Rethinking Context Management of Data Parallel Processors in an Era of Irregular Computing

by

Jonathan Beaumont

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science Engineering)
in the University of Michigan
2019

Jonathan Beaumont
jbbeau@umich.edu
ORCID iD: 0000-0001-5092-9094

ACKNOWLEDGMENTS

First and foremost, I must thank my advisor Trevor Mudge. From my first tentative email inquiring about research through the completion of this dissertation, he has never wavered in doing his utmost to assist me in achieving my goals. His guidance, along with that of my committee members David Blaauw, Ron Dreslinski and Scott Mahlke have been integral to my success as a researcher, and their patience and positive spirit gave me the motivation to persevere during frustrating periods.

Mark Brehob was a fantastic mentor to me throughout my undergraduate and graduate studies. He advised me through the process of applying to grad school, surviving my first (and second and third) semester teaching, getting involved with research, and finally navigating the daunting landscape of applying for faculty positions. Drew DeOrio was also an encouraging and ever helpful resource in my efforts to pursue a role in academia.

My peers in the department were key to my success at Michigan, not just for the collaborations which would spawn the bulk of this dissertation, but also for the sense of community that made it worth trudging through polar vortices to get my work done. In particular, I would like to thank Tony Gutierrez, Qi Zheng, and Nilmini Abeyratne for their mentorship during my inauguration into TRONLab. I am grateful for the experiences shared with my other labmates who have joined and departed over the years: Yajing Chen, Cao Gao, Byoungchan Oh, Johann Hauswald, and Yiping Kang. Amlan Nayak and Steve Zekany provided crucial, sanity-maintaining distractions in the form of inter-lab drop-ins.

John Kloosterman was instrumental in getting me involved in GPU research. The contents of chapter 4 are largely a result of my collaborations with Subhankar Pal and Donghyeon Park. Chapter 5 is the product of the guidance and collaboration from my team while interning at Arm Research: Alex Rico, Jose Joao, Josh Randall, Jesse Beu, Dam Sunwoo, and Francesco Petrogalli.

Finally, none of what I've been able to accomplish would have been possible without the love and encouragement of my family. My parents, Bruce and Susan, were an endless supply of support (both financially and emotionally). I'm not sure they appreciated what a commitment it would be when they agreed to let me use the spare car "while I finished my degree" at the eve of my two year master's program six years ago. But if they had any reservations then or at any other point during my perpetual attempts to stave off adulthood, they didn't show it. Chris and Matt gave me the gift of equanimity in a way that only brothers know how. Since beginning my program, I have had the great pleasure of welcoming my two sisters-in-law, Katie and Maggie, and my precious nibling Ethan into my family. Their presence and love have meant the world to me, and it shan't be soon forgotten.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table

ABSTRACT

Data parallel architectures such as general purpose GPUs and those using SIMD extensions have become increasingly prevalent in high performance computing due to their power efficiency, high throughput, and relative ease of programming. They offer increased flexibility and cost efficiency over custom ASICs, and greater performance per Watt over multicore systems. However, an emerging class of irregular workloads threatens the continued ubiquity of these platforms as general solutions. Indirect memory accesses and conditional execution result in significantly underutilized hardware resources. The nondeterministic behavior of these workloads combined with the massive context size associated with data parallel architectures make it difficult to manage resources and achieve desired performance.

This dissertation explores new strategies for scheduling irregular computational tasks. Specifically, we characterize the performance loss associated with current thread block scheduling policies in GPU architectures and evaluate possible extensions to enable better performance. Common patterns exist in irregular workloads which allow the architecture to dynamically respond to changing execution conditions. We analyze how these strategies can entail high overhead in many-thread architectures due to their large context sizes and explore methods to limit this cost. Our solution is able to achieve significant increases in throughput of up to 17% with minor augmentations to traditional GPU architectures and full support for legacy software. We show that by extending these solutions to incorporate more dramatic alterations to the architecture and programming model, we can increase this improvement to 24%.

We further identify potential correctness issues when generalizing these strategies to heterogeneous multi-core SIMD systems. After presenting data motivating the support for context switching in these systems, we demonstrate how modifications can guarantee correctness and propose simple extensions to the ISA which enable the full benefits of these dynamic solutions.

CHAPTER 1

Introduction

Data parallel machines are effective at executing workloads that apply the same operation across many pieces of data. This type of workload is highly prevalent in signal processing, graphics, and scientific computing, as they often process large amounts of data which are largely independent from one another. Data-parallel machines fall under the "single instruction multiple data" or "SIMD" quadrant of Flynn's taxonomy (see figure 1.1) and offer many advantages over "multiple instruction multiple data" or "MIMD" architectures such as multicore machines when processing highly regular workloads. Because different streams of computation are all executing the same instructions, hardware resources such as instruction fetch and decode can be shared among processing elements. This allows greater power efficiency and scalability over MIMD architectures and, assuming that the execution behavior of these streams does not diverge, allows for much higher peak theoretical throughput.

Examples of data parallel machines include vector architectures, subword-SIMD architectures, and general-purpose graphics processing units. All of these take advantage of data parallelism by fetching multiple data values for each instruction executed, processing them in parallel, and writing the results back. Although they have been demonstrated to show energy savings and throughput improvements over other architectures on applications exhibiting *regular* data parallelism (parallelism involving predictable memory and control patterns), the benefits are reduced on applications with *irregular* data parallelism. This type of parallelism, while still allowing multiple instances of an instruction to be executed over multiple data in parallel, may contain either control flow and memory flow divergence, or both.

