ACCELERATING PARALLEL TASKS BY OPTIMIZING GPU HARDWARE RESOURCE UTILIZATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Tsung Tai Yeh

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Timony R. Rogers, Chair
School of Computer and Engineering
Dr. Rudolf Eigenmann
School of School of Computer and Engineering
Dr. Samuel P. Midkiff
School of School of Computer and Engineering
Dr. T. N. Vijaykumar
School of School of Computer and Engineering

Approved by:

Dr. Timony R. Rogers
Head of the School Graduate Program

ACKNOWLEDGMENTS

I feel so grateful that I have had such a great opportunity over the past eight years to conduct my research work and turn it into my Ph.D. dissertation. I have to admit the path to the Ph.D. was full of challenges. Without the help of many people including my advisor, colleagues, friends, and family, this dream would never come true.

First, I want to appreciate each member of my dissertation committee that continuously cultivate me research skills and courage as I begin my graduate student life. Timothy G. Rogers, my Ph.D. advisor, demonstrates the value of seeking the root-cause of research problems and focusing intensely on pursuing solutions. Rudolf Eigenmann embodies professional effectiveness that inspires me to convey the research work succinctly and effectively. Samuel Midkiff and T. N. Vijaykumar are my role model for how to be a wonderful mentor to young graduate students.

I would also like to thank the colleagues who directly helped me on my research projects and life. They are all talented, hardworking and generous to share their ideas and thoughts. Their kindly sharing always scatters me the light when I was trapped by some difficulties. Here is everyone listed in alphabetical order: Roland Green, Akshay Jain, Mahmoud Khairy, Amit Sabne, Putt Sakdhnagool, and Mengchi Zhang.

I am also deeply appreciative of my internship supervisors at AMD research, Bradford M. Beckmann and Matt Sinclair. I feel so fortunate to have such wonderful mentors to give me guidance on designing practical micro-architectures. Their great jobs enrich my internship life and broaden my knowledge on the GPU architecture design. Larry Bihel was my supervisor at research computing center, Purdue University. I cherish his patience to direct me to understand insights of remote-sensing image analysis.

My work would not nearly be meaningful without unwavering supports from my friends and family. Thank for their accompany to help me go through this adventure.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Tsung Tai Yeh Ph.D., Purdue University, May 2020. Accelerating Parallel Tasks By Optimizing GPU Hardware Resource Utilization. Major Professor: Timothy G. Rogers.

Efficient GPU applications rely on programmers carefully structure their codes to fully utilize the GPU resources. In general, programmers spend a significant amount of time optimizing their applications to run efficiently on domain-specific architectures. To reduce the burden on programmers to utilize GPUs fully, I create several hardware and software solutions that improve the resource utilization on parallel processors without significant programmer intervention.

Recently, GPUs are increasingly being deployed in data centers to accelerate latency-driven applications, which exhibit a modest amount of data parallelism. The synchronous kernel execution on these applications cannot fully utilize the entire GPU. Thus, a GPU contains multiple hardware queues to improve its throughput by executing multiple kernels on a single device simultaneously when there are sufficient hardware resources. However, a GPU faces severe underutilization when the space in these queues has been exhausted, and the performance benefit vanishes with the decreased parallelism. As a result, I proposed a GPU runtime system – Pagoda, which virtualizes the GPU hardware resources by using an OS-like daemon kernel called MasterKernel. Tasks (kernels) are spawned from the CPU onto Pagoda as they become available, and are scheduled by the MasterKernel at the warp granularity to increase the GPU throughput for latency-driven applications. This work invents several programming APIs to handle task spawning and synchronization and includes parallel tasks and warp scheduling policies to reduce runtime overhead.

Latency-driven applications have both high throughput demands and response time constraints. These applications may launch many kernels that do not fully utilize the GPU unless grouped with large batch sizes. However, batching forces jobs to wait, which increases their latency. This wait time can be unacceptable when considering real-world arrival times of jobs. However, the round-robin GPU kernel scheduler is oblivious to application deadlines. This deadline-blind scheduling policy makes it harder to ensure that kernels meet their QoS deadlines. To enhance the responsiveness of the GPU, I also proposed LAX, including an execution time estimate for jobs with one or many kernels. Moreover, LAX adjusts priorities of kernels dynamically based on their slack time to increase the number of jobs that complete by their real-time deadlines. LAX improves the responsiveness and throughput of GPUs.

It is well-known that grouping threads into warps can create redundancy across scalar values in GPU vector registers. However, I also found that the layout of thread indices in multi-dimensional threadblocks (TBs) creates redundancy in the registers storing thread IDs. This redundancy propagates into dependent instructions that can be traced and identified statically. To remove GPU redundant instructions, I proposed DARSIE that uses a per-kernel compiler finalization check that uses TB dimensions to determine which instructions are redundant. Once identified, DARSIE hardware skips TB-redundant instructions before they are fetched. DARSIE uses a new multithreaded register renaming and instruction synchronization technique to share the values from redundant instructions among warps in each TB. Altogether, DARSIE decreases the number of executed instructions to improve GPU performance and energy.

1. INTRODUCTION

Programmers spend a significant amount of time optimizing their applications to run efficiently on domain-specific architectures. However, variants of hardware components in these domain-specific accelerators increase the burden on programmers to utilize these accelerators fully. Graphic Processing Units (GPUs) consist of many computing units to enhance the throughput of parallel execution and provide programmable scratchpad cache memory and vector registers to reduce the data access latency. To increase the scope of GPU applications, GPUs also enable multi-tasking by executing multiple kernels simultaneously on a single device. However, these concurrent applications with difference resource usage and quality-of-service (QoS) constraints exhibit new challenges on GPU computing.

GPUs are increasingly being considered for latency-sensitive applications in data centers. Examples include deep learning inference, network packet, and natural language processing. These data parallel applications exhibit a modest amount of data parallelism with high throughput demands and real-time deadline constraints. Furthermore, these applications may launch many kernels that do not fully utilize the GPU unless grouped with large batch sizes. However, batching forces jobs to wait, which increases their latency.

To reduce the latency on large batch, contemporary GPUs support the execution of multiple kernels on a single device simultaneously when there are sufficient hardware resources. In general, a GPU contains multiple hardware queues to support this type of execution. Thus, programmers can use GPU streams to execute these latency-driven kernels concurrently. However, the GPU concurrent kernel execution exposes several problems on latency-sensitive applications. First, GPUs faces severe underutilization when the space in these queues has been exhausted, and the performance benefit vanishes with the decreased parallelism. Second, the round-robin GPU

kernel scheduler is oblivious to application deadlines. This deadline-blind scheduling policy makes it harder to ensure that kernels meet QoS deadlines. This thesis will present hardware and software solutions for problems when executing GPU kernels concurrently.

Additionally, GPUs achieve high throughput by grouping multiple threads into warps to hide long latency operations with fine-grained *Single Instruction Multiple Thread* (SIMT) execution. However, the layout of thread indices in multi-dimensional threadblocks creates redundancy in the registers storing thread IDs. For instance, each matrix can be chunked into multiple tiling blocks that are loaded in the GPU shared memory with 2D thread in matrix multiplication program. We can observe many repetitive memory load in each row and column of two matrices in a tiling block and present the redundancy across different warps in a threadblock. I run through multiple applications and found over 30% SIMT redundant instructions derived from their operand values. This thesis also focuses on removing these SIMT redundant instructions, Finally, this thesis will also present a hybrid software and hardware solution to reduce the waste of hardware resources on SIMT redundant instructions.

This chapter begins with the discussion of GPU underutilization problems in the presence of latency-sensitive applications. We follow with a close look at the kernel scheduling issues when executing latency-driven applications composed of multiple kernels simultaneously and the redundancy shown in the SIMT executions. At last, I will present several open challenges when increasing GPU resource utilization and reducing the number of redundant SIMT instructions.

1.1 GPU Underutilization on Latency-sensitive Applications

This section addresses the GPU underutilization problem in the presence of latency-sensitive applications and challenges to overcome such an obstacle on contemporary GPUs.

1.1.1 Narrow Task

GPGPU computing has demonstrated an ability to accelerate a substantial class of compute-intensive applications [1, 2]. These applications have a high degree of parallelism, where iterations of large parallel loops are executed on the GPU. The programs see significant performance benefits because they can fully utilize the GPU's hardware resources by launching enough concurrent threads.

The GPU's performance benefits start to diminish as the degree of parallelism lessens. Conventionally, large parallel loops are offloaded to the GPU, while retaining the execution of smaller ones on the CPU. Applications should benefit from using the GPU, provided that the involved task (or CUDA kernel) count is sufficiently high. Each such task, called a *narrow task*, has limited parallelism (< 500 data parallel threads in practice).

Narrow tasks emerge in a number of scenarios. One set of such applications comprises latency-driven, real-time workloads. For example, online sensors that generate small inputs, resulting in tasks with low parallelism. Online sensors can generate many tasks in quick succession and require immediate processing. These workloads have been characterized as having mixed task and data parallelism [3, 4]. Secondly, *irregular* applications can exhibit narrow tasks. These applications often contain varying amounts of computation among different threads, and/or among loop iterations. To reduce load imbalance, these applications are often represented using many tasks with low degrees of parallelism [5]. Irregular workloads may also arise in multi-programmed environments. Different applications with low degrees of parallelism can be co-executed on a node to exploit all the computing resources.

1.1.2 Prior Work to Overcome GPU Underutilization

Prior work has identified the issue of GPU underutilization [6–9]. One approach to solve this problem is to statically fuse multiple smaller tasks [6, 7] to accumulate a large kernel. Advanced approaches [8, 10] use a concurrent kernel mechanism, moni-

toring and time-slicing their execution at runtime to obtain fair sharing. These static approaches require the programmer to fuse tasks *manually* and none of them have been shown to work beyond ten concurrent tasks.

These mechanisms also require static knowledge of the kernels to be fused, which is not always possible in multi-programmed or real-time environments. Additionally, individual tasks in a fused tasks receive the same on-chip resource allocation, e.g., shared memory and registers, thereby limiting occupancy based on the resource requirements of the largest task.

Dynamic (runtime) solutions can mitigate the above issues of static fusion. NVIDIA's current-generation GPUs employ HyperQ [11], which allows 32 kernels (tasks) to concurrently execute on the GPU. However, I show that narrow tasks can still cause underutilization, as 32 such tasks may not occupy the entire GPU. I argue that software mechanisms are needed to achieve flexible kernel concurrency. Prior work, *GPU enabled Many-Task Computing* (GeMTC) [9], presents a runtime task scheduling mechanism, where a task executes as a single *threadblock*. *Threadblocks* are sets of threads constituting the GPU kernel. Because GPU architectures limit the concurrent threadblock count, executing narrow tasks in GeMTC may result in poor utilization. In addition, GeMTC uses batch-based task execution, which results in delayed task launching and load imbalance since the completion time of a batch is determined by its longest running task.

1.1.3 Challenges in Designing a GPU Runtime System for Narrow Tasks

There are three key challenges that must be addressed when attempting to launch and run thousands of short-running tasks on a GPU.

First, CPU-GPU communication overhead must be minimized, while allowing the GPU to asynchronously schedule new tasks on each Streaming Multiprocessor (SM). Launching thousands of short-running tasks increases the importance of minimizing the time it takes for each task to begin execution on the GPU. Since the CPU and

GPU must coordinate task spawning and scheduling over the PCIe bus, which currently has no support for atomic operations, the handshaking required is expensive or impossible if a traditional data structure, such as a queue [12], is used. Previous work that required OS-like co-ordination over PCIe [13, 14] solved consistency issues using a producer-consumer model but did not have to optimize the system for many, short running tasks. The second challenge is to keep the overheads involved in task spawning and scheduling low. Minimizing both the copying of task parameters and the search for free GPU resources is important when task execution times are short. The third issue is supporting native CUDA functionality such as shared memory usage and efficient threadblock synchronization.

1.1.4 Pagoda GPU Runtime System for Narrow Tasks

I proposed Pagoda – a GPU runtime system and is designed to increase the number of concurrent tasks (kernels) on a single GPU. The programmer replaces certain CUDA API calls with equivalent Pagoda calls in the host and device codes, retaining the functionality of the CUDA programming model. Unlike static solutions, the programmer does not have to tediously fuse the available tasks. Pagoda achieves high utilization by continually running a *MasterKernel*, which controls the execution of all GPU *warps* in software. In Pagoda, tasks are spawned by the CPU as soon as they become available, without batching. On the GPU side, the MasterKernel virtualizes the GPU's resource allocation and threadblock scheduling mechanism to allow individual warps to make progress as soon as resources are available.

1.2 Latency-sensitive GPU Applications with Deadline Constraints

This section addresses problems when executing latency-sensitive applications with real-time deadline constraints simultaneously on a single GPU and previous work that improve QoS of GPU latency-sensitive applications.

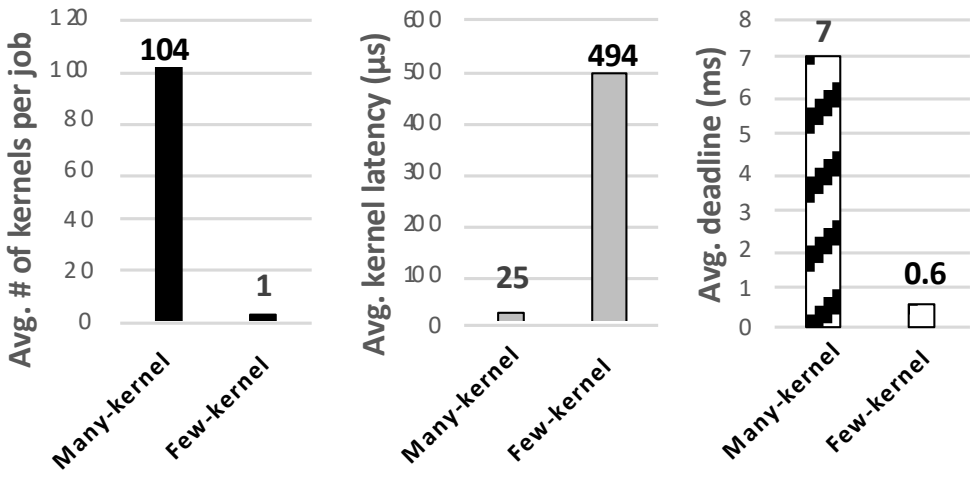

Fig. 1.1.: Characteristics of many-kernel latency-sensitive jobs versus few-kernel latency-sensitive jobs, listed in Table 6.3

1.2.1 Constraints of GPUs on Latency-sensitive Applications

GPUs are the programmable accelerator of choice for massively data-parallel applications that do not have strict latency requirements. However, there is a growing class of latency-sensitive, data-parallel workloads that can benefit from the GPU's throughput. Examples include machine learning (ML) inference for RNNs [15–20], network packet processing [21–23] , and natural language processing (NLP) in Intelligent Personal Assistants (IPAs) [24]. These latency-sensitive applications have become a staple of contemporary datacenters and have response time constraints. For instance, Google TPU [25] paper raises RNN inference applications have 7 milliseconds response time constraint. These latency-sensitive applications have become a staple of contemporary data centers, which increasing include GPUs and other high-throughput accelerators. Given the availability of GPUs in the data center, and the data-parallel nature of the applications, there is significant potential to offload work from overburdened CPUs to an accelerator. However, contemporary GPUs are

deadline-blind and have no mechanism to predict which work can be offloaded and completed in time.

Many deadline-driven applications exhibit a middling amount of data-parallelism [26]. Enough to justify GPU acceleration, but not enough to fully utilize the GPU's resources [24, 27]. As a result, executing one job on the GPU at a time causes severe underutilization. To alleviate this issue, programmers batch similar jobs together [19], greatly improving throughput and utilization at the expense of additional latency. This increase in latency is usually unacceptable for tasks with tight deadlines [25], especially when realistic job arrival rates are considered. GPU programs can avoid batching, while still executing multiple jobs at once with streams. Streams allow kernels from independent jobs to be scheduled concurrently on multiple command queues located between the CPU and GPU [11, 28]. However, software cannot efficiently manage the relative priority of these queues at short time scales, which makes it difficult to efficiently re-prioritize jobs with different deadlines as contention in the GPU changes.

State-of-the-art GPU solutions for managing latency-sensitive tasks are restricted to varying priorities at a coarse granularity on the host CPU [29–31], and thus do not fully utilize the GPU's integrated queue scheduling logic. Consequently, the precision of information available to these CPU-side mechanisms is limited. Dynamic, microsecond-scale information about GPU-side contention, which some latency-sensitive applications require, is difficult to track from host-side software. As a result, these software-only techniques are less effective when scheduling many latency-sensitive jobs and primarily focus on mixing latency-insensitive and latency-sensitive work. In contrast, we target a common situation in many data centers where many homogeneous, latency-sensitive jobs are executing in parallel [32].

Figure 1.1 demonstrates how quickly scheduling decisions must be made when executing concurrent latency-sensitive jobs. To better understand their demands, we subdivide our latency-sensitive applications into two categories: many-kernel and few-kernel. The many-kernel applications we study, which come from ML inference,

are composed of a number of relatively small, short kernels and typically have deadlines on the order of milliseconds. The few-kernel applications, which come from network packet processing and IPAs, execute a single, much longer kernel, but have more aggressive deadlines (usually < 1 ms). To efficiently manage both many-kernel and few-kernel applications, per-kernel scheduling decisions must be made at the microsecond timescale.

We argue that dynamic, integrated stream scheduling is necessary to meet the low-latency scheduling demands of these workloads. An analogy can be made to the memory hierarchy in modern CPUs. At the lower-levels of the CPU memory hierarchy, the operating system is responsible for managing the replacement of relatively large pages in physical memory from the relatively high-latency disk. However, smaller cache blocks, which require nanosecond-scale response times, are managed by hardware. In throughput-oriented GPUs, scheduling relatively few, millisecond- or second-scale kernels in soft-ware is acceptable. However, managing many short-running kernels competing for GPU resources to meet sub-millisecond or millisecond-scale deadlines requires hardware support.

1.2.2 Prior Work to Improve QoS of Latency-sensitive GPU Applications

Contemporary GPUs [11, 28, 33] contain multiple queues to manage independent work submitted asynchronously with streams. This parallel work can be executed concurrently on a GPU when resources are available. In this section, we describe the work to schedule multiple kernels on a GPU.

Preemptive GPUs

Prior work on GPU kernel preemption or re-execution [34–37] are alternative mechanisms that can be used in combination with better stream scheduling. However, for latency-sensitive workloads, the overheads associated with preempting GPU kernel contexts, whose aggregate registers and scratchpad size can be 100s of KBs

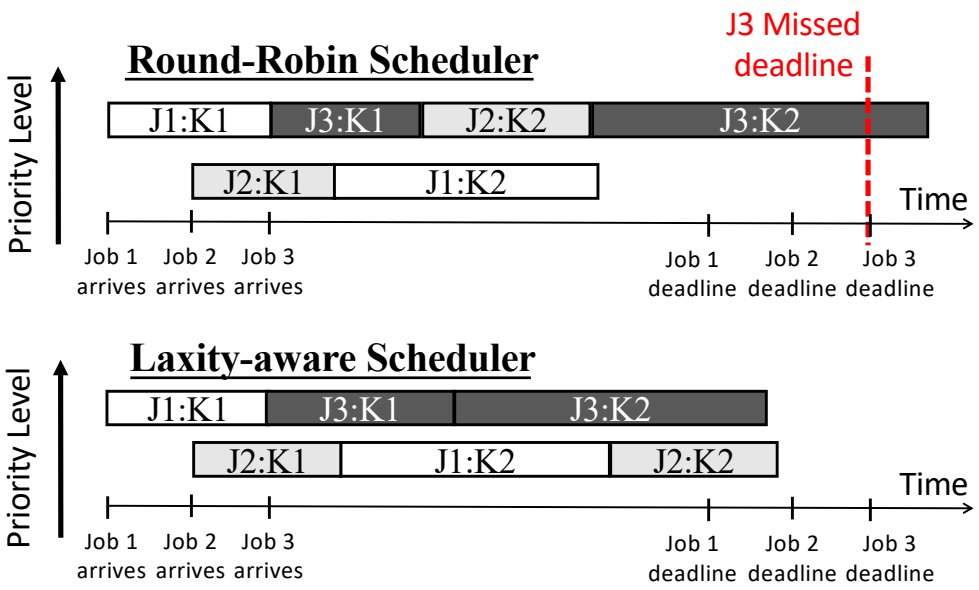

Fig. 1.2.: Comparison of Round Robin and Laxity-aware Schedulers for a GPU that can simultaneously execute 2 jobs

Table 3.1, may be prohibitive. Additionally, the benefits of preemption are muted for short running kernels that finish long before the cost of preemption and rescheduling can be amortized. Specifically, Table 3.1 indicates that the vast majority of kernels in our evaluated latency-sensitive workloads complete within 10 us. Recently proposed preemption-based techniques, such as PREMA, are effective at intelligently preempting and scheduling relatively coarse-grained tasks [38].

GPU Stream Scheduling

The Command Processor (CP) is an integrated microprocessor within a GPU, which parses the kernel contexts and schedules streams. In Figure1.3, each stream is mapped to a queue and each queue can hold multiple kernels from a single stream. Inter-kernel dependencies between kernels in the same stream are maintained, but GPUs can execute kernels from different streams simultaneously. Each queue entry

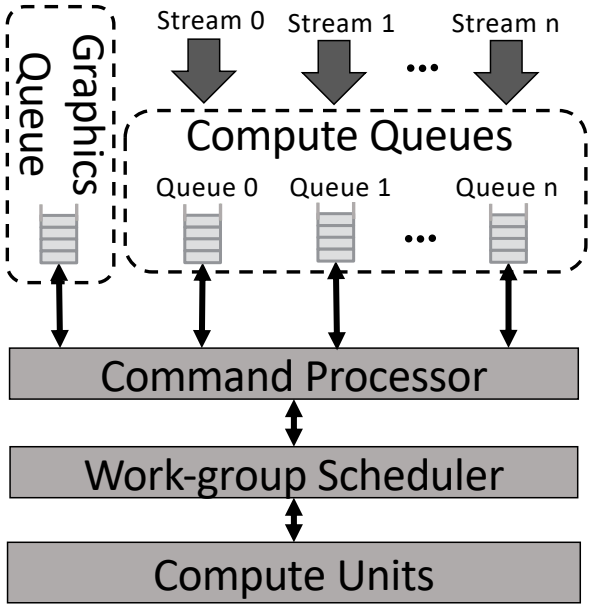

Fig. 1.3.: GPU Queue Scheduler Architecture

describes a separate kernel and includes details such as the kernel's thread dimensions, register usage, and local data store (LDS) size. We assume the CP can parse four different streams in parallel every 2 microseconds in this work. Afterward, a work-group (WG) scheduler reads these fields to dispatch work groups to compute units (CUs). Generally, GPU work group schedulers issue all work groups from one kernel before switching to issuing work groups from a different kernel. Despite this restriction, work groups from kernels in different queues often interleave execution.

Normally, the CP schedules kernels within these queues in a Round Robin (RR) manner [48]. This deadline-blind scheduling policy makes it harder to complete jobs by their real-time deadlines. The top half of Figure 1.2 illustrates the problem with RR. In this example, the GPU is running three jobs with varying arrival times such that the deadline of each job varies. Each job contains two kernels with different execution times. For simplicity, assume that at most two kernels can be concurrently executed. RR will schedule kernel 1 from job 1 (J1:K1) and kernel 1 from job 2

(J2:K1) first because they arrive before job 3. When job 3 arrives, its first kernel is scheduled after J1:K1, and then J3 is not scheduled again until both J1:K2 and J2:K2 have executed. Since J3 is the longest job, if it had been prioritized over J1 and J2, all the jobs could have made their deadlines. However, since RR is unaware of this, J3 misses its deadline.

Priority-based GPU Programming

At the application level, programmers can specify the priority value of streams. However, the limited number of priority levels (e.g., high and low) in GPU programming languages [39] is insufficient. First, the priority level submitted by programmers does not give any information about when the kernel must be completed, only the kernel's relative importance. Second, priorities assigned to individual streams do not provide the GPU a global view of when to complete a chain of dependent kernels. As a result, programmers conservatively set a job's priority to ensure that its deadline is met. Finally, jobs can have different amounts of work despite potentially having the same static priority level.

Laxity-based Scheduling on GPUs

Laxity-based scheduling [40] leverages the laxity which tells us how close to the deadline of a job is predicted to finish. The dynamic laxity value is a job priority in the laxity-based scheduling. Jobs with less laxity have higher priority. Laxity-based scheduling was used in the real-time applications to ensure jobs to be completed by their deadlines. Also, laxity-based scheduling relies on accurate job execution time prediction to adjust jobs' priorities dynamically. However, GPU occupancy and job arrival rates cause the completion time of each job and their associated kernel(s) to vary. These dynamically varying parameters make static predictions of job completion time extremely difficult.

I propose to dynamically adjust the priorities of each job (and its associated queue) based on the estimated execution time of each job's kernels. By adjusting the priorities, kernel launches are re-ordered to increase the number of jobs completed by their deadlines. The bottom half of Figure 1.2 demonstrates that with reasonably accurate execution time estimates, a deadline-aware scheduler can optimize the scheduling of deadline-sensitive jobs (similar to prior work for CPUs [74][75]). The bottom example begins like the top example, with the GPU scheduling job 1 and job 2 first, because they arrive earlier than job 3. However, the LAX scheduler is aware of the deadlines and durations of all 3 jobs, so it prioritizes J3 when it arrives since it will miss its deadline if not immediately scheduled (i.e., it has a zero laxity). As a result, all three jobs completed by the deadlines.

1.2.3 LAX: Laxity-aware GPU Job Scheduler

An effective deadline-aware scheduler must: (1) be aware of each job's deadline, (2) estimate the remaining execution time for each job, and (3) frequently adjust job priority as time progresses and the level of contention in the GPU changes. We propose an integrated laxity-aware stream scheduler (LAX) that achieves all three of these requirements.

LAX leverages the idea that stream-based GPU applications enqueue all their kernels in quick succession. In the many-kernel jobs, although each kernel launch is dependent on the data output by the previous kernel, all the kernels associated with a particular job are known before the GPU begins execution. As a result, LAX uses the GPU's queue scheduler (or command processor (CP)), to perform a novel stream inspection technique that reads the contents of parallel hardware streams and generates an estimate of how much work exists in each job. LAX's scheduling algorithm then combines this information with the job's deadline and fine-grain information about the current per-kernel work completion-rate to generate an accurate estimate of how much laxity the job has. A job's laxity is an estimate of

how much earlier than its deadline it will finish, given current conditions [40]. Based on each job's estimated laxity, our scheduler re-prioritizes jobs to complete as many as possible by their respective deadlines. With the rich, fine-grained information available to GPU stream schedulers, LAX also prevents job oversubscription by using a Little's Law-based queuing delay estimate [41, 42] to reject work that is predicted to miss its deadline. My proposed stream inspection, per-kernel work completion monitoring, and job rejection mechanisms make real-time scheduling possible and practical in GPUs

1.3 Redundancy on GPU SIMT Instructions

This section illustrates the redundancy embedded in GPU SIMT instructions, previous work, and challenges to improve the performance and energy of GPUs through the elimination of SIMT redundant instructions.

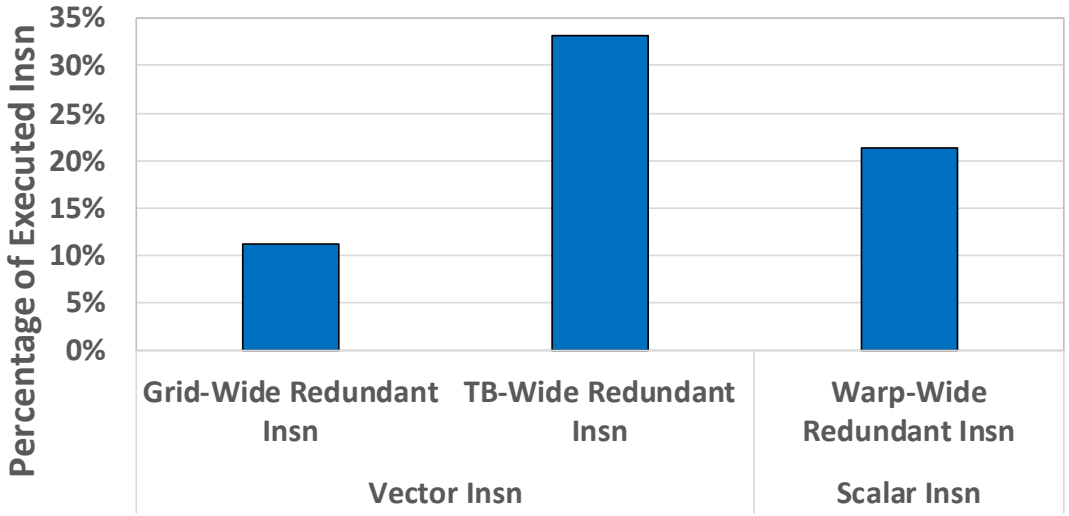

Fig. 1.4.: Redundant instructions in each GPU thread grouping level across different applications

1.3.1 Threadblock-wide Redundancy on GPU SIMT Instructions

Graphics Processing Units (GPUs) run thousands of concurrent scalar threads based on programmer-defined parallelism. Programmers define a three dimensional grid of threadblocks (TBs) for each kernel. TBs are three-dimensional arrangements of scalar threads, grouped into warps behind the scenes for Single Instruction Multiple Thread (SIMT) execution. Although the programming model for GPUs is SIMT, the underlying datapath is Single Instruction Multiple Data (SIMD). Each warp has a set of private vector registers from a vector register file storing per-thread scalar values in each vector lane. For example, each lane in every warp of the TB contains the same scalar value such as the shared constants and TB dimensions [43–45]. Furthermore, affine redundancy occurs when vector registers in different warps have the same value, which can be represented as (base, strand) pair. This redundancy occurs naturally in multi-dimensional TBs where consecutive lanes have consecutive threadId values, replicated in the private register space of each warp in the TB. Unstructured redundancy occurs when registers have the same vector values in each warp of the TB, but those values have no discernible pattern. However, little work exists on statically identifying and removing all three types of redundancy that can exist on GPU programming model.

