Our Approach</td>
<td>72</td>
</tr>
<tr>
<td>4.5</td>
<td>Coordinated CTA Combination and Bandwidth Partitioning</td>
<td>72</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Overview</td>
<td>72</td>
</tr>
<tr>
<td>4.5.2</td>
<td>CCBP Algorithm</td>
<td>74</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Bandwidth Management</td>
<td>77</td>
</tr>
<tr>
<td>4.5.4</td>
<td>CCBP-Parameter Detection</td>
<td>80</td>
</tr>
<tr>
<td>4.5.5</td>
<td>CCBP with Adjustable Priorities</td>
<td>82</td>
</tr>
<tr>
<td>4.6</td>
<td>Architectural Support</td>
<td>83</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Virtual Queues</td>
<td>84</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Request Arbitrator</td>
<td>84</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Tracking and Monitoring DRAM Access for df factors</td>
<td>85</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Overall Hardware Cost</td>
<td>85</td>
</tr>
<tr>
<td>4.7</td>
<td>Evaluation</td>
<td>86</td>
</tr>
</tbody>
</table>
4.7.1 2-Kernel Co-Runs ......................................................... 86
4.7.2 Impact on LS+BI Co-Runs ............................................ 88
4.7.3 Impact on NBI+DBI Co-Runs ........................................ 89
4.7.4 3- and 4-Kernel Co-Runs ............................................. 90
4.7.5 Comparison with Hybrid Approaches ............................... 91
4.7.6 Sensitivity Studies ....................................................... 92
4.8 Related Work ............................................................... 93
4.9 Conclusions ................................................................. 96

Chapter 5 Conclusion and Future Works .................................. 97

BIBLIOGRAPHY ..................................................................... 100
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>Baseline architecture configuration</td>
<td>22</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Benchmark specification</td>
<td>23</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Baseline architecture configuration</td>
<td>35</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Benchmark specification</td>
<td>36</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Throughput utilization classification</td>
<td>47</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>Storage overhead per SM</td>
<td>49</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Baseline architecture configuration</td>
<td>65</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Benchmark specification</td>
<td>65</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>An example of CCBP algorithm. Assume that, without bandwidth management, the CTA/NoC/DRAM consumption ratios for kernels K1, K2 and K3 are 2:1:1, 1:2:4 and 1:3:1.</td>
<td>76</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1 Baseline GPU architecture. ............................................. 7
Figure 2.2 Kernel code of persistent threads. .................................. 8
Figure 2.3 Occupancy of the register file and shared memory. ............ 10
Figure 2.4 Register and shared memory allocation tables before and after K1 is pre-
empted by K2. (a) Before. (b) After. ........................................... 11
Figure 2.5 Possible thread locations when a preemption point is reached. 12
Figure 2.6 Normalized register context sizes after liveness analysis and compression. . 13
Figure 2.7 A kernel code snippet of BP_1. ..................................... 14
Figure 2.8 Warp-Level register value locality analysis. ....................... 14
Figure 2.9 TB-Level register value locality analysis. ......................... 16
Figure 2.10 Warp-level register state compression logic. .................... 18
Figure 2.11 TB-level register state compression logic. ........................ 18
Figure 2.12 Kernel context format. .............................................. 20
Figure 2.13 Saving and restoring pipelines for preemption. ................. 21
Figure 2.14 Normalized spilling latency. ....................................... 24
Figure 2.15 Normalized preemption latency. ................................... 24
Figure 2.16 Normalized worst case preemption latency ..................... 26
Figure 2.17 Normalized preemption latency with in-place context switching. ... 27

Figure 3.1 Instruction per cycle (IPC), L1 D-cache hit rate and DRAM bandwidth utiliza-
tion with different numbers of active warps for the SPMV benchmark. .... 31
Figure 3.2 GPU architecture. ...................................................... 33
Figure 3.3 A crossbar with queueing on inputs, assuming that the input number equals
with the output number [Kar87]. ................................................. 34
Figure 3.4 Benchmark categorization based on throughput utilization of GPU resources. 37
Figure 3.5 The impact of varied numbers of active warps for different interconnect
bandwidth on SPMV. The left Y-axis is the throughput of L1 D-cache and
interconnect (icnt). The right Y-axis is the IPC. The X-axis is the number of
active warps. ........................................................................ 40
Figure 3.6 Speedup for doubling the context resources per SM. .......... 41
Figure 3.7 Occupancy of GPU context resources. ............................ 43
Figure 3.8 Context switching to overlap long-latency stalls. ............... 44
Figure 3.9 Warp state transition diagram. ..................................... 48
Figure 3.10 Block diagram of the GPUDuet architecture. ................... 49
Figure 3.11 Warp and TB context table entry. ................................ 49
Figure 3.12 Normalized speedups of GPUDuet with varied numbers of maximum extra
TBs (ME). ............................................................................ 54
Figure 3.13 Performance comparison of GPUDuet with Virtual Thread (VT) and TB-level
GPUDuet. ............................................................................ 55
Figure 3.14 TLP improvement by Virtual Thread (VT) and GPUDuet over the baseline
architecture. ........................................................................ 56
Figure 3.15 Impact of context switching latency. ........................................... 57
Figure 4.1 GPU architecture. ................................................................. 64
Figure 4.2 Performance and latency impact for BP+CFD co-run using different CTA combinations. ................................................................. 69
Figure 4.3 The instantaneous NoC bandwidth demand of BP+CFD co-runs with different CTA combinations. ................................................................. 70
Figure 4.4 Instantaneous DRAM bandwidth utilization for CFD+FTD co-run and FTD standalone execution. ................................................................. 71
Figure 4.5 A snapshot of the L1 D-cache miss queue when a latency-sensitive kernel (kernel_1) co-runs with a bandwidth-intensive one (kernel_2). ................. 71
Figure 4.6 An overview of the CCBP architecture. ........................................ 73
Figure 4.7 4 memory transaction types, black dots represent the bandwidth is consumed at a particular component. ........................................... 78
Figure 4.8 Architecture of an issue rate controller (IRC). ......................... 83
Figure 4.9 Overall performance of 2-kernel co-runs. ................................... 86
Figure 4.10 Harmonic speedup of 2-kernel co-runs. .................................... 86
Figure 4.11 Case studies of BP+CFD. ..................................................... 88
Figure 4.12 Bandwidth utilization of CFD+FTD. ....................................... 89
Figure 4.13 Performance of 3- and 4-kernel co-runs. ................................... 90
Figure 4.14 Comparison with hybrid approaches. ....................................... 92
Figure 4.15 Normalized harmonic speedups for varied queue sizes. ............... 92
Figure 4.16 Harmonic speedups with various warp scheduling policies. ........... 92
Nowadays, graphics processing units (GPUs) have become the most prevalent accelerator in high performance computing. GPUs have been widely used in various application fields, such as machine learning, computer vision, computational fluid dynamics, medical imaging, etc. Recently, companies such as IBM, Google, Amazon and NVIDIA are providing GPU clouds so that applications from different users/applications may potentially share one physical GPU card. Therefore, we see a strong demand for multiprogramming supports on GPUs.

The workload that is offloaded to a GPU is referred to as a kernel. Programmers use the CUDA [Cuda] or OpenCL [Ope] programming models to define the behavior of each thread in a kernel. A number of threads constitute a thread block (TB) and threads in a TB can communicate through the on-chip shared memory. There are two perspectives to support multiple kernels sharing one physical GPU. One is to enable preemption so that different GPU kernel can share the GPU in a time-multiplexing manner. The other perspective is to spatially partition the GPU resources and
allow multiple kernels running on the same GPU simultaneously. This dissertation presents our works on supporting the GPU multiprogramming from both perspectives.

In the first work, we propose a lightweight context switching approach to enable efficient preemption on GPUs. Similar to the CPUs, context switching requires the GPU to save all the architectural states of the running kernel to the memory so that the GPU can be take over by another kernel. The states will be restored to the GPU when the first kernel is scheduled to continue running. The challenge of supporting context switching on GPUs is that GPUs feature high amount of on-chip resources to accommodate a large amount of concurrent threads. For example, in the NVIDIA GK110 (Kepler) architecture, each stream multiprocessor (SM) has a 256KB register file and up to 48KB shared memory. Such large contexts result in long latency for context switching.

In our work, we propose novel ways to reduce and compress GPU contexts to enable lightweight context switching. Three approaches are proposed. First, based on the observation that for some applications, the on-chip resource is significantly underutilized, we propose in-place context switching, which means that not all resources need to be released/spilled to accommodate a new kernel. Second, liveness analysis is used to exclude dead registers so as to reduce the register context sizes. In this work, we observe the liveness of a vector register is dependent on the thread divergence. So the traditional liveness analysis algorithm is augmented for the GPU architecture. Third, based on register pattern analysis, register contexts can be further compressed. These techniques can greatly reduce the context size that needs to be swapped to/from off-chip memory.

Based on the lightweight context switching support, in our second work, we leverage the context switching to improve the performance of a single kernel. To achieve high computation throughput and memory bandwidth, GPUs exploit high degrees of thread-level parallelism (TLP). The GPU hardware aggregates multiple threads into a warp as the basic execution unit. The warp scheduler seeks for one ready instruction among multiple warps every cycle. Such fine-grain multithreading is the key to hide memory latency. As a result, GPUs feature high amounts of on-chip resources to accommodate the contexts of large numbers of concurrent warps.

We find that the number of concurrent warps is limited by the context capacity, such as the regis-
ter file capacity or warp scheduler capacity, of the GPU. We also noticed that, for many benchmarks, resource usage is unbalanced. Often, shared memory is underutilized and/or the L1 D-cache performance is low. For these benchmarks, shared memory or the L1 D-cache can be used to accommodate more warp contexts. In this work, we propose a novel approach using context switching as another level of multithreading for GPU architecture. The key idea is to switch out stalled warps/TBs to realize much higher degrees of TLP without increasing the size of critical physical resources. In order to achieve fast context switching, we only use on-chip memory to store the switched out contexts. Our experiment shows that the performance of single kernels can be significantly improved by our approach.

**In the third work,** we study the case of concurrent kernel execution (CKE), i.e. multiple kernels co-running on the same GPU. To deliver high throughput, GPUs incorporate a large amount of computational resources and support high memory bandwidth. However, the resource demands across different GPU kernels vary significantly, which may lead to saturation of certain resources and underutilization in others. Some prior works [Wan16; Xu16; Par17] have been proposed to solve the unbalanced resource utilization problem through concurrently executing multiple kernels with complementary characteristics. In their approaches, different methods are proposed to determine the TB combinations of different co-running kernels.

In our work, we highlight that memory interference can significantly affect the throughput and fairness of CKE. And we make a case that even the optimal TB combination does not eliminate the negative memory interference impact. To address this problem effectively, we propose a coordinated approach for TB combination and bandwidth partitioning. In our approach, we first dynamically detect the kernels as latency sensitive or bandwidth intensive. Then it effectively allocates the TB number and bandwidth resources for each co-running kernel based on their resource requirements. In our experiments, we observe that our approach achieves higher performance than the state-of-the-art CKE approaches and the oracle TB combination, which is the result of an exhaustive search of all possible TB combinations.
ELIBLING EFFICIENT PREEMPTION
FOR GPUS WITH LIGHTWEIGHT CONTEXT SWITCHING

2.1 Introduction

State-of-the-art GPUs exploit high degrees of thread-level parallelism (TLP). As a side effect, GPUs feature high amounts of on-chip resources to accommodate the contexts of the large numbers of concurrent threads. For example, in the NVIDIA GK110 (Kepler) architecture, each stream multiprocessor (SM) has a 256KB register file and up to 48KB shared memory. Such large contexts result in long latency for context switching (meaning switching in a new kernel rather than switching among
the running threads/warps, which GPUs support natively). To reduce the overhead, TB-level context switching techniques, SM-draining [Tan14] and SM-flushing [Par15a] have been proposed. The key idea of SM-draining is to wait for all running TBs on an SM to finish, then to launch the TBs from the new incoming kernel to the SM. The drawback of this solution is that the preemption latency can be very high. In the worst scenario, a TB can have a lifetime as long as the kernel [Gup12] [KB14], and the kernel may not be preempted at all. SM-flushing flushes the running TBs and then launches the new kernel. The limitation is that only kernels which conform the idempotent (re-executable) condition [KS11] can be preempted in this way. Also, the useful work is wasted when a running TB is flushed. To overcome such limitations, an integrated solution [Par15a] is proposed based on the progress of a TB. If it is close to the end, TB draining is used. If it just begins execution, TB flushing is employed instead. In other scenarios, the baseline context switching, i.e., swapping the thread contexts, is performed.

In this paper, we propose novel ways to reduce and compress GPU contexts to enable lightweight context switching. Three approaches are proposed. First, based on the observation that for some applications, the on-chip resource is significantly underutilized, we propose in-place context switching, which means that not all resources need to be released/spilled to accommodate a new kernel. Second, liveness analysis is used to exclude dead registers so as to reduce the register context sizes. In this paper, we observe the liveness of a vector register is dependent on the thread divergence. So the traditional liveness analysis algorithm is augmented for the SIMT architecture. Third, based on register pattern analysis, register contexts can be further compressed. The register pattern is explored in both warp-level and TB-level. These techniques can greatly reduce the context size that needs to be swapped to/from off-chip memory.

Based on the lightweight context switching, we use compiler and hardware co-design to enable instruction-level preemption for GPUs. The compiler analyzes the native assembly code to figure out the appropriate points for preemption. We introduce two new preemption instructions to annotate the preemption points, meaning that preemption is only enabled at this point. The preemption instruction checks the interruption signal and becomes a nop if there is no such a signal. If there
is an interrupt signal, the context switching is handled by a special hardware pipeline to reduce and compress the architectural states. At last, the compressed states are saved to global memory. To restore the kernel, the context will be loaded from global memory. After decompressing, the architectural states are restored on the processor.

Besides preemption, our proposed lightweight context switching can also be used for long running applications on supercomputers. The reason is that long running applications on supercomputers are error prone. Therefore, checkpointing mechanisms are commonly used such that the supercomputer can resume from a prior checkpoint in the case of an error. Our proposed scheme enables efficient context saving, i.e., efficient checkpointing of GPU contexts.

We evaluate our context switching enabled preemption approach with the Rodinia [Che09a] benchmarks. Our experiments show that the register context size can be reduced by 91.5% and the preemption latency can be reduced by 59.7% on average with our proposed lightweight context switching.

In summary, this paper makes the following contributions.

• We propose three techniques, in-place context switching, register liveness analysis, and register value compression to achieve lightweight context switching.

• A compiler and hardware co-design is proposed to enable instruction-level preemption for GPUs.

• We show that our proposed approach achieves low preemption latency.

The rest of the paper is organized as follows. Section 3.2 describes the SIMT architecture and motivates the proposed ideas. Section 2.3 presents the techniques for lightweight context switching. Section 2.4 makes use of lightweight context switching for efficient instructional-level preemption. Section 2.5 reports the methodology and experiment results. Section 2.6 discusses the related work and Section 2.7 concludes.
2.2 Background and Motivation

2.2.1 Baseline Architecture

Figure 2.1 shows the baseline GPU or GPU architecture. A GPU is composed of a number of streaming multiprocessors (SMs). The SMs share a multi-banked L2 cache. Typically, one or more L2 banks are backed up with a memory controller to communicate with off-chip memory. The SMs and multiple L2 banks communicate through a crossbar or an interconnect network. The on-chip memory in each SM includes shared memory, the register file and L1 D-cache. The basic execution unit in GPUs is a warp. A warp is a collection of threads that run in the single-instruction multiple-data (SIMD) style. Each warp has a private space in the register file. A per-warp SIMT stack keeps track of the program counters (PCs) of the threads when a divergent branch is encountered [Fun09]. One or more warps constitute a thread block (TB). All threads in one TB can synchronize and share data through shared memory. The threads in the same TB must be executed on the same SM and one SM can accommodate one or more TBs depending on the resource requirement of a TB.

When a kernel is launched, the resource requirement of a TB is provided to the GPU. Based on its available resource, an SM decides whether one more TB can be dispatched to it. There are four
void kernel(float A, float B, int Ahead, int Bhead, int N) {
    int in_id, out_id;
    while ((in_id = atomic_inc(Ahead)) < N) {
        float in_data = A[in_id];
        float out_data = do_work(in_data);
        out_index = atomic_inc(Bhead)
        B[out_index] = out_data;
    }
}

Figure 2.2 Kernel code of persistent threads.

There are several types of resources that can limit the number of concurrent TBs on an SM: the register file, shared memory, the warp scheduler, and the TB slots. For example, in the NVIDIA GT200 architecture, the register file size is 128KB, the shared memory size is 48KB, and there are 48 warp scheduler slots (i.e., up to 48 warps can run concurrently) and 8 TB slots on each SM. For a kernel with 8 warps (i.e., 256 threads) in each TB, if each warp takes 3KB register space and each TB takes 8KB shared memory space, the maximum TB per SM is 5 as limited by the register file size. A warp/TB will hold the resources during its whole lifetime. The resources will be released only after it finishes execution.

2.2.2 Prior Preemption Techniques for GPUs

The large context size on GPUs leads to high preemption overhead. To avoid the overhead, Tanasic et al. [Tan14] proposed the SM-draining technique, in which all current running TBs need to finish before releasing the SM for the new kernel. However, the SM-draining technique can cause long preemption latency. In the experiments from Park et al. [Par15a], the preemption latency of SM-draining can be as high as tens of milliseconds. A more extreme case, as shown in Figure 2.2, is a persistent kernel [Gup12] [KB14]. In this kernel, TBs only exit when all the input elements are processed. In other words, TBs may have the lifetime as long as the overall kernel execution time. Such kernels cannot be preempted with the SM-draining technique.

To address the long preemption latency, Park et al. [Par15a] proposed SM-flushing. Taking advantage of idempotent regions, the execution of the kernel can be stopped immediately and all
intermediate results can be flushed. The kernel can be resumed by relaunching the flushed TBs. SM-flushing only works on idempotent kernels, which means each TB can be re-executed multiple times without affecting the results. This is a strict limitation. Although Park et al. [Par15a] also proposed relaxed idempotent conditions, the scheme does not work on certain cases. For example, for the kernel shown in Figure 2.2, in each iteration, the variables Ahead, Bhead and an element in B will be modified. So it cannot be safely flushed. Moreover, when flushed, all the progress made on the TB is wasted.

Because the SM-draining latency may be too long and SM-flushing wastes useful work, Park et al. [Par15a] also used context switching for preemption. But the naive approach to swap all the occupied registers and shared memory incurs high overhead. For example, in the benchmark HS, the context size for each SM is about 140KB. For GTX480 with 15 SMs and the global memory bandwidth of 177GB/s, even if the global memory bandwidth is fully utilized, it would take at least 12 us to store such a large context. As pointed out in prior works [Tan14], the SMs are completely underutilized during context save and restore.

2.3 Efficient Context Switching

For context switching, in order to properly restore a warp or TB, its architectural states must be preserved. For a warp, the architectural state includes its registers and SIMT stack. The SIMT stack contains thread execution information in the case of divergent branches and also includes the program counters (PCs). For a TB, besides the contexts of all its warps, the architectural state also includes shared memory and barrier states, keeping the information on which warps have reached a barrier and are waiting for others. The SIMT stack and barrier states tend to be very small compared to registers and shared memory. For the SIMT stack, each entry has three 32-bit registers, which are the next PC, active mask and reconvergent PC [Fun09]. Based on the observation by Rhu et al. [RE13], the maximum stack depth is limited (11). So the maximum stack size is relatively small (132B). For barrier states, each barrier only needs 1 bit for each warp to record whether it has reached the barrier. Therefore, we focus on the context of registers and shared memory in this work. Using the BP_1
benchmark as an example, each thread has 13 registers (4B each) and each TB has 256 threads and 1128B shared memory. So the context size is 1664B per warp and 14440B per TB. Our goal is to make such context sizes much more manageable.

Next, we propose three complementary schemes, (a) in-place context switching to leverage unused resource, (b) register liveness analysis for architectural state reduction, and (c) value locality detection for architectural state compression.

2.3.1 In-Place Context Switching

As different applications exhibit different resource requirements, the fixed-size resources on GPUs are commonly underutilized. In Figure 3.7, we report the occupancy of both the register file and shared memory for different benchmarks. We can see that for most benchmarks, either (or both) type(s) of the resources is under-utilized. Take BP_1 as an example, the register file occupancy is 60.9% and shared memory occupancy is 13.7% as the occupancy is limited by the number of threads (or the maximal number of warps). Such resource under-utilization have also been observed in prior works [AMA13] [Geb12]. In this paper, we make use of such unused resource to store the context of the warps/TBs to be switched out.

On the baseline SIMT architecture, when the warps of a thread block are dispatched to an SM, their logic registers are mapped to physical registers. In this paper, an allocation table is used for managing the register allocation. Each launched TB reserves one entry in the table to denote the start
address and allocation size of the register file. When preemption occurs, the old kernel de-allocates the minimum number of TBs to make enough space for the new kernel. The remained TBs will keep reserving the register file. Such in-place context switching reduces the amount of data to be spilled and restored and enables fast preemption between two kernels. A similar but separate allocation table is used to manage the shared memory allocation.

Figure 2.4 (a) shows the register and shared memory allocation table when kernel K1 is running. K1 has 3 TBs on one SM, each TB allocates 300 vector registers and 8KB shared memory. In Figure 2.4 (b), K1 is preempted by K2, which launches 2 TBs per SM and each TB occupies 200 vector registers and 10KB shared memory. There are 1024 vector registers and 48KB shared memory in each SM on our baseline architecture. In this case, the register file deallocates and spills 2 TBs of K1 to accommodate K2, whereas none of shared memory needs to be deallocated.