Such irregular parallelism is highly prevalent in graph processing algorithms, which consist of a diverse range of workloads ranging from the breadth-first search and minimum spanning tree analysis to stochastic gradient descent in machine learning kernels and are becoming ever more important in the processing of big data. Control flow and memory irregularities cause these architectures to make only partial use of their hardware resources,

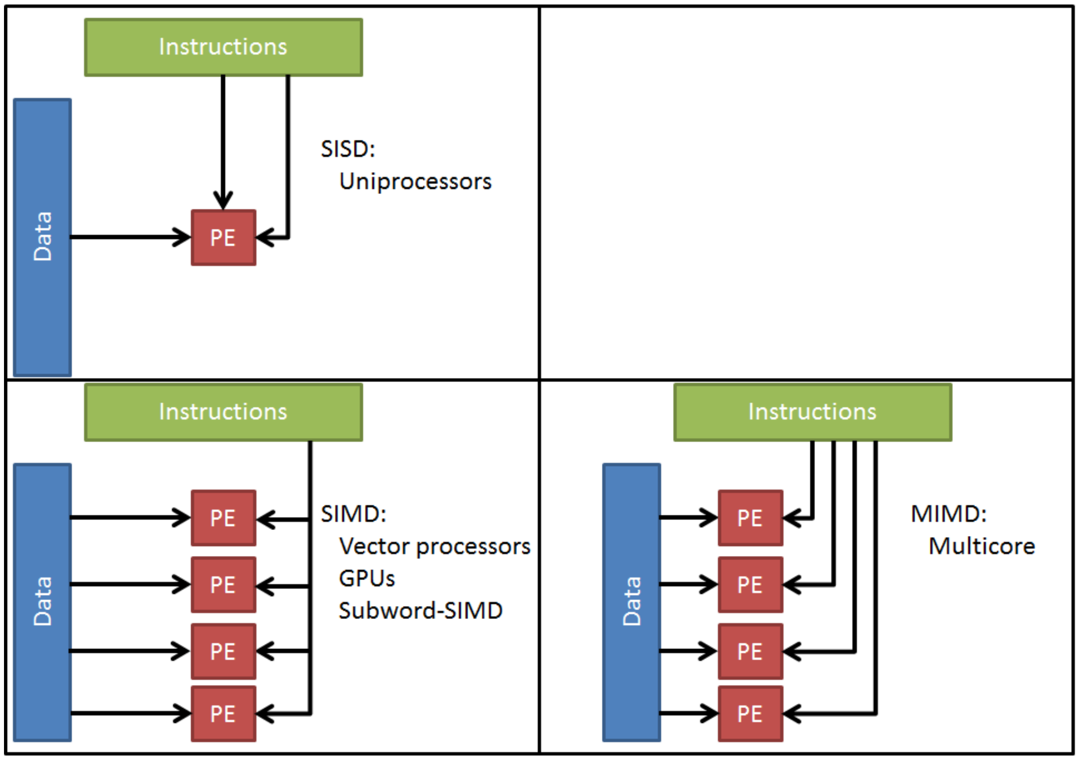


Figure 1.1: Flynn's taxonomy of architectures (MISD not shown)

and splitting computation over longer periods of time, reducing parallelism. Worse still, the exact degree of irregularity is often dependent on the structure of data being processed, which may not be known until runtime and may change dynamically. Several implementations of irregular algorithms have been ported to SIMD architectures and demonstrate speedup over MIMD architectures, but it is a more challenging process and provides much smaller speedups when compared with regular workloads due to underutilized resources.

On GPU architectures, which run a massive number of threads concurrently, co-running multiple workloads which make use of complementary resources (for example, one workload with a high memory bandwidth footprint and low floating-point utilization, and other which uses little memory bandwidth but high use of the floating-point units) is a common strategy to combat underutilization. Unfortunately, most strategies for choosing which workloads to co-run rely on static information which do not account for the dynamic behaviors associated with irregular workloads, and the dynamic strategies that do exist make assumptions regarding the structures of running programs which often do not extend to general workloads.

Reconfigurable architectures offer another promising solution to the problem of resource utilization, as hardware resources can be explicitly reconfigured in response to changing input patterns. Rather than migrating data between different processing elements each equipped with distinct resources ideal for processing a particular type of workload, resources can be dynamically reassigned to these elements while keeping data stationary.

Finally, architectures making use of subword-SIMD instructions offer much less throughput improvement over other data parallel machines, but are a popular approach due to their significantly simpler programming model. The parallelism of these architectures is much more rigid, typically having less flexibility in control flow and having fixed sized vector units for each implementation which may not be ideally suited for all workloads, depending on the amount of parallelism and regularity within the code. Including different sized SIMD vector units within a heterogeneous system could provide better flexibility in executing a more diverse set of workloads. However, doing so creates challenges as changing the vector length of these cores mid-execution has undefined behavior and may be difficult if not impossible to design programs which have the intended results.

1.1 Contributions

This dissertation explores the effects of running irregular workloads on different data parallel architectures. We identify challenges associated with each platform, and present our findings that a greater amount of flexibility with how thread contexts are scheduled, partitioned, and managed throughout their lifetime can help to overcome these obstacles. This increase in flexibility presents challenges of its own, which we explore in detail.

Chapter 2 presents the background information and concepts that will be used throughout this dissertation. Chapter 3 explores the challenges of multi-kernel execution on GPUs when running irregular workloads and how a more sophisticated dynamic scheduling and context placement policy combined with minimal hardware extensions can greatly improve co-location of complementary workloads. Chapter 4 extends these ideas by looking at how enabling reconfigurability in a data parallel architecture can more greatly enhance the ability to allocate resources to thread contexts more effectively and with less overhead. Chapter 5 delves into the benefits of including multiple differently-sized vector units in a single system and enabling context swapping between them as well as how to overcome the correctness challenges that arise as a result.