To help understand this redundancy, Figure 1.4 shows the results of a limit-study measuring the fraction of redundantly executed instructions at the grid, TB, and warp level.Instructions are classified as redundant at the grid-level when all the grid's warp instructions operate on the same vector operands, implying it need only be executed once for the entire grid. Similarly, Figure 1.4 plots redundant instructions for TBs if all warp instructions within a TB use the same vector operands. Warp-wide redundancy occurs if all scalar threads in a warp operate on the same scalar value. We find that the greatest opportunity for redundancy elimination exists at the TB level, where on average 33% of instructions need only be executed once per TB.

GPU languages like CUDA and OpenCL express parallelism defined along multiple (x, y and z) axes, which helps programmers naturally map multi-dimensional data to multi-dimensional thread grids. However, this dimensionality can have a significant impact on redundancy. While opportunities exist for redundancy elimination at the warp granularity (through scalar instructions), eliminating vector instructions at the TB and grid levels improves both performance and energy efficiency without the addition of functional units or register files. Unlike warp-wide redundant operations that are local to one vector instruction, TB- and grid-wide redundant instructions occur across different vector instruction, each of which occupies space in the instruction pipeline. Elimination of these vector instructions frees space in the pipeline, and reduces pressure on the memory system if the instruction is a memory operation. In practice, grid-wide redundancy is both difficult to eliminate and less common than TB-wide redundancy. I therefore focus this thesis on the elimination of TB-wide redundant instructions to improve both performance and energy efficiency.

1.3.2 Previous Work on Removing GPU SIMT Redundant Instructions

Contemporary GPUs from NVIDIA and AMD use a scalar functional unit and register file to perform operations on warp-wide redundant data identified by the compiler [46, 47]. Research has also sought to address warp-level redundancy by masking off lanes in the vector pipeline and skipping partial warp instructions when the SIMD width is less than the warp size [43–45]. Other work has proposed adding expensive value comparison hardware to the pipeline to remove redundancy at the issue stage, or reduce register file space via compression [48, 49]. I use insights gained from creating a redundancy taxonomy for TBs to identify both new opportunities for instruction skipping that are unexplored in these works, and ways to offload redundancy identification to the compiler. This allows my proposed work to both avoid expensive value comparisons in hardware, and improve performance by skipping entire vector instructions before they are fetched in the frontend of the pipeline.

1.3.3 Challenges on Eliminating GPU SIMT Redundant Instructions

Value sharing and instruction elimination is difficult to solve solely in the compiler or hardware. Alone, the compiler is not able to efficiently coordinate value sharing and instruction skipping between parallel warps. Likewise, it is difficult for a solely hardware implementation to detect redundancy in warp-wide vector registers, and a reactive mechanism to detect redundancy in register file accesses and forward PC values marked as redundant to the frontend of the pipeline is complex. My proposed work therefore uses a combined compiler and hardware approach to avoid these problems.

1.3.4 DARSIE GPU SIMT Redundant Instruction Skipper

In this thesis, I examine the root-causes of TB-wide redundancy, and find that many conditionally redundant instructions at the TB level that can be identified during static compilation. We observe that the layout of thread indices in multi-dimensional TBs creates redundancy in the registers storing thread IDs. This redundancy propagates into dependent instructions that can also be traced and identified statically. A per-kernel runtime check of a TB's dimensions can be used to determine if conditionally redundant instructions are actually redundant. This avoids performing expensive vector register comparisons at runtime. Based on these observations, I propose *Dimensionality-Aware Redundant SIMT Instruction Elimination* (DARSIE), a TB-centric instruction skipping mechanism that statically identifies TB-redundant instructions using a combination of compiler markings and runtime TB-sizing information. Once identified, TB-redundant instructions are skipped by the hardware before they are fetched. DARSIE uses a novel multithreaded register renaming and instruction synchronization technique to share the values from redundant instructions among warps in each TB. This differs from CPU register renaming that is used to remove false dependencies in single-threaded pipelines.

Contemporary GPUs are designed to perform well when executing regular kernels with limited control-flow divergence. As a result, these regular applications are the most commonly run in the field today [50]. Although these applications can be computationally-dense, we demonstrate that they also operate a significant number of redundant operations. In contrast to the body of orthogonal work on improving GPUs in the presence of irregular, cache- and scheduling-sensitive workloads [51–55], DARSIE is designed to target common contemporary workloads which prior work has demonstrated are insensitive to locality-optimizing techniques that focus on scheduling [51].

To demonstrate how common multi-dimensional redundancy is, we conducted a survey of 133 applications [56–67] running on a commodity NVIDIA Volta GPU. Over 33% of the applications surveyed demonstrated the multi-dimensional TB characteristics that create implicit redundancy. Interestingly, we find that this characteristic is more pervasive in applications that make use of optimized libraries (CUDNN, CUBLAS, etc.), where 60% met DARSIE's conditional redundancy requirements. Furthermore, in the apps that had at least one kernel that met the sizing requirements, an average of 71% of the application's execution time is spent in those kernels.

1.4 Contributions of This Thesis

In this thesis, I demonstrate the hardware and software designs to improve the GPU utilization and response time on latency-sensitive applications. The main contributions are as follows:

- **A GPU runtime system virtualizing the hardware resources**. To increase GPU throughput in the presence of narrow tasks, I present a software mechanism – Pagoda to schedule multiple tasks on the GPU in parallel, and describes a pipelining scheme to overlap several task processing stages. Pagoda includes new APIs to handle the operation of task processing and software solutions for dynamic shared memory management and sub-threadblock synchro-

nization. Furthermore, Pagoda introduces a continuous task spawning mechanism to reduce CPU-GPU synchronizations to obtain a high task spawn rate.

- **A deadline-aware GPU kernel scheduler for real-time applications with deadline constraints**. Contemporary GPUs supports the concurrent kernel execution, but its round-robin kernel scheduling policy does not satisfy QoS requirements in latency-driven applications. Thus, I propose a laxity-aware algorithm (LAX), which is used in combination with a dynamic, per-kernel work completion rate to generate an accurate estimate of work and time remaining. LAX dynamically varies job priorities to improve throughput while attempting to meet real-time latency requirements.

- **The elimination of GPU redundant SIMT instructions** . I introduce a new taxonomy of redundancy for GPUs, focusing on the TB granularity. I show that the composition of TB-wide redundancy is highly dependent on the dimensions of the TB, and that thread index layouts in multi-dimensional TBs create ample implicit redundancy. Therefore, I propose DARSIE, which combines our redundancy identification software with novel instruction skipping hardware to ensure that redundant instructions are fetched an executed only once per-TB. DARSIE leverages multithreaded register renaming and selective warp synchronization to share vector registers between warps in a TB, allowing TB-redundant instructions to be skipped in the fetch stage of the pipeline.

1.5 Thesis Overview

The following sections present the work that improves 1) the GPU resource utilization in the presence of narrow tasks. 2) the number of concurrent latency-sensitive jobs completed by their real-time deadlines. 3) the waste of GPU hardware resource caused by redundant SIMT instructions. Chapter 2 illustrates background and related work. Chapter 5 presents Pagoda GPU runtime system. Chapter 4 illustrates a taxonomy of GPU redundancy. Chapter 6 describes the work of deadline-aware LAX job

scheduler. Chapter 7 demonstrates DARSIE that eliminates redundant GPU SIMT redundant instructions. The conclusion and future work are placed on Chapter 8.

2. BACKGROUND AND RELATED WORK

In this chapter, I describe the GPU architecture, constructions of GPU programming languages, and GPU microarchitectural model. Then, I examine the prior solutions to helping the GPU utilization and QoS requirements on domain-specific architectures. At last, I also discuss the taxonomy of GPU redundancy and the previous work in removing GPU redundant SIMT instructions.

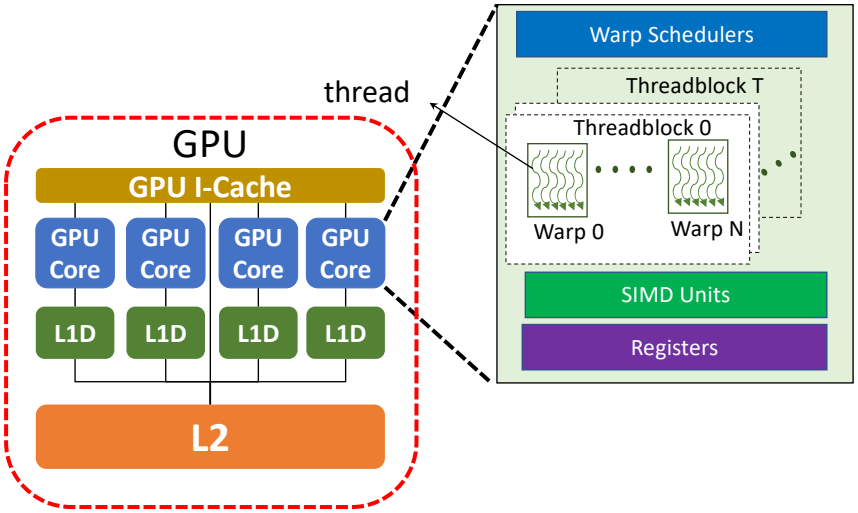

Fig. 2.1.: GPU Architecture: The number of GPU cores is dependant with different GPU generations and the scratchpad cache is shared with L1D cache.

2.1 GPU Architecture and Programming

The GPU hardware typically consists of multiple GPU cores within Streaming Multiprocessors (SMs) on NVIDIA GPUs [67] and Compute Units (CUs) on AMD GPUs [68]. Each GPU core has its private L1 data cache and one on-chip programmer

managed cache, known as shared memory and a number of 32-bit registers. L2 cache and I-cache are shared across different GPU cores. GPU scalar threads typically share register files and the shared memory space in each SM and CU. Within a SM and CU, the SIMD unit contains hundreds of SIMD lanes for parallel thread arithmetic-logic unit (ALU) executions. In NVIDIA GPUs, the *War*p is the basic Single Instruction, Multiple Thread (SIMT) work unit, which comprises 32 threads that march in lockstep, executing the same instruction. In AMD GPUs, a *wavefront* is like the warp and composed of 64 threads. Each SM has multiple warp slots and warp schedulers. The warp scheduler can concurrently schedule up to 64 warps and issues ready warps to SIMD unit for the execution.

In the CUDA programming model, the programmer organizes parallel work in *kernels*. Threads of a kernel are grouped into *threadblocks*. Multiple threadblocks can reside on each SM, the maximum number being 32. The threadblock size is limited to 1024 threads, or 32 warps. Each SM can hold up to 2048 concurrent threads [67]. Both the shared memory and registers of an SM are partitioned among the executing threadblocks. There is no CUDA primitive for global, kernel-wide synchronization; however, threads in a threadblock can use the *__syncthreads()* function as a barrier.

A way of measuring the GPU utilization is *occupancy*. Occupancy is the ratio of the total number of resident GPU warps divided by the maximum number of warps that can co-exist in the GPU (i.e. 64 $\times$ the number of SMs in the GPU). The kernel occupancy is affected by three factors, namely, i) size of threadblocks, ii) kernel's register count, and iii) size of the requested shared memory. Balancing these three factors requires programmer expertise, making high occupancy often difficult to achieve. For exampke, NVIDIA Volta V100 GPU [69] has 80 SMs. Consider a scenario of narrow tasks, where one task has 256 threads, or 8 warps. If only one task is executed at a time, the occupancy would be $(8/(64\times80))\times100\% = 0.156\%$. With HyperQ [11], 32 kernels may co-execute, meaning that 32 narrow tasks can run simultaneously. The achieved occupancy then would still be low, i.e. $(8\times32/(64\times80))\times100\% = 5\%$.

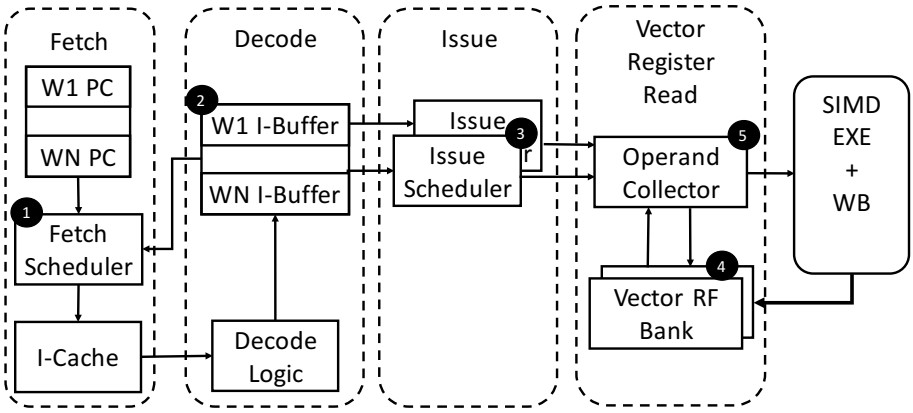

Fig. 2.2.: GPU micro-architectural model

2.2 GPU Micro-architectural Model

Figure 2.2 presents the GPU microarchitectural model. Each cycle, a fetch scheduler (❶ in Figure 2.2) in the frontend of the pipeline initiates a fetch for one of the warps assigned to the core. This scheduler uses loose-round-robin (LRR) prioritization, and initiates instruction cache (I-cache) fetches based on which warps have empty instruction buffer (I-Buffer ❷) entries. Each warp has a two-entry I-buffer that is used to decouple the SIMT frontend (which fetches one PC for an entire 32-thread warp) from the SIMD backend. Each cycle, multiple issue schedulers select at most two instructions from one warp each for execution. Warps are statically partitioned among these issue schedulers.

Once selected to issue, instructions must read their source operands (32-element vector registers with 32-bit elements) from a highly banked register file (❹). To avoid excessive bank conflicts and facilitate high bandwidth, an operand collector [70] schedules register file reads in a way that limits stalls. These operand collectors are the inputs to the execution stage of the pipeline. The mapping between <named vector register, warp ID> pairs and physical vector register contents is programmable, and based on a mapping table initialized when a Threadblock (TB) is launched on an SM.

This is necessary since each warp can be assigned a different number of registers at compile time.

There is little documentation on how this register mapping is achieved, so we make the assumption that blocks of registers are assigned to warps using a simple base register + length mapping table. This avoids storing a unique <named vector register, warp> pair for each physical vector register. We also assume this mapping is done in the operand collection phase. Contemporary GPUs (like the Pascal card we model) do not have scoreboard logic embedded in the core, but rather encode dependencies in their instruction stream. Variable-cycle memory instructions are controlled via *depbar* instructions that ensure source registers read by instructions have received responses to their memory requests before a dependent instruction is issued.

2.3 Processing Narrow Tasks on Domain-specific Accelerators

Task-based models [71,72] employ a runtime system which governs task executions on various engines, such as CPUs and GPUs. These systems, however, always execute narrow tasks on CPUs, believing that their low parallelism degree cannot overcome the overhead of memory copies. This section indicates the work that aims to increase the GPU utilization on workloads composed of narrow tasks.

2.3.1 Static Software Approaches

Static task fusion is the preliminary approach to deal with GPU underutilization. Wang et. al. [6] present a mechanism where such fusion achieves higher utilization, resulting in energy benefits. KernelMerge [7] statically fuses kernels, and explores round-robin and fair-partitioned execution schemes for these kernels. The GPU programming models, such as CUDA and OpenCL, allocate same resources to each thread. Therefore, the resource usage in static fusion schemes gets limited by the requirements of the most resource-hungry task. A more sophisticated approach

is therefore to perform fusion at the runtime. Two approaches [8, 10] perform kernel consolidation leveraging concurrent GPU kernel execution. They launch multiple concurrent kernels, where resources not being used by one kernel can be yielded to another. The first approach [10] relies on a threadblock-level launching scheme. The second approach [8] presents a compiler scheme that transforms kernels so that they can automatically support any threadblock configuration. This ability helps in finding the best sharing configuration for different kernels. Zhong et. al. [73] present an approach where a large kernel is split into independent smaller kernels that co-execute to achieve better utilization. Kato et al. [74] propose a software scheduler at the device driver layer to prevent interference among concurrently running GPU applications, trading off response latency for throughput. However, all these approaches are restricted by the 32 kernel limit imposed by CUDA-HyperQ, and fail to efficiently execute narrow tasks.

2.3.2 Dynamic Runtime Solutions

Runtime systems that virtualize GPU resources can naturally overcome the hardware-imposed kernel limit. Additionally, they offer low execution latencies compared to static fusion. Closest to our Pagoda work is GeMTC [9]. Like the MasterKernel in Pagoda, GeMTC runs a SuperKernel that virtualizes GPU resources. The use of large dameon-like kernels is similar to persistent threading [75]. Unlike the MasterKernel, the SuperKernel does not guarantee an occupancy of 100%, and therefore may face underutilization. Secondly, the GeMTC design uses a single FIFO queue for its batch-based task launching scheme, resulting in significant task scheduling overhead. Third, GPU-specific functionalities, such as the shared memory and threadblock-level synchronization remain unsupported.

GPU researchers have exploited pipelining [76] to overlap data transfers with kernel computations. The distinguishing factor in the Pagoda pipelined task processing is that it overlaps spawning, which comprises the CPU finding a free task entry and

performing a data copy, with GPU scheduling, which is only a sub-part of the overall task processing. Yang et. al. [77] showed that fusing cross-kernel threadblocks can obtain better shared memory performance. Pagoda's shared memory management schedules threadblocks as long as shared memory is found at runtime.

2.3.3 Preemptive Hardware Scheduler and Virtualization

Prior research has explored preemptive hardware techniques to improve GPU utilization in the presence of concurrent low occupancy kernels [78–80]. In contrast to these works, which require hardware changes, Pagoda provides a software only solution that runs on contemporary GPU hardware and could be applied to any future GPU hardware that supports the CUDA programming model.

Virtualizing GPU resources has also been explored to improve GPU utilization via multi-tenancy in cloud computing. Sengupta et al. [81] focus on virtualizing the GPU as a whole in a cloud with multiple GPUs. Becchi et al. [82] study a virtual memory system that isolates the memory spaces of concurrent kernels and allows kernels whose aggregate memory footprint exceeds the GPU's memory capacity to execute concurrently. By contrast, Pagoda virtualizes the compute resources of a single GPU at the granularity of a warp.

2.4 Improving Application Latency on Accelerators

Recent work has optimized GPUs and accelerators for latency-sensitive applications, especially machine learning algorithms. At the architecture level, these optimizations include distributing and pipelining RNNs across FPGAs [83], compressing weights [84], increasing batch size and adding special purpose functional units [25] and redesigning algorithms to move shared weights on-chip [16–18, 37]. At the software and system levels, optimizations include preemptively scheduling kernels [22], increasing data reuse [85], dynamically combining same-sized RNN cells at runtime [19], and persistence [26].

2.4.1 QoS-Aware Scheduling Policies

Recent work has also applied QoS concepts to GPUs. The most relevant related work is Baymax [30] and Prophet [29]. Baymax uses pre-trained regression models to predict the execution time of jobs, then uses its predictions to adjust job priorities to prevent latency-sensitive jobs from missing their QoS targets. Similarly, Prophet [29] leverages offline profiling and prediction models to co-locate kernels and improve GPU utilization and QoS. Wang, et al. [86] measure the GPU's IPC to properly provision GPU resources and enable kernels to be completed by their QoS targets. This prior work provides some of LAX's features, but relies on software-only, CPU-side schedulers, whereas LAX extends the GPU's command processor to better respond to dynamic changes in behavior and avoid host-device overheads. Other orthogonal work adds QoS support at the memory controller [87, 88].

2.4.2 Real-Time Scheduling

Embedded and real-time systems have also utilized laxity, and prior solutions that use laxity have been deployed on CPUs [89–94] and GPUs [74, 95–98] for real-time applications. For instance, TimeGraph [74] uses the driver to assign kernel priorities based on the GPU resource usage. Other work preempts lower priority kernels in order to execute higher priority kernels [34, 37]. However, preemption schemes are usually guided by the operating system and have high overhead on GPUs due to their amount of context state [34,35,37]. Furthermore, the communication latency between the OS and the GPU makes fine-grained updates difficult. In comparison, LAX dynamically adjusts the job's priorities. Prior CPU-side work such as backfilling also exploits similar ideas [94]. Subsequent work extended backfilling to predict job runtime based on the runtime of different jobs [93]. More generally, although these CPU-side ideas utilize similar underlying concepts, they suffer from the same inefficiencies as other CPU-centric solutions.

2.5 Solutions on the GPU Redundancy Removal

This section addresses the hardware and software methods to eliminate GPU SIMT redundant instructions.

2.5.1 Hardware Redundant Instruction Skipper

Instructions operating on identical data has long been observed in CPUs [99–104]. Recent GPU work [43–45, 105–109] has targeted the removal of GPU instructions. Recent work by Wang and Lin [109] proposes Decoupled Affine Computation (DAC) that uses the compiler to identify and isolate an affine instruction stream that is run on a separate pipeline from the SIMT instruction stream. DAC captures the run-ahead effect of Decoupled Access Execution [110] as well as achieves a reduction in SIMT instructions by computing affine *base* + *stride* values only as needed in the affine stream. In contrast, DARSIE exploits redundant instructions, which are fundamentally different than affine instructions. The unstructured redundancy eliminated by DARSIE cannot be eliminated with affine function units. Xiang et al. [45] identified inter-warp uniform values in the decode stage, and skips selective, uniformly redundant instructions using an instruction reuse buffer [100]. Unlike Xian et al.'s design, DARSIE skips redundant instructions before they are fetched, based on the pre-emptive detection of TB-level redundancy. Kim et al. [44] presents a fine-grain(FG-SIMT) execution engine to tackle instructions composed of *affine* and *uniform* value structures. This FG-SIMT architecture aims to improve performance and energy efficiency for irregular kernels, focusing primarily on scalar instructions. Lee et al. [49] proposes compressing GPU vector registers to save energy. Esfeden et al. [111] proposes a register packing mechanism using renaming to that helps save energy and increase performance by combining reads to multiple registers into a single access.

Recent approximate computing research [112, 113] concentrates on removing the execution of similar value structures to reduce energy consumption. Daniel et al. [113]

observed operand value similarity within a warp. Their approximating warp micro-architecture [113] can both detect value similarity, and reduce the execution of identical data across SIMT lanes. G-scalar [114] found that 45% of divergent instructions are eligible for scalarization. G-scalar [114] compares values of the registers and compresses them to reduce the usage of the register file. Concurrent work on Warp Instruction Resuse (WIR) [48] saves energy by reusing registers with identical operand values across warps through a signature-based renaming mechanism. Unlike DARSIE, WIR relies on a complex, hardware-based redundancy detection mechanism, and is still bottlenecked on fetch/issue bandwidth.

2.5.2 Compiler-Assisted Approaches

GPU compiler work on scalarization [106,115–117] discovers invariant instructions, and re-allocates registers to improve performance. Lee et al. [115] exploited convergence and variance analysis to recognize scalar instructions of data parallel programs statically. This compiler analysis transformed invariant values within a loop to a scalar instruction. a loop accessed by each threads become a scalar instruction. The scalarization compiler can pick out these replicated instructions and provide hints for hardware to skip scalar operations.

3. LATENCY-SENSITIVE GPU APPLICATIONS

We study a wide group of latency-sensitive GPU applications that represent different use cases and access patterns to understand how they perform on contemporary GPUs.

3.1 Applications

This section addresses latency-sensitive GPU applications in machine learning, network packet processing and speech recognition domains.

3.1.1 Recurrent Neural Networks

RNNs are well suited for domains such as language translation [118, 119] and speech recognition [10][11] where prior events persist and influence subsequent ones. RNNs contain loops that allow this information to persist across multiple iterations (or time steps). The number of times the loop is un-rolled represents the RNN's sequence length, which varies across jobs and determines the length of the recurrent step. As a result, RNNs behave very differently than Convolutional Neural Networks (CNNs) [120–123]. The hidden state (the memory) is calculated by looking at the previous hidden state and the input at the current step. RNN models such as Long-Short-Term-Memory (LSTM) [124] and Gated Recurrent Unit (GRU) [125] add memory cells to improve accuracy.

Each RNN time step contains multiple kernels with varying degrees of parallelism and execution time. As shown in Table 3.1, a single-batched LSTM with a sequence length is 13 consists of 6 unique kernels and each kernel is called multiple times (we only show LSTM due to space constraints, Vanilla and GRU are similar). In contrast

to the training phase where latency is less critical [126–128], RNN inference jobs have real-time constraints [19, 25, 27, 85, 129]. It is challenging to fully utilize the GPU while minimizing the end-to-end latency of RNN inference applications.

3.1.2 Network Packet Processing

Network packet processing increasingly utilizes GPUs to take advantage of their massive parallelism. For example, IPV6 performs a Longest Prefix Matching computation used in IPV6 network packet table lookups and has a stringent 40 microseconds deadline [21,130]. Similarly, Cuckoo must complete cuckoo hash table lookups to map MAC address to output ports within 600 microseconds [21][66]. Unlike RNNs, these networking applications are composed of a single kernel, and their input sizes are determined by the speed of the network. In Table 1, the input size of 8K represents the number of network packets that arrived per 100 microseconds in 40 Gbps networks.

3.1.3 Intelligent Personal Assistants

IPAs also have significant real-time constraints. Although prior work explores a series of algorithms used in an Automat-ic Speech Recognition (ASR) pipeline by IPAs, we focus on Gaussian Mixture Model (GMM) and Stemmer (STEM), two single kernel pieces that consume the most time in IPAs and thus present the biggest challenge [24]. GMM maps input feature vectors to multi-dimensional space and consumes 85% of ASR's computational time [24]. STEM reduces inflected words to a certain word stem and takes up to 85% of the remaining time in the ASR pipeline [24].

3.2 Small Data-Parallel Kernels on Latency-sensitive Applications

Table 3.1 characterizes each kernel in a single HIP [131]RNN LSTM inference job where its batch size is 1 and its hidden layer is 128. Both LSTM and GRU use 5 unique MIOpen kernels [132] and one rocBLAS [133] GEMM kernel that are called

Table 3.1.: Summary of kernels in latency-sensitive benchmarks

Applications	Kernel Name	# of calls	Execution time	Threads	Context size
LSTM	TensorKernel 1	3	3.96 us	16384	397 KB
	TensorKernel 2	5	1.79 us	128	3.1 KB
	TensorKernel 3	2	4.45 us	2048	106.8 KB
	TensorKernel 4	40	4.74 us	64	9.1 KB
	ActivationKernel 5	39	8.87 us	128	11.1 KB
	rocBLASGEMMKernel 1	13	127.48 us	1024	562.4 KB
IPV6	IPV6Kernel	1	25 us	8192	329 KB
CUCKOO	cuckooKernel	1	300 us	8192	566 KB
GMM	GMMKernel	1	1500 us	2048	195.5 KB
STEM	STEMKernel	1	150 us	4096	317 KB

multiple times in an RNN forward pass. The MIOpen kernels perform tensor and activation operations. Each kernel has a varying number of threads. However, most kernels have few threads, and do not occupy the entire GPU.

The number of threads, registers, and LDS size of kernels determine the GPU utilization. In an AMD Radeon RX 580 GPU with 36 CUs based on the GCN architecture [134], each CU can concurrently execute 2560 threads, has 256 KB 32-bit vector registers, and has 64 KB of LDS. However, LSTM's GEMM kernel only uses 1.11% of thread contexts, 1.26% of registers, and 2.78% of the LDS space. The other LSTM kernels similarly use relatively few resources. Hence, a single RNN job significantly under-utilizes the GPU, as prior work has also shown for other sequence lengths, hidden sizes, and batch size combinations [18,27]. Moreover, although IPV6, Cuckoo, GMM, and STEM are single ker-nel applications, they also complete very quickly and have narrow kernels with few threads that also under-utilize the GPU.

3.3 Impact of Job Arrival Rate

In a real system, the GPU receives job requests from different users or processes with varying arrival rates. Batching improves GPU utilization and throughput when

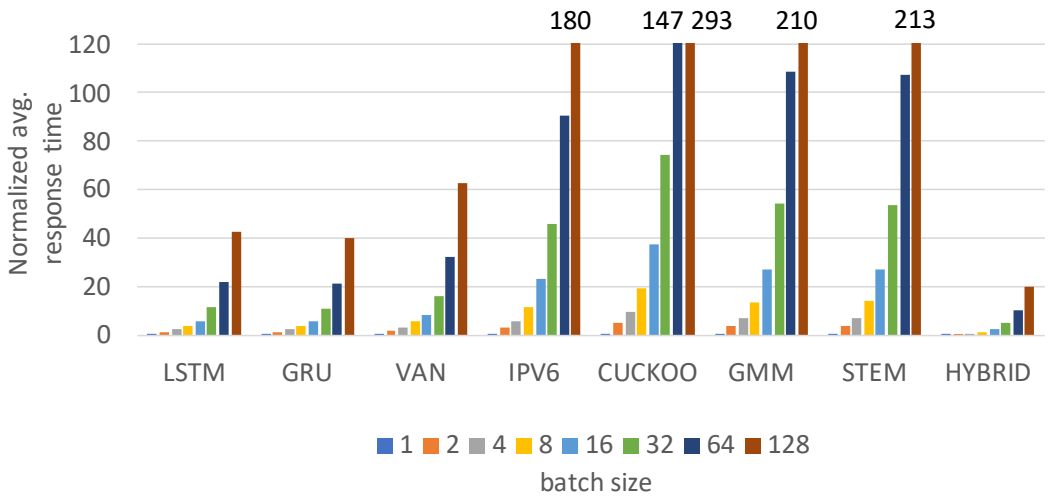

Fig. 3.1.: Comparing response times with varying job arrival rates, normalized to batch size 1.

requests arrive at the same time. However, it will delay individual jobs when requests arrive at varying rates. Streams alleviate this aspect of batching by allowing work to begin as soon as it arrives.