In our implementation, each entry of register/shared memory table is 5B and the capacity for each of the tables is 16 entries. So the total overhead for the allocation tables is 160B.

A more aggressive option is to reallocate the dead register for the new kernel as proposed by H. Jeon et al. [Jeo15]. But such mechanism will increase the hardware complexity by introducing a register renaming table.

### Figure 2.4
Register and shared memory allocation tables before and after K1 is preempted by K2. (a) Before. (b) After.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Start Addr</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1_TB0</td>
<td>0</td>
<td>8KB</td>
</tr>
<tr>
<td>K1_TB1</td>
<td>8KB</td>
<td>8KB</td>
</tr>
<tr>
<td>K1_TB2</td>
<td>16KB</td>
<td>8KB</td>
</tr>
<tr>
<td>K2_TB0</td>
<td>0</td>
<td>8KB</td>
</tr>
<tr>
<td>K2_TB1</td>
<td>8KB</td>
<td>8KB</td>
</tr>
<tr>
<td>K2_TB2</td>
<td>16KB</td>
<td>8KB</td>
</tr>
<tr>
<td>K2_TB0</td>
<td>24KB</td>
<td>10KB</td>
</tr>
<tr>
<td>K2_TB1</td>
<td>34KB</td>
<td>10KB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Start Reg #</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1_TB0</td>
<td>0</td>
<td>300</td>
</tr>
<tr>
<td>K1_TB1</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>K1_TB2</td>
<td>600</td>
<td>300</td>
</tr>
<tr>
<td>K2_TB0</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>K2_TB1</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>K1_TB2</td>
<td>600</td>
<td>300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Start Addr</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1_TB0</td>
<td>0</td>
<td>8KB</td>
</tr>
<tr>
<td>K1_TB1</td>
<td>8KB</td>
<td>8KB</td>
</tr>
<tr>
<td>K1_TB2</td>
<td>16KB</td>
<td>8KB</td>
</tr>
<tr>
<td>K2_TB0</td>
<td>24KB</td>
<td>10KB</td>
</tr>
<tr>
<td>K2_TB1</td>
<td>34KB</td>
<td>10KB</td>
</tr>
</tbody>
</table>

---

11
2.3.2 Architectural State Reduction

We propose to use liveness analysis to reduce architectural register states. Liveness analysis reports that at any program point, which registers are defined and may be potentially used before the next re-define. Only the values in live registers need to be saved during context switching. At compile time, the compiler identifies live vector registers at each instruction and saves the results into a liveness table. Each entry in the liveness table corresponds to one static instruction and the register liveness information is encoded into a bit vector.

One option to provide the liveness bit vector at runtime is to load the liveness table to the GPU when the kernel is launched. At any point of execution, liveness registers can be looked up by the PC of a thread. The problem of such fine-granularity approach is that to store liveness table may take huge hardware resource. In our baseline architecture, the maximum register number is 64, meaning each liveness entry is 8B. For a program with 1K instructions, liveness table will take 8KB storage on hardware. To avoid the overhead of liveness table, we choose selective points to enable preemption. In other words, instead of enabling preemption for each instruction, we only enable preemption at certain selected program points. At each preemption point, a preemption instruction is inserted with encoded liveness bit vector. At runtime, the liveness bit vector is fetched to the instruction buffer. The details of selective preemption is discussed in Section 2.4.1.
In our approach, an entire warp will be stopped and handled by the preemption handler if any thread has reached the preemption point. In the case of thread divergence, the liveness of a whole warp may be different with the threads which reached the preemption point. As shown in Figure 2.5 are the possible thread locations when a preemption point is reached. For example, one warp has two threads, T1 and T2. Assume that T1 and T2 diverge at basic block A, T1 executes path B and T2 executes path C. When T2 reaches the preemption point at path C, T1 can be either at the divergence point (end of A) or the reconvergence point (start of D). In this case, the live vector register for the preemption point should be the union of all these 3 possible thread locations.

In our compiler, we firstly perform the traditional liveness analysis without considering thread divergence. Then the divergence and reconvergence points are analyzed based on immediate post-dominator [Fun09]. At last, the compiler calculates the union of liveness at the original preemption point, the divergence point and reconvergence point. Because the thread divergence can only be determined at runtime, both liveness vector versions are saved. When handling the preemption of a warp, the SIMT stack will be checked to determine whether there is a divergence. If there is, the union version is used. Otherwise, the original version is used.

To evaluate the effectiveness of liveness analysis, we count the number of live registers in the Rodinia benchmarks at runtime. When a warp reaches a preemption point, the total liveness number is accumulated. Then, the sum is divided by the number of warps. The result is shown in Figure 2.6. Take BP_1 as an example, only 39.6% of occupied registers are live ones. Therefore, the average
for (i = 1; i <= LOG_H; i++) {
    int pow = __powf(2, i);
    if (ty % pow == 0) {
        float tmp = s_weight[ty+pow/2][tx];
        s_weight[ty][tx] += tmp;
    }
    __syncthreads();
}

Figure 2.7 A kernel code snippet of BP_1.

Figure 2.8 Warp-Level register value locality analysis.

per-warp context size is reduced from 1664B to 656B. On average across all the kernels, 34.3% of the register context size can be reduced with liveness analysis.

2.3.3 Architectural State Compression

Register state compression is based on the observation that many register values in GPU programs conform certain patterns. S. Collange et al. [Col10] reports that uniform and strided are common patterns for GPU vector registers. A uniform register is defined as all scalar registers in a vector register have the same value, i.e. $V_i = a$. A strided register is defined as the scalar registers in a vector register conform arithmetic progression, i.e. $V_i = ai + b$. In this paper, we show that the TB dimension is an important factor for analyzing GPU register pattern. Also, we explore TB-level register compression to further exploit inter-warp data locality.

Warp-Level Compression

Warp-level register compression is used to leverage intra-warp value locality to compress vector
registers. Take the kernel code snippet of BP in Figure 2.7 as an example. BP_1 has 16x16 TB dimension, tx and ty are the thread ids in the X and Y dimension, respectively. The variable i is uniform across all threads in a warp, and so are the variable pow and the base addresses of array s_weight. However, for a warp with 32 threads, the values of ty for warp 0 is “0, 0, ..., 0, 1, 1, ..., 1”. The uniform pattern occurs for 16 scalar registers instead of the whole vector register. A similar situation happens for tx, which has the values “0, 1, ..., 15, 0, 1, ..., 15” for a warp. So, in this paper, the register pattern analysis is performed at the granularity of a pattern analyzing group (PAG). PAG is the minimum between the vector register width and the lowest TB dimension. In our baseline architecture, the vector register width is 32. So the maximum value of PAG is 32. When one of the TB dimensions is less than 32, PAG is the lowest TB dimension.

For vector registers containing uniform values, we can compress it into 1 scalar register. A strided vector register can be compressed to 2 scalar ones, i.e. a base and a stride. We refer to other registers as random ones, such as tmp in Figure 2.7.

To analyze the warp-level register value locality, we also take samples in the Rodinia benchmarks at runtime. For each sample, we count how many vector registers are uniform, strided or random in each warp. We only analyze the live registers. After execution, the average of all the samples is calculated. Figure 2.8 shows the result. For BP, 47.4% of its live registers are uniform and 52.6% live registers are strided. Therefore, the per-warp context size can be further reduced to 64B on average. From Figure 2.6, we can see that on average 91.5% register context size can be reduced with combined liveness analysis and warp-level register compression.

In our benchmark, BP, HS and LUD has two-dimensional TBs and the PAG is different with the vector register width. In Figure 2.8 we use two approaches for warp-level register pattern analysis. The default approach is to use PAG as the analysis width whereas the BP_vec, HS_vec and LUD_vec use the vector register width, i.e., the warp size, as the analysis width. From the result we can see that our approach can exploit uniform and strided registers more effectively.

Figure 2.10 shows the logic design for register state compression. The inputs are PAG scalar registers. There are PAG-1 subtractors to calculate the differences between two adjacent values. The
first subtraction result will be converted to a Boolean signal $a$, showing whether the difference is 0. The comparator takes the results of all subtractors and outputs 1 if all the results are equal, 0 otherwise. The output of the comparator is marked as signal $b$. This way, the signals $a$ and $b$ encode the value pattern, 01: uniform, 11: stride, 10/00: random. This logic is fully pipelined. Because the vector register width is 32 in our baseline architecture, the maximum of PAG is 32. In our experiments, the compression latency is assumed as 2 cycles. Such assumption is also used in a similar design for register compression [Lee15a].

The decompression process is relatively straightforward. For uniform registers, the value is duplicated PAG times for PAG registers in one warp. For strided registers, the first register takes the base value, and every following register adds the stride value to the former one.

**TB-Level Compression**

TB-level compression leverages inter-warp locality for vector registers. For example, in Figure 2.7, all threads across a TB has the same value of $i$ when they are in the same dynamic program point. Such registers are defined as global uniform. A local uniform register is the registers that have the same value for PAG threads, e.g. variable $ty$. Similarly, global strided registers are the registers that are strided across all threads in a TB. For example, the index of $s_{\text{weight}}$, which equals to $ty \times 16 + tx$, is global strided. Local strided registers are the registers that are strided for PAG threads but are not global strided, e.g. variable $tx$. 

![Figure 2.9 TB-Level register value locality analysis.](image)
For local uniform/strided registers, the compression is the same as the warp-level compression. For a global uniform register, only one scalar register will be saved for the whole TB. Only two scalar registers, base and stride, in a TB will be saved for a global strided register. So, for global uniform/strided registers, the compression ratio is higher than warp-level compression.

Figure 2.11 illustrates the TB-level compression logic. For each logic register, the physical registers of all warps in a TB are analyzed by the warp-level compressor. The registers will be identified as random if any physical vector register is random. Then the random registers bypass the compressor and spill to the global memory. If all the registers are not random, the base and stride are stored in the base or stride vector buffer. After all warps in the TB finished compressing, the pattern of base vector and stride vector are analyzed. The registers are global uniform if the base vector is uniform and the stride vector is zero-uniform, meaning the scalar registers are all zeros. The registers are local uniform if the stride vector is zero-uniform and the stride vector is not uniform. The registers are global strided if the base vector is strided, the stride vector is non-zero uniform, and the stride in the base vector is the same as the stride in the stride vector. Otherwise, the registers are local strided. In this analysis, the base vector and stride vector width equal to \( \frac{TB\_size}{PAG} \). The maximum TB size is 1024, the minimum PAG we support is 8, and the N in Figure 2.11 is 32 in our implementation.

The TB-level register pattern is shown in Figure 2.9. The approach we use to analysis TB-level register pattern is similar to warp-level register patterns except that TB-level is only enabled when the preemption point is a barrier. If the preemption point is not at a barrier, the liveness of different warps may be different and the register pattern becomes difficult to analysis. So only the benchmarks with barriers are shown in the result. In Figure 2.6, we apply TB-level compression at barrier preemption points and warp-level compression for other preemption points. From the result, we can see that TB-level compression can further reduce register context size by 36.1% on average. However, because TB-level compression may be worse on some benchmarks, e.g. PF, and it can only be applied on barriers, we choose not to use TB-level compression for our preemption design.
Figure 2.10 Warp-level register state compression logic.

Figure 2.11 TB-level register state compression logic.
2.4 Context Switching for Preemption

2.4.1 Selective Preemption

As long execution time of a TB mainly results from loops with large numbers of iterations, we insert one preemption point for each loop. For nested loops, only the innermost loop is considered. For loops with one barrier, the barrier will be selected as the preemption point. If any other point is selected as a preemption point, deadlocks may occur when some warps are waiting at the barrier while other warps reach the preemption points and wait for preemption. The barrier with minimum liveness is selected by the compiler if there are more than one barrier in the loop. For loops without barriers, the point with minimum liveness is selected. Outside the loops, we insert one preemption point every K instructions. Similar to the loops, either the minimum liveness point or the barrier is selected. If a kernel does not have a loop or a barrier and the kernel is smaller than K instructions, the execution time of a TB is small and our approach is essentially the same as SM-draining [Tan14].

In our experiment, because the execution time of all benchmarks is dominated by loops, the value of K does not have a great impact on the evaluation results when it varies from 100 to 1000.

We introduce two preemption point (pp) instructions, bar.pp and pp, to annotate the preemption points. After analyzing the preemption points, one preemption instruction is inserted to one point. For preemption point at barriers, bar.pp instruction is inserted to replace the original barrier instruction. bar.pp is a barrier instruction when the preemption signal is off. For the other preemption points, pp instructions are inserted into the program. The pp instruction becomes a nop when the preemption signal is off. When a preemption signal is on, warps keep running until a preemption point is reached. Then the warp stops and waits for preemption.

Both preemption instructions have one operand to provide the liveness bit vector for the program points at which the instructions are inserted. To follow the Fermi ISA format [Asf] [NVI], 10 bits are reserved as opcode. The remaining 54 bits are used as liveness bit vector. Because the architecture may support more than 54 registers, the highest bit is used to denote there are live registers that have higher register number than 53. All higher registers will be saved if such bit is set. In our benchmarks,
we observe that a 53-bit vector is typically enough for representing the liveness. To provide the liveness for thread divergence, a dummy instruction is introduced and it follows the pp instruction. It also encodes 54 bits for the liveness bit vector. At runtime, if thread divergence is detected at the preemption point, the liveness which is encoded in the dummy instruction is used for preemption. The bar.pp instruction doesn't need to be followed by the dummy instruction because the barrier ensures that there should be no divergence [NV15].

2.4.2 Context Format

Due to the in-place context switching as we discussed in Section 2.3.1, register file and shared memory can either be reserved on SM or dumped to global memory. In this paper, the context switching granularity is TB, meaning that the register file or shared memory of one TB cannot be partly spilled. But shared memory and that register file of one TB can reside in different locations, one in SM and the other in global memory, as illustrated in Figure 2.4.

As shown in Figure 2.12 is the context format of a kernel. The context control block (CCB) contains an array of TB context. The array size M is the maximum number of TBs that can be launched on the processor. Each entry contains the global memory pointers for the context of the register file, shared memory and SIMT stack. The pointer is NULL if the register file or shared memory is in-place reserved. Otherwise, a global memory space is allocated. The shared memory size on global memory is the same as the occupied size on GPU. The register file context on global memory has N entries,
Figure 2.13 Saving and restoring pipelines for preemption.

N equals to the warp number in one TB. The register context size for each warp is the maximum liveness number times vector register width. To maximize the bandwidth usage, the compressed register values are stored continuously. For decompressing, a pattern vector is used to store the register pattern and liveness of a warp. Two bits are used to represent the four states of each register. The four states are uniform, strided, random and dead. Because the maximum register number is 64, so the pattern vector length is 128 bits. In our paper, the whole SIMT stack will be saved to global memory. Because the SIMT stack includes the PC for each thread, the PCs are not separately saved. The warp is waiting at a barrier if the PC points to a barrier instruction, so the barrier state of each TB can also be derived from the SIMT stack.

2.4.3 Preemption Pipeline

Because the executions on different SMs are independent, a new kernel may preempt all SMs or only some of them. Here, we focus on preemption in one SM. When an SM receives an interrupt signal for preemption, the active warps keep executing until a preemption point is reached. Then the reached warp is set as inactive so that they will not fetch or issue new instructions. In order to preserve precise states, a warp must be drained before being switched out. A drained warp means that it has no issued instructions in the pipeline and has no pending updates to the register file.

As shown in Figure 2.13 is the spilling and restoring pipeline for preemption. For saving the context, the context saving logic looks up the register and shared memory allocation table, shown in Figure 2.4, to calculates how much resources (i.e. registers and/or shared memory) to be spilled
Table 2.1 Baseline architecture configuration

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. of SMs</td>
<td>15</td>
</tr>
<tr>
<td>SIMT core freq.</td>
<td>700MHz</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>SIMD width</td>
<td>32</td>
</tr>
<tr>
<td>Resources per SM</td>
<td>8 TB slots, 48 warp slots (1536 threads), 128KB register file, 48KB shared memory</td>
</tr>
<tr>
<td>Warp scheduler</td>
<td>2 schedulers, RR policy</td>
</tr>
<tr>
<td>L1 D-cache</td>
<td>16KB per SM, 128B block size</td>
</tr>
<tr>
<td>L2 cache</td>
<td>128KB per channel, 6 channels</td>
</tr>
<tr>
<td>DRAM</td>
<td>924MHz, QDR, 384-bit bus, peak bandwidth = (0.924 \times 4 \times 384 / 8) = 177GB/s</td>
</tr>
</tbody>
</table>

To global memory in order to accommodate the new kernel. Such information is converted to how many resident TBs to be spilled. To save the register of a warp, the liveness vector is fetched from the instruction buffer. Then the live vector registers are compressed and pushed into a buffer. Because the most efficient way to access global memory is by a width of 128B, the compressed data form data segments with the size of 128B through the buffer. After the register states of all warps from one TB are drained, shared memory used by this TB starts to be spilled to global memory.

To restore a TB, the restoring logic waits for there is enough on-chip resource to launch the TB. Then the context control block, shown in Figure 2.12, is accessed to find the TB context. To restore the registers of a warp, the pattern vector is firstly loaded. Then each vector register is decompressed based on its pattern.

2.5 Experiments

2.5.1 Methodology

We implemented our lightweight context switching on GPGPU-sim [Bak09] v3.2.2. Our baseline architecture models the NVIDIA GTX480 GPU, and its configuration is shown in Table 2.1. GPGPU-sim supports both the PTX and GT200 instruction set architecture (ISA). PTX is for a virtual machine with unlimited registers. Therefore, in order to collect the accurate architectural register information, all benchmarks are compiled to the GT200 ISA. We evaluate our techniques on the Rodinia [Che09a]
Table 2.2 Benchmark specification

<table>
<thead>
<tr>
<th>Kernel (Label)</th>
<th>Benchmark (Label)</th>
<th>Warps/TB</th>
<th>TBs/SM</th>
<th>Regs/Warp</th>
<th>Smem/TB (bytes)</th>
<th>Limiting Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>layerforward (BP_1)</td>
<td>backprop (BP)</td>
<td>8</td>
<td>6</td>
<td>13</td>
<td>1128</td>
<td>warp</td>
</tr>
<tr>
<td>adjust_weight (BP_2)</td>
<td>backprop (BP)</td>
<td>8</td>
<td>6</td>
<td>18</td>
<td>40</td>
<td>warp, reg</td>
</tr>
<tr>
<td>Kernel1 (BFS_1)</td>
<td>bfs (BFS)</td>
<td>16</td>
<td>3</td>
<td>7</td>
<td>44</td>
<td>warp</td>
</tr>
<tr>
<td>Kernel2 (BFS_2)</td>
<td>bfs (BFS)</td>
<td>19</td>
<td>3</td>
<td>4</td>
<td>36</td>
<td>warp</td>
</tr>
<tr>
<td>findRangeK (BT_1)</td>
<td>b+tree (BT)</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>48</td>
<td>warp</td>
</tr>
<tr>
<td>findk (BT_2)</td>
<td>b+tree (BT)</td>
<td>8</td>
<td>6</td>
<td>9</td>
<td>60</td>
<td>warp</td>
</tr>
<tr>
<td>initialize_variable (CFD_1)</td>
<td>cfd (CFD)</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>32</td>
<td>warp</td>
</tr>
<tr>
<td>compute_step_factor (CFD_2)</td>
<td>cfd (CFD)</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>48</td>
<td>warp, TB</td>
</tr>
<tr>
<td>compute_flux (CFD_3)</td>
<td>cfd (CFD)</td>
<td>6</td>
<td>4</td>
<td>39</td>
<td>38</td>
<td>reg</td>
</tr>
<tr>
<td>copySrcToComponets (DWT_1)</td>
<td>dwrt2d (DWT)</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>894</td>
<td>warp</td>
</tr>
<tr>
<td>fdwrt53Kernel (DWT_2)</td>
<td>dwrt2d (DWT)</td>
<td>6</td>
<td>5</td>
<td>32</td>
<td>8668</td>
<td>reg, smem</td>
</tr>
<tr>
<td>kernel (HW_1)</td>
<td>heartwall (HW)</td>
<td>8</td>
<td>4</td>
<td>23</td>
<td>11888</td>
<td>smem</td>
</tr>
<tr>
<td>calculate_temp (HS_1)</td>
<td>hotspot (HS)</td>
<td>8</td>
<td>4</td>
<td>31</td>
<td>3144</td>
<td>reg</td>
</tr>
<tr>
<td>histogram1024 (HG_1)</td>
<td>hybridsort (HG)</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>12324</td>
<td>smem</td>
</tr>
<tr>
<td>invert_mapping (KM_1)</td>
<td>kmeans (KM)</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>32</td>
<td>warp</td>
</tr>
<tr>
<td>GICOV_kernel (LK_1)</td>
<td>leukocyte (LK)</td>
<td>6</td>
<td>8</td>
<td>18</td>
<td>24</td>
<td>warp, TB</td>
</tr>
<tr>
<td>lud_diagonal (LUD_1)</td>
<td>lud (LUD)</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>2076</td>
<td>TB</td>
</tr>
<tr>
<td>lud_perimeter (LUD_2)</td>
<td>lud (LUD)</td>
<td>1</td>
<td>8</td>
<td>16</td>
<td>3104</td>
<td>TB</td>
</tr>
<tr>
<td>lud_internal (LUD_3)</td>
<td>lud (LUD)</td>
<td>8</td>
<td>6</td>
<td>9</td>
<td>1056</td>
<td>warp</td>
</tr>
<tr>
<td>reduce (SR_1)</td>
<td>srad_v1 (SR)</td>
<td>16</td>
<td>3</td>
<td>14</td>
<td>4132</td>
<td>warp</td>
</tr>
<tr>
<td>srad (SR_2)</td>
<td>srad_v1 (SR)</td>
<td>16</td>
<td>3</td>
<td>16</td>
<td>128</td>
<td>warl</td>
</tr>
<tr>
<td>dynproc_kernel (PF_1)</td>
<td>pathfinder (PF)</td>
<td>8</td>
<td>6</td>
<td>12</td>
<td>2096</td>
<td>warp</td>
</tr>
<tr>
<td>kernel_compute_cost (SC_1)</td>
<td>streamcluster (SC)</td>
<td>16</td>
<td>3</td>
<td>8</td>
<td>56</td>
<td>warp</td>
</tr>
</tbody>
</table>

benchmarks. Table 4.2 lists all the benchmarks. Each entry in Table 4.2 shows the information of a kernel. Because some benchmarks (e.g., BP) contain multiple kernels, (e.g., BP_1 and BP_2), we combine the results of these kernels in our evaluation.