1.1.1 Improving Co-execution on GPUs

Irregular workloads can have a wide range of performance characteristics depending on the specific structure of the data being processed. This makes it challenging to optimally schedule threads and allocate resources such that they do not interfere with one another.

However, by observing common trends in these workloads, we can respond dynamically to changing conditions by tracking performance metrics over time, and either relocating threads or updating resource prediction schemes. In chapter 3, we show that improved scheduling policies can improve throughput of irregular workloads by up to 17% over existing proposals with trivial storage extensions and full support for legacy code.

1.1.2 Increasing Resource Management Granularity using Reconfigurable Architectures

Architecture types with specific parameters are ideally suited for certain subsets of workloads. Heterogeneous systems with accelerators can provide greater flexibility at a coarse level by transferring control flow between cores, but these transfers can introduce long latencies and be impractical for workloads that display variance at a fine granularity. In chapter 4, we explore how reconfigurable architectures can manage thread placement and resource allocation at a much finer granularity for workloads with diverse execution behaviors. We introduce block shuffling and show how latencies for context transfers and many-thread architectures like GPUs can be reduced by 90% and improving overall throughput by 24%.

1.1.3 Enabling Context Migration in Heterogeneous SIMD Systems

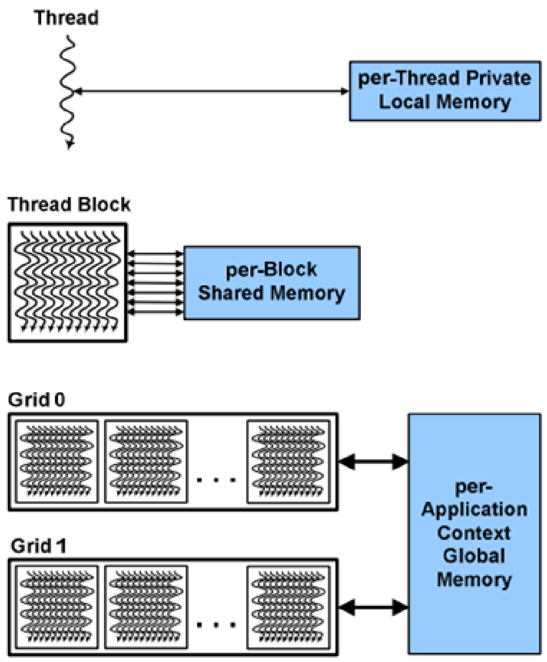
SIMD processors offer parallelism with much less effort from the programmer than GPUs or other accelerators. However, their speedup potential is significantly reduced, and often do not generate the desired power and performance metrics desired for applications with parallelism outside a particular range. In chapter 5, we show how enabling the inclusion of multiple SIMD cores with different vector sizes can allow for a finer navigation of the Power-Performance-Area (PPA) tradeoff landscape for system designers at the cost of undefined behavior for general applications. We describe how this introduces problems with guaranteeing correctness in applications and present a simple extension the ISA which guarantees identical execution for all legacy code while allowing for potential speedups of future software.

CHAPTER 2

Background

2.1 GPUs

NVIDIA's CUDA GPUs use a single instruction multiple thread (SIMT) programming model, in which the programmer writes code for a single *thread* that is replicated hundreds to thousands of times on the hardware to operate on large sets of data in parallel. The GPU hardware consists of several *streaming multiprocessors* (*SMs*), each containing a large register file to support many threads, a cache-hierarchy including a software managed scratch-pad (also called *shared memory*), and several multithreaded single instruction, multiple data (*SIMD*) pipelines.



Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

Figure 2.1: CUDA thread hierarchy [1]

Threads are grouped into *thread blocks* (*TBs*) by the programmer, which are guaranteed to run on the same SM and can cooperate through barrier instructions and a scratchpad called *shared memory*. Thread blocks are dispatched to SMs via the global *thread block scheduler* (*TBS*) as a *grid* when enough resources (namely register file and shared memory space) are available and are run asynchronously from one another until all threads within the thread block have completed. A programmer can indicate that independent kernels can be run concurrently by issuing them on different *streams*. Kernels issued on the same stream are guaranteed to be run serially. Figure 2.1 shows the thread hierarchy on CUDA systems and figure 2.2 shows how threads correspond to hardware.