Figure 3.1 measures our application's response time on an AMD Radeon RX 580 GPU. We use streams to launch 32K jobs for the networking and IPA benchmarks and 512 jobs for the RNN benchmarks based on our GPU's maximum memory space. For the RNNs, we also show data for Hybrid RNNs composed of different RNN models. In this experiment, all streams use the same static priority. We issue 10000 short execution time jobs per second (IPV6, CUCKOO, and STEM) and other applications (RNNs and GMM) at 1000 jobs per second with an exponential arrival rate. Each RNN job may have a different sequence length. We add padding as needed and additional waiting time for the arrival of all jobs in a batch when the batch size is greater than 1.

In general, the high degree of parallelism within large batches increases resource contention and job execution time. For example, the response time of applications where the batch size is 128 can be 20-293X slower than the single-batched job due

to the overhead of waiting for additional jobs to arrive. Additionally, for larger batch sizes, applications such as STEM and GMM require more work groups than the GPU can concurrently schedule, which causes additional delays that further hurt performance. Thus, larger batch sizes may improve utilization for these applications, but this often comes at the cost of not meeting the deadline for the application. In contrast, using multiple streams reduces normalized runtime and allows the GPU to process multiple jobs simultaneously. However, closer inspection of these results reveals that individual job execution times vary tremendously. For example, RNN jobs with long sequence lengths complete much slower than RNN jobs with shorter sequence lengths. The observation exposes an opportunity for a more advanced GPU scheduler to prioritize longer running jobs and allow more overall jobs to meet a given deadline without wasting resources executing jobs that cannot meet the deadline.

3.4 Problems of Mixed Task and Data Parallel Applications

Mixed task and data parallel applications are often shown in many domains [135], [136]. Each independent task has a modest amount of data parallelism. The parallel processors can enhance the throughput and latency of these concurrent tasks. However, there are some performance issues when serving these tasks on parallel processors.

Remote communication overhead: In stream processing applications, tasks consist of a sequence of data sets. These tasks are often delivered over physical and wireless network channels. As a result, the streaming programming requires to manipulate these high volumes of tasks in a timely fashion [137]. Each independent streaming task often has a modest amount of data parallelism, and does not have long execution time. The communication overhead increase the latency of these tiny streaming tasks in distributing to remote machines. As a result, I demonstrate to execute multiple tasks simultaneously on a single GPU to reduce the remote communication overhead in this thesis.

Parallel programming overhead: Embarrassingly parallel tasks [138] and sparse matrix computation [139] exhibit the following problems in the parallel computation. These tasks often lead to load-imbalance because each converge stage yields the different number of tasks. Furthermore, the thread creation of OpenMP programming increases a significant overhead for task-parallel applications with small data parallelism [140]. Hence, it is required the runtime system to relieve the latency when serving these mixed task and data parallel applications.

4. A TAXONOMY OF GPU REDUNDANCY

To help understand why redundancy occurs across vector registers in the same TB, I introduce a new taxonomy of GPU redundancy. I claim that TB-wide redundancy has three classes: *uniform redundancy*, *affine redundancy* and *unstructured redundancy*. Uniform redundancy occurs when every lane in every warp of the TB contains the same scalar value for a particular named register. This typically occurs with shared constants and TB-invariant registers, like TB IDs and dimensions. Affine redundancy occurs when vector registers in different warps have the same value, which can also be represented as a (base,stride) pair. This redundancy occurs naturally in multi-dimensional TBs where consecutive lanes have consecutive threadId values, replicated in the private register space of each warp in the TB. Unstructured redundancy occurs when registers have the same vector values in each warp of the TB, but those values have no discernible pattern. In this paper, I show that all three types of redundancy can be non-speculatively identified at kernel launch time and can be eliminated without expensive runtime comparison hardware.

Figure 4.1 plots the breakdown of TB-redundant instructions under our new taxonomy for a set of benchmarks using either 1 or 2 dimensional TBs. Instructions are classified based on the type of redundancy in their source registers, and applications are subdivided by their TB dimensionality. Figure 4.1 shows that both affine and unstructured redundancy is pervasive in 2D TBs, but largely absent in 1D. The non-uniform redundancy in 2D TBs stems from the layout of the tid.x register. Consecutive threads within a warp are assigned consecutive tid.x values. When the TB's x dimension is <= the warp size, the per-lane values in the tid.x register repeat. For example, with a warp size of 4, and a TB with dimensions 4×4, the value in each of the 4 warp's tid.x register is (0,1,2,3). Furthermore, accesses to memory based on affine redundant addresses create unstructured redundancy. I observe that both

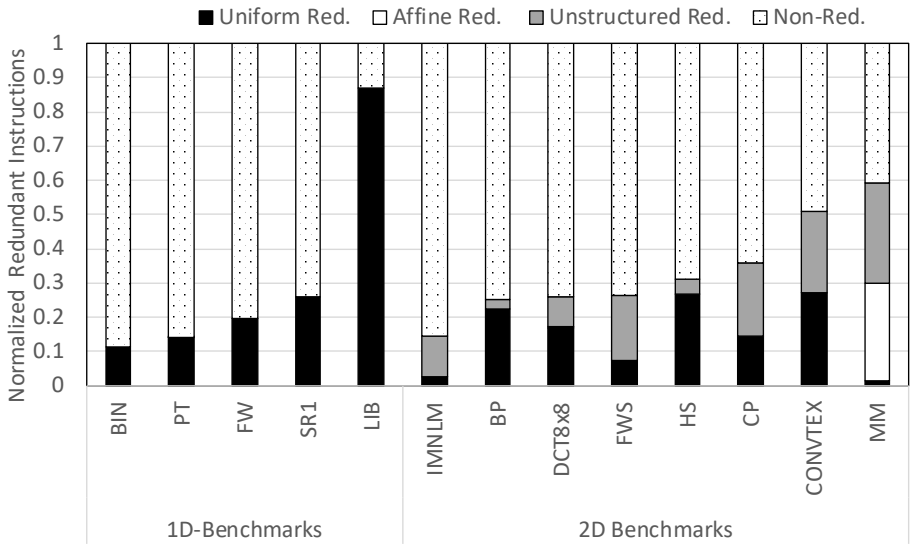

Fig. 4.1.: Fraction of dynamically executed TB-redundant instructions. Instructions executed in diverged control flow are considered non-redundant.

affine and unstructured instructions can be marked *conditionally redundant* during static compilation, and definitely redundant when the TB's dimensions are known at kernel launch time.

Uniform redundancy is simple and can typically be traced back to a collection of known constant and intrinsic variables. These values are often TB invariant, such as TB IDs, and TB dimensions. I observe that most uniform redundancy is *definitely redundant*, and common in both apps with 1D and 2D TBs, as seen in Figure 4.1.

Affine and unstructured redundancy is more subtle and stems primarily from the GPU's threadId register values. Furthermore, they are directly related to the dimensions of an application's TBs. As Figure 4.1 shows, apps with 1D TBs have little to no affine or unstructured redundancy. However, it accounts for a significant portion of total instructions executed for apps with 2D TBs.

To understand how the TB dimensions affect redundancy, consider Figure 4.2. This pseudo-assembly code reads an integer value from an array with base address 10 and is indexed by each thread's tid.x value. I use a warp size of 4 with this example

Memory
Address: [10,14,18,22,26,30,34,38]
Value: [7 , 3, 0,90,55, 8,22,1]

(a) 1D threadblock (xdim=8,ydim=1)

Output Reg. Type	Output Register Value: Warp 0 (tid.x=[0,1,2,3])	Output Register Value: Warp 1 (tid.x=[4,5,6,7])	Instructions
TB-Affine	[0,4,8,12]	[16,20,24,28]	MUL R1, tid.x, 4
TB-Affine	[10,14,18,22]	[26,30,34,38]	ADD R2, R1, #10
Unrelated	[7,3,0,90]	[55,8,22,1]	LD R3, MEM[R2]

(b) 2D threadblock (xdim=4,ydim=2)

Output Register Value: Warp 0 (tid.x=[0,1,2,3])	Output Register Value: Warp 1 (tid.x=[0,1,2,3])	Output Reg. Type
[0,4,8,12]	[0,4,8,12]	Affine Red.
[10,14,18,22]	[10,14,18,22]	Affine Red.
[7,3,0,90]	[7,3,0,90]	Unstructured Red.

Fig. 4.2.: Pseudo-assembly code to read from an integer array with a base address of 10 using tid.x as the index with 1D and 2D TBs. Values in output registers for each instruction are classified based on the pattern they make across the TB. 1D TBs create affine values that are not redundant, while 2D TBs create both affine and unstructured redundant values.

as well. Figure 4.2 (a) details the output register values for each instruction in a 1D threadblock with two warps. Each output register is classified based on the pattern it creates across all warps in the TB. In 1D TBs, the tid.x register is laid out sequentially across warps. This creates register values that are affine across the TB (TB-affine), but not redundant between warps. The two instructions that compute the address for the array lookup are also affine and not redundant because they are based on a 1D tid.x. The load instruction reads from the affine address, and the value loaded into R3 is input data-dependent. R3 is neither redundant nor affine in the TB.

Figure 4.2 (b) illustrates what happens to the same code when the TB is 2D (x=4,y=2). In multi-dimensional TBs, threadIds are assigned to warps by varying the x dimension first, and consecutive threads in a warp will be assigned consecutive tid.x IDs. Both warp 0 and warp 1 have tid.x registers that are affine within the warp, and redundant between warps (affine redundant) in this example. The resulting values computed for R1 and R2 are, therefore, also affine redundant. Since the address computed is redundant between the two warps in the TB, both instructions that load

R3 will produce the same value. R3 is an example of unstructured redundancy. Each warp has an identical value, but there is no clear pattern in the values themselves since they are input data-dependent. In this paper, I classify patterns that cannot be represented with a single <base,stride> pair as unstructured redundancy.

Although Figure 4.2 is a simple example, the conditions under which affine and unstructured redundancy occurs can be generalized. For 2D TBs, each warp's tid.x register will be redundant if the x-dimension is <= the warp size and a power of 2. I refer to this as *conditional redundancy*, since it depends on TB's dimensions that are known at kernel launch time. While this redundancy starts at the threadId registers, it propagates throughout the program through register dependencies. Furthermore, memory instructions that load values from definitely or conditionally redundant addresses will take on the redundancy characteristics of the address they load. These observations also apply to 3D TBs, where both the tid.x and tid.y registers can be conditionally redundant. This is the first work to observe that this redundancy can be identified prior to execution and optimized out by hardware at runtime.

5. PAGODA: FINE-GRAINED GPU RESOURCE VIRTUALIZATION FOR NARROW TASKS

Pagoda [26, 141] is a GPU runtime system, which breaks the hardware constraints and increases the GPU throughput on latency-sensitive applications. Pagoda issues tasks(kernels) via its specific API and reschedules warps and the shared memory within each task in its virtualization layer. Pagoda continuously passes tasks to the GPU until the full of the GPU. Thus, Pagoda increases the number of concurrent tasks and removes the limitation of the hardware queues. This section presents the composition of Pagoda and its specific programming APIs.

5.1 Pagoda Programming APIs

Programmers use Pagoda API functions in their applications to access the Pagoda runtime system. Pagoda supports the CUDA programming model, where-by the Pagoda API functions shown in Table 5.1 override the corresponding CUDA functions. The Pagoda API functions belong to the following two categories:

CPU-side API: The *taskSpawn* function launches a task from the CPU onto Pagoda. The programmer specifies the number of threads per threadblock, and the number of threadblocks as arguments. The programmer also specifies the kernel to execute, along with the parameters. The size of the shared memory needed per threadblock in bytes may be specified. The *sync* flag indicates if threadblock-level synchronization is necessary for the task. *TaskSpawn* is a non-blocking function. The CPU can synchronize with the spawned task(s) using *wait* and *waitAll* functions, or can check the task status with *check* function. One difference in functionality with respect to CUDA is that Pagoda returns a *taskID* for each task. The taskIDs are essential to use functions such as *wait*.

GPU-side API: Since Pagoda virtualizes the GPU resources, the CUDA-based shared memory allocation and threadblock synchronization cannot be directly used. Pagoda allocates shared memory and barriers for each threadblock when it gets scheduled, and the API provides functions that allow the threadblock to obtain a pointer to its shared memory, and to perform barrier synchronizations. Pagoda also offers a function to obtain the threadId of the current thread.

Figure 5.1a shows a possible implementation of Pagoda host code, while Figure 5.1 shows the corresponding device code for FilterBank. The two CPU threads spawn tasks and wait for their completion. Calling *wait()* in a nested task allows the CPU thread to progress, without getting blocked. One key distinction from CUDA is that the task kernels are written as `__device__` functions, instead of `__global__`.

Table 5.1.: Pagoda Programming API

CUDA Function	**Pagoda Function**	**Caller**	**Return Value**	**Arguments**	**Description**
kernel<<<>>>	taskSpawn	CPU	taskId	#threads, #threadblocks, shared memory, sync flag, kernel pointer, kernel args	Spawn a task from CPU onto Pagoda
cudaEventSynchronize	wait	CPU		taskId	Wait until the specified task is over
cudaEventQuery	check	CPU	true if the task is done, else false	taskId	Returns the status of the task
cudaDeviceSynchronize	waitAll	CPU			Wait until all tasks in Pagoda are over
threadIdx	getTid	GPU	thread Id		Get the thread Id of this thread
syncthreads	syncBlock	GPU			Synchronize all threads in the block
__shared__ char *arr	getSMPtr	GPU	32-byte aligned char pointer		Get shared mem pointer for the threadblock

5.2 Pagoda Runtime System

This section introduces the composition of Pagoda runtime system and first describes the design of MasterKernel that achieves GPU resource virtualization. Next, the Pagoda task spawning mechanism is described in Section 5.2.2. Pagoda task spawning mechanism employs TaskTable, a novel data structure that allows simultaneous updates from both the CPU and GPU, and is mirrored in both their memories. TaskTable drastically reduces the CPU-GPU handshaking communication by allowing lazy aggregate updates. Lastly, the section presents Pagoda's GPU scheduling mechanism that parallelizes the scheduling process, and overlaps various task processing stages.

5.2.1 GPU Resource Virtualization

The MasterKernel continually executes on the GPU as a CUDA kernel. It acquires all GPU resources, namely warps, shared memory, and registers. The MasterKernel is launched at the start of Pagoda runtime system. The kernels of tasks are translated into *__device__* functions within the MasterKernel. Later, The MasterKernel allocates its own resources to these tasks dynamically in Pagoda.

Figure 5.1b describes our MasterKernel design on the NVIDIA Pascal Titan X GPU. The MasterKernel acquires all warps of each SM (64) by launching two, 32-warp threadblocks, called *MTB*s (Master Kernel Threadblocks). Each MTB statically allocates 32KB shared memory, which later gets assigned to different tasks. The MasterKernel uses the remaining shared memory of the SM to store some of the scheduling data structures. The register count of each thread is capped at 32 (using NVCC compiler option -maxrregcount) to ensure 100% occupancy for the MasterKernel.

The first warp of each MTB is called a *scheduler warp*, while the rest of the 31 warps are known as *executor warps*. The scheduler warp is responsible for scheduling tasks on the executor warps in the MTB. It also manages shared memory allocations and barriers. The MasterKernel contains two scheduling data structures. The

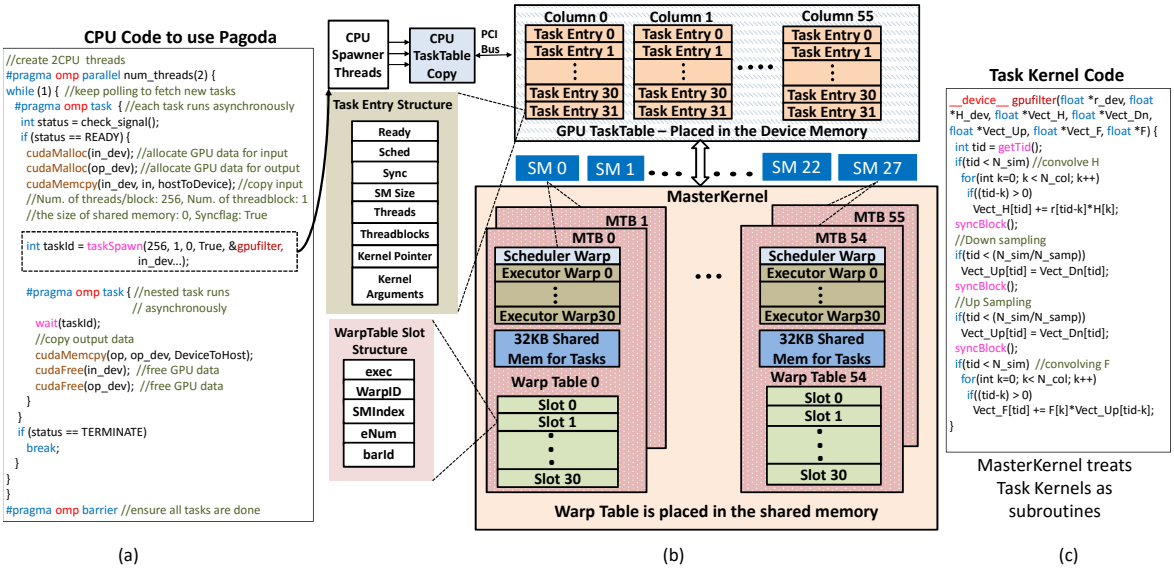

Fig. 5.1.: Pagoda runtime system overview: *The source task kernel and CPU code require few changes to an equivalent CUDA code. The MasterKernel design is shown for Nvidia Pascal Titan X GPU. The 56 MasterKernel threadblocks (MTBs) have 1024 threads each. TaskTable is mirrored on both the CPU and GPU. The CPU threads spawn tasks into the CPU TaskTable, which are then sent to the GPU counterpart. Scheduler warps inside each MTB find free executor warps to launch tasks on. The WarpTable performs bookkeeping for each executor warp.*

first one, called *TaskTable*, is mirrored on the CPU and GPU, and is used for task spawning. Each entry in the TaskTable holds a task. The GPU TaskTable receives online updates from the CPU TaskTable, and is therefore placed in the GPU device memory. The second data structure is called *WarpTable*. Each MTB contains its own WarpTable, which is placed in the shared memory. Every WarpTable contains 31 slots to maintain the status of each executor warp.

5.2.2 Continuous Task Spawning

Scheduling algorithms often involve queues that accumulate tasks, where processing elements pull tasks from the queue [14]. To simultaneously schedule several tasks, multiple pulls must take place in a synchronous/atomic manner, which has long been recognized as a critical source of overhead [12]. Performing global synchronizations or atomic operations on GPUs is extremely expensive. Therefore, to reduce this contention, one solution would be to use multiple queues, and only let a smaller set of GPU threads pull from each queue. Even this solution is impractical. As the CPU-GPU memories are discrete, before the CPU could spawn a task on a GPU queue, it must gather the queue *head* and *tail* pointers from the GPU. Such handshaking is expensive because it requires data copies over the PCIe bus. Another way to spawn tasks is to use a batch-based mechanism [9], where CPU sends a batch of tasks to the GPU. However, such mechanisms are susceptible to load imbalance across tasks.

The Pagoda design therefore employs TaskTable, a data structure that as we will show, drastically reduces the amount of CPU-GPU handshaking. Each TaskTable entry contains the following fields describing the task: 1) number of threadblocks, 2) number of threads in a threadblock, 3) task kernel pointer, 4) size in bytes of the shared memory allocation required per threadblock, 5) a flag indicating whether the task needs thread-block-level synchronization, 6) task inputs, 7) *ready* field, and 8) *sched* flag. Each TaskTable column corresponds to an MTB; The scheduler warp in that MTB schedules tasks in the column's entries onto the executor warps of that MTB. Having multiple rows in the TaskTable allows for high availability of tasks to schedule. Pagoda uses 32 TaskTable rows per MTB.

TaskTable Operation

When a task is launched via the Pagoda API (a call to *taskSpawn*), the tasks's parameters must be copied into an entry in the CPU TaskTable, then the entry must be copied to the GPU for scheduling. Since this copy has to occur while the

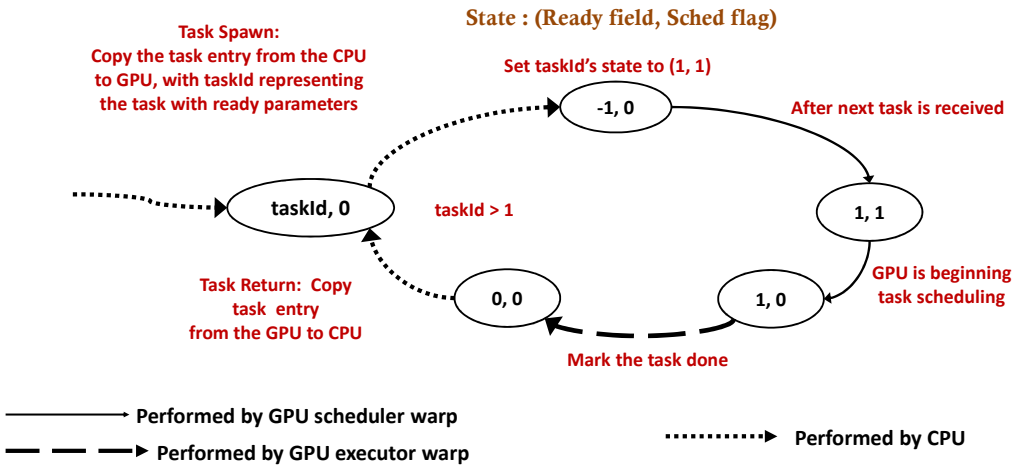

Fig. 5.2.: TaskTable State Diagram : The CPU only touches TaskTable entries with reset *ready* fields, when a task gets scheduled to warps. and the GPU only touches TaskTable entries with non-zero *ready* fields, allowing for simultaneous TaskTable updates from the CPU and GPU. The *sched* flag determines when the task gets scheduled on GPU warps.

MasterKernel is in flight, a *ready* field is necessary to indicate the finishing of the copy to the GPU. A straightforward way of implementing this, where the task's parameter data and the ready flag are copied in one *cudamemcopy* transaction, cannot work because the PCIe bus does not guarantee that the parameters will arrive in the GPU memory before the ready flag. One solution would be to simply split it into two *cudamemcopy* transactions, one for the parameters, and another for the ready flag. However, this doubles the parameter copying overhead, significantly reducing Pagoda performance. To solve this issue, we pipeline the launching of tasks. The launch of a task prompts a copy of its parameters to the GPU, as well as a pointer indicating which task had its parameters copied in the previous *cudamemcopy* transaction. In the steady-state, we achieve 1 *cudamemcopy* per task table entry and the CUDA streams API guarantees that the parameters are copied before the task is scheduled.

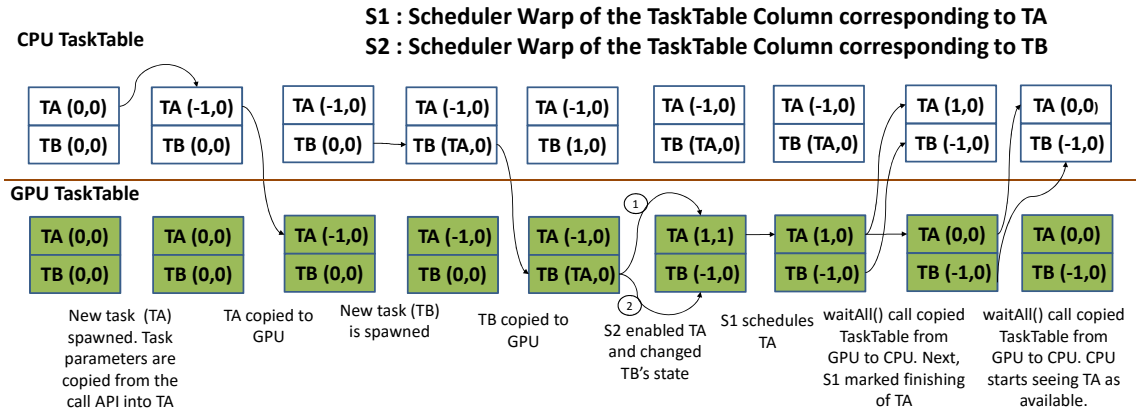

Fig. 5.3.: Example execution of task TA : TA gets scheduled only after TB is spawned. Our design allows for the CPU and GPU TaskTable entries to contain mis-matching values.

Task Spawning Example

Each task's state comprises its *ready* field and *sched* flag. The *ready* field of each TaskTable entry can be in one of four states: 0 meaning the task is not ready, -1 meaning the task's parameters have been copied to the task table, 1 meaning the task is being considered for scheduling on the GPU, or it can be a taskID which is an integer > 1. The taskID provides the necessary indirection to implement the pipelining, indicating which task has already had its parameters copied to the GPU. The *sched* flag has two states: 1, indicating that the task is ready to begin scheduling on an MTB, and 0 meaning otherwise. Figure 5.2 presents a task's state diagram.

Figure 5.3 presents an example execution of task A (TA). When the *taskSpawn* function is executed by the CPU, Pagoda finds a TaskTable entry with a cleared *ready* field and copies the task's parameters into the entry. Since TA is the first task, the CPU sets the *ready* field to -1. For all subsequent tasks, it sets the taskId of the last spawned task, e.g., during task B (TB) spawn, TA is set as the *ready* field. The taskIds generated by Pagoda are references to entries in the TaskTable. Next,

the CPU clears the *sched* flag, and copies the entry to the GPU. If the *ready* field is a taskID, i.e, > 1, the continually polling scheduler warp for the TaskTable column (S2) sets the state of the previous task (TA) to (1, 1). Next, S2 sets the state of the current task to (-1, 0) (see Algo. 1, lines 5-11). S2 waits for the state of TA to be (-1, 0) before changing it to (1, 1). This is achieved through polling orchestrated with CUDA *threadfence* calls. Now, S1, the scheduler warp for TA, finds that TA has a set *sched* flag, and hence schedules TA. To do so, S1 first clears the *sched* flag and then finds executor warps for the task. Once the task execution is finished, the last finishing executor warp for the task sets the *ready* field to 0, marking the end of the task's execution, and freeing up the task entry TA. If the CPU spawner thread observes no new tasks come in, it copies back the status of the last task, i.e. TB, and if it is (-1, 0), then sets it to (1, 1) and copies it to the GPU, ensuring the successful execution of the last task.

Lazy Aggregate TaskTable Updates: The above mechanism allows both the CPU and GPU to simultaneously update the TaskTable. As the CPU only spawns a task if the *ready* field is reset, the CPU can keep spawning as long as it finds an entry with a cleared *ready* field. Similarly, since the GPU only edits TaskTable entries with non-zero *ready* fields, it can keep scheduling as long as it finds a task entry with a set *sched* flag.

This laziness greatly reduces the number of handshaking communication calls. Furthermore, aggregated (bulk) copying achieves better data transfer bandwidth on the PCIe bus. The *wait* and *waitAll* functions return only when the *ready* field(s) of the corresponding task(s) in the TaskTable is/are reset. The laziness of TaskTable updates may block these functions if the CPU is not spawning more tasks; these functions therefore use a timeout, after which they enforce a copy-back of the involved TaskTable entries.

Because the CPU overwrites the TaskTable while the MasterKernel is in flight, coherence issues may arise. We therefore marked the TaskTable as `volatile`, and performed extensive micro-benchmarking to ascertain that the in-flight writes by the

CPU to the TaskTable are visible to the GPU and vice-versa on two GPU architectures, Tesla K40 and Pascal Titan X. This behavior is also confirmed by other researchers [13, 14].

5.2.3 Concurrent Task Scheduling

Task scheduling in Pagoda involves finding free resources (warps, shared memory) on which to execute a task. All warps of a given task execute in the same MTB. This design stems from the fact that narrow tasks need less threads than those available in an MTB.

We found that task spawning and scheduling are high-overhead operations, especially for narrow tasks, which can be short running. To mitigate this issue, Pagoda overlaps the three task processing stages, namely, spawning, scheduling, and execution. Secondly, multiple scheduler warps across different MTBs schedule tasks concurrently, lowering the time to execution for each task.

Two Pagoda data structures facilitate task spawning and scheduling in parallel: the multi-row TaskTable and the per-MTB WarpTable. While the CPU is spawning tasks on a TaskTable row, scheduler warp(s) on the GPU may schedule tasks from the remaining rows. The status of each executor warp is stored in a WarpTable entry, whose fields are described in Table 5.2.

Table 5.2.: WarpTable entry fields

warpId	maintains the warp ID of the warp, for the current task. It is used to generate the threadID in the *getTid()* function.
eNum	refers to the task entry in the TaskTable, which is being executed by the warp. This reference allows each warp to obtain the task kernel arguments.
SMindex	indicates the shared memory starting location for the corresponding threadblock.
barId	maintains the barrier ID that the warp should synchronize on. It is only valid for tasks that request threadblock synchronization.
exec	acts as a flag for the warp to begin task execution. It is also used to query the warp status.

Algorithm 1 describes the operation of the scheduler and executor warps. The scheduler warp (Lines 2-24) scans the corresponding column in the TaskTable, and when it finds an entry with a set *sched* flag (Line 12), it attempts to schedule the task. It begins by resetting the *sched* flag. If the task requires shared memory or synchronization, then the scheduler first allocates shared memory/barrier (Lines 17-20) for them (Section 5.3.1) and performs scheduling for each individual threadblock of the task. If neither shared memory nor synchronization are required, then execution is based solely on available warp slots (Line 24).