We model the potential traffic contention due to context switching at register read/write ports, shared memory read/write ports, the interconnect to memory controllers, and memory read/write bandwidth. For preemption, we found that the contention is limited as the SM essentially stops execution and all the ports are used for context swapping. The context switching requests have lower priority than regular requests from instruction execution.

To evaluate the preemption performance, we add periodic preemption signals (every 10000 cycles) when running the benchmarks. We run each benchmark for at most 200 million cycles or until it exits. When a preemption signal is received, one SM stops running and spill its architectural states to global memory while other SMs keeps running, the same method as used in prior works [Par15a].
2.5.2 Spilling Latency

In Figure 2.14, we evaluate the normalized latency for spilling the architectural states to global memory. Three approaches are compared to show the effectiveness of liveness analysis and register compression. ‘Occup’ shows the latency to spill all the occupied architectural states. The spilling latency is measured from interrupt signal is issued to all the states are spilled. ‘Live’ is to spill only the live registers and ‘live+cp’ is to spill the live registers with warp-level register compression. For ‘live’ and ‘live+cp’, the latency for the spilling registers of a warp starts from the preemption point is reached. Then the latencies of all warps are accumulated. For each mechanism, the normalized latency to spill register file, shared memory and SIMT stack are evaluated.

If we see the results of ‘Occup’, the spilling latency of most benchmarks (except for HG) is dominated by spilling the registers. This is because the register file is the largest on-chip memory that stores the architectural states and it has relatively high occupancy (Figure 3.7). For DWT, HW and HG, shared memory accounts for more than 20% of spilling latency because the shared occupancy...
for these benchmarks is high (Figure 3.7). For most benchmarks, the SIMT stack spilling latency is too small to be observed. The SIMT stack latency appears for BFS because its live register number is very small.

From the results, we can see that liveness analysis and register compression drastically reduce the latency to dump the register state. Since shared memory size is not reduced by these two mechanisms, the latency for saving shared memory states is similar for different approaches. The geometric mean for spilling total architectural states is reduced to 17.7%. With the GPU core frequency of 700Hz, the average spilling latency is reduced from 9.9us to 1.8us. In the special case of BFS, the spilling latency is increased with ‘live+cp’ compared to ‘live’. The reason is that no live registers can be compressed at the preemption points and our preemption mechanism needs to store metadata for the compression patterns of the registers. Therefore, the data to be saved become larger for BFS when compression is enabled.

### 2.5.3 Preemption Latency

The preemption latency evaluation is shown in Figure 2.15. The results are normalized with spilling all occupied architectural states to global memory. ‘Select’ shows the selective preemption latency with register liveness analysis and compression. The total preemption latency is measured from the start of preemption signal to the architectural states are spilled. The latency labeled as ‘spill’ is the context spilling latency. With selective preemption, the warps keep executing until preemption points are reached. So, for some time the spilling pipeline is idle to wait for warps reaching the preemption points. Such latency is called draining latency and labeled as ‘drain’. Because SM-draining lets all current TBs to finish, it doesn't need to save any architectural states for these TBs.

Compared with the baseline, the preemption latency is reduced to 40.3% on average (geometric mean). With the GPU core frequency of 700Hz, the preemption latency is reduced from 9.9us to 4.0us. The draining latency accounts for 55.8% of the selective preemption latency. Note that during draining, some (if not all) warps are still doing useful work.
For SM-draining, although the SM keeps doing useful work during preemption, the latency becomes unbearable for many benchmarks. For example, the average preemption latency for LK is 1431.1us. The newly incoming kernel would have to wait for such long TBs. As a result, fairness cannot be guaranteed with SM-draining because it favors kernels with long TBs. With selective preemption, because the preemption is guaranteed to be done in every loop iteration or every 1000 instructions, the draining latency is much more manageable.

### 2.5.4 Worst Case Preemption Latency

Because selective preemption has to wait for the warps to execute some instructions before being spilled, the latency variation may become higher than naive approach. To evaluate the preemption latency in the worst case scenario, we select 12 kernels which can run long enough to generate 15 times preemption signals. As shown in Figure 2.16, for each mechanism, the worst case preemption latency is normalized to its average latency. From the results, we can see that the naive approach, which is saving all occupied states, has 0.4x difference between average and worst case scenario. The difference may result from the different instructions to drain before preemption or the difference of memory traffic. For selective preemption, the difference between average and worst case scenario is 0.6x, which is slightly higher than the naive approach. The worst case latency for SM-draining is 2.4x compared with the average. For SM-draining, the worst case happens when an interrupt is signaled when a new TB has just being launched.
2.5.5 Impact of In-Place Context Switching

To evaluate in-place context switching, we randomly pick 8 pairs of kernels which are labeled as ‘kernel1-kernel2’ in Figure 2.17. We assume that ‘kernel1’ is preempted by ‘kernel2’ and measure the switching out latency for ’kernel1’. The baseline, which is labeled as ‘complete’, is the preemption latency of the approach using both liveness analysis and compression. For in-place context switched warps, they still have to reach the preemption point until being handled by preemption pipeline. As a result, the warps still need to be drained even if there is no register to spill, e.g. BT_1-HG_1.

From the figure, we can see that the latency for spilling the register and shared memory states can be further reduced with our proposed in-place context switching, by 21.5% on average. On some benchmarks, e.g. BT_1-HG, the draining latency is higher on in-place context switching. This is because spilling can hide the latency for some warps to drain.

2.6 Related Work

On CPUs, there are many works focusing on context reduction to reduce the preemption overhead and improve processor utilization. Some works [Sny95] [ZP06] propose to seek program points with small numbers of live registers for context switching, thereby reducing the context switching latency. Register relocation [WW93] is used to partition the register file into variable-size contexts. The more-often resident contexts are allowed to stay on the processor. Switching between resident...
contexts is very fast, and multiple contexts can tolerate long latencies from cache misses.

To enable fast context switching and exception handling on GPUs, iGPU [Men12] partitions kernel code into idempotent regions and each region is a recovery point. iGPU also leverages liveness analysis when formatting recovery points for context reduction. Register liveness is also used for dynamic register file management [Jeo15]. Lee et al. [Lee15a] leverage register compression for reducing GPU power. They use the base-delta-immediate (BDI) compression algorithm [Pek12] for register file compression. BDI separates a vector register into several trunks and stores the value of first chunk and the delta between adjacent chunks. As delta values tend to be very small, they can be stored in small bins. The compression technique is used for the register file and every register read/write needs to be decompressed/compressed. In comparison, we only perform compression/decompression when a context is spilled/restored.

Some recent works aim to enable the preemption on GPU. RGEM [Kat11] is a user-space solution to reduce the response time of high priority kernels. It splits the input data into multiple chunks so that a kernel can be preempted at a chunk boundary. PKM [BK12] partitions the overall TBs of a kernel into multiple sets where each set has a specific number of TBs. Softshell [Ste12] is a GPU programming model which supports a kernel being preempted at the boundary of TBs. In comparison, our proposed approaches enable efficient preemption at the instruction granularity.

Concurrent kernel execution is another option to support GPU sharing by multiple kernels. KernelMerge [Gre12] and Spacial Multiplexing [Adr12b] study how to use concurrent kernels to better utilize GPU resources and improve overall throughput. Elastic kernel [Pai13] increases GPU utilization by issuing concurrent kernels on one SM. After TBs from one kernel are issued to the SM, the spare resources are distributed to another kernel. In [Lee14], Lee et al. also leverage mixed concurrent kernels to improve GPU utilization.

Compared to these prior works, the novelty of our work includes (a) fast context switching through context reduction and compression (b) efficient instruction-level GPU preemption.
2.7 Conclusions

In this paper, we present lightweight context switching for GPUs and compiler-hardware co-design to enable efficient preemption. We propose three schemes, in-place context switching, liveness analysis and register compression, to address the problem of the large kernel context on GPUs. Our results show that with register liveness analysis and compression, the register context can be reduced drastically by 91.5%. With selective preemption enabling instructions, we can achieve efficient instruction-level preemption with an average preemption latency of 4.0us (with the 700MHz GPU core frequency).
CHAPTER 3

GPU PERFORMANCE VS. THREAD-LEVEL PARALLELISM: SCALABILITY ANALYSIS AND A NOVEL WAY TO IMPROVE TLP

3.1 Introduction

State-of-the-art throughput-oriented processors, like GPUs, have become the dominant accelerator for data-parallel workloads. To achieve high computation throughput and memory bandwidth, GPUs exploit high degrees of thread-level parallelism (TLP). Programmers use the CUDA [Cuda] or OpenCL [Ope] programming models to define the behavior of each thread. The GPU hardware aggregates multiple threads into a warp as the basic execution unit. The warp scheduler seeks for
Figure 3.1 Instruction per cycle (IPC), L1 D-cache hit rate and DRAM bandwidth utilization with different numbers of active warps for the SPMV benchmark.

one ready instruction among multiple warps every cycle. Such fine-grain multithreading is the key to hide the memory latency. As a result, GPUs feature high amounts of on-chip resources to accommodate the contexts of large numbers of concurrent warps. For example, in the NVIDIA GP100 (Pascal) architecture [Pas], each stream multiprocessor (SM) has a 256KB register file and accommodates up to 64 warps.

The inclusion of multiple levels of caches complicates the relationship between TLP and the overall performance. As studied in prior works [Rog12] [Kay13] [Li15], high degrees of TLP cause the cache to suffer from the contention problem, which may lead to performance degradation. For example, Figure 3.1 shows the impact of cache thrashing with various degrees of TLP. The experiment setup is presented in Section 4.3. The figure shows that the L1 D-cache hit rate dramatically decreases when the number of concurrent warps increases from 1 to 24. Cache thrashing causes the performance to drop when the warp number is larger than 16. T. Rogers et al. [Rog12] propose to limit the number of active/concurrent warps to alleviate the cache thrashing problem. However, as the warp number increases from 32 to 64, more TLP should have hidden more latency for accessing memory since the cache performance does not significantly drop. But why isn't the performance improved? Some may conjecture that it is because the off-chip DRAM bandwidth is fully utilized but Figure 3.1 shows that the DRAM bandwidth is fairly underutilized.

In order to find this mysterious performance limitation, we conduct a novel and detailed analysis from the perspective of throughput utilization of GPU components. The result shows that 8 out of
22 benchmarks are actually bounded by the interconnect bandwidth in the baseline architecture. The performance starts to drop when the interconnect throughput is saturated so that higher TLP cannot compensate the loss in cache throughput.

In our experiments, we find that some benchmarks do not fully utilize the throughput of any computation or memory bandwidth resource. These benchmarks would benefit from a higher degree of TLP. However, the number of concurrent warps is limited by the context capacity, such as the register file capacity or warp scheduler capacity, of the GPU. We also noticed that, for many benchmarks, resource usage is unbalanced. Often, shared memory is underutilized and/or the L1 D-cache performance is low. For these benchmarks, shared memory or the L1 D-cache can be used to accommodate more warp contexts.

In this paper, we propose GPUDuet, a novel approach using context switching as another level of multithreading for GPU architecture. The key idea is to switch out stalled warps/thread blocks (TBs) to realize much higher degrees of TLP without increasing the size of critical physical resources. In order to achieve fast context switching, we only use on-chip memory to store the switched-out contexts. Through monitoring the throughput resource utilization, GPUDuet dynamically determines the best TLP degrees.

We evaluate our GPUDuet approach with 10 throughput-unsaturated benchmarks and 5 throughput-saturated ones. We find that it can significantly improve the performance for benchmarks without saturated resources. Compared to the baseline, our technique achieves up to 47% and an average (geometric mean) of 22% performance gain. Compared to the state-of-the-art TLP improvement scheme, Virtual Thread (VT) [Yoo16], our proposed scheme achieves 12% higher performance on average and 16% for unsaturated benchmarks.

In summary, this paper makes the following contributions:

- We present a novel bottleneck analysis on GPU workloads from a resource utilization perspective. It reveals that the interconnect can be a critical bound in GPU performance and provides a detailed categorization of GPU workloads.

- Based on the resource utilization analysis, we reveal insights on GPU performance scalability.
We also characterize the features of GPU workloads, whose performance can scale well with TLP.

- For the benchmarks whose performance scales with TLP, we propose lightweight context switching as another level of multithreading support to improve TLP and our experiments show that our proposed TLP improvement technique can significantly enhance the performance.

The rest of the paper is organized as follows. Section 3.2 provides the background. Section 4.3 describes our experimental methodology. Section 3.4 presents the throughput utilization study and performance scalability analysis. Section 3.5 motivates our context switching approach to improve TLP. Section 3.6 details our GPUDuet approach. Section 3.7 reports the experiment results. Section 3.8 addresses the related work. Section 3.9 concludes the paper.

### 3.2 Background

#### 3.2.1 GPU Architecture

Contemporary GPU architecture, as illustrated in Figure 4.1, consists of multiple stream multiprocessors (SMs). Each SM has a private L1 D-cache, which uses an array of miss status handling registers (MSHRs) to support multiple outstanding cache misses. All SMs share a multi-banked L2 cache, which also uses MSHRs to send memory requests to the off-chip DRAM. The L1 D-caches and L2 cache banks communicate through a crossbar interconnect network [AMD12][Onl].
Each SM hosts a number of warps, each of which is a collection of threads that run in the single-instruction multiple-data (SIMD) manner. Each warp has its private space in the register file and some meta-data, such as the program counter and active thread bit mask, in the warp scheduler to keep track of its execution status. Multiple warps constitute a thread block (TB), within which all warps can communicate and synchronize through shared memory. One SM can accommodate one or more TBs depending on their resource requirement. There are four types of resources that can limit the number of concurrent/active TBs on an SM: register file, shared memory, the warp scheduler slots and the TB slots. The warp/TB slots include some meta data to keep track of the warp/TB execution status. The warp/TB slot size is much smaller than register file or shared memory [Yoo16]. A warp/TB will hold the resources during its whole lifetime. The resources will be released after it finishes execution.

### 3.2.2 Crossbar Interconnect Network

State-of-the-art GPUs use crossbars as the interconnect network to connect the SMs and L2 partitions to provide the tremendous data communication demand between them [AMD12][Onl]. The crossbar is a non-blocking interconnect network, which means a connection can be established if both input and output are idle. In this paper, queueing on inputs (Figure 3.3a) is assumed for simplicity. If more than one packet needs to arrive at the same output at the same time, only one packet can be accepted.
by the network and other packets have to be queued on their inputs. Due to such output contention, the sustainable throughput of the crossbar can hardly achieve the peak bandwidth. Based on the assumption that the destinations of packets are independent and uniformly distributed, M. Karol et al. [Kar87] quantified the ratio of sustainable throughput and the peak bandwidth, as shown in Figure 3.3b. As we can see from Figure 3.3b, the ratio drops sharply as the number of inputs/outputs increases and saturates around 60% quickly.

### 3.3 Methodology

#### 3.3.1 Simulation Infrastructure

We use GPGPU-sim v3.2.2 to investigate the GPU throughput utilization and evaluate our context switching approach. Our baseline architecture models the NVIDIA Maxwell architecture and its configuration is shown in Table 4.1. To get the register and shared memory usage based on Maxwell architecture, we use the nvcc compiler with `-ptxas-options=-v -arch=sm_50` option. We observe that nvcc typically allocates more registers for the Maxwell architecture than for the older architectures, such as Fermi. This configuration is important as it affects how the TLP of a kernel is limited. If we keep using the code generated for the old architecture, e.g., Fermi, the register usage of a kernel is smaller than it would be for the Maxwell architecture. In this case, the kernel is more likely to be scheduling limited (due to the limited warp slots) although it should be capacity limited (due to the limited register file size) if we compile the code for the Maxwell architecture.

We implement the crossbar interconnect based on Section 3.2.2. The crossbar port width can
### Table 3.2 Benchmark specification

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Description</th>
<th>Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>bfs [Che09b]</td>
<td>Breath first search</td>
<td>16 million nodes</td>
</tr>
<tr>
<td>BP_1</td>
<td>backprop [Che09b]</td>
<td>Machine learning</td>
<td>65536 nodes</td>
</tr>
<tr>
<td>BP_2</td>
<td>backprop [Che09b]</td>
<td>Machine learning</td>
<td>65536 nodes</td>
</tr>
<tr>
<td>BT</td>
<td>b+tree [Che09b]</td>
<td>Graph traversal</td>
<td>1 million nodes</td>
</tr>
<tr>
<td>CFD</td>
<td>cfd [Che09b]</td>
<td>Fluid dynamics</td>
<td>0.2 million nodes</td>
</tr>
<tr>
<td>CONV</td>
<td>2DCONV [GG12]</td>
<td>2-D convolution</td>
<td>4096*4096 matrix</td>
</tr>
<tr>
<td>CP</td>
<td>cutcp [Str12]</td>
<td>Coulombic potential</td>
<td>large (96602 nodes)</td>
</tr>
<tr>
<td>DWT_1</td>
<td>dwt2d [Che09b]</td>
<td>Image/video compression</td>
<td>1024*1024 rgb image</td>
</tr>
<tr>
<td>DWT_2</td>
<td>dwt2d [Che09b]</td>
<td>Image/video compression</td>
<td>1024*1024 rgb image</td>
</tr>
<tr>
<td>FDTD_1</td>
<td>FDTD-2D [GG12]</td>
<td>2-D stencil operation</td>
<td>2048*2048 nodes</td>
</tr>
<tr>
<td>FDTD_2</td>
<td>FDTD-2D [GG12]</td>
<td>2-D stencil operation</td>
<td>2048*2048 nodes</td>
</tr>
<tr>
<td>GEMM</td>
<td>GEMM [GG12]</td>
<td>Linear algebra</td>
<td>512*512 matrices</td>
</tr>
<tr>
<td>HG</td>
<td>hybridsort [Che09b]</td>
<td>Sorting algorithms</td>
<td>4 million elements</td>
</tr>
<tr>
<td>HS</td>
<td>hotspot [Che09b]</td>
<td>Physics simulation</td>
<td>512*512 nodes</td>
</tr>
<tr>
<td>KM</td>
<td>kmeans [Che09b]</td>
<td>Data mining</td>
<td>494020 nodes</td>
</tr>
<tr>
<td>LBM</td>
<td>lbm [Str12]</td>
<td>Fluid dynamics</td>
<td>long</td>
</tr>
<tr>
<td>MM</td>
<td>2MM [GG12]</td>
<td>Linear algebra</td>
<td>2048*2048 matrices</td>
</tr>
<tr>
<td>MMS</td>
<td>matrixMul [Cudb]</td>
<td>Matrix multiplication</td>
<td>1024*1024 matrices</td>
</tr>
<tr>
<td>SAD_1</td>
<td>sad [Str12]</td>
<td>Sum of absolute differences</td>
<td>default</td>
</tr>
<tr>
<td>SAD_2</td>
<td>sad [Str12]</td>
<td>Sum of absolute differences</td>
<td>default</td>
</tr>
<tr>
<td>SPMV</td>
<td>spmv [Str12]</td>
<td>Sparse matrix vector multiply</td>
<td>large</td>
</tr>
<tr>
<td>ST</td>
<td>stencil [Str12]</td>
<td>3-D stencil operation</td>
<td>default</td>
</tr>
</tbody>
</table>

be configured as 16B, 32B or 64B. We use 16B wire width (307 GB/s bandwidth) in Figure 3.1 and 32B wire width (614 GB/s bandwidth) if not specified. When multiple inputs are trying to send to the same output, the crossbar randomly selects one input to connect with the output and the other ones wait for the selected one to finish. Our experiment shows that, for interconnect-intensive benchmarks, the attainable throughput saturates close to 60% of peak bandwidth, confirming the study by M. Karol et al. [Kar87].

### 3.3.2 Benchmarks

We choose 22 GPU kernels from various benchmark suites, including Rodinia [Che09b], Parboil [Str12], Polybench [GG12] and CUDA SDK [Cudb], to investigate the throughput utilization on GPU components. Table 4.2 lists the details of the benchmarks along with their sources and input data sets.
Figure 3.4 Benchmark categorization based on throughput utilization of GPU resources.

3.4 Dynamic Resource Utilization

3.4.1 Experiment Description

In this section, we investigate the utilization of six GPU components, including the warp scheduler throughput, L1 D-cache bandwidth, interconnect sustainable throughput along both directions, L2 cache bandwidth, and DRAM bandwidth. In our experiments, the statistic counters are aggregated until the kernel has run for 4 million cycles. The reason is that the variation of statistic counters becomes very small after the kernel is simulated for 2 million cycles. Prior works [Adr12b] [Wan16] [Xu16] made similar observations.