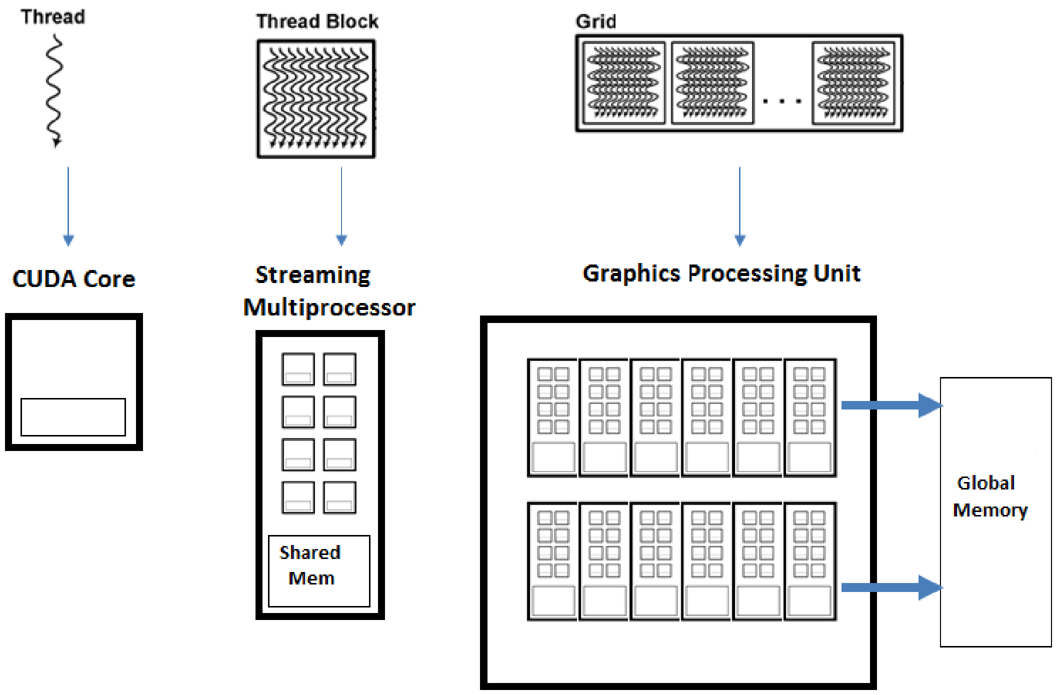


Figure 2.2: CUDA hardware hierarchy [1]

SMs do not employ branch prediction or out-of-order mechanisms, but instead rely on swapping between the many available threads on a cycle-by-cycle basis to hide arithmetic and memory latencies. Thread blocks are transparently divided into smaller bundles of threads called *warps*, which are executed in lockstep, simplifying fetch and dispatch logic. These considerations allow a much greater portion of a GPU's die area to be dedicated towards computation compared with scalar cores, and enable their high-energy efficiency and throughput capabilities [4].

While the SIMT programming model allows for the use of conditional statements within code which create the appearance of threads being able to run independently of one another, thread divergence (the occurrence of multiple distinct branch outcomes within a single warp) causes different branches to be executed in series with certain execution lanes deactivated. This reduces the overall utilization of the SMs and hurts performance. Similarly, while the programming model allows load and store addresses within a warp instruction to be arbitrary, the memory system is optimized for memory accesses within a warp to a single cache lines. Memory operations with irregular patterns are replayed and serialized across several cycles, again reducing utilization. Programmers must take care to ensure that control-flow irregularity (CFI) and memory-access irregularity (MAI) are minimized as much as possible to maximize performance. These values can be quantified accordingly:

$$CFI = \frac{divergent\_branches}{executed\_instructions}$$

$$MAI = \frac{replayed\_instructions}{issued\_instructions}$$

2.2 Irregular Algorithms

Data-parallel architectures, such as GPUs and SIMD extensions on general purpose processors, have been ideal platforms for accelerating algorithms which involve unconditionally applying repeated operations on logically contiguous data. Matrix-matrix multiplication is an archetypal application which can achieve over $100\times$ improvement in performance per Watt on a GPU over serial implementations, as data can be streamed very efficiently through its hundreds of available processing units with minimal control overhead [5].

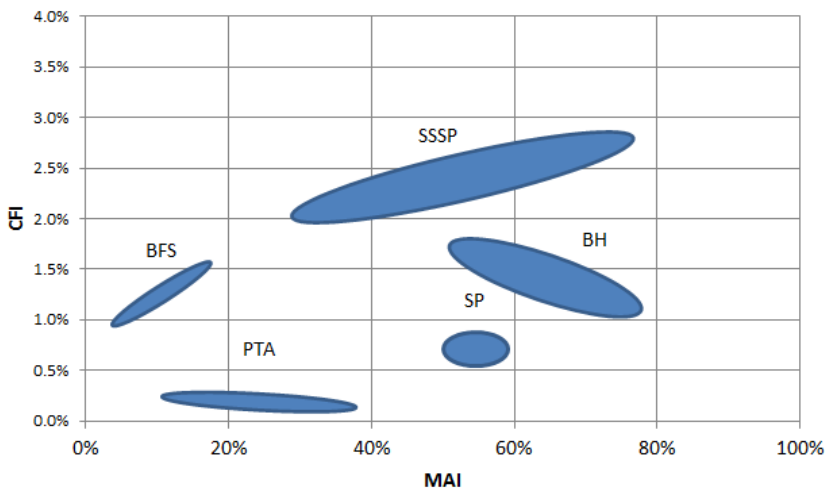


Figure 2.3: Range of memory-access irregularity and control-flow irregularity on several irregular workloads as inputs are varied (data source from Burtscher et al [2])

While not to the same degree, a wider class of algorithms referred to as irregular data parallel algorithms (also called “amorphous data parallel algorithms” or simply “irregular algorithms”), has been demonstrated to show speedups of $2-10\times$ on parallel architectures [2]. These algorithms are still data parallel in the sense that operations are applied repeatedly across multiple pieces of data, but the control flow and memory access patterns are less regular and predictable. Many problems which can be represented as graphs, where operations are propagated through adjacent nodes, are irregular, since the memory layout is data dependent and may change throughout execution. Figure 2.3 shows how the MAI and CFI varies significantly over different input types. Example workloads include breadth-first search, single source shortest path, points-to analysis, Barnes-Hut N-body simulations [6], and survey propagation [7]. While these irregularities make it difficult to fully utilize hardware, several techniques have been demonstrated to show practical speedups on parallel architectures [8][9][10].

```
__global__ void topo(Node *nodes, bool *done) {
  Node node = nodes[threadIdx];
  if( node.active() ) {
    node.process();
    *done = false;
  }
}

int main() {
  //...
  while (!finished) {
    done = true;
    topo<<<N>>>(nodes, &done);
  }
}
```

Figure 2.4: Topology-driven implementation of an algorithm, which spawns N threads per iteration. Each thread checks whether its assigned node is active, and if so processes it. The “process” method may activate neighboring nodes

Two common approaches towards designing these algorithms are topology-driven and data-driven implementations [11]. In both of these styles, the work to be done is represented by a set of connected nodes. Some subset of these nodes are active, indicating that they must be processed by applying an algorithm-specific operator on them. After processing each node, the neighboring nodes may or may not be activated, causing a propagation effect of iteratively processing active neighbor nodes until a steady state or some exit condition is reached. For example, in breadth-first search, the operator simply increments a distance variable by one, and all neighboring nodes are always activated.