The executor warps remain idle until the *exec* flag in their WarpTable slot is set. Once this flag is set, they execute the task (Line 30). Afterwards, they release the shared memory and the synchronization barriers (Lines 33-36). Lastly, they reset the *ready* flag in the corresponding TaskTable entry, and reset the *exec* flag in the WarpTable element, marking the warp to be free (Lines 38-40). In order for this mechanism to work, the scheduler and executor warps must have a consistent view of the WarpTable, which is achieved by the *threadfences*. The scheduler warp does not explicitly monitor the end of a task execution. Hence, it cannot free the shared memory used by the task's threadblocks immediately after they finish execution. The executor warps cannot themselves deallocate the shared memory, since it may lead to inconsistencies if the scheduler warp is simultaneously allocating the shared memory. To overcome this issue, the last executing warp of each threadblock requesting shared memory marks the shared memory region to be freed, and before performing any future shared memory allocation, the scheduler warp first deallocates all memory blocks marked for freeing (Line 19). The allocation/deallocation mechanism is described in Section 5.3.1.

Scheduling is performed by the threads of the scheduler warp in parallel, through the *PSched* function (Algorithm 2). Note that the scheduler warps across different MTBs operate concurrently. The threads in the scheduler warp find free executor warps for a given task by checking their *exec* flags. If a free warp is found, a counter holding the number of warps that are yet to be scheduled is decremented atomically.

ALGORITHM 1: Pagoda Task Scheduling : Each MTB Executes this Algorithm

Input: *gTaskPool* - column of task entries in the TaskTable belonging to the given MTB, numEntriesPerPool - #rows in the TaskTable, *ctr[numEntriesPerPool]* and *doneCtr[numEntriesPerPool*] - counters allocated in shared memory, tid - threadID

```
1  while (1) do
2      if tid < warpSize then // scheduler warp does this
3          for (i = 0; i < numEntriesPerPool; i++) do
4              entry ← gTaskPool[i]
5              taskId = entry.ready
6              if taskId > 0 then
7                  threadfence()
8                  continue
9              else
10                 prevEntry.ready ← 1
11                 prevEntry.sched ← 1
12     if entry.sched then // check if sched flag is set
13         entry.sched ← 0
14         doneCtr[i] ← ctr[i] ← getNumWarps(entry)
15         if entry.SMSize > 0 ∨ entry.sync then // schedule warps per
            threadblock
16             for (j = 0; j < entry.numTB; j++) do
17                 if (entry.sync) then  barId ← getBarId()
18                 if (entry.SMSize > 0) then
19                     (retVal == false)deallocMarkedSM() // avoids deadlocking
20                     retVal ← allocSM(entry.SMSize, &index)
21                 ctr[i]← getNumWarpsPerTB(entry)
22                 pSched(ctr[i]×j, i, index, barId, &ctr[i])
23         else // schedule all warps
24             pSched(0, i, 0, 0, &ctr[i])
25
```

```
26 else // executor warps do this
27     if (warpTable[warpId].exec) then
28         entryId ← warpTable[warpId].eNum;
29         tEntry ← gTaskPool[entryId];
30         *(tEntry.funcPtr))(tEntry.args) // warp executes the task
31         if (laneId == 0) then
32             if lastWarpInBlock() then // only 1 thread per threadblock
                performs this
33                 if (tEntry.SMSize > 0) then // dealloc SM
34                     markSMForDealloc(warpTable[warpId].SMindex)
35                 if (tEntry.sync) then
36                     releaseBarId[tEntry.barId];
37             threadfence_block();
38             if (atomicDec(&doneCtr[entryId])) then
39                 tEntry.ready ← 0; // free the task entry
40             warpTable[warpId].exec ← 0 // warp is free now
```

If the result is positive, then the corresponding warp is scheduled (Lines 6-13). This counter resides in the GPU shared memory, speeding up the atomic operations. Note that both branches on lines 6 and 7 are divergent, i.e, different threads may have different branch outcomes. The threads with a false branch outcome may repeatedly execute the outer *while* loop, in spite of the other threads finding free warps. To remedy this problem, all threads in the scheduler warp must be synchronized after each iteration of the *while* loop. We achieve this using *__all()*, a CUDA warp-level vote function, as opposed to the usual CUDA API for synchronization, *__syncthreads()* which will synchronizes all MTB threads.

5.3 Supporting Native CUDA Functionality

As tasks in Pagoda are launched by the MasterKernel, native CUDA shared memory and synchronization management cannot be used. This section describes how Pagoda supports these functionalities.

ALGORITHM 2: Parallel Warp Schedule Function

Input: *tid* - thread number, *baseWarpId* - base warp number getting scheduled, eNum - number of the TaskTable column entry, index - starting address of the shared memory for the threadblock, barId - barrier Id for the threadblock, warpCtr - count of the number of warps to be scheduled

```
1  pSched (baseWarpId, eNum, index, barId, warpCtr) threadDone ← 1 // private per thread
2  i ← tid; // private per thread
3  while (1) do
4      if (i < numEntriesPerPool) then
5          threadDone ← 0;
6          if (!warpTable[tid].exec) then
7              if (id ← (atomicDec(warpCtr)) >= 0) then
8                  warpTable[i].warpId ← id + baseWarpId;
9                  warpTable[i].eNum ← eNum;
10                 warpTable[i].SMindex ← index;
11                 warpTable[i].barId ← barId;
12                 threadfence_block();
13                 warpTable[i].exec ← 1;
14     if (*warpCtr <= 0) then threadDone ← 1 ;
15     if (__all(threadDone == 1) = true) then // Synchronize threads in the
        scheduler warp
16         break;
17     i ← i + 32;
18     if (i > numEntriesPerPool) then i ← tid ;
```

5.3.1 Shared Memory Management

CUDA lacks support for software-driven dynamic shared memory allocation once a kernel has been launched. Tasks in Pagoda piggyback on the MasterKernel, and hence cannot directly use the shared memory. A need therefore arises for software management of the shared memory. Each MTB reserves shared memory when it starts execution, and allocates this memory to threadblocks of one or more tasks, and frees it after the tasks finish execution.

Our software allocator/deallocator manages small, contiguous regions of shared memory with low overhead. Unlike many general-purpose allocators that rely on freelists [142], Pagoda's algorithm is motivated by the buddy system mechanism [143] to reduce overhead. Threads of the scheduler warp in the MTB are responsible for performing the allocations and deallocations.

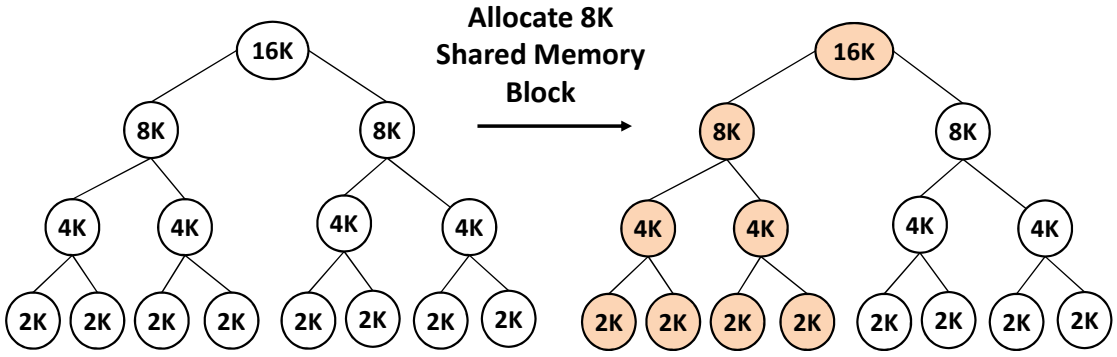

Fig. 5.4.: Allocating 8K of shared memory in Pagoda: The value in each node represents the size of the shared memory block. Note that not all levels of the tree are shown here. The white nodes are free blocks and the shaded nodes are allocated blocks.

Data Structure: The memory blocks are represented as nodes in a tree, as shown in Figure 5.4. This tree is arranged as an array in the shared memory itself, allowing fast access. Each level in the tree corresponds to memory blocks of a given size. The

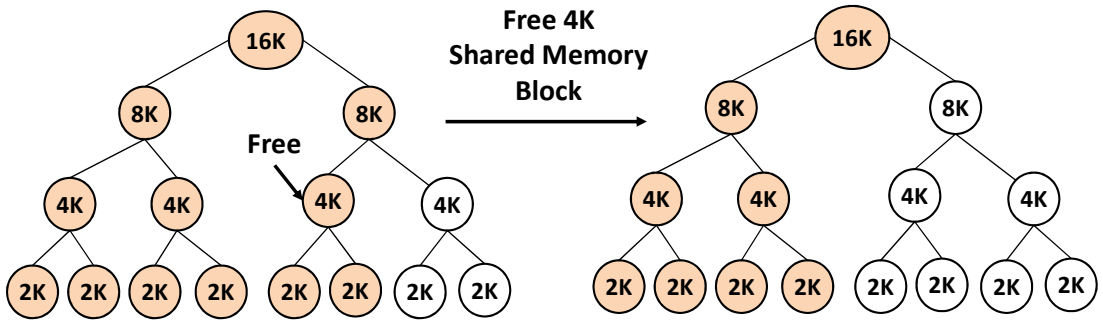

Fig. 5.5.: Deallocating 4K of shared memory in Pagoda: Ancestors of the current node are marked free only if the sibling is free.

lowest node in the tree represents 512 bytes of memory, which is the smallest allocation granularity in our mechanism. The parent of a given node represents a memory block twice as large. Thus, the total number of nodes in the tree is 128, small enough to fit in the shared memory. A marked node means the block is allocated, otherwise, it is free. An invariant of this data structure is that if a node is marked, then its parent must be marked as well.

Allocation: Figure 5.4 shows a case where a completely free tree receives an 8K allocation request. The first step is to find the tree level at which node sizes are no smaller than the request. The static mapping of blocks allows our mechanism to search for a free node on such a level of the tree, an operation which is performed in parallel by the threads of the scheduler warp. One of these threads that finds such a free node marks it. The next operation is to mark all descendants and ancestors of this node. Since the tree contains only 128 nodes, threads of the scheduler warp each check four nodes, and mark them if they are either the descendants or ancestors of the allocated node.

Deallocation: Figure 5.5 shows an example where a block of 4K needs to be freed. First, the threads of the scheduler warp work in parallel to unmark all descendants of this node. Next, the first thread of the scheduler warp unmarks the node itself, and keeps going up the tree unmarking the parent as long as the sibling node is unmarked

as well. Recall that both allocation and deallocation are carried out only by the scheduler warp, and hence no locking is necessary while performing them.

5.3.2 Sub-Thread Block Synchronization

CUDA *__syncthreads()* synchronizes threads within a threadblock. If this function is used directly within the Pagoda kernel code, the synchronization may lead to undefined behavior. This would occur because the MTB may be running two different threadblocks simultaneously, and hence all threads in the MTB may not reach the *__syncthreads()* barrier.

A naive solution to this issue would force all threadblocks running on the MTB to reach the same barrier. However, this would lead to excessive wait times in threadblocks that do not require synchronization. Pagoda presents a sub-threadblock barrier, where only the threads of a given threadblock can synchronize. Pagoda achieves this using named barriers (using *bar.sync* instruction) in the PTX programming model [144]. Each threadblock of a task that annotates the synchronization requirement in the TaskTable entry is provided a unique barrier ID during the scheduling of the threadblock (Algorithm 1, Line 17). When a threadblock encounters the *syncBlock()* function, this barrier ID is used for synchronization. The PTX model allows for only 16 such barriers. The Pagoda design therefore needs to recycle these IDs once the threadblock is finished.

5.4 Evaluation

The following subsections detail our experimental setup and results.

5.4.1 Experimental Setup

The GPU experiments are run on a node with an NVIDIA Pascal Titan X GPU, which contains 3584 1471MHz GPU cores with 12GB RAM. The machine runs

Ubuntu 16.04, with 24GB RAM and an Intel core-i7 4.0GHz quad-core CPU. All experiments were run on Nvidia Pascal Titan X GPU except Section 5.4.3. We enabled 32 concurrent kernels in the HyperQ by setting the CUDA_DEVICE_MAX_CONNECTIONS environment variable to 32. All CUDA and Pagoda benchmarks are compiled using *nvcc* from CUDA 9.0, with *-O3* option. The MasterKernel, along with all task kernels, are forced to use at most 32 registers in the Pagoda versions. The PThreads and sequential programs are compiled with *gcc -O3* and are executed on two hyperthreaded Intel Xeon E5-2660 CPUs each having 10 cores running at 2.6GHz.

Table 7.1 details the applications used in this study. We chose benchmarks from various application domains, such as signal and image processing, network security, and scientific computing where narrow tasks arise often. Table 5.4 shows the workload characteristics of the benchmarks.

5.4.2 Runtime Performance

Figure 5.6 compares the performance of narrow task applications on different CPU (pThread), and GPU runtime systems (CUDA-HyperQ, GeMTC, and Pagoda). Pagoda achieves geometric mean speedup of 1.76X over CUDA HyperQ programs, 1.44X over GeMTC, and 5.52X over the pThread benchmarks. The performance metric in Figure 5.6 is calculated over the entire execution time, including the time of both compute and data copy in the GPU benchmarks. We injected 32K narrow tasks in each application in Figure 5.6, except SLUD (273K). Each task is composed by 128 parallel threads in the GPU benchmarks. The small number of narrow tasks does not fully utilize GPU resources. Increasing the concurrent task counts is helpful for the computational throughtput of narrow tasks. Therefore, the key reason why Pagoda can outperform other runtime systems is the high GPU utilization. GeMTC increases the GPU utilization by launching work in batches and tasks are run within its SuperKernel. We could not implement SLUD in GeMTC; GeMTC needs the

Table 5.3.: Benchmark Description

MB	Mandelbrot sets are used in fractal analysis [**?**]. Each pixel value of the image is calculated in parallel; however, the required computation per pixel is highly irregular. Therefore, computation over each pixel is represented as a task that has low degree of parallelism.
MM	This is a standard matrix multiplication implementation, refactored from the NVIDIA SDK samples. We used small matrix sizes, with each multiplication running as a task to simulate the behaviour seen in an earthquake engineering simulator [**?**]. The behaviour arises from concurrent simulation of various structures, each of which is represented by different but small matrix sizes.
FB	Filterbank is a signal processing algorithm that separates input signals into multiple sub-signals with a set of filters. Multiple radios generate signals, processing each of them represents a task. Each task contains small amount of parallelizable computation.
BF	Beam former is a signal processing method used to control the direction of signal reception and transmission. Many independent signal beams receive inputs asynchronously. Processing individual inputs generate a narrow task.
SLUD	This is a sparse matrix solver using multi-frontal method [139]. A matrix is divided into multiple regular sub-matrices. Sparse LUD is represented as a task-based application owing to the irregularity in the computation size among different iterations of a parallel loop.
3DES	It is used to encrypt electronic data [145]. Network routers encrypt multiple packets as they arrive, each of which is represented as a narrow task. We use NetBench [**?**] to generate varied sizes of network packets that 3DES encrypts.
DCT	The Discrete Cosine Transform (DCT) [146] is commonly used for compression, e.g, JPEG (image), MP3 (audio), and MPEG (video) use it. Online surveillance systems gather image streams from multiple cameras, and operate on images from different streams in parallel [**?**]. Processing each image represents a narrow task.
CONV	Convolution filters [147] are used in blur and edge detection mechanisms in image processing. Each filter operation represents a task, which operates in parallel across pixels.
MPE	Pagoda is able to run multi-programmed workloads, where multiple applications generate narrow tasks asynchronously. To evaluate such a setup, we built a multi-programmed benchmark of our own. Multi-programmed environments often encounter heterogeneity in workloads. To simulate that, we chose 1) 3DES and Mandelbrot, which contain irregular computations, 2) Filterbank, which requires threadblock-level synchronization, and 3) Matrix multiplication, which uses shared memory. Each of the benchmarks contained 8K tasks, totalling 32K tasks.

number of tasks to be pre-defined, which is not the case in SLUD. Both GeMTC and Pagoda can reach 100% GPU occupancy. However, the average performance of GeMTC is 18% less than Pagoda. The reason is due to the complex task queuing and lock-step communication of GeMTC. GeMTC performs worse than CUDA-HyperQ in MB, MPE, and 3DES becuase these applications contain irregular workloads. For a fair comparison with a CPU execution, we have tried to make use of Python-based

Table 5.4.: Benchmark Characteristics

Benchmark	**Source**	**Task Type**	**Input Set per Task(each task is one image, signal, matrix or network packet)**	**Num. of Tasks**	**% Time spent in data copy (CUDA-HyperQ)**	**% Time spent in computation (CUDA-HyperQ)**	**May benefit from Shared Memory**	**Requires thread-block synchronization**	**Default Register Count**
Mandelbrot(**MB**)	Quinn [148]	Irregular	64 × 64 images	32K	24	76	✗	✗	28
FilterBank(**FB**)	StreamIt [149]	Regular	Signals of width 2K	32K	35	65	✗	✓	21
BeamFormer(**BF**)	StreamIt [149]	Regular	Signals of width 2K	32K	13	87	✗	✗	34
Image Convolution(**CONV**)	CUDA SDK [147]	Regular	128 × 128 images	32K	30	70	✗	✗	25
DCT8x8(**DCT**)	CUDA SDK [146]	Regular	128 × 128 images	32K	81	19	✓	✓	33
MatrixMul(**MM**)	CUDA SDK [**?**]	Regular	64 × 64 matrix	32K	51	49	✓	✓	30
Sparse LU Decomposition(**SLUD**)	OpenMP Task Suite [140]	Irregular	32 × 32 matrix	273K	3	97	✗	✗	17
3DES	NIST [145]	Irregular	Network packets sized 2K-64K	32K	74	26	✗	✗	26

thread pooling, OpenMP for data parallelism, pThreads-based task parallelism. We found the pThreads implementation obtained the best results, which are included in Figure 5.6.

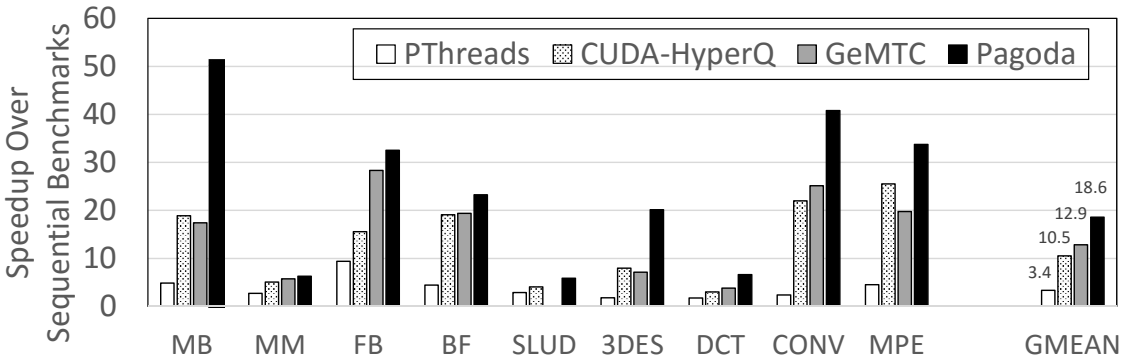

Fig. 5.6.: Overall Performance Comparison: *All applications of this experiment were run on Nvidia Pascal GPU. The number of tasks in each benchmark is constant (32K), except SLUD, which contains 273K tasks. Each GPU task uses 128 threads. The measurement of execution time contains both data copy and compute times. Pagoda significantly outperforms CUDA-HyperQ(1.76x), 20-core PThreads(5.52x), and GeMTC(1.44x) because of the high GPU utilization.*

5.4.3 Pagoda Performance Scalability

Pagoda is able to run on different types of GPUs after changing the number of SMs in its system configuration. Figure 5.7 presents the performance results of Pagoda on Nvidia Maxwell and Pascal Titan X GPU. This section aims to figure out the performance scalability of Pagoda from Nvidia Maxwell to Pascal GPU. Each application in Figure 5.7 includes 32K tasks, 128 threads in one task and the execution time calcuation does not count the time of data copy in. The number of cores on Nvidia Pascal Titan X GPU is 16% more than on the Maxwell architecture. Additionally, its compute frequency is 53% higher than on the Maxwell GPU. The average performance speedup of Pagoda-Pascal is 2.39X and Pagoda-Maxwell achieves 1.64X speedup compared to CUDA-Maxwell in Figure 5.7. The growth of compute cores and speed on Pascal GPU results in this performance improvement. CUDA HyperQ programs attach tasks in different CUDA streams and these tasks are executed when there are available GPU resources. However, the limited number of task queues on

the GPU constraints the degree of task concurrency. As a result, most of the performance benefit of the CUDA-Pascal applications only comes from the higher GPU speed and is 30% faster than on the Maxwell GPU.

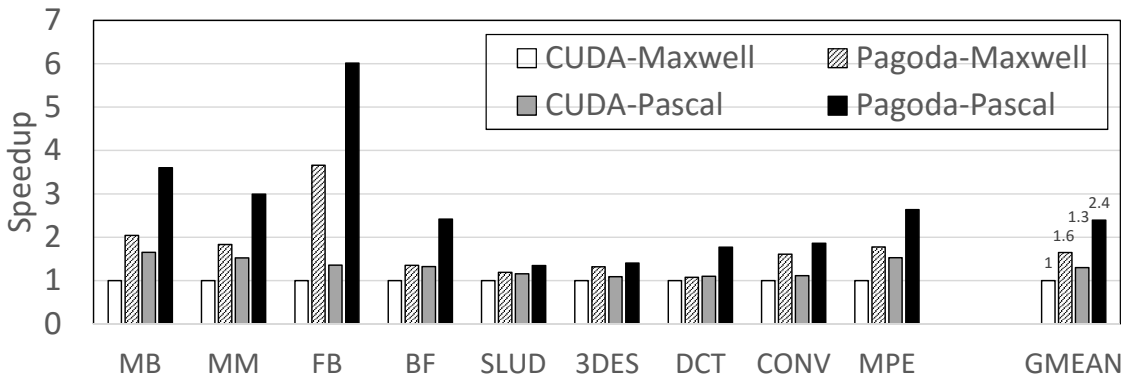

Fig. 5.7.: Pagoda Performance Scalability: *CUDA-Maxwell and CUDA-Pascal indicates CUDA-HyperQ applications are run on Nvidia Maxwell and Pascal Titan X GPU. Pagoda achieves 2.4X speedup compared to CUDA-Maxwell by running benchmarks on Nvidia Pascal GPU.*

5.4.4 Sensivity Analysis for Task Load Imbalance

Task load imbalance impacts the performance and response time of the individual tasks. This section presents the performance results of tasks composed of various computations in static task fusion, CUDA-HyperQ and Pagoda. Static task fustion combines multiple tasks into a monolithic large task [6, 7]. This method is good for tasks consisting of the same computational work, since the task fusion method can decrease the CPU-GPU communication overhead and increases the degree of parallelism within one fused task (kernel). However, this regular workload is not always seen in the real world. In this experiment, we created tasks comprising various input sizes and thread counts by using a pseudo-random geneator. Each application contains 32K tasks. The threadblock size was fixed in the fused kernel, and the

threadblock size is 256 in this experiment. We chose this number heuristically, since selecting the best thread count per task is infeasible in static fusion. The threadblock size in some sub-tasks in a fused kernel are smaller than 256, and this waste is unavoidable. The SLUD application cannot be fused because the number of tasks is not known statically.

Figure 5.8 demonstrates the speedup of static task fusion and Pagoda compared to CUDA-HyperQ. Pagoda gains 1.8X speedup and the static task fusion method is about 10% slower than CUDA-HyperQ applications. There are two reasons for the slowdown shown in the static task fusion benchmarks. First, the longest task in a fused kernel dominates the execution time. This situation is often shown in compute-intensive applications such as MB, FB and CONV in Figure 5.8. Secondly, the underutilization is shown in a fused kernel because the threadblock size of each sub-tasks can be different. In contrast to the static fusion method, Pagoda launches tasks in quick succession without combining tasks in a batch. Additionally, Pagoda follows the availability of the GPU hardware resources to allocate tasks. Pagoda's task allocation mechanism can fully utilize the GPU and satifsy dynamic task workloads.

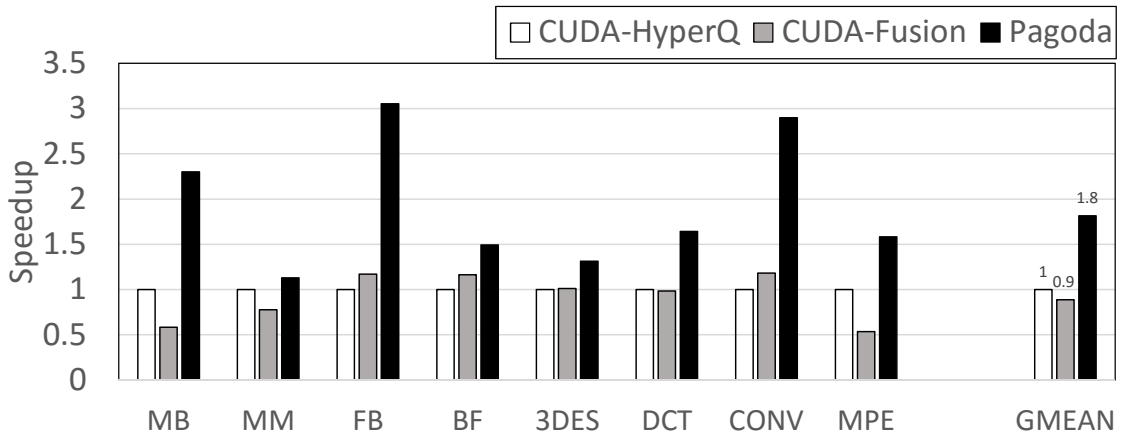

Fig. 5.8.: Performance Comparison of Statis Fusion, CUDA-HyperQ and Pagoda with irregular tasks: *Dynamic task spawning mechanism in Pagoda obtains high performance even with irregular workloads.*

5.4.5 Task Latency Analysis

Figure 5.9 compares the normalized average response time between Pagoda and static fusion applications. The average response time of static fusion benchmarks is 40% and 440% longer than Pagoda shown in Figure 5.9. The batch size in Figure 5.9 means the number of threadblocks in one fused kernels. The average response time measures the total task execution time over the number of injected tasks. Each benchmark in this experiment contains 16K irregular tasks with various threadlblocks and input sizes. Each task in a statically fused kernel and in a batch-based system such as GeMTC, is completed until all tasks in the fused kernel or batch have done the work. Thus, their response time increases with the the number of tasks per batch.

In Figure 5.9, the average response time of BF-Fusion decreases with growing batch size. The reason is that the large batch size increases the degree of parallelism in the fused kernel. The parallelism helps the performance of the fused kernel and response time. However, this growth trend does not always increase linearly and slows down when the batch size is over 256 in FB-Fusion in Figure 5.9. Threadblocks specified in the fused kernel can be over the pre-defined threadblock counts on GPUs. Over-subscribing threadblocks in the fused kernel increases the contention of resources and slows down the performance speedup. Instead of batching tasks to increase parallelism, Pagoda launches tasks successively on the GPU. Pagoda allocates tasks to the GPU dynamically based on the availability of hardware resources and decreases the resource contention.

5.4.6 Lock Step Communication Overhead

To understand the benefit of continuous task spawning, this experiment creates a Pagoda version that spawns tasks in batches. This batch Pagoda version does not spawn tasks untill all tasks in the previous batch are done. Figure 5.10 compares the performance of Pagoda, Pagoda-batch and GeMTC. Each applications in Figure 5.10 contains 32K tasks and their threadblock size is 128. In Figure 5.10, Pagoda

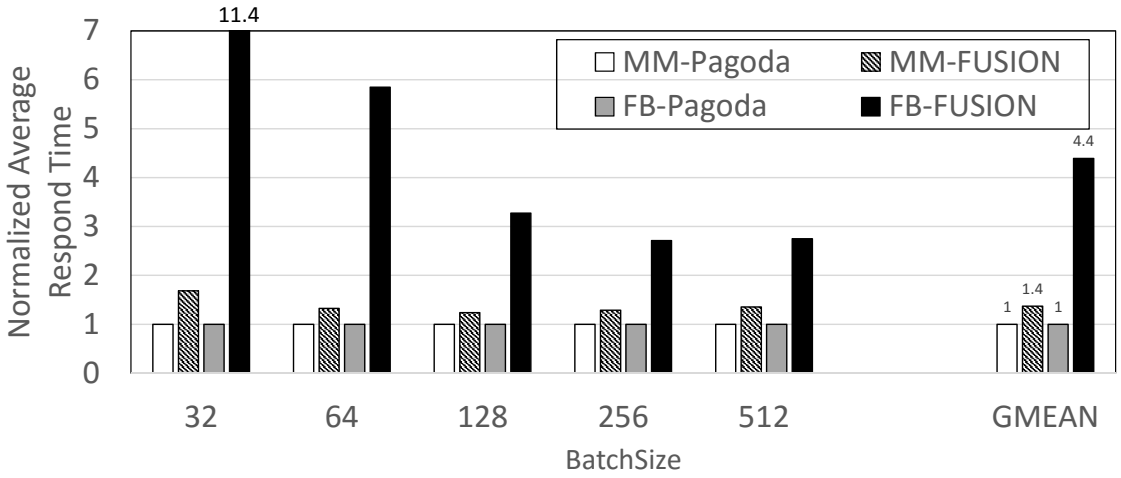

Fig. 5.9.: Average Latency of Tasks *Pagoda achieves much lower latency compared to static fusion.*

gains 2.53X speedup compared to GeMTC. This performance improvement comes from the concurrent task scheduling in Pagoda. GeMTC's complicate task queue method hindered the task concurrency within its batch. In addition, on average, the Pagoda-batch implementation incurs 29% overhead because of its lock-step task launch mechanism. Pagoda overlaps the task spawning and execution to achieve this speedup compared to the Pagoda-batch alternative. In Figure 5.10, CONV only gets 5% performance benefit from continuous task spawning becuase its regular, extremely short running task. However, MPE demonstrates the exceptionally high benefit in the presence of unbalanced tasks.