The scheduler utilization is the ratio of total instructions issued over the total cycles of all schedulers (i.e., overall cycles X number of schedulers). It also reflects the latency hiding effectiveness for a benchmark.

The L1 D-cache and L2 cache throughput utilization is the ratio of achieved throughput over the peak bandwidth of the cache data port. The data port is utilized when there is a cache hit. The duration depends on the data port width and the requested data size.

The memory requests that miss the L1 D-cache, go through the interconnect along the SM-to-L2 direction. The corresponding replies go through the interconnect along the L2-to-SM direction. Ideally, each input node can transmit a flit (32B) in one cycle so the peak bandwidth is 614 GB/s per
direction (Table 4.1). However, due to the output contention as discussed in Section 3.2.2, at most 60% of the peak bandwidth can be achieved. So, we define the interconnect utilization as the ratio of the actual throughput over the sustainable throughput (i.e., 60% of the peak bandwidth) rather than the ratio over the peak bandwidth. If we directly use the peak bandwidth, all benchmarks would show low interconnect utilization overlooking the nature of the sustainable bandwidth of such a crossbar interconnect network.

The DRAM throughput utilization is measured as the ratio of the achieved throughput of the DRAM data transfer over the peak DRAM bandwidth.

The bandwidth utilization of the register file and shared memory is also tested in our experiment. We don’t report the details in the paper because they are not the performance bound for any of our benchmarks.

We also tested the instantaneous throughput utilization by collecting the statistic counters every P cycles. We find that the instantaneous utilization is stable if P is large enough (e.g., 10000).

### 3.4.2 Overall Results

Figure 3.4 presents the overall results of throughput utilization. The Y-axis shows the ratio of achieved utilization of the six components as discussed in Section 3.4.1. The benchmarks are classified into 5 categories according to their most utilized component.

The first category is compute-intensive benchmarks, including SAD_1, BP_1 and ST. In this category, the scheduler utilization is high (over 60%), which means they have enough computation instructions to hide the memory latency such that instructions are issued without significant pipeline bubbles.

The second category is L2-to-SM interconnect-intensive benchmarks, including KM, CFD, SPMV, FDTD_2, MMS and CONV. Those benchmarks have over 60% of the sustainable interconnect throughput along the L2-to-SM direction. The memory replies from either the L2 cache (L2 hits) or DRAM (L2 MSHRs for L2 misses) need to go through this interconnect. Note that neither L2 nor DRAM throughput has been fully utilized for those benchmarks. It means the interconnect fails to provide
the data to the SMs at the speed of L2 and DRAM replying the data. The bottleneck in the L2-to-SM interconnect will cause the buffer between L2 and interconnect to become full, which slows down the L2 and DRAM. The cascading effect will stuff all buffers along the data path and eventually cause the pipeline to stall at the SM load store units (LSUs).

The third category is SM-to-L2 interconnect-intensive benchmarks, including GEMM and MM. The SM-to-L2 direction of interconnect is responsible for transferring the memory load and store requests. And because load requests do not contain any payload data, the major consumption is from the store requests. Both GEMM and MM have excessive global memory stores because they store all intermediate results to global memory rather than registers.

The fourth category is DRAM-intensive benchmarks. Because the DRAM throughput may suffer from row buffer misses, row activation or refreshing, the peak bandwidth is hard to achieve. X. Mei et al. [MC16] observed that the maximum achievable throughput can be 20% – 30% lower than the peak bandwidth. In this paper, we define the benchmarks which consumed more than 50% of DRAM peak bandwidth as DRAM-intensive, which includes SAD_2, LBM, FDTD_1, BP_2, DWT_1 and ST. Note that a common approach to categorizing the workloads on GPUs is based on the ratio of the LSU stalls [Set15][Wan16]. With this approach, both DRAM-intensive and interconnect-intensive benchmarks are classified as memory-intensive. However, such categorization can be misleading as our results clearly show that the DRAM bandwidth is not highly utilized for many interconnect-intensive benchmarks.

The fifth category, which includes BFS, BT, DWT_2, HG and HS, doesn't fully utilize the throughput of any of the resources. Such benchmarks essentially lack active warps to fully utilize the throughput of the computation or memory resources. In Section 3.5, we show that increasing the number of active warps can significantly increase the performance of the benchmarks in this category.

From Figure 3.4, we can see that the L1 D-cache bandwidth utilization is low for most benchmarks, only 7 benchmarks have higher than 5% utilization. This is because the L1 D-cache size is too small for so many concurrent warps [Rog12]. The L2 cache bandwidth utilization is relatively better than L1. There are 13 benchmarks that achieve higher than 10% L2 cache bandwidth utilization.
Figure 3.5 The impact of varied numbers of active warps for different interconnect bandwidth on SPMV. The left Y-axis is the throughput of L1 D-cache and interconnect (icnt). The right Y-axis is the IPC. The X-axis is the number of active warps.

Because L1 D-cache fails filtering the memory traffic effectively, L2 cache and DRAM become the main resources to answer the memory accesses. When L2 cache filters a large amount of memory traffic, the interconnect between L1 and L2 becomes very busy.

3.4.3 Scalability Analysis

In this section, we examine how and why the performance varies with increased TLP. In Figure 3.1, we have shown that the L1 D-cache performance is not enough to explain the performance degradation for the increased TLP. Here we tackle the problem from the throughput perspective. Figure 3.5 presents 3 diagrams with the interconnect bandwidth varies from 307 GB/s to 1228 GB/s through varying the wire width from 16B to 64B. Each diagram shows the IPC (right Y-axis), the throughput of L1 D-cache and interconnect (left Y-axis) with varied numbers of active warps (x-axis). As shown in the first diagram, the L1 D-cache throughput reaches the peak when the warp number is 16. Then the interconnect bandwidth is saturated. Because the interconnect is saturated, the lost throughput from the L1 D-cache cannot be compensated by the interconnect. So the performance keeps decreasing as increasing the warp number. In the second diagram, the performance degradation rate becomes smaller because the interconnect can provide more throughput. In the third diagram, the performance degradation disappears because the increased interconnect throughput can entirely compensate the decreased throughput of L1 D-cache. From the three diagrams, we can see that the performance reaches the peak when both L1 D-cache and
Figure 3.6 Speedup for doubling the context resources per SM.

interconnect throughput reach the peak.

Figure 3.5 also confirms that the interconnect becomes the bottleneck when the bandwidth utilization is close to the sustainable bandwidth. As we double the interconnect bandwidth, the performance increases accordingly.

3.5 Motivation for High TLP

3.5.1 Which Benchmarks Benefit From Increased TLP

As shown in Section 3.4.3, when a dynamic resource has been saturated, a benchmark cannot benefit from a higher degree of TLP. Instead, increasing TLP for the benchmarks which have not saturated any resource, should be helpful. To justify our hypothesis, we double the context resources per SM (Table 4.1) to accommodate the doubled number of warps and test the performance gains.

In Figure 3.6, the benchmarks from ‘underutilized’ category in Figure 3.4 achieve the highest speedups. It means lacking active warps is the reason for their resource underutilization. For benchmarks that have saturated a certain resource, increasing the number of active warps can lead to either little improvement (e.g. SAD_1) or degradation (e.g. KM). For the other benchmarks, such as BP_1 and FDTD_1, although they have achieved relatively high utilization of certain resources, some improvement can still be achieved.
3.5.2 Virtual Thread

Virtual Thread (VT) [Yoo16] is a recent work leveraging warp slot virtualization to improve TLP. VT observes that the TLP of some benchmarks is limited by warp or TB slots while the register file and shared memory have some leftover space. VT refers to the benchmarks which are limited by warp or TB slots as scheduling limited while the others are referred as capacity limited. For scheduling limited benchmarks, VT accommodates extra TBs using the spare register file and shared memory. The extra TBs are in the inactive state initially. When an active TB is stalled by memory operations, VT inactivates the stalled TB by moving the warp/TB slot data to shared memory. Then an inactive TB can become active using the vacant warp/TB slots.

There are several limitations of VT to improve TLP.

First, VT only targets at warp/TB slots and it does not work for capacity limited benchmarks. Figure 3.7 shows the occupancy of the context resources for the top 10 benchmarks that can benefit from increasing of TLP. Although 5 benchmarks are scheduling limited, the other 5 benchmarks are capacity limited, either by the register file or shared memory.

Second, VT only leverages TB-level context switching, which requires all warps of a TB to be stalled by the memory operations. Thus, VT loses the latency hiding opportunities for TBs with some but not all stalled warps.

Third, for some scheduling limited benchmarks, their TLP may become capacity limited after removing the scheduling limit with VT. Take BT as an example. Its TLP is initially limited by the warp slot. As its register file occupancy is 88%, the spare register file can only accommodate one more inactive TB using VT. Shared memory, on the other hand, remains highly underutilized.

Fourth, VT does not consider the dynamic resource utilization for improving TLP. As we highlighted in Section 3.5.1, the performance does not necessarily scale with increased TLP. Especially, for the benchmarks with saturated dynamic resources, increasing the TLP may even lead to performance degradation.

To address these limitations, we propose a novel warp-level context switching scheme. It improves TLP for both scheduling and capacity limited benchmarks, offers latency hiding at both warp
3.6 Improving TLP with Fast Context Switching

3.6.1 General Idea

As pointed out in Section 3.5, increasing TLP can improve the performance for those benchmarks which have not saturated any resources. In this section, we describe our approach to taking advantage of fast context switching for improving TLP. The basic idea is: when a warp/TB is stalled by a long latency operation, the context of the warp/TB is switched out so that the vacant resource can accommodate a new warp/TB. In order to achieve that, the context switching latency must be much smaller than the latency which we want to hide.

In this paper, we consider two reasons that may cause a warp to experience long-latency stalls. The first is global memory loads which miss or bypass the L1 D-cache. On GPUs, the latency to access off-chip memory may take 400 – 800 cycles [Won10]. Barrier instructions are the second reason that may cause a warp to stall. When a warp reaches a barrier, it has to wait for other warps in the same TB to arrive. Y. Liu et al. [Liu16] report that the barrier latency can be as high as 3800 cycles. We discuss additional motivation to enable context switching at barriers in Section 3.6.3.

In order to minimize the latency overhead of context switching, we analyze the context resources’ occupancy in Figure 3.7. The number of active warps per SM is limited by 4 factors: the register file

Figure 3.7 Occupancy of GPU context resources.

and TB levels, and determines the proper degree of TLP based on dynamic resource utilization.
size, shared memory size, scheduler warp slots and TB slots. From Figure 3.7 we can see that not all resources are fully occupied. So we only need to switch out the critical resource that limits the number of active warps. Another key technique that enables fast context switching is that we use spare fast on-chip resources to store the switched-out contexts. Notice that the utilization of register file and shared memory is unbalanced for most benchmarks. So we can use the spare resource to spill the limited resource. Based on our discussion in Section 3.4.2, we also use the L1 D-cache as another context buffer if its throughput is low. In addition, we leverage register liveness analysis to reduce the register number which needs to be swapped during context switching. The detail of resource management is described in Section 3.6.6. We refer to our approach as GPUDuet.

### 3.6.2 Latency Hiding with Two-Level Context Switching

We propose to use both warp-level and TB-level context switching for latency hiding. Let us use an example to illustrate how such a scheme works.

As shown in Figure 3.8, we assume that a kernel has 2 TBs and each TB contains 2 warps. In the baseline architecture, each SM can only accommodate 2 warps or 1 TB. As shown in Figure 3.8a, TB1 can only be launched after TB0 is finished. There are several long-latency events during execution.

In Figure 3.8b, we assume that the kernel is either register limited or warp-slot limited (e.g., the
warp scheduler only has two entries). In this case, we show that warp-level context switching can be used to overlap the execution latency. When warp0 in TB0 is stalled, we save its architectural states and release its register and warp slot. Then we can launch warp0 from TB1 into the SM. When warp1 in TB0 is stalled and dumped, depending on whether warp0 of TB0 is ready, we can switch warp0 of TB0 back or launch a new warp from TB1. In this way, long-latency gaps can be well hidden as shown in Figure 3.8b.

When the kernel is shared memory limited and only one TB can be accommodated in the SM, warp-level context switching does not work because no more TBs or their warps can be dispatched onto the SM. So we propose to use TB-level context switching to hide latency in such cases. As shown in Figure 3.8c, after warp0 in TB0 is stalled, we cannot launch warps from another TB. But when both warps in TB0 are stalled, we can switch out the architectural states of TB0 such that TB1 can be launched. When both warps in TB1 are stalled, TB0 can be switched back.

Between TB-level and warp-level context switching, TB-level context switching has to wait for all warps in a TB to stall before being switched out, thereby losing latency hiding opportunities compared to warp-level context switching. Therefore, we use TB-level context switching only for shared-memory limited benchmarks.

Note that our approach is not applicable for the benchmarks that are limited by both register and shared memory. In such a scenario, the spare on-chip resource to spill the context is very limited. Besides, the context switching overhead to switch in/out both register and shared memory would be much higher.

### 3.6.3 Deadlock Avoidance

If we only enable context switching for memory operations, the warp-level context switching may suffer from deadlocks in the scenario when all TBs are partially active, meaning that some warps are active and some warps in the same TB are switched out. If all active warps have reached the barriers, they will be waiting at the barriers forever. This is because the inactive warps may not have enough resource to be switched back, they will never reach the barriers.
The deadlock problem can be solved in two ways.

The first is to always maintain one TB as fully active, which means to disable the context switching of one TB per SM. In this way, when the other TBs are deadlocked, the inactive warps can be switched back when the fully active TB has finished and deallocated all its resources. However, because there is no guarantee of the TB execution time, other TBs may suffer a long period of idle time. Besides, latency hiding effect is reduced as we need to maintain a fully active TB.

The second is to make all warps that wait at the barriers eligible to be switched out. Then, the stalled inactive warps that have not reached the barrier will be able to switch back. Note that a barrier does not require all warps in a TB to reach the barrier at the same time. As long as all warps reach the barrier, even if some of them are switched out after reaching the barrier, the synchronization is achieved and all the warps can continue execution past the barrier. In this paper, we use the second way to avoid deadlocks.

3.6.4 Determining the Number of Extra TBs

To determine how many extra TBs to accommodate in an SM, we first introduce some terms. We use MC to denote the maximum number of TBs that can be natively accommodated per SM. And ME is used to denote the maximum number of TBs that can be launched without all their contexts being active. In other words, \((M C + ME)\) is the maximum number of concurrent TBs per SM using our proposed GPUDuet approach. However, only \((N \times MC)\) warps can be scheduled to run in each cycle, where \(N\) denotes the number of warps per TB. We refer to the type of context (e.g., register file) that needs to be switched out to accommodate the ME extra TBs as the limiting context.

As observed in Section 3.5, increasing the TLP is most helpful for benchmarks that have relatively low throughput utilization. For benchmarks that have high throughput utilization, increasing TLP can lead to little performance improvement or even performance degradation. To determine the most beneficial ME, we first compute its upper bound statically. Then, we initialize ME as 0 when a kernel is launched and adjust it based on dynamic resource utilization. The upper-bound of ME is set as the minimum among the following factors:
Table 3.3 Throughput utilization classification

<table>
<thead>
<tr>
<th>Classification</th>
<th>Throughput Utilization Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underutilized</td>
<td>Scheduler &lt; 60% and Interconnect &lt; 60% and DRAM &lt; 50% and L1 &lt; 60% and L2 &lt; 60%</td>
</tr>
<tr>
<td>Saturated</td>
<td>Scheduler &gt; 80% or Interconnect &gt; 90% or DRAM &gt; 70% or L1 &gt; 80% or L2 &gt; 80%</td>
</tr>
<tr>
<td>Moderately Utilized</td>
<td>Others</td>
</tr>
</tbody>
</table>

- The maximum number of concurrent warps, which are managed by the warp context table (Section 3.6.6). The number used in this paper is 128, compared to the baseline number of warp slots as 64.

- The maximum number of concurrent TBs, which are managed by TB context table (Section 3.6.6). The number used in this paper is 32, which equals to the maximum TB number in the baseline architecture. This is because we observe that the TB number rarely becomes the limitation (Table 3.7).

- The maximum number of TBs that can be supported considering the unified resource of the register file, shared memory and L1 D-cache, i.e., (unified resource size / TB context size). The reason is that the total register, shared memory and warp slot usage cannot exceed the total capacity of on-chip memories.

- The maximum number of TBs when the register and shared memory are not the limiting context at the same time. When accommodating one extra TB would need both registers and shared memory data to be switched out, the TB-level context switching overhead (solely using the L1 D-cache) becomes too much. In other words, our approach does not work for the benchmarks which are limited by both register file and shared memory.

To collect dynamic resource utilization, GPUDuet implements a throughput monitor to collect the utilization of the scheduler, L1 D-cache, interconnect, L2 cache and DRAM. The monitor skips the first 10000 cycles to let the GPU warm up. Then the monitor collects the counters for 10000 cycles and updates the ME for the first time. As shown in Table 3.3, the benchmarks are categorized into ‘underutilized’, ‘moderately utilized’ and ‘saturated’. Moderately utilized and underutilized
kernels are also referred as ‘unsaturated’ in this paper. For underutilized kernels, the ME is set at the upper-bound while the ME is set to zero, i.e., disabling GPUDuet, for saturated kernels. For moderately utilized kernels, the ME is initialized to zero and GPUDuet increases the ME by 1 every 10000 cycles until it reaches the upper-bound or any resource becomes saturated. For unsaturated benchmarks, if there is no extra space in the register file or shared memory and the L1-D cache bandwidth utilization is low (less than 3%), the L1-D cache is entirely bypassed and used to store the spilled context.

### 3.6.5 Warp States

In GPUDuet, each warp can be in one of the following four states: running, stalled, absent and ready. Figure 3.9 shows the state transition diagram. A running state means all contexts of a warp are present and the warp is not stalled by a long latency operation. When a running warp is stalled by a long latency operation, its state is changed to the stalled state. A stalled warp will wait to be switched out. Before switching out, GPUDuet checks the switching out condition of the warp. If the condition is not met, the warp state returns to running. Otherwise, the warp is switched out and its state is changed to be absent. When the long operation is finished, the absent warp becomes ready and wait to be switched in. When there is enough context resource, the warp is switched in and the warp state is changed to running.
3.6.6 GPUDuet Architecture

Our proposed GPUDuet architecture is shown in Figure 3.10. It manages all the on-chip storage and determines when to switch the context of a warp or a TB. The context of a warp includes the registers and meta-data which keeps track of its execution state. The meta-data includes the program counter, the reconvergence stack entries [Fun09] and the score board entries. In Figure 3.10, the meta-data

![Block diagram of the GPUDuet architecture.](image)

**Figure 3.10** Block diagram of the GPUDuet architecture.

<table>
<thead>
<tr>
<th>Component</th>
<th>Entry size</th>
<th># Entries</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation bit map</td>
<td>3072 bits</td>
<td>1</td>
<td>3072 bits</td>
</tr>
<tr>
<td>Warp context table</td>
<td>27 bits</td>
<td>128</td>
<td>3456 bits</td>
</tr>
<tr>
<td>TB context table</td>
<td>45 bits</td>
<td>32</td>
<td>1440 bits</td>
</tr>
<tr>
<td>Stalled/ready warp queues</td>
<td>7 bits</td>
<td>16/16</td>
<td>224 bits</td>
</tr>
<tr>
<td>Stalled/ready TB queues</td>
<td>5 bits</td>
<td>8/8</td>
<td>80 bits</td>
</tr>
</tbody>
</table>

**Table 3.4** Storage overhead per SM

If the ME is 0, all warp states are initialized as running. If the ME is larger than 0, the state of a newly launched warp is initialized as running if all contexts are present. Otherwise, the state is initialized as ready.
structures are abstracted as a warp slot array indexed by the (physical) warp id. Besides the contexts of its warps, the context of a TB includes its shared memory. In our context management policy, the contexts can be either present in their original structure or switched out to the register file/shared memory/L1 D-cache. Only when all contexts of a warp and the context of its corresponding TB are present, the warp can be scheduled by the warp scheduler. As discussed in Section 3.6.2, GPUDuet performs TB-level context switching only when the TLP of a kernel is limited by shared memory.

The on-chip memory managed by GPUDuet includes the register file, shared memory and L1 D-cache. To manage them, GPUDuet unifies their logical address spaces. The addressable unit is 128B, the same as the vector register width. Note that these memory components remain as separate physical components and there is no change in their hardware organizations. Because the total capacity of the on-chip memory is 384KB (256KB+96KB+32KB), the address space for register file is set as 0x000-0x7ff; the address space for shared memory is set as 0x800-0xaaff; and the address space for the L1 D-cache is set as 0xb00-0xbff. A bit map is used to keep track of allocated context resource. Each set bit represents whether a corresponding 128B is allocated. Because the total memory size is 384KB, the bit map has 3072 bits.

**Warp Context Table and TB Context Table:** In the GPUDuet architecture shown in Figure 3.10, the warp context table and TB context table keep the states of the warps and TBs, respectively. The warp context table (WCT) has 128 entries to support up to 128 active warps and is indexed by the warp id. As shown in Figure 3.11a, each entry includes 4 fields: a 1-bit ‘Valid’ field, a 2-bit ‘State’ field indicating the warp state (Figure 3.9), a 12-bit ‘Regaddr’ field representing the base address of the warp, which can point to anywhere in the unified on-chip memory space and the register context is present only when it falls into the register file address space, and a 12-bit ‘Metaaddr’ field showing where the meta data of the warp is maintained: the index to the warp-slot array if it is present or the spilled address otherwise. The size of the WCT is 3456 bits (=128x(1+2+12+12)).