Topology and data-driven implementations are different in how processing elements are assigned active nodes to process. In topology-driven implementations (an example is shown in figure 2.4), threads are generated to process each node regardless of whether there is useful work to be done at that node. Such implementations are usually simple to design, but for data sets where the percentage of active nodes at any given time is low, this approach will be very inefficient, as few threads are performing meaningful work by executing the "process" method.

```
__global__ void data(Node *nodes, WL *wl)
  while(idx = wl->pop()) {
    Node node = nodes[idx];
    node.process();
    for(i=0; i<node.num_neighbors; i++) {
      wl->push(node.neighbor(i));
    }
  }
}

int main() {
  //...
  init<<<N>>>(nodes, wl);
  data<<<M>>>(nodes, wl);
}
```

Figure 2.5: Data-driven implementation of an algorithm, which spawns M threads, each iteratively popping work assignments of a shared worklist and adding new work items back on

To address this, data-driven implementations (see figure 2.5) dispatch a fixed set of threads which persist throughout the application's execution and are assigned nodes to process by a shared worklist maintained in software. Nodes are only added to this worklist when they are activated, so there are no idle "spin-loops" as there are in the topology-driven implementation.

Data-driven implementations are more algorithmically efficient as they avoid unnecessary work when processing inactive nodes. However, memory contention caused by accessing a shared worklist with atomic memory operations means such implementations are rarely used without aggressive software optimizations.

Double-buffering replaces a single worklist with separate push and pull worklists [11].

This reduces memory contention since atomic memory operations only need to be used when writing to a worklist (otherwise separate threads might clobber each other's additions when writing at the same time). Reads from the worklist can proceed freely since they don't modify it. This has the downside of limiting load balancing within the worklist, as processing elements that finish their work quickly need to wait for the push worklist to be swapped with the pull worklist before they can process more nodes.

Work donating can offset this penalty by using an auxiliary "donation worklist" in shared memory that threads with an excess amount of work can push to, and threads within the same thread block with less work to do can pull from.

Atomic-reduced updates synchronize several threads before pushing elements onto the worklist. It then performs a prefix-sum to calculate how many total elements across all threads will be pushed to the worklist, and does so as a single atomic memory operation, thus reducing the amount of memory contention.

Distributed worklists using thread-block partitions [11][10] potentially sacrifices parallelism when load-balancing is needed but allows scaling to much larger workloads with low *density*, which we define as

$$D = \frac{2|E|}{|V|(|V|-1)}$$

where V is the total number of vertices or nodes, and E is the total number of edges or adjacencies between the nodes.

The work presented in this dissertation makes use of the LonestarGPU benchmark suite Version 3.0[2] to study the behavior of irregular algorithms. The benchmark suite has provided many implementations of several key irregular algorithms, but as the primary concern how computation will scale with the proliferation of large, sparse data sets, analysis is limited to data-driven implementations to those which use distributed worklists as part of their optimizations. Thus, the following benchmarks are used:

Breadth-first search (BFS) counts the minimum number of nodes between a specified source node and all other nodes. Starting with the source node, neighboring nodes are activated and updated with the current distance. There is a single kernel invocation which iterates until the graph is traversed, and the operator is very simple [12].

Delaunay mesh refinement (DMR) is provided with a mesh of nodes and iteratively modifies edges between them to resolve constraint violations. Because it modifies the underlying data structure, it is a "morph algorithm" [13], which often invoke extra overhead since extra synchronization via barrier instructions is needed to prevent partially updated graphs from being read. It alternates invoking two separate kernels, one to modify the mesh, and the other to check for new violations.

Minimum spanning tree (MST) is another morph algorithm which finds a subset of the input graph that spans all nodes and has the smallest overall cost. It uses Boruvka's algorithm and alternates between two kernels to iteratively find the minimum weight edge coming from each component and then merge partners across those minimum edges. The kernels become more computationally intense as the graph is reduced.

Points-to-Analysis (PTA) determines the set of addresses a pointer variable can access given a set of constraints. The algorithm uses a pipeline of low-latency (<100 μs) kernels to iteratively propagate constrains to different nodes until a steady state is reached. [14]

Single-Source Shortest Paths (SSSP) calculates the shortest distance from a specified source node to all other nodes in a weighted graph, in a similar manner to BFS, but with slightly higher bandwidth needed to read the weights and slightly higher computation intensity to process them.

Stochastic Gradient Descent (SGD) completes unknown entries in a supplied sparse matrix. It involves several dot product calculations between vectors and is the most computationally intensive workload considered.

CHAPTER 3

Dynamic Thread Block Scheduling

3.1 Introduction

There has been a significant trend in mapping an increased variety of workloads to data parallel architectures. We have seen numerous enhancements to GPU hardware by vendors to enable greater flexibility and programability [15][16][17] as well as the emergence of new architectures for specific domains, such as Google's Tensor Processing Unit [18] to accelerate machine learning algorithms. A prevalent challenge is the increasing amount of control and data irregularities present in many non-trivial algorithms [2], making it difficult to fully utilize hardware. Multi-kernel execution and virtualization of hardware across multiple users have been employed on GPUs to mitigate these problems [19], but it is difficult to scale these solutions in light of nondeterminism introduced by irregular algorithms.
In this work, we will explore specifically how multi-kernel techniques are hampered by irregular data parallelism, and present our design for an extension to thread block scheduling and placement that will circumvent these challenges. Our contributions are as follows:

- We identify and characterize the variability of resource usage within and across thread blocks when running irregular algorithms as a function of input density, resulting in distinct phases of computation
- We demonstrate that previous methods of resource allocation, both static and dynamic, are insufficient at accounting for this variability and result in interference between thread blocks
- We propose a solution which adds hardware to track and predict future resource usage for each thread block. This hardware is used to make more sophisticated decisions for allocating resources, as well as dynamically detecting phase shifts and reallocating resources when appropriate

- We show that our solution results in a 17% and 13% average increase in throughput over established static and dynamic scheduling strategies, respectively, while only increasing total SRAM requirements by less than 0.5%

The remainder of this chapter is organized as follows. Section 3.2 discusses the relevant background of GPUs, irregular algorithms, and multi-kernel workloads. Section 3.3 lays out how irregular algorithms can prevent the ideal placement of work items in a GPU. Section 3.4 gives a high level description of our design. Section 3.5 elaborates on the parameters chosen for the design and gives a description of the simulation testbench. Section 3.6 analyses the increase in computational throughput we see in our technique as well as the overheads incurred, and finally section 3.7 discusses similar work in literature.

3.2 Background

3.2.1 Multi-Kernel Execution

Not all kernels are created equal: some utilize certain resources more than others. A popular technique to mitigate the challenges of underutilized hardware is to run multiple workloads simultaneously, which are expected to make use of complementary resources. For instance, work on GPUs has shown that applications with distinct register file, shared memory, functional unit, and memory bandwidth usage can be co-run with minimal interference from one another, and can improve overall throughput by over 30% [20][19][21]. NVIDIA has introduced Hyper-Q [22] and Multi-Process Service [23] to allow concurrent execution of multiple kernels. Cloud providers have begun virtualizing GPU resources to multiple users in an effort to maximize the number of available kernels that can be run concurrently. Kernels can be co-run using either spatial multitasking [24] in which different kernels are run on separate SMs, or by simultaneous multi-kernel (SMK) execution [25] in which different kernels can be run on the same SM. SMK provides better utilization when the thread blocks being co-run use complementary resources.

Previous techniques classify at the kernel level what resources a thread block is likely to consume through profiling across several inputs or augment the kernels during compilation or runtime to enable better flexibility [26]. Others monitor the kernels dynamically to characterize the resource usage at a coarse-level dynamically [27].

3.3 Motivation

While previous works [20][27] have recognized the problem of interference across thread blocks, none have considered the added difficulty when processing irregularly structured data and how resource usage of threads changes as a result. This is of paramount importance when considering the increased prevalence of data sets with non-uniform patterns [28]. Real world data sets often consist of graphs with clusters of highly-connected nodes dispersed across many sparsely-connected nodes. For example, figure 3.1 shows the distribution of different local densities in a graph representing which Netflix movies were enjoyed by which users [2]. What we see is that while about three quarters of the graph's nodes are quite sparsely structured, having connections between .001% and 1% of the other nodes, the remaining quarter of the nodes have a wide range of densities. This is an example of a "power-law" graph, which are becoming ever more prevalent in the era of big-data computing [29].

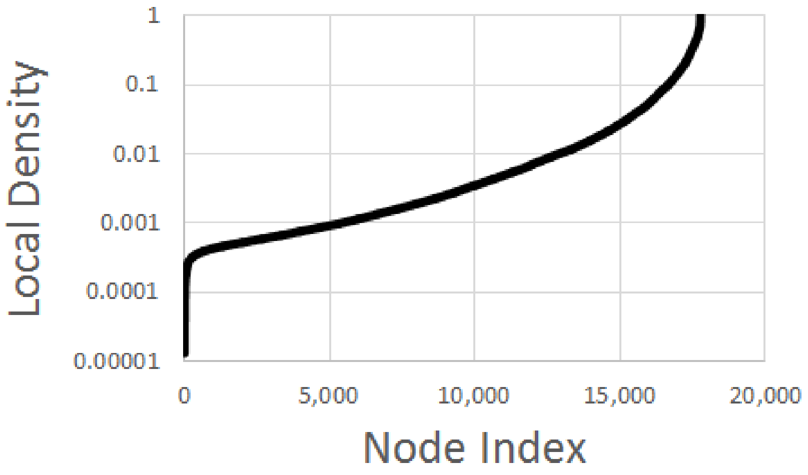


Figure 3.1: Distribution of densities of a graph where nodes are sorted by number of connected neighbors. Local density of a given node is the number of connected neighbors for that node divided by the total number of nodes

Figures 3.2 and 3.3 show two prominent effects this distribution has on the performance of algorithms processing such input data sets. Using an NVIDIA Tesla V100 GPU (the specifications of which are described in section 3.5), we measured how the rate of stalls resulting from memory throttling (i.e. when the load-store-unit (LSU) is fully saturated and can't accept more requests) changes as the density of the data is increased across benchmarks from the LonestarGPU suite. We summarize the range of this metric in the second column of table 3.1, i.e. the total reduction of stalls in the densest data set compared to the sparsest data set. Across all benchmarks, there is a decrease in the rate of these stalls as