5.4.7 Pagoda Task Scheduling Overlead Analysis

Figure 5.11 shows the overhead of Pagoda task scheduling for various threadblock and input size. In Figure 5.11, the threadblock size of the HyperQ applications is 256, and both benchmarks contain 32K tasks. In the CUDA-HyperQ programs, the GPU hardware distributes threadblocks of one task (kernel) to other SMs. However, Pagoda only uses 31 executor warps from the MasterKernel ThreadBlock (MTB) to feed the

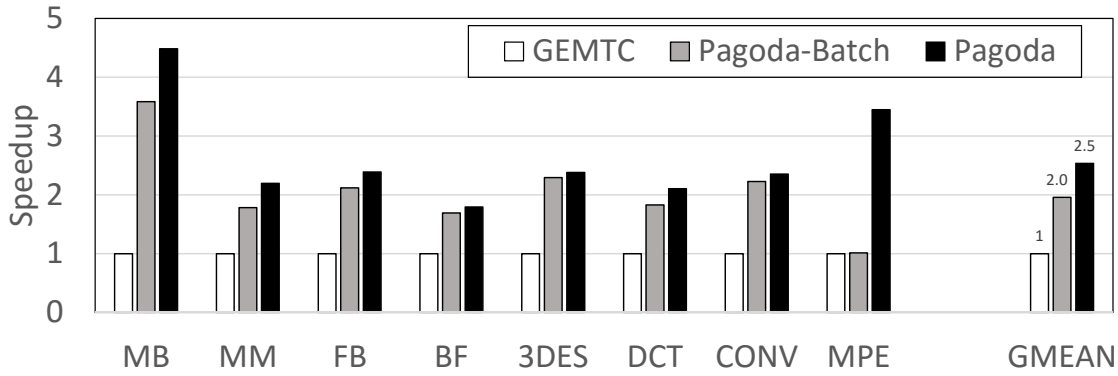

Fig. 5.10.: Benefit of Pagoda Continuous Spawning and Concurrent, Pipelined Task Processing *Pagoda performs both continuous task spawning and concurrent, pipelined task processing. Pagoda-batching only performs task processing. GeMTC performs neither. Pagoda outperms GeMTC in all cases.*

request of one task. Hence, some warps in one task must wait for executor warps to finish. This case occurs in the big task comprising a large number of threadblocks in Pagoda. As shown in Figure 5.11, Pagoda obtains 3% runtime overhead in MM task as its input size is 256 $\times$ 256 and 32K threads. The speedup of HyperQ programs increases linearly with increasing of threadblock size from 32 to 512. The increase of parallelism in one task facilitates this speedup in HyperQ applications. However, the speedup of HyperQ does not improve beyond their threadblock size of 1024, as the GPU is fully utilized. On the other hand, the performance of Pagoda does not fluctuate with changing threadblock size, since Pagoda increases the number of concurrent tasks in small threadblock size configurations. Furthermore, 256 $\times$ 256 input size and 64K threads of CONV, the speedup of Pagoda improves again. We attribute this behavior to the warp-level scheduling in Pagoda versus the threadblock-level scheduling in CUDA. CUDA prevents a new threadblock from launching until all warps of the previous threadblock finish, where as Pagoda can schedule a warp from a new threadblock as soon as another warp completes.

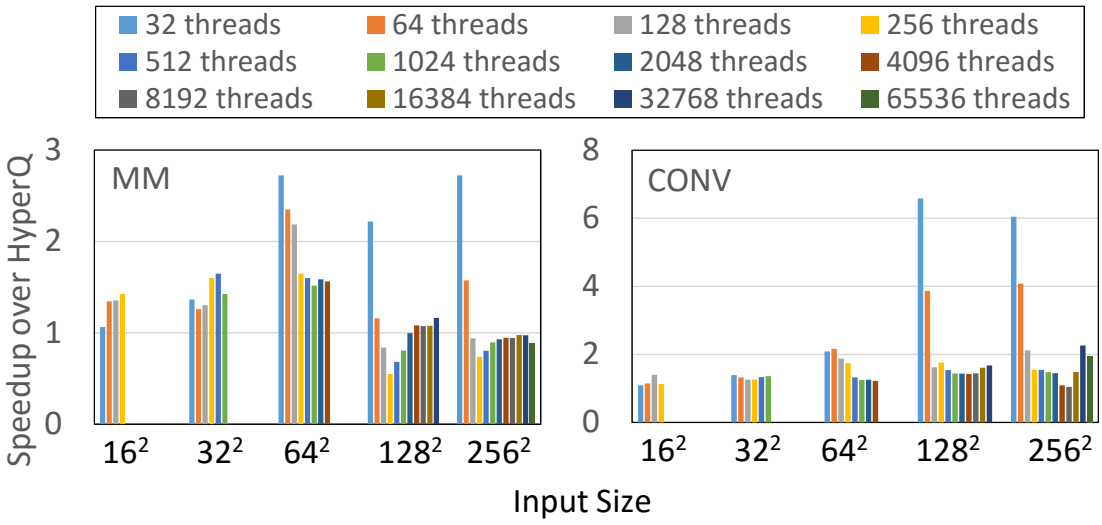

Fig. 5.11.: Effects of varying threads per task for different input size *For small threads, Pagoda outperms HyperQ in all input sizes. For large thread counts, Pagoda may still outperform HyerQ because its finer grain of scheduling.*

Table 5.5.: Compute performance comparison of tasks run in Pagoda with and without shared memory allocation: Each version runs 32K tasks. DCT tasks have 64 threads, MM tasks contain 256 threads. Only the compute time is compared. The shared memory usage offers considerable benefits.

Benchmark	**Pagoda with Shared Memory**		**Pagoda without Shared Memory**	
	Speedup over HyperQ using Shared memory	Achieved Occupancy	Speedup over HyperQ using Shared memory	Achieved Occupancy
DCT	1.13x	25%	1.02x	97%
MM	1.47x	97%	1.32x	97%

5.4.8 Pagoda Shared Memory Analysis

Pagoda performs software management of the GPU shared memory, as described in Section 5.3.1. To compare the obtainable performance benefits from the use of

shared memory, we show performance results on the DCT and MM benchmarks. These two codes can potentially benefit from the use of shared memory. We created two versions for each: with and without using shared memory. Table 5.5 compares the speedups achieved by these versions over the CUDA-HyperQ versions, which also use the shared memory. The shared memory requirement may reduce the achieved occupancy; yet, Pagoda shared memory versions achieve performance benefits. None of the other static-fusion or runtime batching solution offer shared memory utilization, and miss out on such benefits.

5.5 Summary

This chapter presents Pagoda, a GPU runtime system that overcomes under-utilization in the presence of narrow tasks. Pagoda virtualizes GPU resources via MasterKernel, a continually executing daemon on the GPU. Pagoda launches tasks on the GPU as long as some free warps are available. Unlike previous work, Pagoda supports most functionality of the native CUDA model. A key distinction in Pagoda is the task spawning and scheduling mechanism. It contains a novel data structure, called TaskTable, that greatly reduces CPU-GPU handshaking during task spawning. Pagoda achieves concurrent task scheduling, and overlaps task spawning, scheduling, and execution through pipelining. The experimental evaluation showed that Pagoda achieves a geometric mean speedup of 1.76x over CUDA- HyperQ, 1.44x over GeMTC, and 5.52x over 20-core CPU PThreads. The evaluation also showed that Pagoda can outperform static fusion schemes by 1.79x, and achieves much lower latency per task. We believe that the Pagoda design makes it easy to exploit GPUs for applications exhibiting narrow tasks, and will encourage porting of many non-traditional workloads to GPUs.

6. LAX: DEADLINE-AWARE JOB SCHEDULING ON THE GPU

LAX is a deadline-aware GPU job scheduler. LAX dynamically adjusts kernel priorities, allowing more jobs to meet their deadlines. LAX has three key components: a mechanism to estimate the time remaining in each job (Section 6.2), a method for estimating the queuing delay of incoming jobs to prevent oversubscription (Section 6.3), and a laxity-aware scheduling algorithm that changes jobs' priorities based on estimated laxity (Section 6.4).

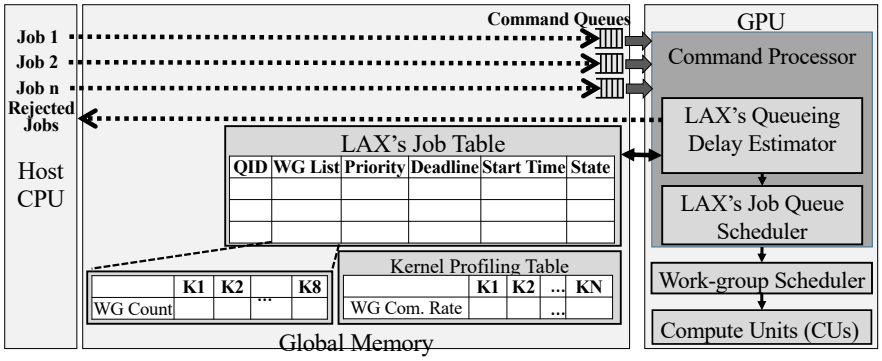

Fig. 6.1.: LAX procedure and system overview

6.1 LAX System Overview

Figure 6.1 presents an overview of our LAX framework. In multi-job GPU applications, all kernels associated with a single job are enqueued on the same stream or underlying GPU compute queue. Before running a job, LAX performs stream inspection to look ahead, parsing all the kernels in a queue to determine their names and associated number of WGs. To store this information, LAX introduces a Job Ta-

ble that stores information about the work remaining in each compute queue. After parsing the WG information for every kernel in a queue, this information is added to the corresponding WGList for the queue. To compute the estimated time remaining in each job, LAX uses the number of WGs from the WGList and a per-kernel work completion rate stored in a Kernel Profiling Table. Given the estimated time remaining in the job and a programmer-specified deadline (passed when initializing the job on the GPU stream), LAX computes the laxity of each job using Equation 6.1. The LaxityTime tells us how close to its deadline a job is predicted to finish. Comparing each job's LaxityTime effectively tells LAX their relative priority. Jobs with less laxity have higher priority.

$$LaxityTime = Deadline - (TimeRemaining + DurationTime) \tag{6.1}$$

Similar to prior work, LAX uses a pull-based model for offloading work from a CPU server [29, 30]. LAX successfully offloads as many jobs as possible. However, unlike prior work, LAX uses its per-job completion time estimates to generate a queuing delay estimate for new jobs entering the system. Based on current contention, if LAX estimates that the new job will not meet its deadline, it will not attempt to offload the job to the GPU.

6.2 Job Remaining and Laxity Time Estimates

In order to estimate the laxity of currently executing jobs and the queuing delay encountered by incoming jobs, LAX must generate an estimate of the time remaining in each job. To generate this estimate, LAX makes use of an in-memory Job Table. As illustrated in Figure 5, each entry in the table contains the following fields: 1) QueueID (QID); 2) Priority (used by the CP to make scheduling decisions); 3) WGList (the list of kernels that comprise the job and the number of WGs that need to be executed for each kernel); 4) Deadline (provided by the programmer); 5) StartTime; and 6) State (either init, ready, or running). In addition to the Job Table, LAX stores

ALGORITHM 3: LAX Queuing Delay Calculation

```
1  totRemTime = 0
2  // new jobs are pushed to the end of the queue
3  for i = JobQ.begin() to JobQ.end() do
4      holdJobTime = 0
5      durTime = curTick() – JobQ[i].startTime
6      for j = 0; j < JobQ[i].WGList.size; j++ do
7          kernelID = JobQ[i].WGList[j].kernelID
8          if JobQ[i].state != init then
9              /* sum the total remaining time of jobs */ totRemTime +=
                JobQ[i].WGList[j].numWG / kernelTable[kernelID].WGCompRate
10         /* initialize new job's estimate */
11         else
12             holdJobTime += (JobQ[i].WGList[j].numWG /
                kernelTable[kernelID].WGCompRate)
13     if (totRemTime + (holdJobTime + durTime) < JobQ[i].Deadline) then
14         JobQ[i].state = readyState
15         totRemTime += holdJobTime
16     /* Cannot complete job in time, tell CPU */
17     else
18         rejectJob()
```

the per-kernel WG completion rates in a Kernel Profiling Table which is periodically updated (empirically set at 100 microseconds) to reflect contention conditions in the GPU. Overall, LAX re-quires only 4240 bytes of memory to store this information for a 128-compute queue system.

To predict a job's remaining time, LAX scans the WGList to generate an estimate for how long each kernel in the job will take. For each entry in the list, LAX looks up the current WG completion rate for this particular kernel in the Kernel Profiling Table. By dividing the number of WGs in each kernel by the current WG completion

rate for that kernel type, LAX generates a time estimate for the kernel. Since the kernels in a job have sequential dependencies and thus must be executed sequentially, LAX simply sums the estimated execution time of each kernel to generate the per-job estimate. As WGs complete, the WGCount entry in the Job Table is decremented to reflect the fact that the job has less work remaining. LAX combines the job's remaining time estimate with the user-specified Deadline and the job's StartTime to generate the estimated LaxityTime. We describe the State and Priority fields in more detail in Sections 6.3 and 6.4.

ALGORITHM 4: LAX: Laxity-aware Scheduling

```
1  for i = JobQ.begin() to JobQ.end() do
2      JobQ[i].RemTime = 0
3      for j = 0; j < JobQ[i].WGList.size; j++ do
4          kernelID = JobQ[i].WGList[j].kernelID
5          JobQ[i].RemTime += JobQ[i].WGList[j].numWG /
             kernelTable[kernelID].WGCompRate
6      JobQ[i].durTime = curTick() – JobQ[i].startTime
7      ComplTime = JobQ[i].RemTime + JobQ[i].durTime if JobQ[i].deadline >
         ComplTime then
8          /*laxityTime = deadline – ComplTime*/
9          JobQ[i].prior = JobQ[i].deadline – ComplTime
10     else
11         JobQ[i].prior = ComplTime
12     /*deprioritize job if LAX cannot make deadline */
13     if JobQ[i].durTime > JobQ[i].deadline then
14         JobQueue[i].prior = INF
```

6.3 Preventing Oversubscription with Queuing Delay Estimation

When designing a GPU to accept multiple jobs, each of which has a tight deadline, preventing oversubscription is critical. As discussed in Section 6.1, LAX only accepts jobs predicted to complete before their deadlines. To make this prediction, LAX must: (1) estimate how long a job J will take on the GPU under current conditions and (2) estimate how long J may be delayed behind other jobs already sent to the GPU, i.e. its queuing delay. Using each job's time remaining estimate from Section 6.2, LAX first computes how long J should take, given current completion rates.

Estimating J's queuing delay is more challenging, because the deadlines and arrival rates of latency-sensitive jobs vary significantly. However, Little's Law works well independent of arrival rate [41,42]. Thus, LAX uses Little's Law to model the queuing delay of the jobs running on the GPU. Accordingly, Algorithm 3 uses Little's Law to sum up the predicted remaining time of all jobs currently execution in the system, including jobs that are ready but not running. Combining this estimate with the runtime estimate, if LAX predicts J will complete by its deadline, it accepts J and changes its state from init to ready, informing the CP that J's first kernel is ready to be executed on the GPU.

6.4 Laxity-Aware Job Scheduling Algorithm

Next LAX needs to determine which job(s) should be run next. A job's laxity determines its priority in the laxity-aware job scheduler. The scheduler assigns each queue (job) a priority level and may adjust it over time. The job with the smallest current laxity is assigned the highest priority.

Algorithm 4 describes LAX's priority update mechanism, where priority zero is the highest priority level. Every 100 microseconds, the priorities are updated, as we empirically found this gave the best performance. Since we want to prioritize jobs with the least laxity, any job that is predicted to complete by its deadline is assigned its laxity value as its priority (Line 9, Algorithm 4). LAX decreases a

job's priority when it predicts the job will not reach the deadline. When a job is predicted to miss a deadline, its completionTime (remainingTime + durationTime, where durationTime is the amount of time since this job was enqueued) is greater than the deadline. To achieve de-prioritization, LAX sets the job's priority to be equal to the completionTime (Line 11, Algorithm 4), because it is greater than the deadline, guarantees that the job has a lower priority than any other job that still has positive laxity. Once it has adjusted the priority for all jobs, the laxity-aware queue scheduler issues all WGs from the highest priority job. If additional WG slots are available, it will then move on to the next highest priority ready job, and so on until all WG slots are filled. When all WGs are issued, LAX updates the associated job's status to running.

Table 6.1.: Key simulated system parameters

GPU Clock	1500 MHz
The number of CUs	8
Number of SIMD units per CU	4
Max wavefronts per SIMD unit	10
Vector register size per CU	256 KB
The number of compute queues	128
CPU Clock	4000 MHz
# CPUs	2
GPU L1-D$ per CU	16 KB, 64B line
GPU L1-I$ per 2 CUs	32KB, 64B line, 16 way
GPU L2 cache per 64 CUs	4MB, 64B line
Main Memory	16 GB HBM2, 16 channels, 16 banks/channel, 1000 MHz

6.5 Methodology

I use the gem5 simulator [150, 151], which offers native GPU ISA support [47, 150] and a cycle-level GPU microarchitecture model to evaluate the latency-driven applications. Prior work has shown that gem5's GPU model provides high correlation

with modern GPUs [151]. The simulated system assumes the CPU and GPU share a single unified cache coherent address space and do not require explicit copies [152]. By default, the original codes in our benchmarks assume discrete memory spaces between the CPU and GPU and use device copies. Thus, we modified the evaluated all our bench-marks to remove the device copies wherever possible. We analyze energy consumption with per-instruction energies [84].

Table 6.1 presents the simulated parameters we used in gem5. The gem5 simulator provides the sophisticated GPU front-end model which allows us to faithful simulate all memory operations associated with asynchronous streams and queue management. Additionally, the modeled WG scheduler assigns WGs in an oldest-first manner.

6.5.1 Evaluated Compute Queue Scheduling Policies

To determine how our laxity-based scheduler compares to prior work, we compare it against eight other queue scheduling policies, which are detailed in Table 6.2. These schedulers leverage various policies with the static and dynamic information to schedule kernels and can be broken into three groups: state-of-the-art CPU-side schedulers [28][53][54], GPU approaches that extend the CP, and variants of our laxity-based scheduler that vary the amount of required changes. We implemented all of these schedulers in gem5.

CPU-side scheduling mechanisms such as BAT [19], BAY [30], and PRO [29] improve throughput without requiring hardware changes. However, BAT, BAY, and PRO incur overheads for communicating between the CPU and GPU. For a tightly coupled GPU like the one in our system, this adds 4 microseconds of host-device communication overhead per kernel in a job. Similarly, we added 50 microseconds of overhead to BAY for calls to its regression model, based on reported data [30].

Modern GPUs perform deadline-blind RR scheduling, but since the CP is programmable (although no API has been disclosed by GPU vendors), it is possible to extend it for other widely used scheduling approaches such as LJF, MLFQ, SJF, and

Table 6.2.: Scheduling Policies

Scheduler	Description
CPU-Side Scheduling	
BatchMaker (BAT) [19]	A dynamic batching technique where each stream can have a different batch size.
Baymax (BAY) [30]	Uses pre-trained models to predict a jobs execution time and re-orders the priorities of jobs based on their QoS headroom.
Prophet (PRO) [29]	Uses offline profiling to choose which concurrent jobs to issue in order to fully utilize the GPU and improve QoS.
GPU Command Processor Scheduling	
Round-Robin (RR)	The baseline scheduler that processes compute queues in a cyclic manner.
Multi-Level Feedback Queue (MLFQ) [153]	Moves jobs between two priority queues based on their runtime and uses RR to schedule jobs in the high priority queue.
Shortest-Job First (SJF)	A static scheduling policy that schedules kernels with the shortest job first.
Shortest Remaining Time Job First (SRF)	A dynamic policy that uses LAX's remaining execution time estimator to assign job priorities. It then assigns the job with the shortest estimated remaining time the highest priority.
Longest-Job First (LJF)	A static scheduling policy that schedules kernels from the longest jobs first.
PREMA [38]	A multi-task scheduler for heterogeneous systems that predicts job priorities and preempts lower priority jobs
LAX	Our laxity-aware scheduling policy
Laxity-Aware Scheduling Variants	
LAX-SW	A variant of LAX that uses CPU-side scheduling.
LAX-CPU	A variant of LAX that does CPU-side scheduling but changes the API to allow rapid changing of the priority of the jobs.

SRF. Like LAX, SJF, SRF, and LJF utilize predicted runtime information for each job to determine what to schedule; however, they do not model queuing delay or laxity of jobs. Finally, for MLFQ we found that it per-formed best with two priority levels, demoted jobs to the lower priority level [153] when its runtime exceeded one-third of

the jobs' deadline, and promoted the job back to the higher priority when its runtime exceeded 2/3 of its deadline.

Finally, we also examine three variants of our laxity-aware scheduler: LAX, which extends the CP; LAX-SW, which performs CPU-side scheduling (and incurs overheads for host-device communication); and LAX-CPU, which also does CPU-side scheduling, but changes the API to allow dynamic job priority changes from user-level software. To do this, the API writes the updated priorities to memory-mapped registers that control the priorities of each queue [151].

Table 6.3.: LAX Benchmarks

Benchmark	Deadline	Input / hidden layer size	Input / hidden layer sizeHigh Job Arrival Rate (jobs/s)	Medium Job Arrival Rate (jobs/s)	Low Job Arrival Rate (jobs/s)
Many Kernel					
LSTM [154, 155]	7 ms	128	8000	5000	3000
GRU [154, 155]	7 ms	128	8000	5000	3000
VAN [154, 155]	7 ms	128	8000	5000	3000
HYBRID [154, 155]	7 ms	128/256	8000	5000	3000
Few Kernel					
IPV6 [130]	40 us	8192	64000	32000	16000
CUCKOO [130]	600 us	8192	8000	5000	3000
GMM [156]	3 ms	2048	32000	16000	8000
STEM [156]	300 us	4096	64000	32000	16000

6.5.2 Benchmarks

To evaluate these schedulers, we use the eight latency-sensitive benchmarks. Table 6.3presents the input size, deadline, and arrival rates for each benchmark. To determine the appropriate deadlines for each benchmark, we used deadlines from recent work where available: 7 ms for RNNs [25], 40 microseconds for IPV6 [21, 23],

and 600 microseconds for Cuckoo [21]. For the IPA benchmarks, we used the same methodology as the authors: we ran each benchmark in isolation, then doubled the worst case latency [29, 30].

To demonstrate how GPUs can simultaneously execute kernels with different degrees of parallelism, we also include a Hybrid RNN benchmark that includes the two most popular RNN variants, LSTM and GRU, with a mixed hidden layer size of 128 and 256, respectively. The input for all RNNs is based on the WMT '15 language translation trace [157] , which has an average sequence length of 16. Furthermore, we share weight data across RNN inference jobs with the same hidden size [19, 84]. Although our technique is applicable to any data width, we use DeepBench's provided precision for the RNNs. Additionally, our schedulers do not affect the RNN inference accuracy since they do not change the underlying algorithms.

6.5.3 Job Arrival Rate

I simulate 128 jobs per benchmark with different arrival times and map one job to one GPU stream in our simulator. Real world system continually receives requests with varying arrival rates. As with determining the proper deadlines, wherever possible we used the same arrival rates as previous work on these benchmarks. For CUCKOO, GMM, and STEM, we modified these rates to account for the difference in system size. Moreover, we sweep multiple levels of contention (high, medium, and low arrival rates) for each benchmark to evaluate the effect of contention on the GPU schedulers. For each arrival rate, we randomly generate the arrival time of a specific job based on an exponential distribution.

6.6 Experimental Results

Overall, LAX successfully offloads more jobs than prior approaches. At the highest arrival rate LAX completes a geometric mean (geomean) of 2.7X – 4.8X and 1.7X – 5.0X more jobs by their deadlines than CPU-side schedulers and schedulers that

extend the CP, respectively. Moreover, CPU-side laxity-aware scheduling outperforms other CPU-side schedulers but requires CP extensions to obtain the full benefits of laxity. Finally, LAX wastes less work, accurately predicts job laxity, provides a better combination of energy consumption and performance, and provides a better combination of throughput and 99-percentile latency.

6.6.1 Completing Jobs by Their Deadlines

CPU-Side Schedulers

Figure 6.2 plots the number of jobs completed by their dead-lines for each arrival rate, normalized to RR, for the CPU-side schedulers, RR, and LAX. In general, most schedulers do well for the lower arrival rates, where contention is low. At the high job arrival rate, contention increases, and all schedulers start missing more deadlines.

RR: As expected, RR does not do very well because it schedules jobs in deadline-blind fashion. However, for the single-kernel benchmarks (IPV6, CUCKOO, GMM, and STEM), which also have equal job sizes, RR does better, especially at higher arrival rates, because a new job will some-times be chosen to run soon by RR if RR is near the end of the queue when the job is added, reducing queuing delay. Although this also occurs for the multi-kernel jobs, since these jobs may have long sequence lengths, the benefit is smaller.

BAT: BAT dynamically combines kernels in a batch. When jobs arrive simultaneously, and are executing the same kernel, this significantly improves efficiency. However, BAT executes these kernels in a lock-step manner and is not aware of the job's deadlines. As a result, BAT performs poorly for many of these latency-sensitive workloads, especially as contention increases. Overall, BAT completes a geomean of 23% fewer jobs than RR by their deadlines.

BAY: BAY generally outperforms deadline-blind schedulers like RR and BAT by effectively predicting the execution time of jobs and using its QoS headroom calculations to control the number of concurrent jobs. However, BAY's prediction overhead

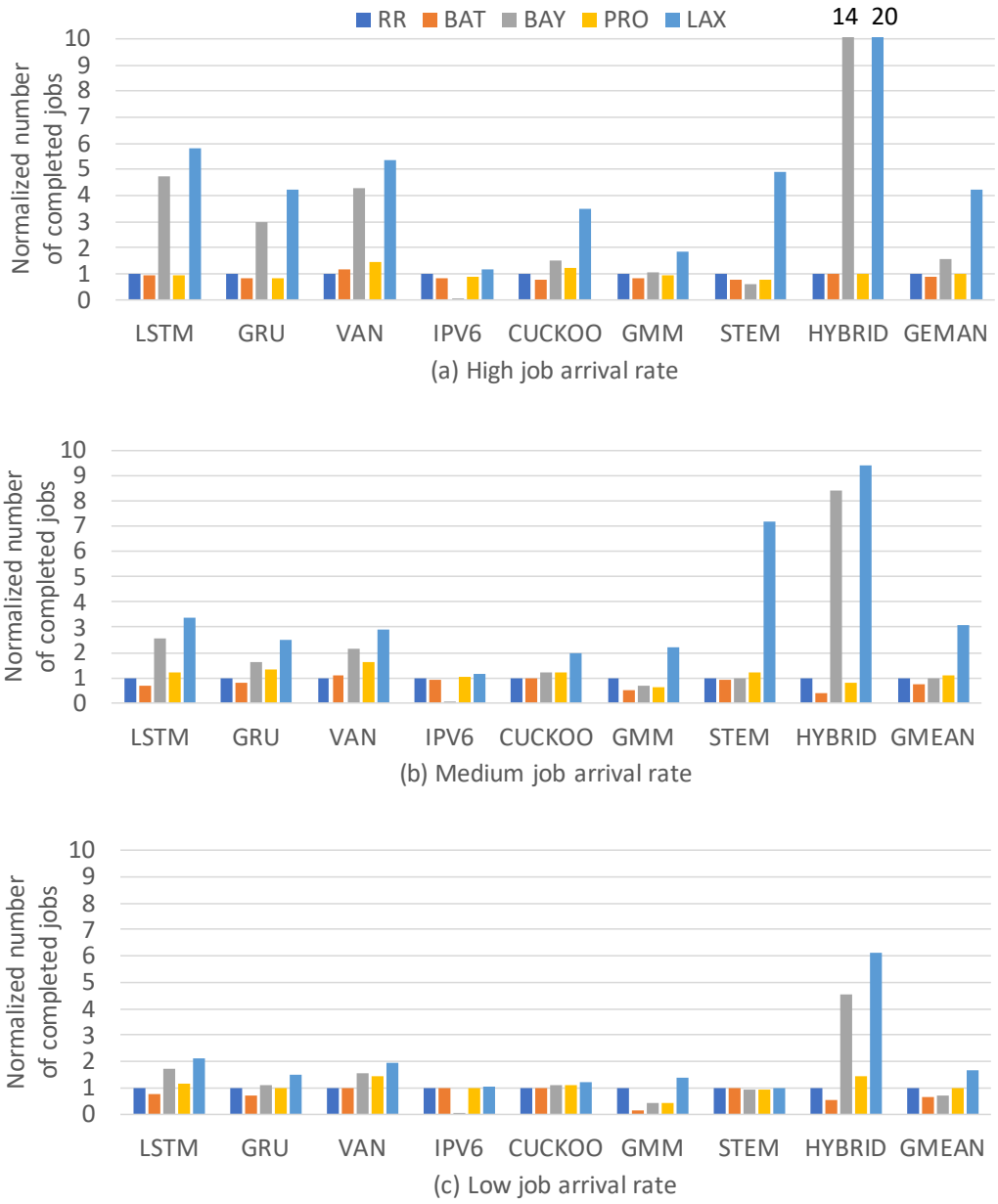

Fig. 6.2.: Jobs completed by their deadlines for CPU-side schedulers, RR, and LAX, normalized to RR

prevents it from completing any IPV6 jobs by their 40 microsecond deadlines, which significantly decreases BAY's overall performance such that RR and BAY complete the same number of jobs by deadline by geomean. Otherwise, BAY is the best CPU-

side scheduler for latency-sensitive workloads. Compared to LAX, the host-device and prediction overheads hamper BAY's ability to dynamically respond, especially at the high arrival rate for applications with multiple kernels, where LAX's accurate queuing delay estimate and faster responsiveness help it complete a geomean 2.7X more jobs than BAY by their deadlines.