The TB context table (TBCT) is indexed by the TB id. There are 16 entries in the TBCT and each entry contains the following fields as shown in Figure 3.11b. ‘Valid’ indicate whether the TB has been launched. ‘Absmask’ is a 32-bit mask indicating which warps in the TB are absent. ‘Smemaddr’
is a 12-bit field pointing to the base address of the shared memory context. The shared memory context is present if the address falls into the shared memory address space. The size of the TBCT is 1440 bits = $(32 \times (1 + 32 + 12))$.

**Context Switching Logic:** The context switching logic (CSL) is used to spill or restore the limiting context resource of warps/TBs. If the limiting context resource is registers and/or warp slots, warp-level context switching is used. If the limiting resource is shared memory or both shared memory and warp slots, TB-level context switching is used. When both registers and shared memory are limiting resources, GPUDuet is disabled.

In warp-level context switching, the CSL switches out a stalled warp when there is enough space in shared memory or the L1 D-cache. It switches in a ready warp when there is enough space in the register file and warp-slot array. When there is both a stalled warp and a ready warp, the CSL will switch the context spaces of them. After switching out/in a warp, the CSL notifies the WCT to update the warp state and changes the address if necessary.

In TB-level context switching, if all warps of a TB have been stalled, the TB is considered as stalled and being pushed to the stalled TB queue. Then the CSL spills shared memory and the warp slots to the register file or L1 D-cache if either has enough space. When all warps, except the ones that are stalled by a barrier, in a TB are ready, the TB is ready to be switched back. Also, if there is a TB in the stalled queue and a TB in the ready queue at the same time, the CSL swaps the context space of them.

**Register Context Reduction:** We leverage register liveness and compression as proposed by Z. Lin et al. [Lin16] to reduce register context size before context switching. Such liveness and compression can be done either with additional architectural support at run-time or by static analysis using the compiler. In our experiments, we use the compiler approach to simplify the hardware complexity. In our approach, the switch out points can be either the first uses of registers to be loaded from global memory or the barriers. The compiler analyzes the reduced register size at such points with liveness and compression. Then, GPUDuet uses the maximum size among those points as the register allocation size so that we can use the same fixed size to spill live and
compressed registers during context switching. In our experiment, the register allocation size is 44% of the original size on average.

### 3.6.7 Context Switching Latency

In this paper, we model the latency of context switching as follows:

$$\text{Latency} = \frac{\text{State}}{\text{BW}}$$

(3.1)

In Equation 3.1, the \textit{State} represents the total size of the architectural states which are going to be spilled or restored. We assume the bandwidth (\textit{BW}) of the on-chip memory is 128 bytes/cycle for the baseline architecture.

For warp-level context switching, the architectural state size for a warp is calculated as:

$$\text{State}(\text{warp}) = ST + LR \times 128$$

(3.2)

\textit{ST} denotes the size of a warp slot, which includes the reconvergence stack entries and the scoreboard entries. Fung et al. [Fun09] discussed the details of the size of warp slots. Because GPUDuet only saves live registers, \textit{LR} is the number of live registers, each of which is 128 bytes.

For TB-level context switching, the kernel is limited by shared memory. So the architectural state size for a TB is:

$$\text{State}(\text{TB}) = N \times ST + SS$$

(3.3)

In Equation 3.3, \textit{N} is the number of warps in a TB and \textit{SS} is the size of shared memory per TB. As discussed in Section 3.6.4, GPUDuet does not allow registers and shared memory to be spilled at the same time. So the registers do not need to be saved for TB-level context switching.
3.6.8 Hardware Overhead Comparing to Virtual Thread

The storage overhead of Virtual Thread (VT) [Yoo16] mainly includes the TB status table and the virtual warp IDs in the scoreboard. The TB status table is used to keep track the number of stalled warps in a TB. When all warps in a TB are stalled, the TB is marked as inactive and prepared to be swapped out. In this paper, the TB status table size is 32(maximum TB number)x7(entry size)=224 bits. The active virtual warp ID storage is used to map the physical warp IDs to virtual warp IDs. We use the following setup: 64(maximum number of physical warps)x7(size of each virtual warp ID)=448 bits.

Table 3.4 summarizes the storage overhead per SM of GPUDuet architecture. Compared to VT, GPUDuet requires higher storage overhead (1034 bytes per SM vs. 84 bytes per SM) and additional logic to manage the register file, shared memory, and L1 D-cache as a unified logical memory region and support both warp-level and TB-level context switching, whereas VT only supports TB-level context switching. To justify such hardware overhead, the performance improvement of GPUDuet compared to VT will be discussed in Section 3.7.2.

3.7 Evaluation

In this section, we evaluate the performance of GPUDuet compared to the baseline architecture and a state-of-the-art approach, VT, to improve GPU TLP. As described in Section 3.6.4, GPUDuet leverages a training phase to determine the number of extra TBs and the overall performance of GPUDuet includes such training phases. In our evaluation, we run each benchmark for 4 million cycles unless it finishes within 4 million cycles. The same methodology is used for the baseline, VT, and GPUDuet.

3.7.1 Impact of Maximum Extra TBs

In Figure 3.12, we select 15 benchmarks (5 from each category in Table 3.3) to evaluate the performance speedups with a varied number of maximum extra TBs (ME). For each benchmark, the ME
can be either dynamically adjusted using the algorithm as described in Section 3.6.4 or statically selected. The upper-bound of the static ME for each benchmark depends on the context occupancy and the reduction ratio of the registers.

For DWT_2, the L1 D-cache is bypassed and used to store the spilled context. Therefore, we use the baseline architecture with the bypassed L1 D-cache for this benchmark to compute the normalized performance. The performance of DWT_2 decreases by 5% with a bypassed L1 D-cache. However, from Figure 3.12 we can see that the performance improvement is much higher than 5% with higher TLP.

As shown in Table 3.3, the benchmarks are classified into 3 categories. For underutilized benchmarks, which have low utilization for any computation and memory bandwidth resources, GPUDuet has the most significant improvement. These benchmarks show good performance scalability as their performance increases as ME increases and all 5 underutilized benchmarks achieve the best performance at the upper-bound of ME. For benchmarks in this category, the dynamic ME algorithm can effectively find the ME with the optimal performance. The performance improvement of the underutilized benchmarks ranges from 23% to 47%.

For moderately utilized kernels, because the computation or bandwidth resources are not fully utilized, increasing TLP with GPUDuet can still achieve better performance for most benchmarks. For MMS, CONV, MM and ST, the performance speedups range from 7% to 25%. For SPMV, however, the increasing of TLP aggravates the cache thrashing, which causes the interconnect bandwidth to be saturated. So, increasing ME can only degrade the performance of SPMV. For MM, because the
interconnect bandwidth is not saturated in the baseline architecture, increasing the ME from 1 to 3 improves the performance. However, the performance starts to drop when ME is larger than 3 due to the cache thrashing and saturated interconnect bandwidth. In this category, the dynamic ME selection algorithm searches for the optimal ME which can maximize the performance.

For saturated benchmarks, the computation or memory bandwidth cannot provide much room for higher TLP. So, BP_1 have very small performance improvement. For CFD, KM, LBM and SAD_2, the cache throughput suffers from higher TLP while the interconnect or DRAM cannot compensate the loss. For KM, the performance degradation can be as high as 26%. Therefore, the algorithm in Section 3.6.4 dynamically sets ME to 0, disabling GPUDuet for such benchmarks with saturated throughput utilization.

### 3.7.2 Comparing with Virtual Thread

In Figure 3.13, we report the normalized IPC of VT [Yoo16] and GPUDuet compared to the baseline GPU described in Table 1. To highlight the benefit of warp-level context switching, we also show GPUDuet with only TB-level context switching, which is labeled as 'GPUDuet_TB'.

Because the TLP of CFD, LBM, ST, MMS, DWT_2, HG and HS is limited by the register file, no extra TBs can be accommodated by VT. However, by spilling registers to shared memory or L1 D-cache, GPUDuet can achieve higher degrees of TLP for these benchmarks. Take DWT_2 as an example, the baseline architecture can accommodate 10 TBs. With register size reduction, the register context size can be reduced to 50%. So the L1-D cache can accommodate the registers of extra 2 TBs. The
Figure 3.14 TLP improvement by Virtual Thread (VT) and GPUDuet over the baseline architecture.

shared memory of the extra 2 TBs can reside in the spare shared memory. As HG is limited by shared memory, it is not supported by VT either. With GPUDuet, the shared memory data can be moved to the register file to accommodate 8 extra TBs.

Even for the scheduling-limited benchmarks, (i.e., their TLP is limited by the warp slots), GPUDuet can achieve much higher degrees of TLP. Take BT as an example, the spare space of the register file can only accommodate 1 extra TBs. Whereas for GPUDuet, 6 extra TBs can be accommodated by using shared memory to spill registers.

For KM, VT performs worse than the baseline. This is because KM saturates the interconnect bandwidth, increasing the TLP will cause L1 D-cache thrashing and hurt the performance. By leveraging dynamic resource utilization information, GPUDuet disables context switching for KM.

For benchmarks which are limited by registers or warp slots, such as HS and BFS, warp-level context switching performs better than TB-level context switching. This is because warp-level context switching takes advantage of more latency hiding opportunities, as discussed in Section 3.6.2.

Compared to VT, GPUDuet improves the overall performance by 12% and 16% for unsaturated benchmarks.

3.7.3 TLP Improvement

In Figure 3.14, we compare the TLP improvement over the baseline architecture by VT and GPUDuet. The TLP is evaluated as the maximum number of TBs which can be launched to each SM. For the unsaturated benchmarks, such as HG and BT, GPUDuet achieves much higher TLP than VT.
This is because VT only works for scheduling-limited benchmarks while GPUDuet also works for capacity-limited ones. In this category, compared to the baseline architecture, the TLP improvement by VT is 18% whereas 70% for GPUDuet. For the saturated benchmarks, such as BP_1 and KM, GPUDuet disables the context switching, therefore VT has higher TLP. However, higher degrees of TLP for these saturated benchmarks lead to little performance improvement or even performance degradation.

### 3.7.4 Impact of Context Switching Latency

In Figure 3.15, we evaluate the impact of context switching bandwidth of GPUDuet. In the baseline architecture, we assume the read/write port width of the register file, shared memory and L1 D-cache is 128 bytes. In Figure 3.15, we evaluate the performance of GPUDuet with 256-byte, 128-byte and 64-byte port width. In general, benchmarks with high register file or shared memory utilization, such as HG and BT, tend to be more sensitive to the swapping bandwidth. Because such benchmarks have larger architectural states to move during context switching. Comparing to the baseline architecture, the performance improvement of GPUDuet with 256-byte, 128-byte and 64-byte port width are 24%, 22% and 18% respectively.

### 3.8 Related Works

Prior studies on the impact of TLP on throughput-oriented processors, including [Rog12] [Kay13] [Li15], are discussed in Section 3.4.
A few prior works focus on improving the utilization of GPU resources. In Section 3.5.2 and 3.7.2, we discussed Virtual Thread [Yoo16]. Warped-Slicer [Xu16] proposes an analytical model to determine the optimal resource partition on one SM for concurrent kernels. Elastic kernel [Pai13] increases register and shared memory occupancy by issuing concurrent kernels on one SM. SMK [Wan16] proposes a dynamic sharing mechanism for concurrent kernels to improve the resource utilization while maintaining the fairness. KernelMerge [Gre12] and Spacial Multiplexing [Adr12b] study how to use concurrent kernels to better utilize GPU resources and improve overall throughput. Lee et al. [Lee14] also leverage mixed concurrent kernels to improve GPU utilization. In this work, we target at resource utilization for a single kernel.

Some works observe the GPU underutilization problem for particular scenarios. SAWS [Liu15] and BAWS [Liu16] propose barrier-aware scheduling policies to avoid warps from waiting at barriers for too long. CAWS [LW14] and CAWA [Lee15b] predict and accelerate the warps that lag behind so that the TB can finish faster. Warpman [Xia14] points out the spacial and temporal resource underutilization due to TB-level resource allocation and propose warp-level resource utilization to improve the GPU resource utilization. Although those works can improve the effective TLP at barriers or TB terminations, they are limited by the maximum warp number that allowed being issued to the GPU.

Zorua [Vij16] is a recent work which leveraging context virtualization on GPU to provide programming portability and achieve higher levels of TLP. There are two key differences between Zorua and our approach. First, Zorua allocates/deallocates on-chip resources at the phase boundaries. Whereas our approach deallocates the resources when a warp/TB suffers from a long latency operation. Second, Zorua spills the oversubscribed register file and shared memory to global memory while our work leverages the spare on-chip resources to achieve much faster context switching.

Gebhart et al. [Geb12] propose to unify the L1 D-cache, shared memory and register file to improve GPU on-chip resource utilization. We have a similar benefit in terms of increasing occupancy for register- or shared memory-limited applications. But the unified design requires extensive hardware changes and also needs software support. Besides, they have to pay overhead for re-partitioning
as different kernels have different resource requirements.

Some prior works [Par15b, Lin16] leverage context switching for preemption on GPUs. Similar to proposed by Lin et al. [Lin16], we leverage liveness analysis and register compression to reduce the context size. However, in this work we use spare on-chip resources to store the spilled contexts to enable much faster context switching.

Majumdar et al. [Maj15] study the scalability of the GPU kernels with computation units and memory bandwidth. Dublish et al. [Dub16] perform bottleneck analysis on different levels of GPU memory hierarchy, including L1/L2 caches and DRAM. However, neither of them discusses the performance impact of the interconnect between L1 and L2.

## 3.9 Conclusions

In this paper, we analyze the relationship between GPU performance and TLP through a novel resource utilization perspective. The GPU performance can be bounded by the scheduler throughput, L1 bandwidth, interconnect bandwidth, L2 bandwidth or DRAM bandwidth. We reveal that many memory-intensive benchmarks are actually bounded by interconnect bandwidth. Then we highlight that for benchmarks not saturating the throughput of any resources, increasing TLP can lead to significant performance improvement. We propose GPUDuet, a fast context switching approach, to increase TLP to better hide the latency of memory load and barrier operations. GPUDuet leverages spare on-chip resources to enable fast context switching. Liveness analysis and register compression are used to reduce the spilled context size. The evaluation shows that GPUDuet achieves up to 47% performance improvement and 22% on average for a set of benchmarks with unsaturated throughput utilization.
CHAPTER 4

COORDINATED CTA COMBINATION AND BANDWIDTH PARTITIONING FOR GPU CONCURRENT KERNEL EXECUTION

4.1 Introduction

To deliver high throughput, GPUs incorporate a large amount of computational resources and support high memory bandwidth. However, the resource demands across different GPU kernels vary significantly, which may lead to saturation of certain resources and underutilization in others. One solution to such unbalanced resource utilization is to concurrently execute multiple kernels with complementary characteristics. Prior works [Pai13] [Wan16; Xu16; Par17; Dai18] have shown that
both the GPU utilization and throughput can be improved by co-running heterogeneous kernels. Moreover, with GPUs being increasingly deployed in cloud servers, there is a strong need for GPU resource to be shared among multiple users.

There are different ways to support concurrent kernel execution (CKE) on GPUs. A simple way is to assign different kernels to different sets of streaming multiprocessors (SMs), such as Spatial Multitasking [Adr12a]. However, as pointed out in prior works [Wan16; Xu16; Par17; Dai18], Spatial Multitasking does not address resource underutilization within SMs. SMK [Wan16], WS, [Xu16] and Maestro [Par17] are the recent works focusing on intra-SM sharing. They propose different Cooperative Thread Array (CTA), aka thread block, combination algorithms to determine how many CTAs from each kernel should be dispatched to the same SMs. To achieve fair partitioning, SMK leverages a ‘Dominant Resource Fairness’ metric to fairly partition the static resources (such as the register file and shared memory) to each kernel. To further ensure fair performance, SMK periodically assigns each kernel a fixed time quota in the warp scheduler. The time quota partitioning is based on profiling of the standalone execution of each individual kernel. In comparison, WS first determines the scalability curves, i.e., the performance vs. the number of CTAs per SM, for each individual kernels. Then WS uses the scalability curves to determine the CTA combination that generates the minimum combined performance slowdown. Maestro proposes a dynamic search approach to find the optimal CTA combination. In addition, Maestro argues for loose round-robin (LRR) as the kernel-aware scheduling policy for CKE.

In this paper, we highlight that memory interference can significantly affect the throughput and fairness of CKE. And we make a case that even the optimal CTA combination does not eliminate the negative memory interference impact. To address this problem effectively, we propose a coordinated approach for CTA combination and bandwidth partitioning. Our proposed approach is based on the following observations.

First, bandwidth over-subscription from the bandwidth-intensive kernels introduce high queuing delay and drastically increase the memory latency, which may lead to severe performance degradation for other co-running kernels. Although it is well-known that compute-intensive kernels
leverage massive thread(warp)-level parallelism to hide the memory latency, the increased memory latency is so high (up to 8.2x higher than the standalone execution) that it can hardly be fully hidden. In this paper, we refer to the kernels which are sensitive to the increased memory latency as latency sensitive.

Second, we observe that, besides the DRAM bandwidth, the L2-to-L1 NoC bandwidth can be a critical resource in the GPU memory system. The reason is that all the data replied either from the L2 cache or DRAM go through the L2-to-L1 NoC. As a result, the L2-to-L1 NoC can become a bottleneck when the L2 cache filters a substantial amount of memory requests and the DRAM bandwidth is not saturated. Therefore, both the NoC bandwidth and the DRAM bandwidth need to be managed carefully. In prior works, both DRAM-intensive and NoC-intensive kernels are both considered memory intensive. In this work, we highlight the difference between them and show that these two types of kernels can benefit from CKE, i.e., running them together would improve resource utilization, as they stress different parts of the memory system.

Third, we find that the above-mentioned issues cannot be addressed even with the oracle CTA combination, which is the best CTA combination from an exhaustive search. There are three fundamental reasons: (a) the bursty memory traffic makes CTA-level control less effective; (b) the lack of application-aware memory scheduling fails to prioritize requests from latency-sensitive benchmarks to reduce their memory access latency; and (c) CTA-level management can be too coarse-grained in controlling thread-level parallelism (TLP).

Forth, different kernels favor different GPU resources. The latency-sensitive kernels require high levels of TLP to better hide the memory latency. On the other hand, the DRAM-intensive kernels are more sensitive to the bandwidth utilization and reducing the CTA number from a bandwidth-intensive kernel may not degrade its performance.

In our proposed coordinated approach for CTA combination and bandwidth partitioning (CCBP), we first dynamically detect the kernels as latency sensitive or bandwidth intensive. Among bandwidth-intensive benchmarks, CCBP further classifies them as NoC intensive and DRAM intensive. Then it effectively allocates the CTA number and bandwidth resources for each co-running kernel based
on their resource requirements. We derive the bandwidth consumption of both NoC and DRAM as functions of the memory request issue rate and achieve bandwidth allocation by controlling the memory request issue rate according to a kernel’s assigned bandwidth quota.

We compare CCBP with three state-of-the-art CKE approaches and the oracle CTA combination, which is the result of an exhaustive search of all possible CTA combinations. Compared to SMK and WS, our approach improves the average harmonic speedup by 78% and 39%, respectively, for 2-kernel co-runs. Compared to a recently proposed approach [Dai18] combining WS and memory-instruction scheduling, our approach improves the average harmonic speedup by 19%. Even compare to the oracle CTA combinations, our approach improves the harmonic speedup by up to 51% and 11% on average.

In this paper, we make the following contributions:

• This work reveals the memory interference problem in GPU CKE cannot be sufficiently solved using CTA-level management alone.

• We highlight that NoC and DRAM bandwidth are different bottlenecks in the GPU memory subsystem. This is the first work that shows it is beneficial to co-run NoC-intensive kernels with DRAM-intensive kernels. To our knowledge, this is also the first work that unifies the management of both the NoC and DRAM bandwidth.

• We propose a dynamic approach to classify GPU kernels into three categories, latency sensitive, NoC intensive or DRAM intensive. Based on the detected kernel types, our scheme selectively assigns resources to the kernels which would benefit the most from such resources.

• Our approach effectively reduces the memory latency for the latency-sensitive kernels. In the meanwhile, the bandwidth utilization is also improved for the bandwidth-intensive kernels.
**4.2 Background on GPU Architecture**

Contemporary GPU architecture, as illustrated in Figure 4.1, consists of multiple stream multiprocessors (SMs). Each SM has a private L1 D-cache and all SMs share a multi-banked L2 cache. The L1 caches and L2 cache banks communicate through an NoC (Network-on-Chip).

Each SM hosts a number of cooperative thread arrays (CTAs). Each CTA has a number of warps, each of which is a collection of threads running in the single-instruction multiple-data (SIMD) manner. Every cycle, a warp scheduler selects a ready instruction to issue to the execution pipeline.

Since the Hyper-Q [Hyp] technology was introduced by NVIDIA on the Kepler GPUs, recent GPUs support concurrent kernel execution on the same GPUs. The state-of-the-art GPU partitioning schemes [Wan16; Xu16; Par17] leverage intra-SM sharing, which allow warps/CTAs from different kernels to reside on the same SMs. As shown in Figure 4.1, warps from different kernels contend for the warp scheduler as well as the memory subsystem.