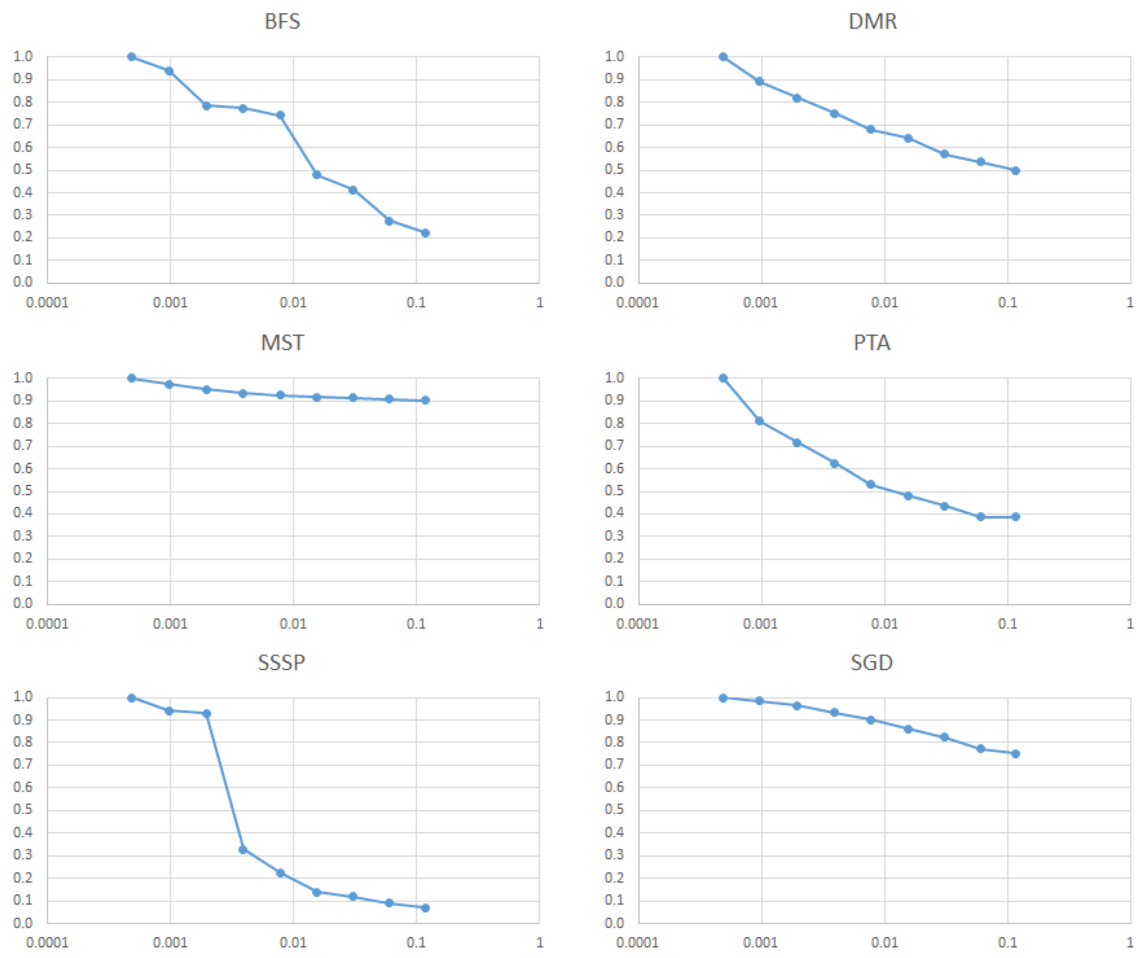


Figure 3.2: How the rate of stalls due to memory throttling, normalized to the sparsest baseline, (y-axis) varies as a function of input density (x-axis)

density increases. This is a consequence of higher data-reuse when processing dense data. As long as the working data set can be fit within the cache, tightly packed clusters of nodes will result in the compute operator processing the same nodes multiple times in quick succession, resulting in a higher cache hit rate. Similarly, sparse segments of data will result in pointer-chasing with little data reuse and the LSU is more likely to be saturated with long latency cache misses. BFS and SSSP are both memory bound workloads with very simple compute operators, and accordingly show significant changes (a total of 78% and 93%, respectively) in the rate of stalls as the density and data reuse increases. On the other extreme, MST and SGD have much more computationally intense operators which hide the latencies associated with cache misses. The penalties associated with processing highly sparse data is correspondingly less than workloads with simple compute operators, with a 10% and 25% reduction of stalls in processing the densest data sets, respectively. DMR and PTA show moderate reductions in stalls when processing dense data (50% and 61%,