PRO: PRO leverages offline profiling to infer the QoS of kernels, which reduces prediction overhead compared to BAY. However, since PRO focuses on co-scheduling memory- and compute-intensive workloads, it suffers with the purely latency-sensitive workloads we are studying. As a result, it only completes a geomean of 1.02X more jobs by their deadlines than RR. As contention increases, PRO especially suffers for LSTM, GRU, and GMM, where the increased contention exacerbates its focus on co-scheduling.

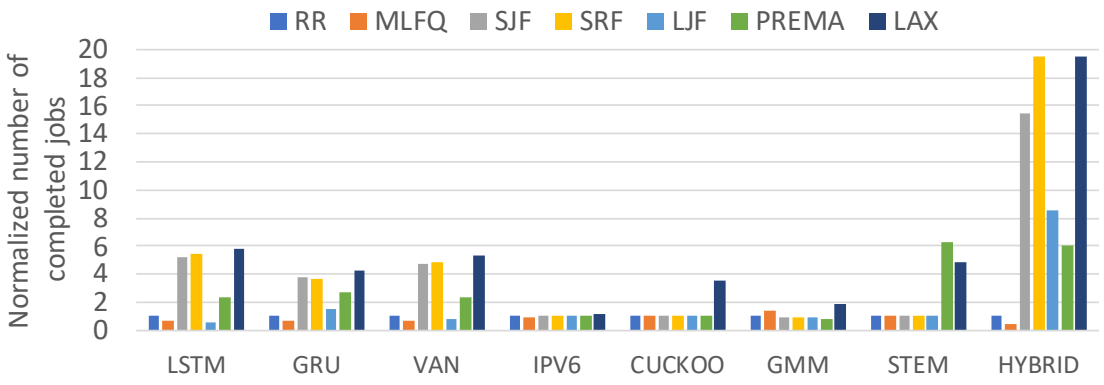

Fig. 6.3.: Jobs completed by their deadlines at the high job arrival rate, for schedulers that extend the CP, normalized to RR

LAX: LAX completes a geomean of 1.7X, 3.1X, and 4.2X more jobs by their deadlines compared to RR, respectively, for the low, medium, and high arrival rates. Unlike other schedulers, LAX utilizes the laxity of jobs, which increases the number of medium and large size jobs it can complete by their deadlines, especially as contention increases. Additionally, extending the CP helps LAX adjust more quickly and accurately to dynamically changing conditions. Finally, LAX's accurate queuing de-

lay model helps it avoid oversubscription. Thus, the combination of accurate queuing delay modeling, rapid, accurate responsiveness, and laxity allow LAX to significantly outperform the CPU-side schedulers.

Overall, LAX significantly outperforms state-of-the-art CPU-side schedulers for both many- and few-kernel workloads. Although some of these schedulers also model job runtime or utilize QoS calculations to avoid oversubscription, LAX's combination of laxity, rapid responsiveness, and accurate queuing delay modeling help it successfully offload more jobs, especially those with deadlines $<$ 1ms. Since the high arrival rate magnifies the differences between the schedulers, we focus on it due to space constraints.

Extending the Command Processor Schedulers

Figure 6.3 compares the number of jobs completed by their deadlines for each scheduler that extends the CP and utilizes hardware information. Overall, we find that other CP schedulers obtain some of benefits of LAX, but without accurate queuing delay estimations and laxity estimates, they are unable to complete some jobs that LAX can.

SJF and SRF: SJF and SRF greedily schedule kernels from the shortest jobs (e.g., RNN jobs with the shortest sequence lengths). As a result, SJF and SRF complete 2.46X and 2.54X more jobs by geomean, respectively, over RR at the highest job arrival rate. However, SJF and SRF do not work very well for the single-kernel benchmarks because all jobs have the same input size. This causes SJF and SRF to default to scheduling jobs in First-Come-First-Serve (FCFS) order, so queuing delay dominates the SJF and SRF's response time for these applications. Nevertheless, exploiting runtime information is still useful, as it allows SJF and SRF to complete more jobs than any other schedulers beside LAX. Moreover, compared to the CPU-side schedulers, extending the CP al-lows SJF and SRF's to improve performance over BAY, the best CPU-side scheduler, by 1.6X at the highest arrival rate.

MLFQ: In theory, MLFQ multiple priority levels should help it perform well. However, MLFQ often performs poorly – by geomean only 0.85X jobs complete by their deadlines compared to RR. For the RNNs and the networking processing applications, MLFQ completes relatively few jobs because once long-running jobs get promoted back to the higher priority queue, they take up high priority resources even after their deadline has passed [67]. However, for GMM and STEM, deprioritizing jobs long running jobs (e.g., from queuing delay), allows newer jobs can be scheduled sooner.

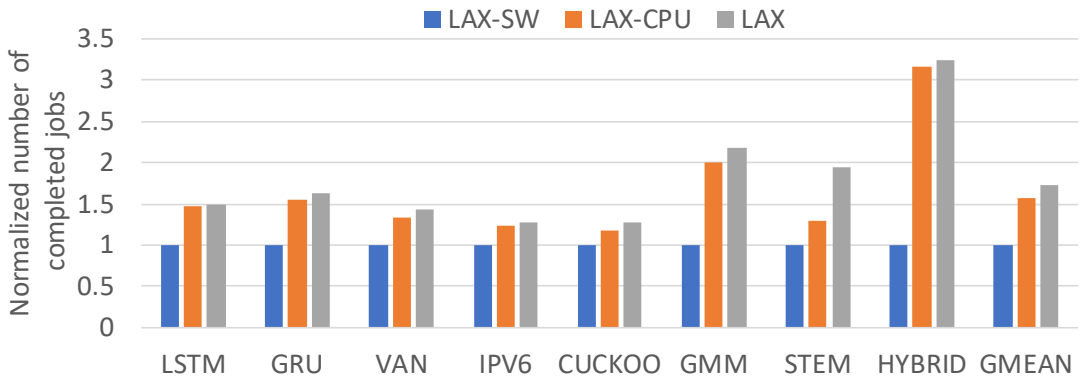

Fig. 6.4.: Jobs completed by their deadlines over different laxity-aware implementations, normalized to LAX-SW

LJF: Compared to RR, LJF completes 1.24X more jobs by their deadlines because it reorders jobs and schedules the longest jobs (e.g., RNN jobs with long sequence lengths) first. Although this allows some longer jobs to complete by the deadline, in general LJF does not perform well because it sacrifices the smaller jobs to complete these longer ones (for jobs like the RNNs with different sized jobs). LJF works the same as SJF and SRF for benchmarks composed of a single kernel and has a consistent performance number.

PREMA: PREMA's user-defined priorities and slowdown calculations help it complete geomean 2.2X more jobs than RR. PREMA performs particularly well for

STEM, which completes around 250 us. However, overall LAX completes geomean 1.9X more jobs than PREMA because LAX makes finer-grained decisions and does not need to preempt.

Overall, extending the CP can significantly improve the number of jobs that meet their real-time deadlines versus CPU-side schedulers, especially for CP schedulers that are able to predict the remaining runtime. However, these advantages alone are insufficient: LAX completes a geomean of 1.7X more jobs by their deadlines than SJF and SRF (the next best CP schedulers) because it also utilizes laxity and an accurate queuing delay model to better schedule the jobs.

LAX Scheduler Variants

LAX significantly outperformed both CPU-side schedulers and schedulers that extend the GPU's CP. However, since LAX utilizes multiple components , we also evaluate the three LAX variants to identify if laxity-aware scheduling could provide the same benefits without extending the CP. Figure 6.4 compares the number of jobs completed by their deadlines for these three laxity-aware schedulers.

LAX-SW shows how well laxity-aware scheduling can perform as a CPU-side scheduler, like BAT, BAY, and PRO. Although LAX-SW suffers from the same host-device overheads and is neither able to obtain nor rapidly respond to information about the GPU's current conditions as quickly as the CP schedulers, it still performs well. BAY, the best CPU-side scheduler (not shown in Figure6.4), outperforms LAX-SW for benchmarks with deadlines $>$ 1 ms (GMM and the RNNs) by a geomean of 27%. However, for the benchmarks with tight deadlines $<$ 1 ms (IPV6, CUCKOO, and STEM), LAX-SW completes many more jobs by their deadline due to its more accurate queuing delay model. Overall, LAX-SW completes a geomean of 1.8X more jobs by their deadlines than BAY. Thus, laxity-aware scheduling can improve on the state-of-the-art, even without hardware support.

However, LAX-CPU and LAX complete a geomean of 1.6X and 1.7X more jobs by their deadlines, respectively, than LAX-SW. This shows that CPU-side scheduling alone is not sufficient to fully exploit the benefits of laxity-aware scheduling, especially when contention is high. Interestingly, LAX-CPU, which only requires changing the API to allow user-level software to dynamically vary the priority of jobs, provides the majority of the benefits of LAX, which requires extending the CP. Overall, LAX completes a geomean of 1.1X more jobs than LAX-CPU, because it can respond more rapidly than LAX-CPU and has access to higher fidelity information about the system. Thus, to obtain all the benefits of laxity-aware scheduling, extending the CP is necessary. However, changing the API can provide most of LAX's benefits. Nevertheless, since LAX provides the best performance, we focus on LAX in the remaining results.

6.6.2 Scheduling Effectiveness

To measure how efficiently the schedulers utilized GPU resources, Figure 6.5 plots the percentage of the WGs completed that are part of jobs that meet the deadline. This metric shows how effective the schedulers were at identifying and performing useful work in general. Unsurprisingly, the deadline-blind schedulers (RR, BAT) waste a geomean of 67% - 71% of their resources on jobs that will not make the deadline. BAY utilizes its QoS prediction model to reduce contention and waste fewer compute resources (27% geomean). Finally, PRO wastes geomean 65% of its effort on jobs that cannot make their deadlines. In particular, PRO struggles with the RNNs. We believe this is because PRO has conservative QoS estimates that do not consider over-lapping many kernels.

Since SJF and SRF always issue small jobs first, they waste less work than deadline-blind schedulers (only 41% and 38%, respectively). Intuitively, since LJF always schedules large jobs first, which are less likely to be completed, LJF wastes more work (56% at the highest job arrival rate). In comparison, LAX's queuing delay

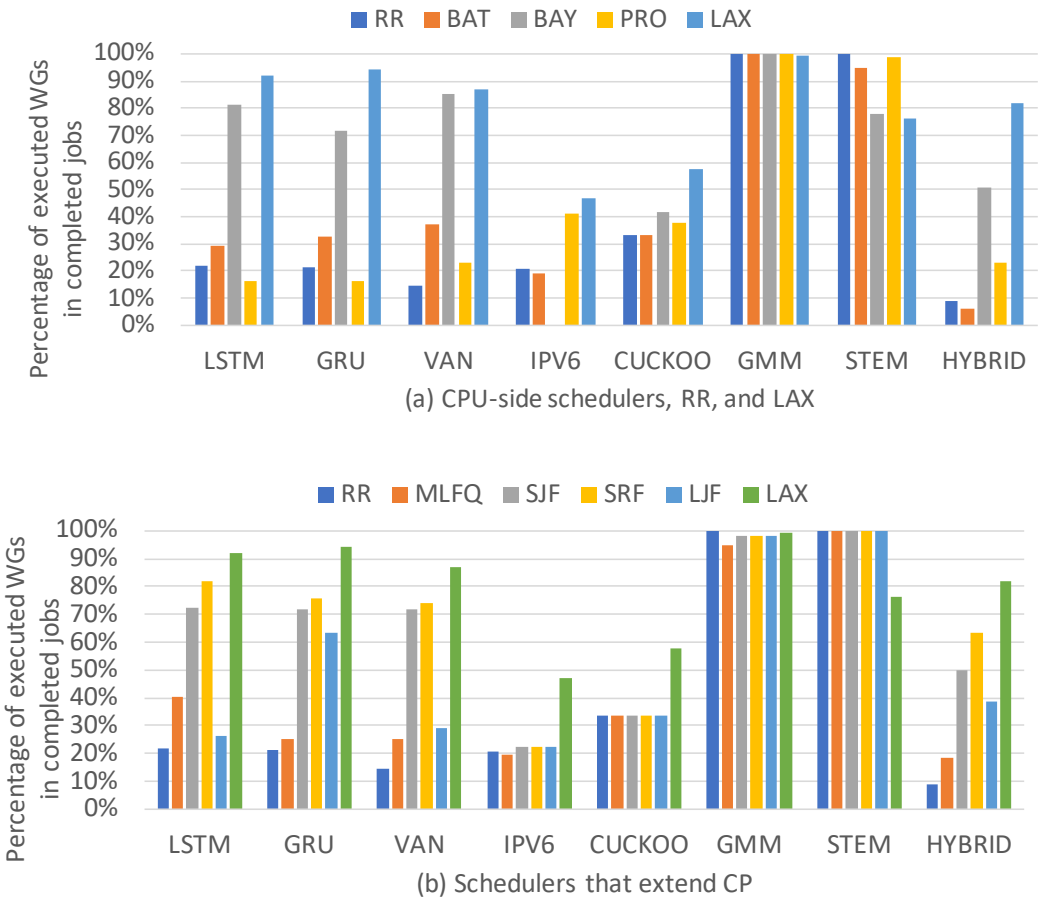

Fig. 6.5.: Percentage of completed WGs from jobs that meet their dead-lines at the high job arrival rate.

model helps it waste the least work of all schedulers – a geomean of 23% of compute resources. Overall, LAX again outperforms the other schedulers, and makes better decisions about which jobs to work on.

6.6.3 Execution Time Prediction and Priority Over Time

To examine how well LAX's execution time predictions track over time, Figure 6.6 plots the predicted execution time and the priority of a sample job for the LSTM RNN. (i.e., time the job is in the running state, where its work groups were actively

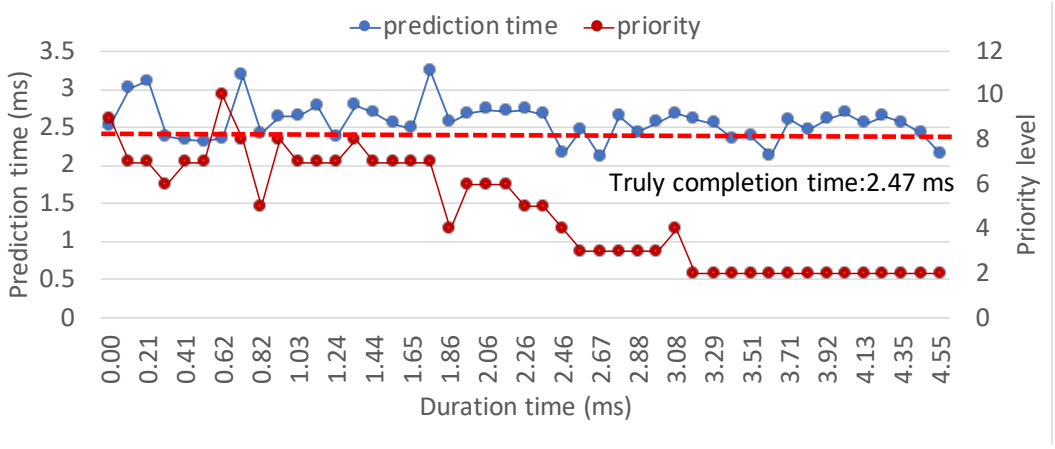

Fig. 6.6.: LAX's Job Time and Priority Prediction in LSTM. P0 is the highest priority.

being executed). The x-axis indicates the duration time of the job and the endpoint of the x-axis shows the job's actual completion time (i.e., time in ready and running). Initially, the LSTM job's priority stays relatively steady until its laxity starts to decrease. This shows that LAX correctly deprioritizes this job while it has plenty of laxity and prioritizes it once its laxity is small. Additionally, LAX's execution time prediction tracks very closely to its actual time in the running state. Overall, LAX effectively varies the dynamic priority and tracks the laxity effectively.

Table 6.4.: Energy rate (consumed energy over the number of successful jobs) (mJ))

	RR	MLFQ	BAT	BAY	PRO	LJF	SJF	SRF	PREMA	LAX
LSTM	5.52	7.36	6.03	0.37	0.30	9.47	1.05	1.02	2.37	0.36
GRU	2.47	3.30	2.91	0.30	0.24	1.60	0.66	0.67	0.92	0.29
VAN	1.81	2.42	2.27	0.18	0.20	3.30	0.55	0.53	1.10	0.21
IPV6	0.21	0.23	0.21	0.00	0.22	0.21	0.21	0.21	0.21	0.13
CUCKOO	5.97	5.97	7.96	0.35	0.34	5.97	5.97	5.97	5.97	0.97
GMM	1.49	1.02	1.77	0.12	0.13	1.62	1.62	1.62	1.76	0.14
STEM	1.36	1.36	1.82	0.09	0.10	0.91	0.91	0.92	0.22	0.10
HYBRID	38.63	77.30	38.63	0.63	0.82	4.53	2.48	1.98	6.44	0.45

6.6.4 Energy Consumption

Table 6.4 compares the schedulers in terms of their normalized energy consumption per successful job. In general, LAX provides comparable or better energy consumption relative to most CPU-side schemes (0.85X – 12.1X geomean less energy) and schedulers that extend the CP (4.1X – 12.3X geomean less energy). LAX outperforms all schedulers in this regard except for BAY and PRO (15% and 5% less energy per job than LAX, respectively). However, BAY and PRO are overly conservative and do not accept larger jobs that consume more energy, whereas LAX completes many more jobs that are both small and large.

Table 6.5.: The Successful Job Throughput (the number of successful jobs per second)

	RR	MLFQ	BAT	BAY	PRO	LJF	SJF	SRF	PREMA	LAX
LSTM	511	419	458	2651	465	372	2883	3069	1302	3348
GRU	912	700	775	2828	775	1551	3466	3558	2463	3877
VAN	729	515	750	2574	987	472	2832	2960	1416	3346
IPV6	13158	13816	11842	0	13816	13158	13158	13158	12500	20120
CUCKOO	289	276	651	295	289	289	289	289	289	868
GMM	2242	2841	2242	2446	2242	2242	2242	2242	1921	4484
STEM	3937	3937	2624	1969	2624	3937	3937	3937	23622	18207
HYBRID	85	43	85	1147	85	766	1277	1702	511	1702

6.6.5 Throughput and 99-percentile Tail Latency

Table 6.5 and Table 6.6 also shows the scheduler's throughput and 99-percentile tail latency. Overall, LAX provides a better blend of throughput and tail latency. LAX has better or comparable tail latency than all CPU-side schemes (0.8X-6.8X geomean faster) and has geomean 1.7X-4.9X better throughput. Moreover, LAX's throughput is 1.3X-5.4X better than the CP schedulers and has 5.3X–6.8X better tail latency. BAY and PRO are the most competitive schedulers in terms of throughput and tail latency – their queuing models help them avoid offloading jobs that are

Table 6.6.: 99-percentile job latency(millisecond))

	RR	MLFQ	BAT	BAY	PRO	LJF	SJF	SRF	PREMA	LAX
LSTM	47.7	38.2	51.9	21.4	6.7	50.1	46.4	46.3	43.2	5.6
GRU	35.1	25.6	37.9	20.4	6.5	36.9	33.7	33.4	27.6	5.7
VAN	43.9	34.2	38.7	9.4	7.0	47.0	43.6	42.9	38.7	7.0
IPV6	0.2	0.2	0.2	0	0.4	0.2	0.2	0.2	0.2	0.1
CUCKOO	9.7	9.0	9.2	1.0	1.3	9.2	9.2	9.2	9.4	4.4
GMM	41.5	42.3	42.2	3.3	1.8	42.2	42.2	42.2	40.2	2.7
STEM	3.1	3.1	3.2	0.3	0.3	3.1	3.1	3.1	4.8	0.5
HYBRID	84.5	75.7	88.4	20.9	2.4	85.7	81.9	83.9	83.7	6.9

unlikely to be completed by their deadlines. However, PRO and BAY complete far fewer jobs by their deadlines than LAX.

6.7 Summary

Although GPUs have traditionally been used for through-put-oriented workloads, latency-sensitive workloads are of increasing importance. Traditional solutions such as batching are less effective for these workloads, especially once realistic arrival times are considered, since batching actively harms their performance by forcing jobs to wait for additional work to arrive. GPU streams help address this issue and permit multiple streams to execute concurrently. However, these streams only have static information on relative priority making it a challenge to meet real-time deadlines.

To address this inefficiency, we propose a new GPU queue scheduler, LAX, that significantly improves the throughput and latency of latency-sensitive GPU applications. By tracking the WG completion rates and monitoring the queuing delay, LAX accurately estimates the overall execution of individual latency-sensitive jobs. Our results show that LAX completes a geomean of 1.7X-5.0X more jobs by their deadlines compared to eight GPU queue schedulers.

7. DARSIE:DIMENSIONALITY-AWARE REDUNDANT SIMT INSTRUCTION ELIMINATION

DARSIE [158] ensures that TB-redundant instructions are fetched and executed only once in each TB. As all instructions are 64-bits in length, redundant ones can be skipped in the frontend of the pipeline by simply adding eight to the program counter. However, detecting and marking redundancy, sharing values of skipped instructions, properly handling redundant load instructions, and handling branches with the potential for divergence requires more care. Therefore, we start by presenting DARSIE's operation at a high level in Section 7.1. Here, we introduce our extensions to the compiler that marks instructions as either redundant, conditionally redundant, or non-redundant. This is explained in full detail in Section 7.2. We next describe our changes to the microarchitecture, explained fully in Section 7.3. We additionally describe how DARSIE handles the skipping of load instructions, and its operation in the presence of divergence. These are explained fully in Sections 7.4 and 7.5 respectively.

7.1 High Level Operation

DARSIE operates by first declaring PCs as *skippable*. Using a novel compiler pass, we detect when values are conditionally-redundant across a TB. At kernel-launch time, every static PC in the program is marked at as either TB-redundant or not. The hardware itself treats uniform, affine redundant, and unstructured redundant instructions the same. Section 7.2 details these compiler markings. The compiler analysis also assumes that warps in a TB are executed in lock-step, such that there is only one version of each TB-redundant instruction at any given time. There are two options in hardware to ensure that warps read the correct version of a TB-redundant register without enforcing costly lock-step execution on every instruction: (1) Syn-

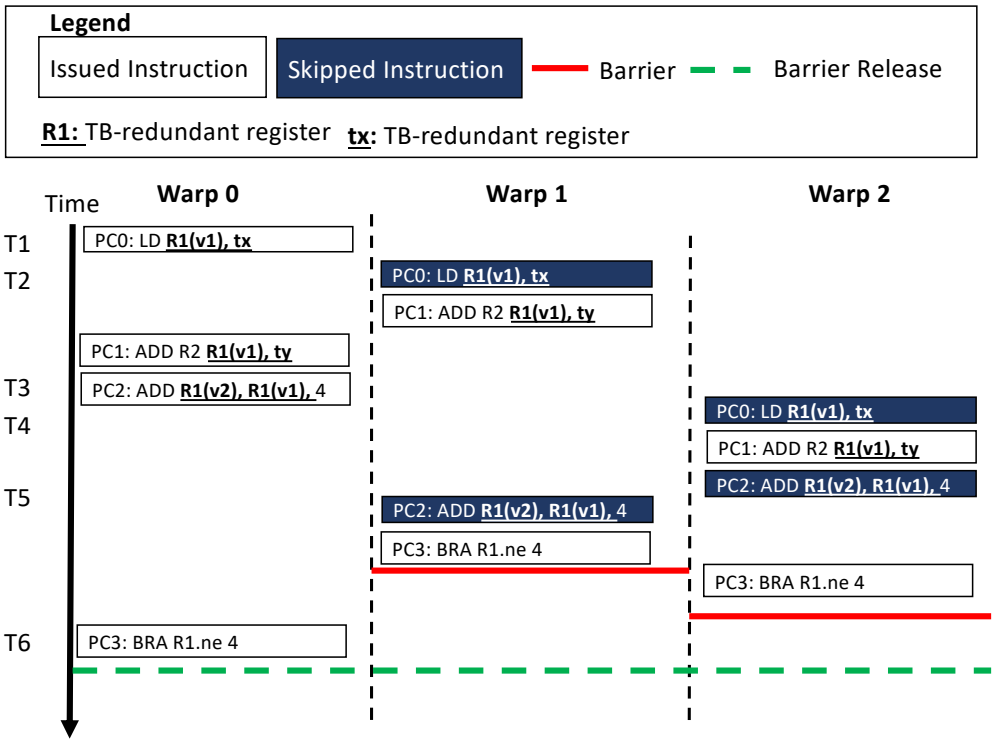

Fig. 7.1.: DARSIE's Instruction Skipping Flow: Branch instructions always force a TB-wide barrier to determine what the majority-path is. In this example, TBs are three warps wide.

chronize the warps in the TB when any warp writes to a TB-redundant register. This ensures there is only ever one live version of a particular TB-redundant register. (2) Store multiple versions of each TB-redundant register such that each warp can progress at a different pace. As long as warps are executing on the same control-flow path (hence executing the same instructions in the same order), the correct register version for each warp can be attained by keeping track of writes to the register. When warps in the TB take different control flow paths, DARSIE continues operating on the path taken by the majority of warps. To avoid excessive synchronization, DARSIE adopts solution (2).

In hardware, DARSIE elects a single warp from a TB to execute a redundant instruction, and share the result with other warps in the TB. We call the warp that executes the instruction the *leader warp*. Warps that skip the instruction and read the value are referred to as *follower warps*. Subsequent instructions in follower warps that are dependent on the result of an eliminated instruction access the leader warp's values through a multithreaded register renaming mechanism. The existing programmable GPU register mapping mechanism helps facilitate this process, as each warp has a configurable number of physical registers based on the per-thread register demands of an application. Our updates to this mechanism are described in Section 7.3.1.

Figure 7.1 provides a visual overview of DARSIE's operation. In Figure 7.1, warp 0 becomes the leader for instruction PC0 at $T1$ because it arrives first. Warp 0 executes the instruction and stores the result as R1(v1), since this is the first write to R1. At time $T2$, warp 1 skips PC0, updating its register mapping table to point to R1(v1) which warp 0 just produced. Warp 1 then executes PC1, which is a true vector instruction (since ty is not TB-redundant). Warp 0 does the same. At time $T3$, warp 0 writes to R1 again. Each time a redundant register is written, we create a new version of the register tagged with the number of times it has been written by this TB. At this point, there are now 2 active versions of R1 across this TB. At $T4$, warp 2 finally reaches PC0, skips the instruction then executes PC1 using the old (v1) version of R1. Warp 2 uses R1(v1) for its source operand because there has only been 1 write to R1 in warp 2's instruction stream. Warp 2 then skips PC2, and increments the number of writes the warp has seen to R1. At $T5$, warp 1 skips PC2. Since all the warps in the TB are done with R1(v1), it can be released. Warp 1 executes the branch instruction, then waits for all other warps in the TB to reach this branch. Synchronization at branches is necessary to ensure that all warps skipping instructions in a TB execute on the same control-flow path. Warps that diverge off the majority control-flow path are no longer considered for skipping. Similarly, warps with intra-warp control-flow divergence do not participate in instruction skipping. Note that, except at branch instructions, the warp scheduling order in DARSIE is

not prescribed. The scheduler will still make throughput-oriented decisions. The ordering in this example is therefore one of many that are possible.

7.2 Compiler Annotations

DARSIE's static compilation phase detects both definitely and conditionally redundant registers and instructions. In relation to our taxonomy described in Section 4, uniform redundant values are always definitely redundant. Affine and unstructured redundant values are conditionally redundant. The compiler starts by identifying all the intrinsic values known to be uniform across a TB. The values we consider in this work are: blockIdx, blockDim, scalar constants, global kernel input parameters and the base value of shared memory. These values are all marked as definitely redundant. Next, the compiler marks intrinsic registers that are conditionally redundant. Based on our observations in Section 4, both the threadIdx.x and threadIdx.y registers are conditionally redundant and depend on the TB dimensions known at runtime. All studied applications use 2D TBs at most, as is typical for GPU workloads. We therefore limit the analysis to only threadIdx.x. All other registers are considered true vector registers. The compiler then creates the program-dependence graph and iteratively propagates our redundancy information through registers and instructions. Unlike previous works that focus exclusively on finding affine and uniform instructions [44, 105, 109], we introduce and exploit *conditional* redundancy as well. The conditionally redundant instructions comprise a significant portion of total executed instructions for applications with 2D TBs. Load instructions that access redundant or conditionally redundant addresses (and their corresponding output registers) are also marked. If more than one of our three redundancy definitions (redundant, conditionally redundant or vector) reaches a source operand of an instruction, we assign the weakest of the definitions.