**4.3 Methodology**

**4.3.1 Simulation Specifications**

We use GPGPU-sim v3.2.2 to investigate GPU memory interference and evaluate our proposed approach for CKE. The baseline architecture configuration is shown in Table 4.1. In our experiments, the loose round-robin (LRR) scheduling policy is used by default and the greedy-then-oldest (GTO)
Table 4.1 Baseline architecture configuration

| Overall config. | 16 SMs, 32 threads/warp, 16 L2 banks, 16 DRAM chips |
| SM config.      | 1800MHz, 4 schedulers, LRR warp scheduler |
| Static resource per SM | 32 CTAs, 2048 threads, 256KB register file, 96KB shared memory |
| L1 D-cache     | 24KB, 128B block, 8-way associativity, 128 transaction queue entries, 16 store buffer entries, 256 MSHRs |
| NoC            | Two 16*16 crossbars, 32B flit size, bandwidth = 16 ports * 1.2GHz * 32B = 614GB/s |
| L2 cache per bank | 128KB, 128B block, 8-way associativity, 256 MSHRs, WBWA policy |

Table 4.2 Benchmark specification

<table>
<thead>
<tr>
<th>Bench.</th>
<th>Source</th>
<th>Thro.</th>
<th>NoC BW</th>
<th>DRAM BW</th>
<th>L2 Miss</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFD</td>
<td>cfd [Che09b]</td>
<td>7%</td>
<td>54%</td>
<td>14%</td>
<td>15%</td>
<td>NBI</td>
</tr>
<tr>
<td>LAV</td>
<td>lavaMD [Che09b]</td>
<td>5%</td>
<td>53%</td>
<td>1%</td>
<td>1%</td>
<td>NBI</td>
</tr>
<tr>
<td>SMV</td>
<td>spmv [Str12]</td>
<td>10%</td>
<td>52%</td>
<td>25%</td>
<td>30%</td>
<td>NBI</td>
</tr>
<tr>
<td>KM</td>
<td>kmeans [Che09b]</td>
<td>2%</td>
<td>52%</td>
<td>22%</td>
<td>17%</td>
<td>NBI</td>
</tr>
<tr>
<td>LBM</td>
<td>lbm [Str12]</td>
<td>17%</td>
<td>15%</td>
<td>61%</td>
<td>88%</td>
<td>DBI</td>
</tr>
<tr>
<td>FTD</td>
<td>FDTD-2D [GG12]</td>
<td>29%</td>
<td>32%</td>
<td>65%</td>
<td>66%</td>
<td>DBI</td>
</tr>
<tr>
<td>SAD</td>
<td>sad [Str12]</td>
<td>5%</td>
<td>17%</td>
<td>64%</td>
<td>78%</td>
<td>DBI</td>
</tr>
<tr>
<td>SRT</td>
<td>bucketsort [Che09b]</td>
<td>25%</td>
<td>37%</td>
<td>59%</td>
<td>69%</td>
<td>DBI</td>
</tr>
<tr>
<td>BFS</td>
<td>bfs [Che09b]</td>
<td>44%</td>
<td>14%</td>
<td>24%</td>
<td>100%</td>
<td>LS</td>
</tr>
<tr>
<td>BP</td>
<td>backprop [Che09b]</td>
<td>68%</td>
<td>19%</td>
<td>36%</td>
<td>51%</td>
<td>LS</td>
</tr>
<tr>
<td>BT</td>
<td>b+tree [Che09b]</td>
<td>24%</td>
<td>27%</td>
<td>19%</td>
<td>44%</td>
<td>LS</td>
</tr>
<tr>
<td>HG</td>
<td>histogram [Che09b]</td>
<td>28%</td>
<td>19%</td>
<td>25%</td>
<td>74%</td>
<td>LS</td>
</tr>
<tr>
<td>PF</td>
<td>pathfinder [Che09b]</td>
<td>72%</td>
<td>28%</td>
<td>39%</td>
<td>90%</td>
<td>LS</td>
</tr>
</tbody>
</table>

policy is used for standalone kernel running. The memory access latency is configured according to the memory hierarchy study by Mei et al. [MC16]. In our GPU model, we use crossbars for the NoC [AMD12]. Our crossbar is a non-blocking interconnect network, which means a connection can be established if both the input and output are idle. If more than one packet needs to reach the same output at the same time, only one packet can be accepted by the network and other packets have to be queued at their inputs.

4.3.2 Benchmark Categorization

We studied a wide range of 42 GPU kernels from the Rodinia [Che09b], Parboil [Str12] and Polybench [GG12] benchmark suites. Based on their NoC and DRAM bandwidth utilization, the benchmarks are grouped into three categories: NoC intensive (labeled as ‘NBI’), DRAM intensive (labeled as...
‘DBI’) and latency sensitive (labeled as ‘LS’). To evaluate the mechanisms proposed in this paper, as shown in Table 4.2, we selected a total of 13 benchmarks using the following criteria. For the NoC-intensive kernels, we ranked the NoC bandwidth utilization of all the 42 kernels and selected the highest 4 kernels. Similarly, for the DRAM-intensive kernels, we selected the top 4 kernels with the highest DRAM bandwidth utilization. For the kernels with low bandwidth utilization, they leverage a large amount of threads/warps to hide the memory latency. Note that even with a high number of warps, their overall bandwidth usage is low since either they are compute intensive or they are limited by other resources such as shared memory or branch divergence, which constrain their memory requests. However, when such kernels co-run with bandwidth-intensive kernels, the memory latency will be significantly increased such that higher levels of TLP are required to fully hide the latency. On the other hand, as the static resources are shared with all co-running kernels, each kernel may have fewer threads/warps to run concurrently on each SM to hide the memory latency. As a consequence, they become sensitive to increased latency and we refer to such kernels, i.e., those with low bandwidth utilization, as latency sensitive. To select the 5 kernels which are most sensitive to the increased memory latency, we doubled the access latency to L2 cache and DRAM and selected the ones with the most performance slowdowns.

Table 4.2 lists the details of the selected 13 benchmarks along with their resource utilization in standalone execution. The throughput (labeled ‘Thro.’ in Table 4.2) utilization is defined as the ratio of the achieved IPC (instruction per cycle) over the peak IPC. The NoC/DRAM bandwidth utilization is the ratio of the achieved bandwidth over the corresponding peak bandwidth. For the NoC-intensive kernels, we observe the NoC bandwidth utilization is saturated around 50% – 60% of the peak bandwidth. This is because the sustainable bandwidth of the crossbar NoC is 60% as pointed out in prior works [Kar87; Lin18]. For the DRAM-intensive kernels, the DRAM sustainable bandwidth saturates around 60% – 70% of the peak bandwidth, which is consistent with the observation by X. Mei et al. [MC16]. When a kernel has bandwidth utilization close to the sustainable bandwidth, the memory system latency is significantly increased due to queuing delays. For example, the DRAM access latency in SAD becomes 6.8x higher than the idle latency, i.e., the access latency when the
system is idle.

In the latency-sensitive category, BP and PF are compute intensive and they achieve high throughput. BFS and BT are thread-divergent kernels. The dynamic resources of HG are low because it is limited by the shared memory size. These kernels are latency-sensitive because they suffer from higher than 1.7x performance slowdown when the L2 and DRAM latency is doubled. Because the memory bandwidth is underutilized, the average L2/DRAM latency for these kernels is close to the idle latency.

In summary, we list the definitions of our kernel categorization as follows.

- **Bandwidth-intensive (BI) kernels**: kernels that are sensitive to the NoC or DRAM bandwidth. BI kernels include both NoC-intensive and DRAM-intensive kernels.
- **NoC-intensive (NBI) kernels**: kernels that are sensitive to the NoC bandwidth.
- **DRAM-intensive (DBI) kernels**: kernels that are sensitive to the DRAM bandwidth.
- **Latency-sensitive (LS) kernels**: kernels that are sensitive to the increased memory latency or the static resource capacity. LS kernels include both compute-intensive kernels and static-resource-limited kernels.
- **Compute-intensive kernels**: kernels that are limited by the ALU resources.
- **Static-resource-limited kernels**: kernels that are limited by the static resources, including register file, shared memory and the number of threads.

### 4.3.3 Evaluation Metrics

We evaluate the performance of CKE using two metrics: harmonic speedup (HSpeedup) [Luo01] and weighted speedup (WSpeedup) [EE08]. The HSpeedup is a balanced metric for both system throughput and fairness [Luo01] whereas WSpeedup is a metric for system throughput. The higher these metrics, the better performance is achieved. Because the goal of this paper is to improve both system throughput and fairness, HSpeedup is used as our primary evaluation metric. We
also observe that these two metrics correlate well. The reason is that when the co-running kernels stress different resources, higher overall resource utilization means both higher and more balanced utilization, thereby higher throughput and better fairness.

\[
HSpeedup = \frac{N}{\sum_k \frac{IPC_{\text{alone}}^k}{IPC_{\text{share}}^k}}
\]

\[
WSpeedup = \sum_k \frac{IPC_{\text{share}}^k}{IPC_{\text{alone}}^k}
\]

To ensure our simulation sufficiently captures co-running kernels’ dynamic behavior, we run 100 million cycles in each simulation experiment. If one kernel finishes before the 100 million cycles, it is re-executed. Because the IPCs of our benchmarks vary from 10s to 1000s. Therefore, the range of simulated instructions is 1 to 100 billions.

### 4.4 Limitations of CTA Management

A common way to improve GPU CKE is to manage the CTA combination of co-running kernels [Wan16; Xu16; Par17]. Although appropriate CTA combinations throttle the TLP of bandwidth-intensive kernels and therefore their memory requests, such approaches are not sufficient to address the memory interference problem. The reasons include (1) the bursty nature of memory traffic, (2) the inability to prioritize memory requests from different kernels, and (3) coarse-grain TLP control at the CTA level. Next we dissect these effects.

#### 4.4.1 Effects of Bursty Memory Requests

**High memory access latency** First, we use a case study of the BP+CFD co-run to illustrate the impact of bursty memory traffic on the memory access latency. BP is latency sensitive and CFD is NoC intensive. These two kernels are considered complementary as they stress different GPU resources. Through an exhaustive search of all possible CTA combinations, we find (7,4), meaning 7 CTAs from BP and 4 CTAs from CFD co-running on each SM, achieves the highest weighted speedup.
Figure 4.2 Performance and latency impact for BP+CFD co-run using different CTA combinations.

Figure 4.2(a) shows the slowdown of BP and CFD in the co-run, normalized to their standalone execution using this CTA combination. Normalized slowdown is computed as $\frac{IPC_{alone}}{IPC_{shared}}$. Even though BP has higher CTA utilization than CFD, it suffers from higher performance slowdown when the CTA combination (7,4) is used. Figure 4.2(b) shows the normalized memory access latency, comparing to the BP standalone execution in this case. We can see that the L1-L2 latency is much higher due to the NoC congestion caused by CFD. Because neither BP nor CFD is DRAM intensive, the L2-DRAM latency is not significantly increased. Regardless the high latency, we notice that 4 CTAs of CFD only consume 62% NoC bandwidth on average.

To pinpoint the reason for the increased L1-L2 latency, Figure 4.3 shows the instantaneous NoC bandwidth demand, which is normalized to the NoC sustainable bandwidth. To measure the bandwidth demand, we count the number of memory requests in the L1 D-cache miss queue to be sent to the NoC every 400 cycles. The request rate can be transformed to the bandwidth demands using the derivations to be discussed in Section 4.5.3. We call the bandwidth as ‘over-subscribed’ when the demand is higher than 100% of sustainable bandwidth. In such scenarios, the requests have to be lined up in the memory queues. In Figure 4.3(a), we can see the bursty memory traffic of CFD leads to frequent bandwidth over-subscription, resulting in very high queuing delays in the memory system. As a result, the memory requests from BP suffer from the long access latency.

Reducing the CTA number of bandwidth-intensive kernels can improve the performance of
the co-running latency-sensitive kernels. For example, as shown in Figure 4.2(b), the performance of BP can be improved if we reduce the CTA number of CFD from 4 to 2. However, Figure 4.2(b) shows that the memory access latency is still much higher than it when BP runs alone. The average NoC bandwidth consumption is reduced to 44% with 2 CTAs from CFD. However, as seen in Figure 4.3(b), the bandwidth over-subscription problem remains due to the bursty behavior of the CTAs from CFD. On the other hand, reducing the CTA number of CFD results in the overall bandwidth being underutilized. Therefore, the combination of (7,2) shows worse weighted speedup than (7,4) (weighted speedup of 1.2 vs. 1.3).

**Resource underutilization** When an NoC-intensive kernel co-runs with a DRAM-intensive kernel, the GPU utilization is maximized when both NoC bandwidth and DRAM bandwidth are fully utilized. However, the bursty behavior may cause the memory system being dominated by either NoC or DRAM demands in a certain period, leading to underutilized bandwidth of the other. We illustrate such a phenomenon by co-running an NoC-intensive kernel, CFD, and a DRAM-intensive kernel, FTD. We find that the optimal CTA combination of CFD+FTD co-run is (4,7) through an exhaustive search. Figure 4.4 shows the instantaneous DRAM bandwidth utilization for 7 CTAs of FTD running alone and CFD+FTD co-run. The sampling period is 400 cycles. As we can see from the

---

**Figure 4.3** The instantaneous NoC bandwidth demand of BP+CFD co-runs with different CTA combinations.
Figure 4.4 Instantaneous DRAM bandwidth utilization for CFD+FTD co-run and FTD standalone execution.

Figure 4.5 A snapshot of the L1 D-cache miss queue when a latency-sensitive kernel (kernel_1) co-runs with a bandwidth-intensive one (kernel_2).

The co-run shows lower but more bursty DRAM bandwidth consumption than FTD running alone. This is because CFD has a lot of L1 D-cache misses (which would hit in the L2 cache and therefore stress the L2-to-L1 NoC) in certain periods. Those requests dominate the memory queues between L1 and L2 caches so that FTD fails to issue requests. As a result, the DRAM bandwidth is wasted/underutilized in such periods even though FTD has DRAM bandwidth requests to be issued.

4.4.2 Prioritization of Memory Requests

We observe that the performance of CKE can be significantly improved if we can prioritize the memory requests from different kernels based on their different latency-hiding characteristics. For example, Figure 4.5 shows a snapshot of the L1 D-cache miss queue in a 2-kernel co-running case, where kernel_1 is latency sensitive and kernel_2 is bandwidth intensive. As kernel_2 typically generates much more requests than kernel_1, in this snapshot, kernel_1 has 2 requests while kernel_2 has 6 requests waiting in the miss queue. We assume that the critical bandwidth resource can only service 5 requests in a certain period. Given the FIFO policy of the miss queue, 1 request of kernel_1 and 4 requests of kernel_2 can be serviced in the first period. However, if we could change the
policy to prioritize the latency-sensitive kernel, 2 requests of kernel_1 and 3 requests from kernel_2 would be serviced instead. The throughput improvement of kernel_1 could be as high as 100% while the throughput of kernel_2 is only reduced by 25%. Such memory request prioritization is hard to achieve with CTA-level TLP management.

4.4.3 Coarse Granularity of TLP Control

Another limitation of CTA combination is the coarse granularity of TLP management. For some kernels, even one CTA can cause high bandwidth utilization and long memory access latency. For example, a single CTA of the SAD kernel utilizes 89% DRAM bandwidth and causes 2.3x higher memory access latency than the idle latency. More often, adding one CTA of the bandwidth-intensive kernel may lead to congested memory systems while reducing one CTA may result in underutilized bandwidth. Therefore, the CTA-level management alone is likely to be too coarse-grained to achieve optimal system throughput or fairness.

4.4.4 Motivation of Our Approach

This section illustrates that, without considering the bandwidth impact of the co-running kernels, the CTA management alone is insufficient to achieve the optimal performance. In order to solve the bandwidth over-subscription problem due to the bursty memory traffic, we propose to strictly control the NoC and DRAM bandwidth consumption of each kernel. To determine the optimal CTA combination and bandwidth partition, our approach detects the characteristics of each kernel and allocates CTA and bandwidth resources accordingly.

4.5 Coordinated CTA Combination and Bandwidth Partitioning

4.5.1 Overview

In this paper, we propose a coordinated CTA combination and bandwidth partitioning (CCBP) approach to improve the concurrent kernel execution on GPUs. Figure 4.6 presents the architecture
of our CCBP approach. The CCBP logic executes the CCBP algorithm to determine the CTA combination and bandwidth partitioning for each kernel. It monitors the bandwidth consumption and achieves bandwidth partition through the issue rate controllers (IRCs). The CTA combination is controlled through the CTA dispatcher.

The issue rate controllers (IRCs) in Figure 4.6 determine the bandwidth consumption of both NoC and DRAM. We manage the bandwidth demands by controlling the memory request issue rate, i.e., how many memory requests to be issued per time unit (200 cycles in this paper), for each kernel. Depending on the memory request type (load or store) and whether it hits the L2 cache, a request may result in different consumption of NoC and DRAM bandwidth. As will be discussed in Section 4.5.3, our approach collects some factors of a kernel to derive the average NoC and DRAM consumption of a memory request and then achieve the overall NoC and DRAM bandwidth partitioning by managing the request issue rate.

There are three possible locations to place the IRC in an SM: the warp scheduler, the load-store unit, and the L1 D-cache miss queue. The warp scheduler cannot control the bandwidth consumption accurately because it is not aware of the memory coalescing effect. For the option of controlling the request rate at the load-store unit, the accesses, which turn out to be L1 D-cache hits, can be affected even they have no impact on the NoC (or DRAM) bandwidth consumption. Therefore, our choice is to control the issue rate of each L1D miss queue, i.e., how many L1D misses are sent to the L1-to-L2 NoC per time unit.

Based on the kernel type detection and the feedback from the IRCs, the CCBP algorithm deter-
mines the optimal CTA combination and bandwidth partitioning. Our key insight is that different types of kernels have different ‘dominant’ resources. The dominant resource of the latency-sensitive kernels is the CTA number because they require higher levels of TLP to hide the memory latency. For the NoC- or DRAM-intensive kernels, the dominant resource is the NoC or DRAM bandwidth, respectively.

4.5.2 CCBP Algorithm

As observed in Section 4.4, if we let the co-running kernels freely compete for the memory resources, the overall bandwidth demand may exceed the peak bandwidth. The resulting high queuing delays will significantly affect latency-sensitive kernels. However, if we evenly partition the bandwidth, some CTA and bandwidth resources may be wasted. So, there are three objectives of CCBP: (1) fully utilize both the CTA and bandwidth resources; (2) avoid bandwidth over-subscription by the bandwidth-intensive kernels; (3) fairly distribute the resources among the co-running kernels based on their dominant resources.

To achieve the objectives, our CCBP algorithm, as shown in Algorithm 1, leverages the ‘Dominant Resource Fairness’ (DRF) [Gho11] approach to assign the CTA and bandwidth resources to different kernels. Compared to SMK [Wan16], which determines the CTA combination using the DRF algorithm on static resources, our CCBP approach takes both static resources and dynamic bandwidth resources into account. In Algorithm 1, the \textit{type} input contains the kernel type of each kernel. In Section 4.5.4, we will describe how the CCBP approach classifies the kernels into different types, i.e., latency sensitive, NoC intensive and DRAM intensive. The \textit{nbw} and \textit{dbw} inputs contain the NoC and DRAM bandwidth utilization and the computation of such utilization is described in Section 4.5.4. The \textit{priorLSK} input is used to adjust the priority of the latency-sensitive kernels, which will be discussed in Section 4.5.5.

The \textit{domk} value maintains the dominant resource utilization for kernel \(k\). If the kernel is latency sensitive, the dominant resource are the static resources (register file, shared memory or thread number) that limit the CTA number. On the other hand, the dominant resource is the NoC or DRAM bandwidth.
Data: \( \text{type}_{1..K} \): kernel types; 
\( nbw_{1..K}[1..N] \): NoC BW util with varied CTA numbers; 
\( dbw_{1..K}[1..N] \): DRAM BW util with varied CTA numbers; 
\( \text{priorLSK} \): priority for latency-sensitive kernels;

Result: \( CTA_{1..K} \): number of CTAs for each kernel; 
\( NBW_{1..K} \): NoC bandwidth quota; 
\( DBW_{1..K} \): DRAM bandwidth quota;

1. \( CTA_{1..K} \leftarrow [0], NBW_{1..K} \leftarrow [0], DBW_{1..K} \leftarrow [0], \text{dom}_{1..K} \leftarrow [0]; \)
2. \( \text{unFinished} \leftarrow [1..K]; \)
3. \( \text{while} \ \text{find} \ k \ \text{with} \ \text{lowest} \ \text{dom}_k \ \text{in} \ \text{unFinished} \) do
4.  Allocate resources tentatively
5.  \( \text{if} \ \text{type}_k = \text{Latency\_Sensitive} \) then
6.     \( \text{tmpCTA}_k \leftarrow CTA_k + 1; \)
7.     \( \text{tmpNBW}_k \leftarrow nbw_k[\text{tmpCTA}_k]; \)
8.     \( \text{tmpDBW}_k \leftarrow dbw_k[\text{tmpCTA}_k]; \)
9.  \text{else if} \ \text{type}_k = \text{NoC\_Intensive} \) then
10.     \( \text{tmpNBW}_k \leftarrow NBW_k + 1; \)
11.     \text{if} \ nbw_k[CTA_k] < \text{tmpNBW}_k \) then
12.         \( \text{tmpCTA}_k \leftarrow CTA_k + 1; \)
13.     \text{else}
14.         \( \text{tmpCTA}_k \leftarrow CTA_k; \)
15.         \( \text{tmpDBW}_k \leftarrow dbw_k[\text{tmpCTA}_k] \times \frac{\text{tmpNBW}_k}{nbw_k[\text{tmpCTA}_k]}; \)
16.  \text{else}
17.     \( \text{tmpDBW}_k \leftarrow DBW_k + 1; \)
18.     \text{if} \ dbw_k[CTA_k] < \text{tmpDBW}_k \) then
19.         \( \text{tmpCTA}_k \leftarrow CTA_k + 1; \)
20.     \text{else}
21.         \( \text{tmpNBW}_k \leftarrow nbw_k[\text{tmpCTA}_k] \times \frac{\text{tmpDBW}_k}{dbw_k[\text{tmpCTA}_k]}; \)
\> Check if there are enough resources
22. \text{if} \ \text{CanAlloc}(\text{tmpCTA}, \text{tmpNBW}, \text{tmpDBW}) \) then
23.     \( \text{CTA}_k \leftarrow \text{tmpCTA}_k; \)
24.     \( \text{NBW}_k \leftarrow \text{tmpNBW}_k; \)
25.     \( \text{DBW}_k \leftarrow \text{tmpDBW}_k; \)
26.     \( \text{dom}_k \leftarrow \text{DomShareCal}(k, \text{CTA}_k, \text{NBW}_k, \text{DBW}_k, \text{priorLSK}); \)
27. \text{else}
28.     \( \text{unFinished} \leftarrow \text{unFinished} - k; \)
29. end

Algorithm 1: CCBP algorithm based on DRF.

bandwidth for NoC- and DRAM-intensive kernels, respectively.