This analysis assumes that warps within the TB proceed through the program in lock-step. Since enforcing this requirement can be expensive (or impossible if warps

```
// Loading Parameters and thread ID
       PC
DR    0x000      cvt.u32.u16 $r2, %ctaid.y;
...
CR    0x068      cvt.u32.u16 $r4, $r0.lo // $r0.lo: tid.x
...
CR    0x0f0      add.u32 $10, $r4, 0x0000040c
...
CR    0x108      shl.b32 $ofs3, $r10, 0x00000007
...
// Start of loop
V     0x150      l0x00000150: ld.global.u32 $11, [$r1]

// Unrolled Loop
CR    0x178      mov.u32 $r0, s[$ofs3+0x0000];
CR    0x180      add.u32 $ofs4, $ofs3, 0x00000080;
V     0x188      mad.f32 $r10, s[$ofs2+0x0000], $r0, $r10;
CR    0x190      mov.u32 $r0, s[$ofs4+0x0000];
CR    0x198      add.u32 $ofs4, $ofs3, 0x00000100;
V     0x200      mad.f32 $r10, s[$ofs2+0x0004], $r0, $r10;

...

V     0x480      set.le.s32.s32 $p0/$o127, $r8, $r9;
V     0x488      add.u32 $r1, $r1, 0x00000080;
V     0x490      add.u32 $1, $1, 0x0000080;
V     0x498      add.u32 $r5, $r6, $r5
V     0x500      @$p0.ne bra 10x00000150;

// End of loop
...
```

Fig. 7.2.: Example of compiler marking TB-redundant instructions for matrix multiply kernel. DR:Definitely Redundant, CR:Conditionally Redundant

traverse different control-flow paths), we rely on hardware to create the illusion that warps leveraging DARSIE are proceeding in lockstep (described in Section 7.3.3) with respect to TB-redundant operations.

Promoting conditionally redundant registers to definitely redundant requires run-time information about a TB's dimensions. For example, if the there are 32 threads in the x-dimension, then the tid.x value per warp will vary from [0 to 31]. Likewise, if there are 16 threads in the x-dimension, each warp will have tid.x IDs [0 to 16].

In both cases, values based on tid.x will be redundant across the TB, and repeat for every warp. Conditionally redundant instructions are evaluated at kernel launch time based on the kernel's specified TB size, and are static for the duration of the kernel. This marking could either be implemented in the GPU driver's JIT-ing finalization pass, or determined with a minor hardware modification that compares conditionally redundant instructions to the launched TB size. Supporting CUDA dynamic parallelism, where the GPU launches kernels to itself is possible with latter option, as the code does not need to be recompiled. In either case, the check simply tests if the kernel has 2D TBs, and that the width of the x-dimension is a power of 2, and less than or equal to the warp size. If so, conditionally redundant instructions are marked as definitely redundant, or are otherwise marked as true vector instructions. We note that the majority of multi-dimensional GPU applications meet the above x-dimension criteria. Of the 128 unique 2D kernels from the application surveyed in Section 1.3.4, only one fails to meet this requirement.

Figure 7.2 illustrates the compiler annotations made for the matrix multiply kernel. Note that this code is register-allocated PTXPlus code, which is used for all our experiments. As shown in Figure 7.2, the value *threadIdx.x* propagates to the register *$ofs3*. Thus, the unrolled loop in the program contains 2 redundant instructions and one true vector operation. This highlights the granularity at which redundancy elimination takes place using DARSIE.

The compiler can only mark static instructions as being skippable based on an analysis that assumes that threads in a TB are executing in lock-step. It is up to the hardware fetch scheduler to ensure that all warps in the TB skip the same version of the redundant instruction. For example, if a redundant instruction is in a loop, all threads skipping the instruction must be on the same iteration of the loop. This ensures that dependencies between loop iterations are maintained for each warp. Ordering is enforced using either register-versioning or forcing the hardware to barrier when TB-redundant registers are written.

We note that the compiler changes in DARSIE do not change the instruction stream in any way, other than adding hints about redundancy. This is dissimilar to techniques like DAC [109] that completely transform the compiled code into a format requiring hardware support for affine and non-affine instruction streams. If the hardware does not support DARSIE, the markings are simply ignored. Likewise, the hardware does not require the compiler to support DARSIE. Binaries compiled without DARSIE markings will run seamlessly on DARSIE hardware, but without instruction skipping.

We encode the three-state <vector, conditionally redundant, redundant> classification in two bits of the GPU's virtual ISA (PTX in NVIDIA) that is produced by the static compiler. GPUs employ a two-step compilation process where the virtual ISA in the binary is transformed into the real machine instruction set (known as SASS) when the kernel is launched. Although the encoding of SASS is proprietary, reverse engineering efforts indicate that there are many unused bits in this 64-bit RISC-like ISA [159]. We use one of these extra bits to encode if an instruction is TB-redundant or not, which is known when the SASS is loaded into the GPU. Two bits would be required to maintain the three states of redundancy if the decision is delayed until after the code is JIT compiled.

7.3 DARSIE microarchitecture

Figure 7.3 shows the changes to the microarchitecture needed to support DARSIE. We add the instruction skipping hardware in the fetch stage, consisting of a PC coalescer (Ⓐ in Figure 7.3), a PC Skip Table (Ⓑ) and fixed-size adders to allow each PC to be incremented by 8 (Ⓒ).

7.3.1 Remapping Registers

To enable dynamic remapping of vector registers, we add a register renaming table that is probed prior to looking up the register in the baseline's linear register

renaming table (Ⓓ). This allows follower warps to read register values produced by leader warps. Registers in contemporary GPUs are not mapped on a per-thread basis, but rather on a per-warp basis. There is a 1:1 correspondence between vector lanes and threads within the warp, and cross-lane communication is generally not supported outside of special instructions. DARSIE does not change this assumption and remaps whole vector registers. Our register renaming table contains one entry for every currently renamed register in each warp. The register rename table, version table, and physical register freelist implement the versioning detailed in Figure 7.1.

The register rename table maps <warp,reg#> pairs to this warp's <reg#,version#> pair. A separate version table (Ⓔ) stores the <reg#,version#> to physical register mapping. Both the version and rename table are banked on a per-TB basis. When a kernel with TB redundancy is launched, we allocate a portion of the physical register file space for renaming. Many GPU applications are not limited by the register file size, so this allocation will not typically affect occupancy. How much register space to consume could be made on an application basis. In this work, we allow DARSIE to consume up to 32 vector registers per TB for renaming. DARSIE uses as many registers as it can before affecting occupancy when registers are limited.

We allocate our renamed register space in a strided fashion across the vector RF banks at kernel launch and maintain a physical register freelist (Ⓕ). Physical registers are freed when a register version number is no longer in flight for the TB. When the freelist empties, synchronization must be performed to ensure all required versions of a register are available. Our evaluation accounts for the increase in register bank conflicts that occur when all follower warps attempt to read from the renamed register's space.

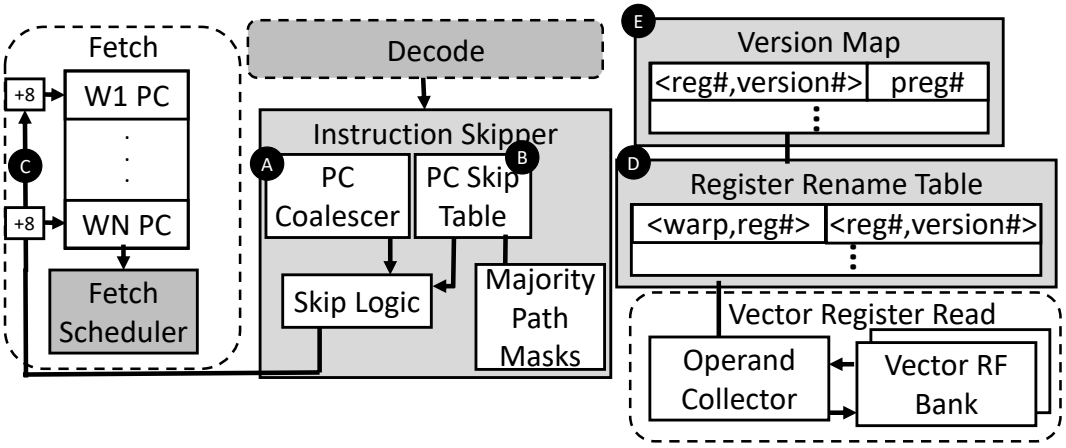

Fig. 7.3.: Detailed breakdown of DARSIE uarch operation.

7.3.2 PC Skip Table

DARSIE skips instructions in the front of the pipeline before the I-cache is probed. To achieve this, we add hardware that acts in parallel with the fetch scheduler, to skip some instructions while initiating a fetch for another. This effectively increases our throughput at fetch without increasing the width of any of existing structures like the fetch scheduler and I-cache. The instruction skipper relies on our compiler annotations to decide which instructions should be skipped, and the PC Skip table (Ⓑ) controls the skipping logic. Each entry in the Skip PC table contains five fields:

1. **PC** : The program counter that should be skipped
2. **Warps waiting bitmask** : A mask that indicates which warps are waiting at this PC to skip it. Required if synchronizing between warps.
3. **Majority-path bitmask**: A mask with 1-bit per warp in a TB that indicates which warps are executing on the majority-path.
4. **IsLoad**: A bit that is 1 if this instruction is a load instruction. This is necessary since load instructions must be removed from the Skip PC Table if a store is executed, or if global atomics/synchronization events occur.

5. **LeaderWB**: A bit that is 1 if the leader warp has written back the redundant-register value. To ensure correct operation, follower warps must wait for the leader to writeback before they can leave the skippable instruction.

7.3.3 Achieving the Illusion of Lockstep Execution

Our skipping mechanism is dependent on all warps having the same branch history as the leader warp (i.e., all warps following the same control-flow path). To ensure this condition, we synchronize TBs at branch instructions. The path with the majority of warps will continue skipping. Warps on any other path will not. We store 1-bit per warp to indicate if it is on the TB-majority path. When warps deviate from the path, their bit is cleared. These bits are all set back to one upon the execution of *syncthreads* instructions which require the entire TB to be in sync.

7.3.4 PC Coalescer

A PC coalescer is used to minimize the skip table read port requirements (Ⓐ). The PC coalescer acts like the global memory coalescer in the load/store unit, except instead of coalescing global memory addresses to cache lines, it coalesces PCs based on exact matches. This helps limit the number of accesses made to the Skip PC Table each cycle. The PC skip table contains one entry for each PC currently being skipped. We experimentally determine that the PC coalescer reduces the port requirement on the PC skip table to 2, while providing both reasonable throughput and minimal area and energy overheads.

7.3.5 Instruction Skipping Flow

After TB-redundant instructions are decoded, the PC skip table is probed to see if they are currently being skipped. If there is no PC skip table entry and the accessing warp is on the majority-path, it becomes the leader warp. Upon the creation of a

new leader warp, an entry is created in the PC skip table, a new physical register is taken from the freelist and allocated in the version table, the LeaderWB bit is cleared, and the leader updates its register's version number in the register renaming table. If there are no other entries in the freelist, the warps waiting bitmask is updated to indicate that this instruction will act as a synchronization point.

Only warps on the majority control-flow path can skip instructions. When the leader writes back a TB-redundant value, it updates the LeaderWB bit.

When another warp in the TB gets to the PC being skipped, the PC skip table is probed, and an entry is found. If the warps waiting bitmask is empty, and the leaderWB bit is set, this follower warp is able to skip the instruction. If the warps waiting bitmask has a non-zero value, it is updated to indicate that the new warp is now waiting for all other warps in the TB to reach the TB-skippable instruction. The follower warp then updates its version number in the register renaming table to reflect the fact that it needs to read a newer version of this register. If this was the last warp in the TB using a particular register version number, the physical register is returned to the freelist. The PC of the skipping warp is then incremented by 8 (Ⓒ). If synchronization is necessary, the warp is removed from the fetch scheduler.

As more warps from the same TB reach the PC to be skipped, their PCs are incremented, and their registers' versions updated. If synchronization is necessary, we determine if all the warps on the majority control-flow path have arrived at the instruction to be skipped by matching the warps waiting bitmask with the majority-path bitmask for this TB. Once all follower warps in the TB have skipped the instruction, it is removed from the PC Skip Table.

When warps leave the majority path, they copy their redundant register values into their warp-private space, and clear their state in the register renaming and version table. We also note that the execution of warps is different than execution of individual scalar threads, in that a warp may proceed in both branch directions using the SIMT stack. Our technique is not applied in the presence of SIMD divergence

(see Section 7.5). If SIMD divergence is encountered on a branch, the warp is no longer considered for skipping and is removed from the majority path.

7.4 Skipping Load Instructions

Skipping load instructions presents a unique challenge since memory dependence information is not embedded in the decoded instruction. Without complex and potentially expensive memory dependence tracking hardware, we cannot guarantee that a store instruction does not update memory at the location a skipping load instruction reads. To simplify the design, complexity and size of our proposed hardware, DARSIE avoids the memory dependence problem by removing load PCs from the skip table when one of two events happens: **(1) This TB executes any store instruction**: Stores are relatively infrequent, so we conservatively assume that any store can update memory referenced by any load instruction to be skipped. **(2) Any global communication primitives are executed**: Our baseline GPU does not guarantee any particular memory ordering between TBs executing on different SMs, or TBs on the same SM, unless global communication primitives are used. When we detect that an SM executes any instruction used to perform global communication, such as global atomic instructions, we remove all global load PCs from the skip table. In our benchmarks, and the in the bulk of contemporary GPU workloads, these global communication primatives are not used.

To disable skipping load instruction is a simple solution when DARSIE can statically identify any global communication in the program. A dynamic mechanism could take advantage of the fact. For instrance, there are no inter-threadblock communication guarantees unless the local core uses special instructions that imply global communication in CUDA [160]. Supporting the current GPU memory model with DARSIE only requires us to examine the local core's instructions and would not introduce any inter-core mechanisms. Note that the vast majority of GPU workloads (including the ones we evaluate) do not use inter-threadblock communication.

7.5 Handling SIMD Divergence

DARSIE specifically targets highly regular code that does not exhibit large levels of SIMD divergence. This is the common case for GPU applications. We therefore simplify our design by not skipping instructions with inactive threads in its active mask. While we note that divergent workloads exist, prior work from industry has shown them to be a minority of contemporary applications run by GPU customers [**?**]. We evaluated the effects of allowing diverged instructions to be considered redundant, but found that it provided minimal returns. We note that warp-level control-flow divergence is different from SIMD divergence. Warp-level divergence indicates that the entire warp took a different execution path, and not just some threads within a warp. If warp-level divergence occurs, instruction skipping is still possible among warps that traverse the majority control-flow path.

7.6 Methodology

Table 7.1.: Applications studied

Name	**Abbr.**	**TB dim**	**Name**	**Abbr.**	**TB dim**
binomial-Option [67]	BIN	(256,1)	ImageDenois-ingNLM [67]	INLM	(16,16)
pathfinder [57]	PT	(1024,1)	Backprop [57]	BP	(16,16)
fastWalsh-Transform [67]	FW	(256,1)	DCT8x8 [67]	DCT	(8,8)
SRADV1 [57]	SR1	(512,1)	Floyd-Warshall [56]	FWS	(16,16)
LIB [59]	LIB	(256,1)	HotSpot [57]	HS	(16,16)
			CP [59]	CP	(16,8)
			convolution-Texture [67]	CONVTEX	(16,16)
			MatrixMul [67]	MM	(32,32)

C: CUDA SDK [67], P: Parboil benchmark [58], R: Rodinia benchmark suite [57], I: GPGPU-sim distribution benchmark [59]. P: Pannotia benchmark [56]

We use GPGPU-sim v4.0 [161] and GPUWattch [162] to estimate the performance and energy consumption of DARSIE respectively. We simulate our applications using PTXPlus, and extension of NVIDIA's virtual ISA PTX that is converted from the native machine ISA SASS. We use the NVIDIA GTX-1080Ti Pascal GPU as our baseline. Table 7.2 describes our baseline, and is verified to have a 90% correlation to the real card. We swept different warp schedulers and observed that these regular applications are insensitive to scheduler choice, with GTO being the best performing option. We implement DARSIE's compiler pass inside GPGPU-Sim on register-allocated PTXPlus code, similar to the methodology used by Wang et al. [109]. We use Cacti 7.0 [163] to model the energy and area overhead of DARSIE's additional hardware components. The PC skip table has 2 read ports, 1 write port and 1 entry per TB. Each SM also contains a register rename table, which has 32 entries per TB, based on the maximum number of registers renamed in our applications. We compare DARSIE as described in Section 7 with two previously proposed techniques:

Table 7.2.: Baseline GPU

Parameters	Values
GPU	Pascal (GTX1080Ti), 28 SMs, 64 warps/SM
	32 thread blocks/SM
SM	32 SIMD Width, 2K vector registers per SM
Scheduler	4 warp scheduler/SM, GTO scheduling
L1	96KB shared memory/SM
Register	14.2pJ/read 25.9pJ/write [162]

Uniform Vector (UV): UV [45] is a recently proposed technique to remove redundant inter-warp instructions. UV makes use of an instruction reuse buffer [100] to eliminate instructions that read uniform scalar register values. UV prevents instructions from executing at the issue stage of the pipeline after being loaded into the instruction buffer. It does not consider non-uniform redundant vectors, and does not skip memory operations. We choose this technique to compare against because it is

the only related work to remove redundancy without major pipeline modification. A more detailed description of UV is located in Section 2.5.1.

Idealized Decoupled Affine Computation (DAC): DAC-IDEAL [109] proposes a compiler and hardware mechanism that detects affine (not necessarily redundant) operations. The DAC compiler separates instructions into affine and non-affine streams, and synchronizes the two when the vector stream reads values from the affine stream. We model an idealized version of DAC by detecting affine instructions at runtime, and assuming that all affine instructions (both redundant and otherwise) will be executed only once. We also assume there is no synchronization cost between affine and non-affine instruction streams. This implementation was validated to be as good or better at instruction reduction compared to the original results in [109]. We choose DAC-IDEAL to compare against because it covers both uniform and affine redundancy, and is the most recently proposed technique.

7.7 Experimental Results

The following subsections evaluate the performance and energy-efficiency of DARSIE, the effects of synchronization and provide an area estimate.

7.7.1 Performance and Energy

Figure 7.4 compares the speedup of UV [45], DAC-IDEAL [109], and DARSIE over our baseline GPU. DARSIE achieves a geometric mean speedup of 1.3, significantly better than UV (1.02) and DAC-IDEAL (1.11) for benchmarks with 2D TBs. DARSIE-IGNORE-STORE doesn't reset the skip table in the occurrence of store instructions. DARSIE significantly outperforms the two alternatives because of the elimination of the unstructured redundancy in 2D benchmarks. As mentioned in Section 4, neither UV nor DAC-IDEAL remove unstructured redundancy. UV is typically limited by fetch throughput since it can only remove uniform redundancy at the issue stage. DARSIE has significantly higher instruction skipping bandwidth because it can

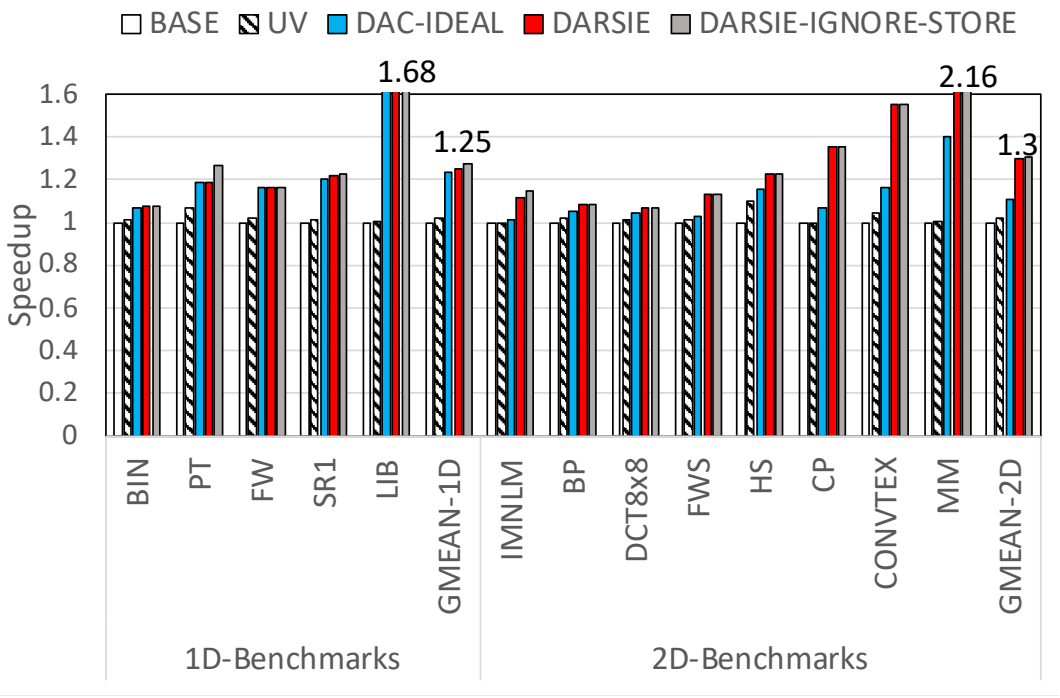

Fig. 7.4.: Performance of DARSIE against prior work. Speedup is normalized to the baseline GPU.

skip multiple instructions in a single fetch cycle with only an increment of the PC. DAC-IDEAL's performance with 1D TB applications is roughly equal to DARSIE's since it is similarly able to remove all uniform and affine-redundant instructions. To evaluate the effect store instructions have on performance, DARSIE-IGNORE-STORE doesn't reset the skip table when store instructions occur and demonstrates that the performance impact is minimal. Further investigation reveals that stores usually occur at the end of the register-use chain. Therefore; the value in the register is typically not used again after the store, so clearing it's redundancy data has little effect on DARSIE's performance. Since DARSIE remaps follower warps to the same register bank, it does cause additional register file bank conflicts. However, we find that artificially removing all DARSIE-induced bank conflicts results in just a 1% performance improvement.

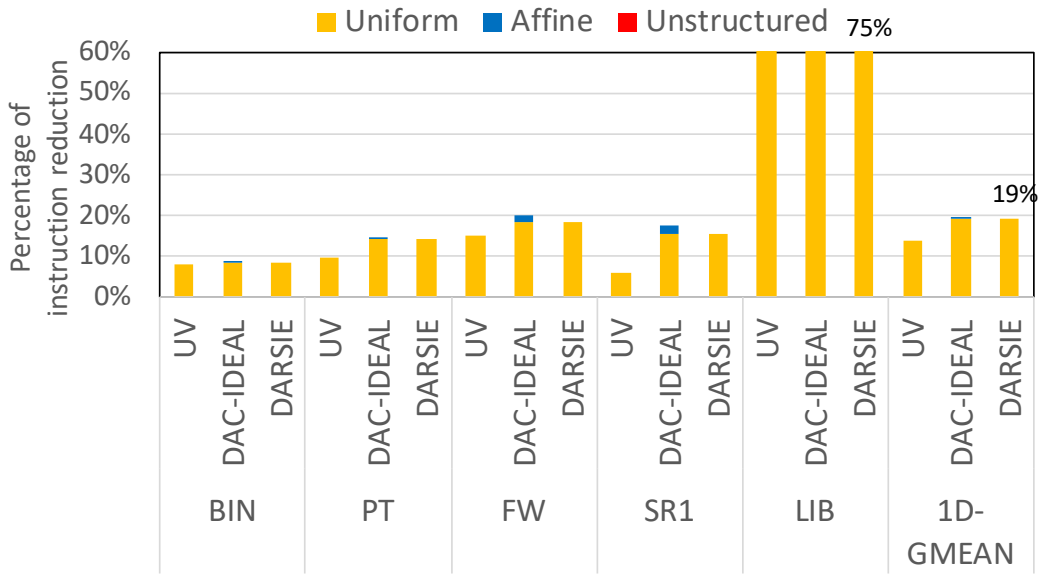

Fig. 7.5.: Percent reduction in 1D benchmark instructions versus the baseline

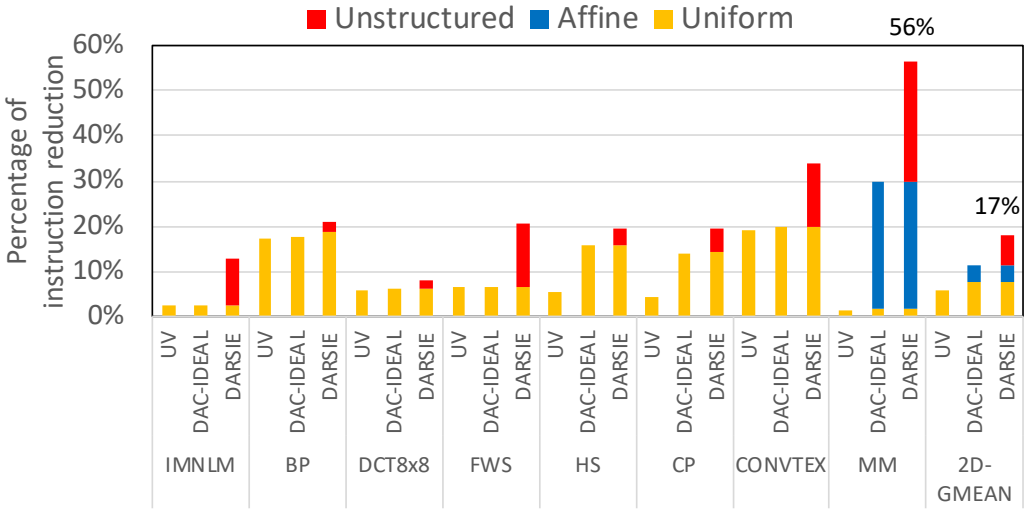

Fig. 7.6.: Percent reduction in 2D benchamrk instructions versus the baseline

The performance gain of DARSIE is not always proportional to the number of instructions eliminated. Some memory-bound applications have a high number of redundant compute operations, but few redundant memory accesses. For example,

DARSIE improves the performance of FWS by 13%, despite the fact that 21% of its instructions are skipped. This is because memory operations dominate the application runtime but are not redundant. Conversely, MM has a significant number of unstructured-redundant accesses to shared memory. MM tiling causes multiple warps in one TB to access the same shared memory blocks with affine memory addresses. This results results in excessive affine and unstructured redundancy.

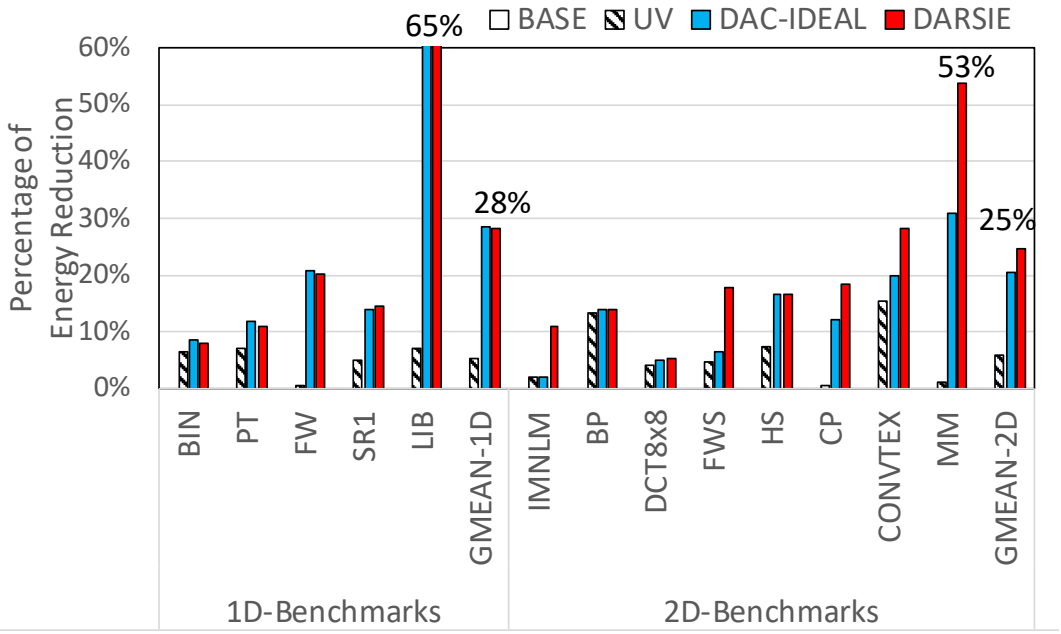

Fig. 7.7.: Percent Energy Reduction versus the baseline

Figures 7.5 and 7.6 plot the number of instructions eliminated by DARSIE and prior work. In Figure 7.6, DARSIE decreases instructions by a geometric mean of 17% in 2D TB benchmarks. UV [45] is able to remove uniform redundant instructions, but doesn't improve performance. UV removes instructions in the execute stage of the pipeline, requiring them to still be fetched and decoded. In applications like LIB, the fetch bandwidth becomes the bottleneck. Since both DARSIE and DAC eliminate instructions before they are fetched, they are able to see significant performance gains in LIB. DAC-IDEAL [109] eliminates redundant instructions by a geometric mean of 11%. We make the idealized assumption that DAC-IDEAL is able to remove all affine

values in both 2D and 1D applications, but is not able to remove the unstructured redundancy we identify in this paper. We also assume that DAC-IDEAL is able to remove *non-redundant* affine values that occur in 1D applications for example, tidx.x in Figure 4.2(a). Only DARSIE removes unstructured redundant instructions, which accounts for the improvements over UV and DAC-IDEAL in 2D TB benchmarks. As a result, DARSIE is able to match the performance of DAC-IDEAL on 1D benchmarks, while outperforming DAC in 2D applications.

Figure 7.7 shows the total energy consumption of UA, DAC-IDEAL and DARSIE is normalized to the baseline GPU. DARSIE reduces energy by a geometric mean 25%, while UV and DAC-IDEAL reduce energy by a geometric mean of 7% and 20% respectively. This improvement can be traced back to our microarchitecture preventing redundant instructions from even probing the I-cache and saves energy throughout the pipeline. The overhead of DARSIE is only 0.95% of the dynamic energy consumption. Most of the overhead comes from accessing the PC Skip Table, majority path mask and register rename table. This minimal energy overhead stems from the small size of the added hardware (roughly 82 bytes for the majority path mask, and 84 bytes per TB bank).