Similar to previous GPU intra-SM sharing approaches [Pai13; Wan16; Xu16; Dai18], in CCBP, all SMs share the same CTA combination. And the bandwidth allocation for each kernel is evenly distributed to each SM. The reason is that the CTAs in the same kernel typically exhibit very similar
Table 4.3 An example of CCBP algorithm. Assume that, without bandwidth management, the CTA/NoC/DRAM consumption ratios for kernels K1, K2 and K3 are 2:1:1, 1:2:4 and 1:3:1.

<table>
<thead>
<tr>
<th>Iter 1</th>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>CTA Number</th>
<th>NoC Quota</th>
<th>DRAM Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td>1</td>
<td>T</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>K3</td>
<td></td>
<td></td>
<td>1</td>
<td>T</td>
<td>0.5</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter 2</th>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>CTA Number</th>
<th>NoC Quota</th>
<th>DRAM Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>T</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td>1</td>
<td>T</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>K3</td>
<td></td>
<td></td>
<td>1</td>
<td>T</td>
<td>0.5</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter 3</th>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>CTA Number</th>
<th>NoC Quota</th>
<th>DRAM Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>T</td>
<td>1</td>
<td>T</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>K3</td>
<td></td>
<td></td>
<td>1</td>
<td>T</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter 6</th>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>CTA Number</th>
<th>NoC Quota</th>
<th>DRAM Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>T</td>
<td>2</td>
<td>T</td>
<td>6</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>2.5</td>
</tr>
<tr>
<td>K3</td>
<td></td>
<td></td>
<td>2</td>
<td>T</td>
<td>5</td>
<td>1.66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter 11</th>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>CTA Number</th>
<th>NoC Quota</th>
<th>DRAM Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>T</td>
<td>4</td>
<td>T</td>
<td>6</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2.5</td>
</tr>
<tr>
<td>K3</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter 12</th>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>CTA Number</th>
<th>NoC Quota</th>
<th>DRAM Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>T</td>
<td>4</td>
<td>T</td>
<td>7</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1.66</td>
</tr>
<tr>
<td>K3</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter 16</th>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>CTA Number</th>
<th>NoC Quota</th>
<th>DRAM Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>T</td>
<td>6</td>
<td>T</td>
<td>9</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>K2</td>
<td></td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>K3</td>
<td></td>
<td></td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>1.33</td>
</tr>
</tbody>
</table>

characteristics, comparing to the CTAs in different kernels.

The algorithm works as follows. Initially, resources assigned to all kernels are set to zeros. Each iteration in Algorithm 1 finds a kernel with the lowest $dom_k$ value. Then the dominant resource allocated to that kernel is tentatively increased by 1 unit. For latency-sensitive kernels, as their dominant resource is the CTA number per SM, 1-unit resource means the static resources needed to accommodate one CTA on every SM. For NoC and DRAM bandwidth sensitive kernels, we evenly divide the dynamic bandwidth resource into M units. And 1-unit resource means 1 unit of dynamic bandwidth. The higher M, the finer granularity of bandwidth control. We find that when M is higher than 20, the algorithm output as well as the overall performance has very little difference. Therefore, we set M as 20 in this paper. For bandwidth-intensive kernels, their CTA numbers are increased only when the assigned bandwidth exceeds the overall bandwidth demand of the current CTA number.

In each iteration of resource allocation, CCBP checks whether the total static resource and dynamic bandwidth available can accommodate such allocation. If so, the resources are allocated accordingly and the $dom_k$ is updated using the function $DomShareCal$, which is described in Section 4.5.5. Otherwise, the resource allocation is finished.

Table 4.3 shows an example of the CCBP procedure on three kernels, K1, K2 and K3. Assume that K1 is latency sensitive and each CTA of K1 consumes 0.5 NoC bandwidth unit and 0.5 DRAM unit. K2 is DRAM intensive and we assume that each CTA of K2 consumes 2-unit NoC bandwidth and 4-unit DRAM bandwidth. K3 is NoC intensive and each CTA consumes 3-unit NoC bandwidth and 1-unit DRAM bandwidth. We assume each type of resource has a total of 10 units. In each iteration of CCBP, the algorithm allocates one unit of the dominant resource for the kernel with the minimum dominant share. Table 4.3 shows the status after a certain number of iterations. In each iteration, we
highlight the kernel which is selected to increase its resources and its dominant resource. In the first iteration, when all kernels has zero dominant shares, CCBP selects the latency-sensitive kernel K1 and increases its CTA number by 1. And its NoC and DRAM quotas are increased proportionally to 0.5. In iteration 2, K2 is selected and its dominant resource, i.e., the DRAM bandwidth, is increased by 1 unit and the NoC bandwidth quota is increased proportionally by 0.5 unit. Similarly, the NoC quota of K3 is increased in iteration 3 to 1 unit and its DRAM quota is increased proportionally to 0.33 unit. At the end of iteration 11, the NoC-intensive kernel K3 has the minimum dominant share (3 units of NoC bandwidth) compared to the other two kernels. So, in iteration 12, its NoC quota is increased to 4. Because 1 CTA of K3 is not able to consume 4 units of NoC bandwidth, its CTA number is increased by 1. The CCBP algorithm terminates at iteration 16 when no more CTAs can be allocated. At this time, we can see that the dominant resources of all kernels have been fairly allocated.

After the resource allocation for all kernels is finished, CCBP returns the newly determined CTA combination and bandwidth quotas. Then, the static resources are re-partitioned to accommodate the new CTA combination. The assigned bandwidth quotas are converted to the L1 D-cache miss request issue rate quotas using the bandwidth management approach discussed in Section 4.5.3.

The CCBP algorithm can also be applied for kernels with more than one limiting resource. For example, a kernel can have similar NoC and DRAM resource utilization and both are limiting resources. In this scenario, the dominant resource can be detected as either limiting resource. With the CCBP algorithm, whenever one share of the dominant resource is allocated, a proportional quota in the other resources, including the other limiting resources, are allocated accordingly. Therefore, the algorithm remains effective.

4.5.3 Bandwidth Management

We first illustrate some of the key concepts of our approach using Figure 4.7.

Figure 4.7 shows 4 types of memory transactions, load or store and hit or miss in the L2 cache. The black dots in NoC and DRAM denote that the bandwidth is consumed by such a transaction.
Figure 4.7: 4 memory transaction types, black dots represent the bandwidth is consumed at a particular component.

For stores hit in the L2 cache, as we focus on the bandwidth consumption of NoC in the L2-to-SM direction and the size of store acknowledgements is small, we ignore the NoC bandwidth consumption by such store transactions. For a load transaction, the NoC bandwidth is consumed no matter it hits or misses the L2 cache. So, the NoC bandwidth consumption is proportional to the number of replied L1 cache lines per time unit, which equals to the number of load requests per time unit. The DRAM bandwidth is consumed only when a load/store transaction misses the L2 cache. Therefore, the DRAM bandwidth consumption is proportional to the product of the request issue rate and the average DRAM transactions per request. In our approach, we collect the statistics of the average DRAM transactions per memory transaction in the L2 cache. And such statistics is piggybacked in the replied packets. This architectural support will be discussed in Section 4.6.

Next, we derive the relationship between the L1 D-cache miss queue issue rate and the NoC/DRAM bandwidth consumption.

For kernel $k$, given the L1 D-cache miss request issue rate $IssueRate_k$ (issued requests per time unit), the L2-to-SM NoC bandwidth consumption $BWReq_n^k$ can be derived as:

$$BWReq_n^k = L2L1ReplyRate \times \#L1Caches \times ReplySize$$

$$= L1L2ReadRequestRate \times \#SMs \times ReplySize$$

$$= IssuedReadReqPerTimeUnit \times \#SMs \times ReplySize$$

$$= rf_k \times IssueRate_k \times \#SMs \times (L1BlkSize + FlitSize)$$

(4.1)
In which, $r f_k$ is defined as the ratio of the read requests over the total memory requests:

$$rf_k = \frac{\#ReadRequests_k}{\#Requests_k} \quad (4.2)$$

Because we only consider the NoC bandwidth consumption by load transactions, the number of transactions sent to each SM is $rf_k \times IssueRate_k$ per time unit. Each transaction or packet through the NoC includes the payload and a header. The payload is the L1 D-cache block replied from the L2 cache. The header, which occupies a flit, contains the request meta data such as the destination SM id and warp id.

Likewise, the DRAM bandwidth consumption for kernel $k$ can be derived as a function of the L1 D-cache miss queue issue rate $IssueRate_k$:

$$BWReq_d^k = df_k \times IssueRate_k \times L2BlkSize \times \#SMs \quad (4.3)$$

In which, $df_k$ denotes the number of DRAM accesses generated per L2 cache request:

$$df_k = \frac{\#DRAMAccesses_k}{\#Requests_k} \quad (4.4)$$

Therefore, $df_k \times IssueRate_k$ is the number of DRAM accesses per time unit. Note that $df_k$ has a subtle difference from the L2 cache miss rate. The reason is that when an L2 cache request misses the L2 cache and the victim cache block is dirty, the number of DRAM accesses is 2. The DRAM access granularity is $L2BlkSize$ per request in our model.

From Equations 4.1 and 4.3, we can see that the factors, $rf$ and $df$, of each kernel are the keys to control the NoC and DRAM bandwidth consumption using the request issue rate. Therefore, before partitioning the NoC/DRAM bandwidth, $rf$ and $df$ need to be collected using Equations 4.2 and 4.4, which can be done during the kernel-type detection step.
4.5.4 CCBP-Parameter Detection

In this section, we discuss how the input parameters of CCBP algorithm are detected. These input parameters include the kernel types and the relationship between the CTA number and the corresponding NoC/DRAM bandwidth utilization.

**Even Partitioning**

The input parameters are detected using even CTA combination and bandwidth partitioning. Under even CTA combination, the CTA number of each kernel is set to \( \frac{\# \text{Alone Run CTA}_k}{\# \text{Kernels}} \). Then, the NoC and DRAM bandwidth are also partitioned evenly, which means to assign each kernel an even share of the NoC and DRAM bandwidth. To achieve even partitioning, the IssueRate\(_k\) of each kernel is set to the maximum value that satisfies the even-share conditions, which can be represented as:

\[
I_{\text{Even}}^k = \text{Max}(\text{IssueRate}_k): \text{BWReq}_n^k \leq \frac{\text{SustainBW}_n}{\# \text{Kernels}}, \\
\text{BWReq}_d^k \leq \frac{\text{SustainBW}_d}{\# \text{Kernels}}.
\] (4.5)

The issue rate under such conditions can be solved using Equations 1 and 3 and is named \( I_{\text{Even}}^k \) for kernel \( k \). For example, between 2 co-running kernels, either is assigned with 1/2 sustainable NoC bandwidth and DRAM bandwidth. Such an allocation is then converted to two issue rates using Equations 1 and 3 respectively. The \( \text{SustainBW}_n \) and \( \text{SustainBW}_d \) denote the NoC and DRAM sustainable bandwidth. As discussed in Section 4.3, in our model, the NoC and DRAM sustainable bandwidth are 60% and 70% to their theoretical peak bandwidth. When other types of NoC or memory are used for GPU, these two parameters need a one-time update to reflect the sustainable bandwidth on the corresponding technologies.

**Kernel-Type Detection**

With the issue rates of all kernels being set based on Equation 4.5, we decide a kernel as bandwidth intensive if its average memory request demand has an equal or higher rate than the even-partitioned issue rate, i.e., \( \text{IRDemand}_k \geq I_{\text{Even}}^k \). Otherwise, the kernel is considered latency
sensitive.

A bandwidth-intensive kernel can be further classified as NoC intensive or DRAM intensive based on its relative NoC and DRAM bandwidth utilization. Specifically, a kernel is NoC intensive if:

$$\frac{BWReq_k^n}{SustainBW_n} > \frac{BWReq_k^d}{SustainBW_d}$$  \hspace{1cm} (4.6)

Take Equations 4.2-4.3 into Inequation 4.6, we get:

$$\frac{df_k}{rf_k} < \frac{SustainBW_d}{SustainBW_n} \times \frac{L1BlkSize + FlitSize}{L2BlkSize}$$  \hspace{1cm} (4.7)

If this inequality holds, the kernel is NoC intensive. Otherwise, it is DRAM intensive. Intuitively, $df_k$ in Inequation 4.7 is positively correlated to the L2 cache miss rate. If the L2 cache miss rate is so low that a substantial amount of requests are filtered, the NoC becomes the bottleneck in the memory system. Otherwise, the DRAM is the main resource to reply the requests and the DRAM bandwidth is the bottleneck. This conclusion can be confirmed from Table 4.2, which shows that NoC-intensive kernels have much lower L2 miss rates than DRAM-intensive kernels.

**Bandwidth-Utilization Detection**

Once the kernel types are detected, our CCBP approach collects the bandwidth resource demand/utilization with different CTA numbers, i.e., the $nbw$ and $dbw$ inputs in Algorithm 1, for each kernel.

For latency-sensitive kernels, CCBP assumes the bandwidth utilization scales linearly with the CTA number. Therefore, it leverages the average bandwidth utilization per CTA to calculate the bandwidth utilization for different CTA numbers. For a bandwidth-intensive kernel, CCBP learns its bandwidth utilization with different CTA numbers using a throttling approach. First, it releases the bandwidth limitation for this kernel and throttles a certain number of CTAs in this kernel. When a CTA is throttled, the warps in the CTA cannot be selected by the warp scheduler to issue any instructions. Under such circumstances, the bandwidth utilization is collected for this particular number of active CTAs.
Static or Dynamic Detection

In our approach, the CCBP parameters can be either detected online or offline. In order to capture the interference among the co-running kernels, either online or offline detection is performed under co-running environment with even CTA combination.

With online detection, our approach skips the first 50K cycles for warming up. The next 20K cycles are used to collect the $r_f$ and $d_f$ factors as defined in Section 4.5.3. Then, the NoC and DRAM bandwidth can be partitioned evenly using Equation 4.5. Under even CTA combination and bandwidth partitioning, we determine the kernel types as described above. To detect the bandwidth utilization with varied CTA numbers of a bandwidth-intensive kernel, we release the bandwidth limitation of such a kernel and throttle its active CTA number. For each CTA number of interest, we skip the 10K cycles and use the next 10K cycles to collect the bandwidth utilization.

In offline detection, the procedure is the essentially same as online detection except that we use 10x longer detection time in each step to improve the accuracy.

4.5.5 CCBP with Adjustable Priorities

Although the CCBP algorithm as described in Section 4.5.2 fairly allocates the resources, we observe that the latency-sensitive kernels can still suffer from higher performance slowdown when co-running with bandwidth-intensive kernels. The reason is that although our approach can significantly reduce the memory latency in the CKE environment, it is still higher than the latency when a latency-sensitive kernel runs alone. In order to achieve the fairness as well as high throughput, the CCBP approach offers higher priority to latency-sensitive kernels.

As shown in Algorithm 1, the \textit{DomShare Cal} function calculates the dominant resource utilization of a kernel. For a NoC-/DRAM-intensive kernel, the result is the ratio of assigned NoC/DRAM bandwidth and the total sustainable NoC/DRAM bandwidth, i.e.,

$$dom_{bwk} = \frac{\text{Assigned DomBW}_{bwk}}{\text{Sustain DomBW}_{bwk}}$$
For a latency-sensitive kernel, the result is scaled with the priority:

\[
dom_{lsk} = \frac{\text{Assigned } \text{Dom Static Res}_{lsk}}{\text{Total Dom Static Res}_{lsk}} \times \text{priorLSK}
\]

The static resources of a kernel are the registers, shared memory and thread number and the dominant static resource is the resource with the highest utilization. The priorLSK is the priority for the latency-sensitive kernels. Lower priorLSK means higher priorities. When priorLSK is 1.0, latency-sensitive kernels have equal priority as bandwidth-intensive kernels.

In our approach, the priorLSK is set to 1.0 in the beginning. Then it is adjusted using a hill climbing algorithm and priorLSK is decreased by 0.1 in each iteration. For each priority, the CCBP algorithm determines a certain resource assignment and monitors the performance while running for 100K cycles using such an assignment. The performance goal can either be Wspeedup or Hspeedup as shown in Section 4.3. The adjustment of priorLSK stops when the performance reaches the highest value.

4.6 Architectural Support

In this section, we discuss how the issue rate controller (IRC) in Figure 4.6 is implemented. Figure 4.8 shows the architecture of an IRC. To monitor the bandwidth consumption, IRC collects memory request issue rate of each kernel. In order to transform the issue rate to the bandwidth consumption, the rf and df factors are monitored according to the Equations 4.2 and 4.4. After the CCBP
algorithm determines the bandwidth partitioning, the corresponding issue rate quotas are sent to
the IRC.

4.6.1 Virtual Queues

As shown in Figure 4.8, we implement the miss queue as a set of sub-queues between the L1 D-cache
and the NoC such that each co-running kernel has its private queue, which is referred to as a virtual
queue. The total number of the sub-queues equals the maximum number of kernels that an SM can
accommodate. When the kernel number is less than the number of sub-queues, the sub-queues are
evenly assigned to co-running kernels. When a memory request is generated (i.e., an L1 D-cache
miss), it is sent to a certain virtual queue based on the kernel id of the request. A virtual queue
partitioning vector is maintained to map the kernel ids to the tails of the virtual queues.

4.6.2 Request Arbitrator

The Issue Rate Credit Vector (IRCV) contains the number of credits assigned to each kernel. At each
time interval (200 cycles in our model), the credits are reset to the issue rate quotas. In each cycle, the
arbitrator only selects requests from kernels with non-zero credits. The credit of one kernel decreases
by one when the arbitrator issues one request from this kernel. Among the kernels with non-zero
credits, the arbitrator prioritizes the requests of latency-sensitive kernels. This way, memory requests
from latency-sensitive kernels will not be delayed by those from bandwidth-intensive co-running
kernels.

In our evaluation, a 1-cycle latency is simulated for the arbitrator. The performance impact of
the memory request arbitrator is negligible because it is not on the critical path. The NoC in our
model can only transmit one 32-byte packet per cycle. However, one memory transaction contains
a 128-byte cache block and a 32-byte header. So, the maximum number of requests can be issued
in a 200-cycle time unit is 40. In our experiments, we vary the arbitrator latency from 1 to 4 cycles.
The results show that the overall performance slowdown is only 0.11% – 0.72%. The reason is that
the added latency is only for memory accesses that miss the L1 D-cache. In addition, the L2 cache
access latency is already high on GPUs even without queuing delays. And it becomes much higher when a bandwidth-intensive kernel is running, as shown in Figure 4.2. Therefore, the few-cycle arbitrator latency does not affect performance much.

### 4.6.3 Tracking and Monitoring DRAM Accesses for $df$ factors

To retrieve the number of DRAM accesses for each request, we add a 2-bit field in the memory transaction header. As the NoC transmits the transactions in the unit of a flit, the header size is one flit (32B). So this extra 2-bit field does not increase the NoC bandwidth consumption.

We also add the following relatively simple logic in the L2 cache to track the number of DRAM accesses resulting from a request. For an L2 hit, the number is filled with 0. For an L2 miss which leads to a non-dirty block eviction, the number is 1. If the victim block is dirty, the number is 2. The DF monitor collects the DRAM access numbers in the replied transaction header to calculate the $df$ factor defined in Equation 4.4.

### 4.6.4 Overall Hardware Cost

**Sub-Queues** We reuse the existing L1 D-cache miss queue to implement the sub-queues as shown in Figure 4.8.

Given the number of sub-queues $N$, which is also the number of maximum kernels in each SM, $3N – 1$ sets of wires and $N – 1$ multiplexers are added to implement the sub-queues.

**Per Kernel Counters/Registers** The Virtual Queue Partitioning Vector has $N$ entries, each of which has $\log_2 N$ bits to point to the tail sub-queue of a kernel.

The Issue Rate Credit Vector has $N$ entries and each entry has 8 bits as the maximum issue rate is 200 per interval.