7.7.2 Saving Memory Bandwidth

Figure 7.9 and Figure 7.10 compare the reduction of each type of instruction in 1D and 2D benchmarks that is normalized to the baseline. DARSIE can save 45%-90% shared memory bandwidth over the baseline in 2D benchmarks by eliminating redundant memory accesses in 2D threadblock constructs. In particular, DARSIE removes over 90% shared memory LD instructions in MM, since its repetitive memory accesses within each tiling block. DARSIE also reduces over 98% global memory LD instructions in LIB benchmark, most of them are uniform redundant instructions. Also, DARSIE can save execution bandwidth. For instance, DARSIE removes 12% to 19% ALU redundant instructions in 2D benchmarks. Uniform and affine redundant

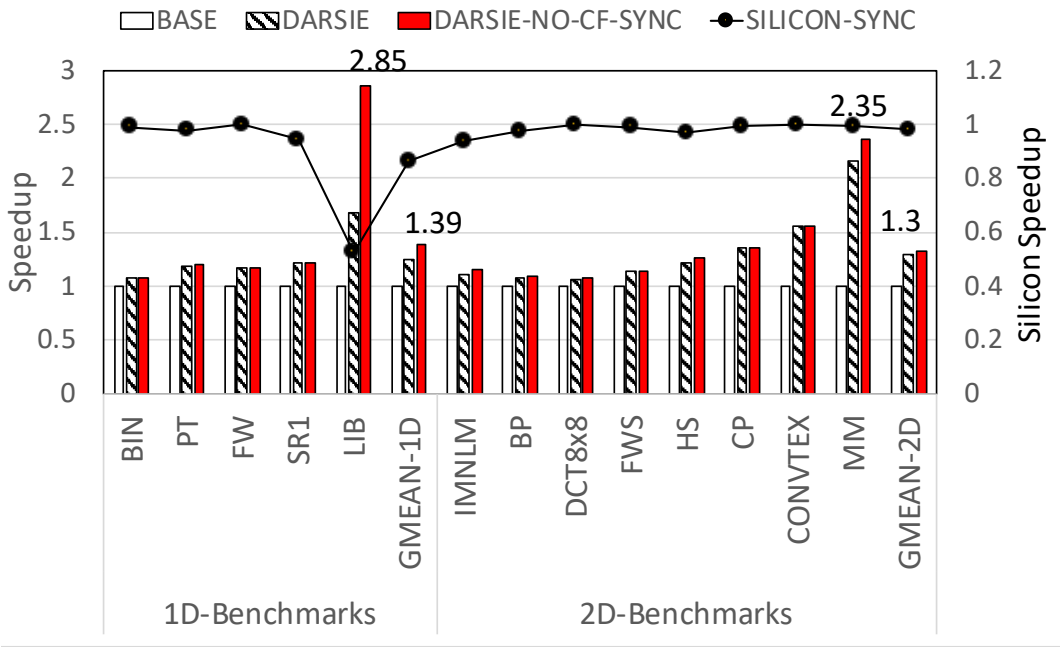

Fig. 7.8.: Effects of Synchronization.

instructions and some unstructured instructions composed of half warp size (16, 16) contribute this improvement.

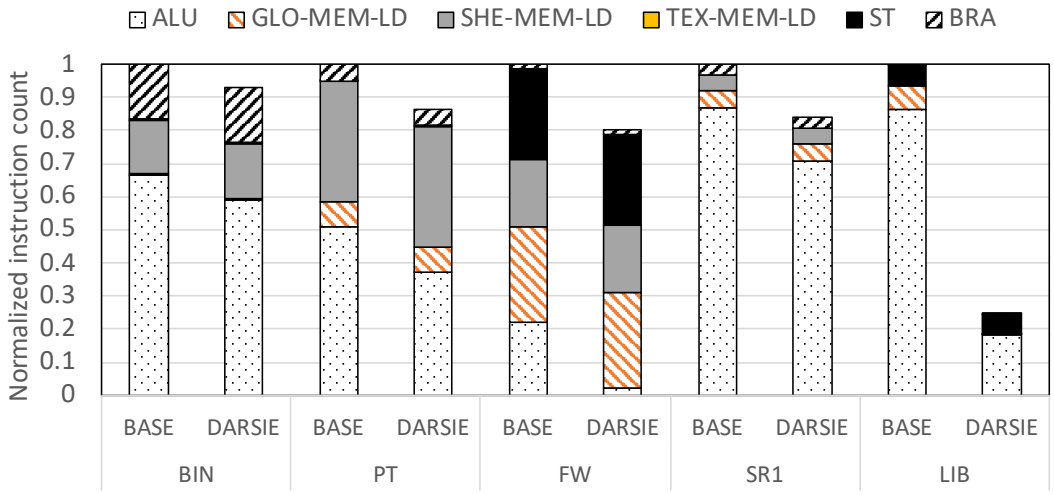

Fig. 7.9.: Instruction Reduction of 1D-benchmarks

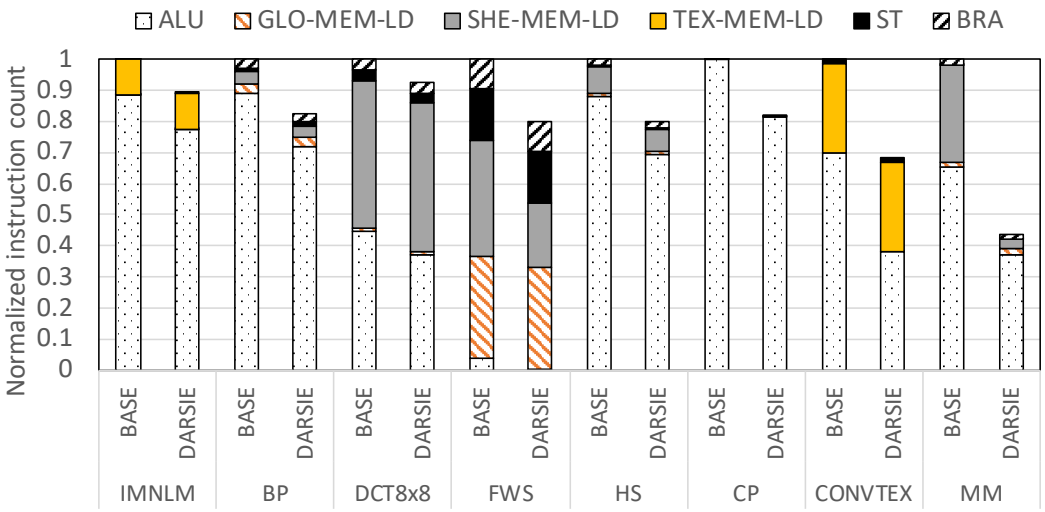

Fig. 7.10.: Instruction Reduction of 2D-benchmarks

7.7.3 Effect of Synchronization

Figure 7.8 presents the performance of an idealized DARSIE, *DARSIE-NO-CF-SYNC*, that has no DARSIE-related synchronization.

To measure the effect of DARSIE's synchronization overhead, without any of DARSIE's benefits, on a real machine, we instrumented the applications with __syncthreads() calls at basic-block boundaries and measured their performance. SILICON-SYNC in Figure 7.8 plots the effect synchronization has on performance on a silicon NVIDIA Pascal Titan X GPU. The overhead in most applications is small. Interestingly, many of the 2D applications already had __syncthread() operations at basic block boundaries, limiting DARSIE's synchronization effects. On LIB, there is a 50% performance reduction becasue the baseline application contains no __sycthreads(). However, the 75% instruction reduction DARSIE provides (Figure 7.6) on LIB makes up for the overhead.

7.7.4 Area Estimation

The three major sources of area in DARSIE are the PC Skip Table, majority path mask and register renaming/version tables. One PC Skip Table entry includes a PC value, the warp waiting bitmask (which consists of one bit for each warp that can be allocated on one TB), a bit to indicate if this instruction is a memory load (IsLoad) and a bit to indicate if the leader warp's output register has been written back (LeaderWB). One TB is allocated 8 PC skip table entries that are replaced dynamically. These fields consume 82 bits: 48 bits for the PC + 32 bits for the warp mask (since there are at most 32 warps can be allocated by one thread block) + 1 bit for the IsLoad flag + 1 bit for LeaderWB. The PC Skip Table is 256 entries based on their being at most 32 TBs in one SM, and consumes 20092 bits (2624 bytes). DARSIE allocates one majority path mask entry for one TB in one SM. These fields cost 32 bits for the warp bitmask. The total size of majority path mask is 32 $\times$ 32 = 1024 bits (128 bytes). We conservatively estimate that each entry in the register rename and version table consists of 21 bits: 8 bits for the named register (CUDA allows 255 potential named registers per thread) + 8 bits for physical register tag + 5 bits for the version numbers. DARSIE allocates 32 entries for one TB, based on the max register usage of our workloads (32). These entries therefore consume: 21 $\times$ 32 (entries per TB) $\times$ 32 (TBs in one SM) = 21504 bits (2688 bytes). Altogether, the additional structures consume an additional 5.31 kB (2.1% of the Pascal GPU register file size).

7.8 Summary

In DARSIE, I detail the root cause of massively multithreaded redundancy at the programming language level, and quantitatively explore how much redundancy exists at the grid, threadblock and warp levels. I show that a significant portion of redundancy in GPU applications is TB-wide. Moreover, I observe that much of the seemingly unstructured redundancy that occurs at runtime can be non-speculatively

identified based on TB sizing information known at kernel launch time. Using a novel compiler pass for conditional redundancy and an aggressive instruction-skipping microarchitecture that skips instructions in fetch, our proposed DARSIE design both increases performance and decreases energy consumption by 30% and 25% respectively.

DARSIE is a vertical solution that delegates each aspect of complexity to the appropriate system level. Static compilation techniques are first leveraged to propagate our newly observed *conditionally redundant* registers, then simple TB sizing information available at kernel launch time finalizes the static compiler's incomplete picture. My light-weight hardware modifications then provides what the compilation system cannot: the illusion of TB-wide lockstep execution, efficient access to warp-private registers and the ability to skip instructions from multiple warps before they are fetched.

8. CONCLUSION AND FUTURE WORK

As the increase of parallel cores within a GPU, GPUs become increasingly attractive to accelerate applications composed of massive parallelism. It is also possible to execute multiple kernels with a modest parallelism simultaneously within a single device to increase the GPU utilization. Thus, GPUs are increasingly being considered for latency-sensitive applications in data centers. These applications often demand both high throughput and real-time constraints. It is challenging to schedule these concurrent kernels with different resource usages while fulfilling their QoS requirements and fully utilizing the GPU.

This thesis addresses the problem of GPU resource underutilization shown in latency-sensitive applications and the GPU SIMT redundant instructions. The work focus on the scheduling parallel kernels and threads on the runtime system and the integrated command processor in the GPU. The fundamental idea behind the approach is the virtualization that increases resource utilization and real-time scheduling policy with the job execution time estimate. These approaches ensure the demand for latency-sensitive applications on the high throughput and real-time deadline. Additionally, this thesis also proposes a compiler-architecture co-design solution for recognizing and skipping GPU SIMT redundant instructions. This work avoids the waste of hardware resources to save GPU energy and improve performance. In the following sections, I discuss the future work based on the work within this thesis.

8.1 GPU Virtualization on the Cloud

Streaming applications include video frames from the surveillance cameras, time-series data from the internet of things (IoT) sensor devices, and speech recognition in deep learning inference. Each data stream request is often delivered over network

protocols with varying arrival times. These streaming applications often have high throughput and real-time deadline demand. The container and virtual machine can serve each data stream applications and provide security protection through the isolation. The server engine of the container and virtual machine often distributes each instance across accelerators to increase the utilization of accelerators. However, the existing virtual machine and container server engine cannot map applications based on their resource usage and the responsiveness of accelerators and can increase the tail latency of applications significantly. In general, the container and virtual machine server can obtain the global view of resource usage and estimate the job execution time in each accelerator. An intelligent job scheduler can use this information to distribute jobs properly across accelerators and reduces the tail latency of applications.

8.2 Memory Model for GPU Concurrency

Lock-free applications were tailored for shared-memory multiprocessor machines. These applications often leverage atomic operations and cache coherence protocols for conflict detection among concurrent transactions. For instance, the backpropagation operation in Convolutional Neural Network (CNN) atomically adds the product of gradient inputs and outputs in each convolutional layer to update weights. Fine-grained locking enables the high concurrency in applications but requires programmers to ensure the deadlock-free and the performance scalability of atomic operations, especially when running tens of thousands of threads concurrently on a GPU.

Contrary to CPUs, GPUs are designed to quickly switch between concurrent threads on SIMT cores to hide long-latency operations. Threads within single TB can communicate efficiently through scratchpad memory local to each SIMT core, and threads in different blocks must communicate through global memory. Contemporary GPUs provide atomic operations that can be used for inter-block communication. GPUs have an L1 cache that is local to a SIMT core and is not coherent. To ensure the correctness of atomic operations, GPUs evict L1 cache data back to the shared L2

cache in each atomic operation [67] and increase the memory access latency. KILO TM [164] exhibits the GPU hardware transactional memory but introduces additional overhead when applications have many concurrent transactions with high contention. With new support for fine-grained locking and blocking algorithms on GPUs, it is worth rethinking which applications we map to massively parallel accelerators and how we map them.

REFERENCES

REFERENCES

[1] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2008, pp. 1–11.

[2] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High Performance Discrete Fourier Transforms on Graphics Processors," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2008, pp. 1–12.

[3] J. Subhlok and G. Vondran, "Optimal Use of Mixed Task and Data Parallelism for Pipelined Computations," *Journal of Parallel and Distributed Computing*, vol. 60, no. 3, pp. 297–319, 2000.

[4] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1993, pp. 13–22.

[5] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The Case for Tiny Tasks in Compute Clusters," in *Presented as part of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2013.

[6] G. Wang, Y. Lin, and W. Yi, "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU," in *Green Computing and Communications (GreenCom), IEEE/ACM International Conference on International Conference on Cyber, Physical and Social Computing (CPSCom)*, 2010, pp. 344–350.

[7] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained Resource Sharing for Concurrent GPGPU Kernels," in *Presented as part of the USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2012.

[8] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013, pp. 407–418.

[9] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "Design and Evaluation of the GeMTC Framework for GPU-Enabled Many-Task Computing," in *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2014, pp. 153–164.

[10] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework," in *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2011, pp. 217–228.

[11] NVIDIA, "Hyper-Q Example," http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2012.

[12] A. Morrison and Y. Afek, "Fast Concurrent Queues for x86 Processors," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 103–112.

[13] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 201–216.

[14] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013, pp. 485–498.

[15] J. Appleyard, T. Kocisky, and P. Blunsom, "Optimizing Performance of Recurrent Neural Networks on GPUs," *arXiv preprint arXiv:1604.01946*, 2016.

[16] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring Sparsity in Recurrent Neural Networks," *arXiv preprint arXiv:1704.05119*, 2017.

[17] F. Zhu, J. Pool, M. Andersch, J. Appleyard, and F. Xie, "Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip," *arXiv preprint arXiv:1804.10223*, 2018.

[18] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh, "Persistent RNNs: Stashing Recurrent Weights On-Chip," in *International Conference on Machine Learning (ICML)*, 2016, pp. 2024–2033.

[19] P. Gao, L. Yu, Y. Wu, and J. Li, "Low Latency RNN Inference with Cellular Batching," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018, pp. 1–15.

[20] X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu, "Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018, pp. 162–174.

[21] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the Bar for Using GPUs in Software Packet Processing," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 409–423.

[22] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as Packet Processing Accelerator," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 83–96.

[23] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, High Performance Ethernet Forwarding with Cuckooswitch," in *Proceedings of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2013, pp. 97–108.

[24] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and et al., "Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2015, pp. 223–238.

[25] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[26] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017, pp. 221–234.

[27] T. Zhao, Y. Zhang, and K. Olukotun, "Serving Recurrent Neural Networks Efficiently with a Spatial Accelerator," *arXiv preprint arXiv:1909.13654*, 2019.

[28] S. Puthoor, A. M. Aji, S. Che, M. Daga, W. Wu, B. M. Beckmann, and G. Rodgers, "Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture," in *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2016, pp. 53–62.

[29] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2017, pp. 17–32.

[30] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2016, pp. 681–696.

[31] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "GRNN: Low-Latency and Scalable RNN Inference on GPUs," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019, pp. 1–16.

[32] C.-J. W. B. Y. C. M. H. B. R. X. Z. D. B. V. C. U. D. A. F. K. H. B. J. H.-H. S. L. M. L. B. M. D. M. M. N. M. S. M. S. X. W. [78] Liu Ke, Udit Gupta, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," *arxiv preprint arXiv: 1912.12953*, 2019.

[33] AMD, "AMD's Asynchronous Shaders White Paper." https://developer.amd.com/wordpress/media/2012/10/Asynchronous-Shaders-White-Paper-FINAL.pdf, 2016.

[34] B. Wu, X. Liu, X. Zhou, and C. Jiang, "FLEP: Enabling Flexible and Efficient Preemption on GPUs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2017, pp. 483–496.

[35] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014, pp. 193–204.

[36] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2015, pp. 593–606.

[37] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017, pp. 3–16.

[38] Y. Choi and M. Rhu, "PREMA: A Predictive Multi-task Scheduling Algorithm For Preemptible Neural Processing Units," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[39] NVIDIA, "CUDA Streams: Best Practices and Common Pitfalls ," http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf, 2014.

[40] A. K.-L. Mok, "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment," Ph.D. dissertation, Massachusetts Institute of Technology, 1983.

[41] J. D. Little and S. C. Graves, "Little's Law," in *Building intuition*, 2008, pp. 81–100.

[42] D. Simchi-Levi and M. A. Trick, "Introduction to Little's Law as Viewed on Its 50th Anniversary," *Operations Research*, vol. 59, no. 3, pp. 535–535, 2011.

[43] Z. Chen and D. Kaeli, "Balancing Scalar and Vector Execution on GPU Architectures," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 973–982.

[44] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013, pp. 130–141.

[45] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2013, pp. 433–442.

[46] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVIDIA Turing T4 GPU via Microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[47] AMD, "AMD Graphics Cores Next (GCN) Architecture," https://www.amd.com/documents/gcn_architecture_whitepaper.pdf, 2016.

[48] K. Kim and W. W. Ro, "WIR: Warp Instruction Reuse to Minimize Repeated Computations in GPUs," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018, pp. 389 – 402.

[49] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-Compression: Enabling Power Efficient GPUs through Register Compression," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 502–514.

[50] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A Variable Warp Size Architecture," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 489–501.

[51] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012, pp. 72–83.

[52] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 157–166.

[53] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013, pp. 395–406.

[54] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013, pp. 332–343.

[55] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013, pp. 99–110.

[56] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2013, pp. 185–195.

[57] S. Che., M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[58] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[59] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009, pp. 163–174.

[60] PolyBench, "The Polyhedral Benchmark Suite," http://web.cse.ohio-state.edu/~pouchet/software/polybench, 2016.

[61] NVIDIA, "NVIDIA CUDA SDK 4.2," https://developer.nvidia.com/cuda-downloads, 2016.

[62] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2010, pp. 63–74.

[63] M. Kulkarni, M. Burtscher, C. Cascava, and K.Pingali, "Lonestar: A Suite of Parallel Irregular Programs," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 65–76.

[64] J. Wang and S. Yalamanchili, "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2014, pp. 51–60.

[65] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench-the Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," 2014.

[66] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf, "Software Design Space Exploration for Exascale Combustion Co-design," in *International Supercomputing Conference (ISC)*, 2013, pp. 196–212.

[67] NVIDIA, "NVIDIA CUDA SDK 10.0," https://developer.nvidia.com/cuda-downloads, 2018.

[68] AMD, "AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE White Paper," https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf, 2012.

[69] NVIDIA, "NVIDIA TESLA V100 GPU ARCHITECTURE," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2019.

[70] S. Liu, J. Lindholm, M. Siu, B. Coon, and S. Oberman, "Operand Collector Architecture," https://www.google.com/patents/US7834881, 2010, uS Patent 7,834,881.

[71] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[72] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2012, pp. 557–568.

[73] J. Zhong and B. He, "Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, 2014.

[74] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments," in *USENIX Annual Technical Conference (ATC)*, 2011, pp. 17–30.

[75] K. Gupta, J. A. Stuart, and J. D. Owens, "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads," in *Innovative Parallel Computing (InPar)*, 2012, pp. 1–14.

[76] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Scaling Large-data Computations on Multi-GPU Accelerators," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2013, pp. 443–454.

[77] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, "Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 283–292.

[78] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous Multikernel GPU: Multi-Tasking Throughput Processors via Fine-grained Sharing," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 358–369.

[79] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2015, pp. 593–606.

[80] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014, pp. 193–204.

[81] D. Sengupta, R. Belapure, and K. Schwan, "Multi-tenancy on GPGPU-based Servers," in *Proceedings of the International Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2013, pp. 3–10.

[82] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar, "A Virtual Memory Based Runtime to Support Multi-tenancy in Clusters with GPUs," in *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2012, pp. 97–108.

[83] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, and et al., "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[84] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[85] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "DeepCPU: Serving Rnn-based Deep Learning Models 10x Faster," in *USENIX Annual Technical Conference (ATC)*, 2018, pp. 951–965.

[86] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of Service Support for Fine-Grained Sharing on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017, pp. 269–281.

[87] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *Design Automation Conference (DAC)*, 2012, pp. 850–855.

[88] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, 2016.

[89] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[90] S. K. Baruah and J. R. Haritsa, "Scheduling for Overload in Real-time Systems," *IEEE Transactions On Computers (TOC)*, vol. 46, no. 9, pp. 1034–1039, 1997.

[91] K. S. Hong and J.-T. Leung, "On-line Scheduling of Real-time Tasks," *IEEE Transactions on Computers (TOC)*, no. 10, pp. 1326–1331, 1992.

[92] A. K. Mok and D. Chen, "A Multiframe Model for Real-time Tasks," *IEEE Transactions on Software Engineering*, vol. 23, no. 10, pp. 635–645, 1997.

[93] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling Using System-generated Predictions Rather Than User Runtime Estimates," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789–803, 2007.

[94] W. Smith, V. Taylor, and I. Foster, "Using run-time predictions to estimate queue wait times and improve scheduler performance," in *Workshop on Job scheduling strategies for Parallel Processing*, 1999, pp. 202–219.

[95] U. Verner, A. Schuster, and M. Silberstein, "Processing Data Streams with Hard Real-Time Constraints on Heterogeneous Systems," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2011, pp. 120–129.

[96] K. Sajjapongse, X. Wang, and M. Becchi, "A Preemption-Based Runtime to Efficiently Schedule Multi-Process Applications on Heterogeneous Clusters with GPUs," in *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2013, pp. 179–190.

[97] H. Lee and M. A. Al Faruque, "GPU-EvR: Run-time Event Based Real-time Scheduling Framework on GPGPU Platform," in *Design, Automation and Test in Europe Conference (DATE)*, 2014, pp. 1–6.

[98] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A Framework for Real-Time GPU Management," in *IEEE Real-Time Systems Symposium (RTSS)*, 2013, pp. 33–44.

[99] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong, "Minimal Multi-threading: Finding and Removing Redundant Instructions in Multi-threaded Processors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2010, pp. 337–348.

[100] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1997, pp. 194–205.

[101] K. M. Lepak and M. H. Lipasti, "On the Value Locality of Store Instructions," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000, pp. 182–191.

[102] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 1996, pp. 138–147.

[103] J. A. Butts and G. Sohi, "Dynamic Dead-Instruction Detection and Elimination," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2002, pp. 199–210.

[104] S. Wen, M. Chabbi, and X. Liu, "REDSPY: Exploring Value Locality in Software," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2017, pp. 47–61.

[105] S. Collange, D. Defour, and Y. Zhang, "Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations," in *European Conference on Parallel Processing(Euro-Par)*, 2009, pp. 46–55.

[106] Z. Chen, D. Kaeli, and N. Rubin, "Characterizing Scalar Opportunities in GPGPU Applications," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 225–234.

[107] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, , D. Qunfeng, and H. Zhou, "A Case for a Flexible Scalar Unit in SIMT Architecture," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 93–102.

[108] A. Yilmazer, Z. Chen, and D. Kaeli, "Scalar Waving: Improving the Efficiency of SIMD Execution on GPUs," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 103–112.

[109] K. Wang and C. Lin, "Decoupled Affine Computation for SIMT GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017, pp. 295–306.

[110] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 289–308, 1984.

[111] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "CORF: Coalescing Operand Register File for GPUs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2019, pp. 701–714.

[112] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014, pp. 127–139.

[113] D. Wong, N. S. Kim, and M. Annavaram, "Approximating Warps with Intra-warp Operand Value Similarity," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 176–187.

[114] Z. Liu, S. Gilani, M. Annavaram, and N. S. Kim, "G-Scalar: Cost-Effective Generalized Scalar Execution Architecture for Power-Efficient GPUs," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2017, pp. 601–612.

[115] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovic., "Convergence and Scalarization for Data-parallel Architectures," in *International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–11.

[116] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 86–97.

[117] A. Kerr, G. Diamos, and S. Yalamanchili, "Dynamic Compilation of Data-parallel Kernels for Vector Processors," in *International Symposium on Code Generation and Optimization (CGO)*, 2012, pp. 23–32.

[118] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, "Massive Exploration of Neural Machine Translation Architectures," *arXiv preprint arXiv:1703.03906*, 2017.

[119] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation," *arXiv preprint arXiv:1609.08144*, 2016.

[120] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017, pp. 548–560.

[121] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016, pp. 1–13.

[122] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[123] J. Hestness, N. Ardalani, and G. Diamos, "Beyond Human-Level Accuracy: Computational Challenges in Deep Learning," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019, pp. 1–14.

[124] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[125] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[126] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient Data Encoding for Deep Neural Network Training," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018, pp. 776–789.

[127] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu, "Asynchronous Stochastic Gradient Descent for DNN Training," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 6660–6663.

[128] R. Kaleem, S. Pai, and K. Pingali, "Stochastic Gradient Descent on GPUs," in *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2015, p. 81–89.

[129] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, and et al., "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018, p. 1–14.

[130] A. Kalia, "G-Opt Benchmark Suite," https://github.com/efficient/gopt, 2016.

[131] AMD, "HIP: Heterogeneous-computing Interface for Portability," https://github.com/ROCm-Developer-Tools/HIP/, 2018.

[132] AMD, "MIOpen: AMD's Machine Intelligence Library," https://github.com/ROCmSoftwarePlatform/MIOpen, 2018.

[133] AMD, "rocBLAS library document," https://rocm-documentation.readthedocs.io/en/latest/ROCm_Tools/rocblas.html, 2018.

[134] AMD, "AMD Graphics Core Next (GCN) Architecture," https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2018.

[135] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010, pp. 195–206.

[136] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang, "Concurrent Analytical Query Processing with GPUs," in *International Conference on Very Large Data Bases (VLDB)*, 2014, pp. 1011–1022.

[137] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: High-throughput Stream Programming in Java," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, 2007, pp. 211–228.

[138] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., 1999.

[139] J. W. H. Liu, "The Multifrontal Method for Sparse Matrix Solution: Theory and Practice," *SIAM review*, vol. 34, no. 1, pp. 82–109, 1992.

[140] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2009, pp. 124–131.

[141] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: A GPU Runtime System for Narrow Tasks," *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 4, pp. 21:1–21:23, 2019.

[142] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," in *Proceedings of the International Workshop on Memory Management (IWMM)*, 1995, pp. 1–116.

[143] K. C. Knowlton, "A Fast Storage Allocator," *Communications of the ACM*, vol. 8, no. 10, pp. 623–624, 1965.

[144] NVIDIA, "PTX," http://docs.nvidia.com/cuda/parallel-thread-execution/, 2016.

[145] P. FIPS, "46-3: Data Encryption Standard (DES)," *National Institute of Standards and Technology*, vol. 25, pp. 1–22, 1999.

[146] NVIDIA, "The White Paper of Discrete Cosine Transform for 8x8 Blocks with CUDA," http://docs.nvidia.com/cuda/samples/3_Imaging/dct8x8/doc/dct8x8.pdf, 2012.

[147] NVIDIA, "Texture-based Separable Convolution," http://docs.nvidia.com/cuda/cuda-samples/#graphics, 2007.

[148] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[149] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction*. Springer, 2002, pp. 179–196.

[150] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and et al., "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[151] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor *et al.*, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018, pp. 608–619.

[152] J. Alsop, M. D. Sinclair, S. Bharadwaj, A. Dutu, A. Gutierrez, O. Kayiran, M. LeBeane, S. Puthoor, X. Zhang, T. T. Yeh *et al.*, "Optimizing GPU Cache Policies for MI Workloads," *arXiv preprint arXiv:1910.00134*, 2019.

[153] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[154] Baidu research lab., "DeepBench benchmark suite ," https://svail.github.io/DeepBench/, 2016.

[155] S. Narang and G. Diamos, "An update to DeepBench with a focus on deep learning inference," https://svail.github.io/DeepBench-update, 2016.

[156] J. Hauswald, "Lucidia Benchmark Suite," https://github.com/jhauswald/lucida, 2016.

[157] M.-T. Luong and C. D. Manning, "Achieving open vocabulary neural machine translation with hybrid word-character models," *arXiv preprint arXiv:1604.00788*, 2016.

[158] T. T. Yeh, R. Green, and T. G. Rogers, "Dimensionality-Aware Redundant SIMT Instruction Elimination," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2020.

[159] NIRVANA, "Maxas SASS Assembler," https://github.com/NervanaSystems/maxas, 2016.

[160] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "A Formal Analysis of the NVIDIA PTX Memory Consistency Model," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2019, pp. 257–270.

[161] T. M. Aamodt, "GPGPU-Sim 3.x Manual," http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual, University of British Columbia, 2012.

[162] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013, pp. 487–498.

[163] S. J. E. Wilton and N. P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[164] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011, pp. 296–307.