In the RF monitor, two 32-bit counters per kernel are maintained to monitor the $rf$ factor as defined in Equation 4.2. Similarly, in the DF monitor, two 32-bit counters are used to monitor the $df$ factor as defined in Equation 4.4.
4.7 Evaluation

4.7.1 2-Kernel Co-Runs

In this section, we evaluate all 2-kernel combinations under the categories of 2 latency-sensitive kernels (labeled as ‘LS+LS’), 1 latency-sensitive and 1 bandwidth-intensive kernel (labeled as ‘LS+BI’), 1 NoC-intensive and 1 DRAM-intensive kernel (labeled as ‘NBI+DBI’). We do not combine two kernels that stress the same bandwidth resource because the overall throughput cannot be improved by such co-runs. Figure 4.9 shows the average results for each category using the geometric mean and the overall results are labeled as ‘ALL’. In Figure 4.10, we report the harmonic speedup of each individual combination from ‘LS+BI’ and ‘NBI+DBI’ categories. SMK [Wan16] is a prior work which leverages both CTA-level and warp-scheduler-level partitioning to improve CKE. WS [Xu16] is another prior work which uses the scalability curves to determine the CTA combinations. The
CTA_oracle is the best result through exhaustively searching all possible CTA combinations using intra-SM sharing. Spatial_oracle shows the results using Spatial Multitasking [Adr12a; Wan18], in which co-running kernels are evenly partitioned to different SMs. As pointed out by Wang et. al [Wan18], the CTA combinations also have significant performance impact on Spatial Multitasking. Therefore, in Spatial_oracle, all possible CTA combinations are searched to achieve the optimal performance. Note that, for both CTA_oracle and Spatial_oracle, the optimal CTA combinations maybe different for the highest harmonic speedup or the highest weighted speedup. CCBP_DRF is the CCBP approach without adjustable priorities for latency-sensitive kernels. CCBP_DRF+adj is the CCBP approach with adjustable priorities. For these two approaches, the inputs of CCBP algorithm are detected online using the method described in Section 4.5.4. CCBP_offline is the CCBP approach with the parameters and priorities of latency-sensitive kernels determined offline. In the subsequent sections, the CCBP_DRF+adj approach is used as default if not specified.

For the ‘LS+LS’ category, bandwidth partitioning has limited impact. So, our CCBP approach degrades to choosing the CTA combination by fairly partitioning the static resources, which is the same as the DRF approach used by SMK. For this category, we find that the performance of DRF static resource partitioning is very close to CTA_oracle. For Spatial_oracle, however, the performance is worse than intra-SM sharing approaches because it fails to improve the utilization of on-chip resources, such as register file and shared memory. For the ‘LS+BI’ category, our CCBP approach can significantly reduce the memory latency, which leads to the performance improvement of the latency-sensitive kernels. Compared to CTA_oracle, CCBP improves the harmonic speedup by 11% and improves the weighted speedup by 7% on average. For the ‘NBI+DBI’ category, the harmonic speedup and weighted speedup can be improved by 12% and 5% on average, respectively, over CTA_oracle. Compared to SMK and WS, CCBP improves the average harmonic speedup by 78% and 39% respectively. For the weighted speedup, the average improvements are 21% and 12% respectively. Compared to Spatial_oracle, the CCBP approach improves the harmonic speedup and weighted speedup by 28% and 18%, respectively.
4.7.2 Impact on LS+BI Co-Runs

We use a case study, BP+CFD co-run, to illustrate the impact of CCBP when a latency-sensitive kernel (BP) co-runs with a bandwidth-intensive kernel (CFD). Figure 4.11(a) shows the performance slowdown of BP and CFD. With the oracle CTA combination, the latency-sensitive kernel BP suffers from much higher performance slowdown than the bandwidth-intensive kernel CFD. The reason is that the average memory access latency is significantly increased by the bandwidth-intensive kernel. Figure 4.11(b) shows the average memory access latency for BP, normalized to the latency of its standalone execution. With the CCBP approach, the memory access latency can be effectively reduced, leading to the performance improvement of the latency-sensitive kernel. On the other hand, the critical bandwidth utilization, as shown in Figure 4.11(c), is maintained at high levels. So the performance of the bandwidth-intensive kernel is not impaired significantly. The harmonic speedup for BP+CFD is improved by 25% with CCBP over CTA_oracle as shown in Figure 4.10.

For kernel BP, we illustrate the stall cycle breakdown of the warp scheduler in Figure 4.11(d). When a warp instruction is stalled, we accumulate the total stall cycles and attribute each stall into one of three reasons. If a dependent register is waiting for a memory fetch that missed the L1 D-cache, the stall is categorized as ‘RAW_long’ in the figure. If the instruction cannot be issued because the LSU is full, the stall cycle is categorized as ‘LSU_full’. We categorize the stalls due to any other reasons as ‘Others’ in the figure. The figure shows the percentage of each category to the total stall cycles. When BP runs alone, ‘RAW_long’ accounts for 43% of the stall cycles and ‘LSU_full’ accounts for 88%.
for a very small portion. When BP co-runs with CFD, the percentage of ‘RAW_long’ is increased to 56% of the total stalls because the memory latency is increased. In addition, the percentage of ‘LSU_full’ is increased to 20% because the memory pipeline is dominated by CFD. With the CCBP approach, the ‘RAW_long’ is reduced to 49% because the memory latency is reduced. Besides, the ‘LSU_full’ percentage is reduced to 8% because the sub-queue structure can effectively prevent the memory pipeline being dominated by CFD. In this figure, we compare against the CTA_oracle with the MIL scheduling policy [Dai18]. MIL throttles the memory instruction issuing when the L1 D-cache reservation failure rate is high and releases the limit when the rate is low. As shown in the figure, although MIL can also reduce the congestion of the memory pipeline, it is less effective than CCBP for the same reasons that are discussed in Section ??.

### 4.7.3 Impact on NBI+DBI Co-Runs

Figure 4.12 shows the instantaneous NoC and DRAM bandwidth utilization of CFD+FTD. CFD is NoC intensive while FTD is DRAM intensive. As shown in Figure 4.12(a), with the oracle CTA combination, both NoC and DRAM exhibit very bursty behavior. As pointed out in Section 4.4, over-subscribed NoC bandwidth in a period can cause underutilized DRAM bandwidth and vice
versa. Because CCBP accurately controls both bandwidths, as shown in Figure 4.12(b), the burstiness of NoC and DRAM bandwidth utilization is significantly reduced. Compared to CTA_oracle, CCBP improves the average NoC and DRAM bandwidth by 11% and 30%, respectively, leading to a 16% improvement in harmonic speedup, as shown in Figure 4.10.

4.7.4 3- and 4-Kernel Co-Runs

In Figure 4.13, we evaluate CCBP with 3 and 4 co-running kernels, each set has 40 randomly selected kernel combinations. Because the searching space for the oracle CTA combinations becomes too large, we choose to compare CCBP with prior works SMK [Wan16] and WS [Xu16]. For 3-kernel co-runs, CCBP improves the harmonic speedup by 46% and weighted speedup by 18% over WS. For 4-kernel co-runs, CCBP improves the harmonic speedup by 48% and weighted speedup by 21% over WS.

When there is more than one bandwidth-intensive kernel stressing the same critical bandwidth, the memory access latency is much more aggregated without bandwidth-aware management. Take HG+LBM+SAD as an example, because both LBM and SAD are DRAM intensive, the latency-sensitive kernel HG suffers from extremely high memory access latency. Compared to its standalone execution, the memory access latency is increased to 4.8x and 4.5x with SMK and WS, respectively. In contrast, with CCBP, the average memory access latency is only 40% higher than it in its standalone execution.
4.7.5 Comparison with Hybrid Approaches

We compare CCBP with four hybrid approach that integrates WS [Xu16] CTA combination and different cache partitioning and memory scheduling approaches, including TCM [Kim10b], BMI [Dai18], MIL [Dai18], and UCP [QP06]. Figure 4.14 shows the harmonic speedup of these approaches normalized to the WS approach alone for 2-kernel co-runs.

TCM is an application-aware memory scheduling policy which prioritizes latency-sensitive applications in the DRAM memory controller. This approach is not effective for ‘LS+NBI’ and ‘NBI+DBI’ co-running kernels because it does not consider the NoC bandwidth utilization. For the ‘LS+DBI’ category, the memory request prioritization can improve the performance of latency-sensitive kernels through reducing the memory latency.

BMI and MIL are memory instruction scheduling policies for GPU CKE. Through controlling the memory instruction priority, BMI issues memory requests from co-running kernel in a balanced manner. BMI does not improve the performance for ‘LS+NBI’ and ‘LS+DBI’ kernels because it cannot effectively reduce the memory latency. MIL throttles the memory instruction issuing when the L1 D-cache reservation failure rate is high. Reservation failure happens when the L1 D-cache doesn’t have enough resource to handle cache misses. In our experiments, we observe that NoC-intensive kernels have high L1 D-cache reservation failure rate. However, for DRAM-intensive kernels, the contention mainly occurs at L2-to-DRAM instead of L1 D-cache. Therefore, MIL is less effective for ‘LS+DBI’ than ‘LS+NBI’. In contrast, our proposed CCBP is effective for both cases.

UCP is a cache partitioning method based on the cache way utility of each co-running application. It assigns more cache resources to an application if it is likely to reduce more cache misses. In our experiment, UCP is implemented on the L1 D-cache. As shown in Figure 4.14, in general, UCP has a negative performance impact on GPU CKE. The reason is that the L1 D-cache is entirely dominated by the bandwidth-intensive kernels without cache partitioning. With cache partitioning, the actual cache size for the bandwidth-intensive kernel is reduced, which further increases the bandwidth contention. This phenomenon is also observed by H. Dai et al. [Dai18].

A common limitation of the hybrid approaches is that they are not able to control the bandwidth
coordinately with the CTA combination. When the bandwidth is saturated, some CTA resources for the bandwidth-intensive kernels are wasted. In our CCBP approach, these CTA resources can be re-allocated to the latency-sensitive kernels to further improve their performance.

4.7.6 Sensitivity Studies

In this section, we analyze the sensitivity of CCBP to the sub-queue sizes and warp scheduling policies. In Figure 4.15, we vary the sub-queue size from 8 to 32 entries while fixing the number of sub-queues to 8. So the total queue size varies from 64 to 256. The figure shows the normalized harmonic speedup compared to the unlimited queue size. With 64-entry queues, the 2-, 3- and 4-kernel co-runs achieves 96%, 91% and 89% of the harmonic speedup with unlimited queue entries. With 128-entry queues, the achieved harmonic speedup is 99%, 96% and 94% respectively. And the harmonic speedup achieved with 256-entry queues is 100%, 98% and 98%. In general, more kernels concurrently running on each SM, more queue entries are needed to maintain high performance.
For 2-kernel co-runs, the 64-entry queues are sufficient to achieve high harmonic speedups.

Figure 4.16 shows the harmonic speedups of the CCBP approach for 2-kernel co-runs with different warp scheduling policies. The LRR policy is used as the default in our experiments. The greedy-then-oldest (GTO) policy runs a single warp until it stalls then picks the oldest ready warp. In the two-level (2LVL) policy [Nar11], the warps are divided into groups and it executes from only one group until all warps in that group are stalled. The round-robin policy is used for both intra- and inter- group scheduling. In Figure 4.16, ‘2LVL-x’ denotes the two-level policy with group size x. Maestro [Par17] proposed the LRR+GTO scheduling policy for CKE. In this policy, the GTO warp scheduling policy is used within each kernel to increase the single kernel performance while the LRR policy is used to determine which kernel to schedule. With the WS CTA partitioning algorithm, the LRR+GTO scheduling policy achieves the highest harmonic speedup, which is 15% higher than the 2LVL-8 policy. On the other hand, we observe that with our proposed CCBP, different scheduling policies achieve similar performance. The reason is that the memory requests are fairly managed with CCBP regardless the warp scheduling policy.

4.8 Related Work

GPU Concurrent Kernel Execution To support CKE on GPUs, Spatial Multitasking [Adr12a] partitions the GPU at the SM granularity, i.e., different kernels are assigned to different SMs. In order to improve the intra-SM resource utilization, SMK [Wan16], WS [Xu16] and Maestro [Par17] propose to dispatch the CTAs from different kernels to the same SMs and different approaches are proposed to determine the CTA combination. Based on intra-SM sharing, H. Dai et al. [Dai18] propose memory instruction scheduling policies to reduce the memory pipeline stalls. The differences from these works have been discussed in Sections 4.4.

A recent work [Wan18] proposes to control the TLP of each kernel through monitoring the effective bandwidth. Similar to Spatial Multitasking, their work evenly distributes the co-running kernels to different SMs, therefore the resources within an SM, including ALU, static resources and NoC ports, cannot be fully utilized. On the contrary, our work can achieve better resource utilization
because we leverage intra-SM sharing and the resources are assigned according to the kernels’ characteristics.

Jog et al. [Jog14; Jog15] propose memory scheduling policies on the GPU memory controller. Similar to TCM [Kim10b], they do not address the congestion in the NoC or the queues between L1 and L2 caches. Li et al. [LL16; LL17] propose a framework for CTA partitioning and cache bypassing based on off-line profiling. In comparison, our work dynamically detects the kernel type and partitions the critical resources accordingly. Targeting the single kernel running on GPUs, some memory scheduling approaches have been proposed to improve the GPU memory efficiency. Yuan et al. [Yua09] propose a memory scheduling policy at the NoC to improve the DRAM row buffer locality. A DRAM scheduling has been proposed by Lakshminarayana et al. [Lak12] to select between Shortest-Job-First (SJF) and FR-FCFS policies. Based on warp divergence characterization, a set of cache management and memory scheduling policies have been proposed by Ausavarungnirun et al. [Aus15] to reduce the negative effects of memory divergence. Jog et al. [Jog16] observe the latency tolerance of GPU cores are different for some applications and they propose a memory scheduling policy to prioritize the critical memory requests.

Besides the architectural support, some software approaches have also been proposed. Elastic kernel [Pai13] and SM-Centric [Wu15] are two compiler approaches to enable scheduling CTAs from different kernels on the same GPUs. Jiao et al. [Jia15] propose a power-performance model to select concurrent kernels which can mostly improve the energy efficiency. Kernelet [ZH14] divides a GPU kernel into multiple slices, each of which has variable occupancy to allow co-scheduling with other slices. Aguilera et al. [Agu14] improve the fairness of Spatial Multitasking by balancing the individual performance and the overall performance. Ukidave et al. [Uki14] extend the OpenCL run-time environment to explore several dynamic spatial multiprogramming approaches. Compared to the architectural approaches, the software approaches require source code modification.

Baymax [Che16] and Prophet [Che17] focus on system-level Quality of Service (QoS) support for GPUs. They leverage the PCIe data transfer model and kernel execution model to meet the QoS target. Wang et al. [Wan17] assumes the QoS target of a kernel can be transformed to an IPC goal. In order to
achieve the IPC goal, they propose to manage the warp scheduler quota and CTA combination. Our proposed accurate bandwidth allocation can be leveraged to provide more accurate QoS control.

**Memory Scheduling on CMP or Heterogeneous Systems** Numerous works have been proposed to improve memory scheduling on multi-core or chip-multiprocessor systems. Fair Queue [Nes06] provides a fair allocation of the memory system to each thread regardless its memory demand. N. Rafique et al. [Raf07] propose a bandwidth management scheme that periodically tunes the bandwidth to achieve desirable memory latency. PAR-BS [Str12] processes memory requests in batch to avoid unfairness and it exploits bank-level parallelism to achieve high throughput. ATLAS [Kim10a] prioritizes the threads that have attained least from the memory scheduler. TCM [Kim10b] groups the threads into latency-sensitive and bandwidth-intensive clusters and prioritizes latency-sensitive clusters. STFM [MM07], FST [Ebr10], MISE [Sub13] and ASM [Sub15] propose dynamic slowdown models to estimate the slowdown of each individual thread. Then, they prioritize the application that suffers from higher slowdown or throttle the one with lower slowdown. Their models cannot be directly used on GPUs because they don't consider the performance impact of the number of threads/CTAs. In addition, with their mechanisms, the thread/CTA number, i.e., TLP, is not managed to achieve the optimal performance for GPUs. MITT [ZW16] provides a fine-grain memory traffic shaping mechanism for cloud systems.

On heterogeneous architectures, where the CPU and GPU share the last level cache (LLC) and off-chip memory, some works [LK12; Aus12; Kay14; Zha16; Usu16] observe that the heavy memory traffic from the GPU can be detrimental to the CPU. They propose to prioritize the CPU memory requests or throttle the GPU requests to improve the overall performance. In contrast, we focus on managing the memory traffic among the SMs within a GPU.

**NoC Optimization on CMP or GPU systems** On CMP architectures, the NoC has been widely studied [Bol04; Lee08; Gro09; Das09; Das10; Gro11; Li12]. GSF [Lee08] proposes a QoS support for NoC by providing a minimum bandwidth and maximum latency guarantee. R. Das et al. [Das09; Das10] propose application-aware prioritization mechanisms on NoC routers to improve system-level fairness and throughput. Some previous works [Zia15; Jan15; Bak10] propose NoC optimizations
for GPU systems. However, none of these prior works considers concurrent kernel execution, in which the bandwidth traffic is mixed from different kernels and the latency impact is also different for different kernels.

4.9 Conclusions

In this paper, we focus on improving GPU concurrent kernel execution. We observe that bandwidth over-subscription from bandwidth-intensive kernels results in high queuing delays in the GPU memory system, which leads to severe performance degradation for co-running latency-sensitive kernels. In addition, among bandwidth-intensive kernels, the memory bandwidth can also be unfairly consumed due to their different memory request intensities. To address these problems, we first make a case that the CTA-level management alone is insufficient and reveal the fundamental reasons. Based on these observations, we propose a coordinated approach to control CTA combination and bandwidth partitioning.

Our approach allocates the CTA and bandwidth resources based on kernel-type detection and resource requirement of each kernel. It assigns more CTA resources for the latency-sensitive kernels and assigns more bandwidth resources for the bandwidth-intensive kernels. Besides effective bandwidth utilization, the memory latency is highly reduced due to bandwidth control and burstiness reduction. The experimental results show significant improvements in both throughput and fairness, comparing to the oracle CTA combination and the state-of-the-art GPU concurrent kernel execution methods.
This dissertation presents our studies on supporting GPU architecture for efficient GPU multiprogramming. The first work proposes a lightweight context switching to enable the instruction-level preemption efficiently. Besides the multiprogramming enabling, in our second work, we also leverage the fast context switching for improving the TLP levels of single kernels. In the third work, we propose a coordinated approach for improving the GPU concurrent kernel execution, which allows multiple kernels co-running on the same GPU simultaneously.

In the first work, we present lightweight context switching for GPUs and compiler-hardware co-design to enable efficient preemption. We propose three schemes, inplace context switching, liveness analysis and register compression, to address the problem of the large kernel context on GPUs. Our results show that with register liveness analysis and compression, the register context can be reduced drastically by 92%. With selective preemption enabling instructions, we can achieve efficient instruction-level preemption with an average preemption latency of 4.0us (with the 700MHz
In the second work, we analyze the relationship between GPU performance and TLP through a novel resource utilization perspective. The GPU performance can be bounded by the scheduler throughput, L1 bandwidth, interconnect bandwidth, L2 bandwidth or DRAM bandwidth. We reveal that many memory-intensive benchmarks are actually bounded by interconnect bandwidth. Then we highlight that for benchmarks not saturating the throughput of any resources, increasing TLP can lead to significant performance improvement. We propose GPUDuet, a fast context switching approach, to increase TLP to better hide the latency of memory load and barrier operations. GPUDuet leverages spare on-chip resources to enable fast context switching. Liveness analysis and register compression are used to reduce the spilled context size. The evaluation shows that GPUDuet achieves up to 47% performance improvement and 22% on average for a set of benchmarks with unsaturated throughput utilization.

In the third work, we focus on improving GPU concurrent kernel execution. We observe that bandwidth over-subscription from bandwidth-intensive kernels results in high queuing delays in the GPU memory system, which leads to severe performance degradation for co-running latency-sensitive kernels. In addition, among bandwidth-intensive kernels, the memory bandwidth can also be unfairly consumed due to their different memory request intensities. To address these problems, we first make a case that the TB-level management alone is insufficient and reveal the fundamental reasons. Based on these observations, we propose a coordinated approach to control TB combination and bandwidth partitioning. Our approach allocates the TB and bandwidth resources based on kernel-type detection and resource requirement of each kernel. It assigns more TB resources for the latency-sensitive kernels and assigns more bandwidth resources for the bandwidth-intensive kernels. Besides effective bandwidth utilization, the memory latency is highly reduced due to bandwidth control and burstiness reduction. The experimental results show significant improvements in both throughput and fairness, comparing to the oracle TB combination and the state-of-the-art GPU concurrent kernel execution methods.

There are some challenges left to be explored in our future work. First, as architecture security is
becoming more and more important, some multiprogramming designs need to be reconsidered. For example, the sharing of register file and shared memory using intra-SM sharing may result in vulnerability to attackers. Therefore, how to share the GPUs safely and efficiently is left as a future work. Second, the machine learning applications are becoming the most important workloads on GPU accelerators. For a GPU cluster which is concurrently training multiple machine learning models, how to co-run their kernels to improve the overall throughput is a challenge. In addition, as model inferences are much more sensitive to the execution latency compared to the model training, how to efficiently preempt the training kernels to meet the response requirements for the inference kernels is another challenge. Third, in this work, we assume the GPU memory capacity is large enough to accommodate the concurrent kernels. However, as GPU memory capacity is limited, how to share the GPU when the capacity is not large enough to accommodate the concurrent kernels is left as another future work.
BIBLIOGRAPHY


“NVIDIA Tesla P100 Whitepaper”. NVIDIA Corporation (2016).