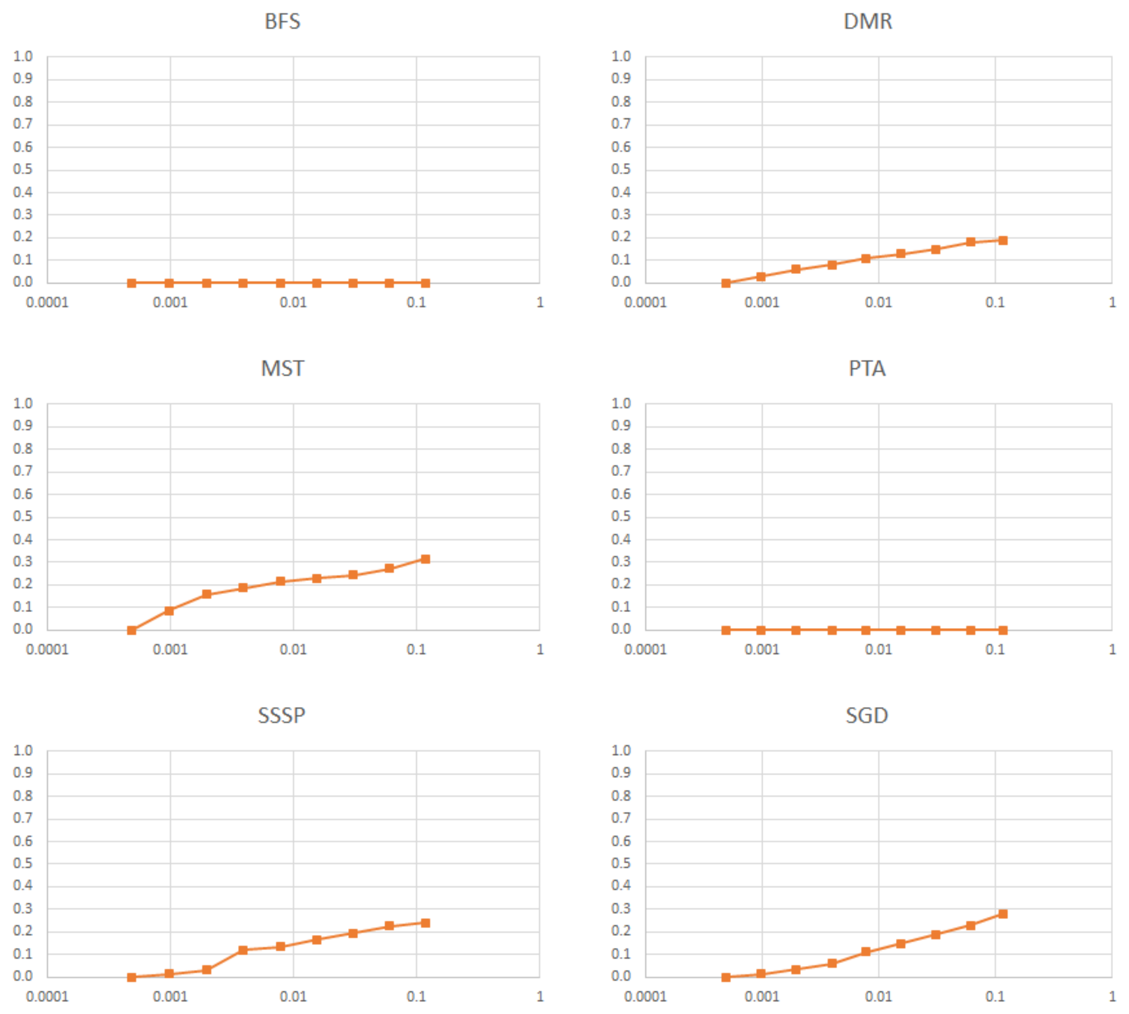


Figure 3.3: How the increase in floating-point unit utilization, normalized to the sparsest baseline, (y-axis) varies as a function of input density (x-axis)

respectively).

Application	**Memory Throttle Range**	**FPU Utilization Range**
BFS	78%	0%
DMR	50%	19%
MST	10%	31%
PTA	61%	0%
SSSP	93%	24%
SGD	25%	28%

Table 3.1: Range of Resource Variance across kernels

We also measured how the utilization of floating point units (FPUs) are affected by density in figure 3.3. The third column of table 3.1 summarizes the total increase in FPU utilization in the densest data set processed compared to the sparsest. Both BFS and PTA do not contain any floating point operations and are thus unaffected. DMR and SSSP show increased utilization (19% and 24%, respectively), but this is largely a side effect of the reduction of memory stalls; since less time is spent waiting for memory, a larger percentage of time is spent processing data and utilization is increased. The amount of computation needed to process each node is largely unaffected. MST and SGD both have operators which grow in complexity as the density increases. For example, SGD performs numerous dot products on segments of the input data. As the number of non-zero elements within a row increases (i.e. as the density increases), more elements must be multiplied and summed together for each dot product, increasing the computational intensity of the operator. Accordingly, MST and SGD both see an increase in FPU utilization (31% and 28%, respectively) that is partially independent of the decrease in memory stalls. This fact will be significant in identifying benefits of tracking both metrics in tandem.

Because different thread blocks process independent data, there is little guarantee about how sparsity levels of the data correlate between active thread blocks. If, for example, the graph summarized in figure 3.1 were being processed, one thread block might be processing a highly dense (for example, 5%) neighborhood, while another thread block processes one of the several sparser neighborhoods (say 0.1%). Over time, the threads processing these nodes may migrate to neighborhoods of different density, and correspondingly the resource usages of the containing thread blocks may also change. Accordingly, an optimal thread block scheduler should treat individual thread blocks within a given kernel differently depending on the properties of the data being processed. The importance of resource allocation has grown over the years and will likely continue to do so as the number of threads placed on a single SM increases. While earlier, CUDA GPUs limited the number of allowable threads on a single SM to 768 [30], that number has increased to 1024, 1536, [31], and 2048 in recent architectures [22], which we summarize in figure 3.4. Improvements in fabrication technologies allow for more hardware resources to be placed on an individual SM, increasing the number of threads that are likely to be allocated. As the number of threads resident on an SM increases, the variance of resource usage has a greater compounded effect. We make two observations that guide our design decisions for a finer granularity scheduling and resource allocation policy:

- The density of graphs changes smoothly as the graph is traversed. To illustrate this, let $NDR(n, d)$ be a function that represents the "Neighbor Deviation Range" of the subset of a graph centered at node n with a diameter d. This function represents the difference in number of neighbors between the most highly connected node and the least connected node in the subgraph. For example, in figure 3.5, $NDR(1, 1) = 5$, as the most connected node in the subgraph starting at node 1 and extending outwards 1 hop has 6 connections, and the least connected nodes all have 1 connection. Similarly, $NDR(2,1) = 4$, and $NDR(2,2) = 6$. Figure 3.6 shows the frequency of $NDR(n,5)$ values across all values for n in graph shown in figure 3.1. About 86% of subgraphs spanning 5 nodes have a deviation less than 10 neighbors between the most and least connected graphs. This graph illustrates how often a particular range in the number of neighbors at each node

- In both topology-driven and data-driven workloads which use distributed worklists, thread blocks with a given block index traverse graphs smoothly (that is, nodes accessed within in a short amount of time are logically near one another in the graph) even across separate kernel invocations. This is not the case with data-driven implementations which use a single shared worklist, as which particular subset of data accessed by a given thread block will be completely random across separate kernel invocations depending on the precise interleaving of other thread block's access to the worklist.

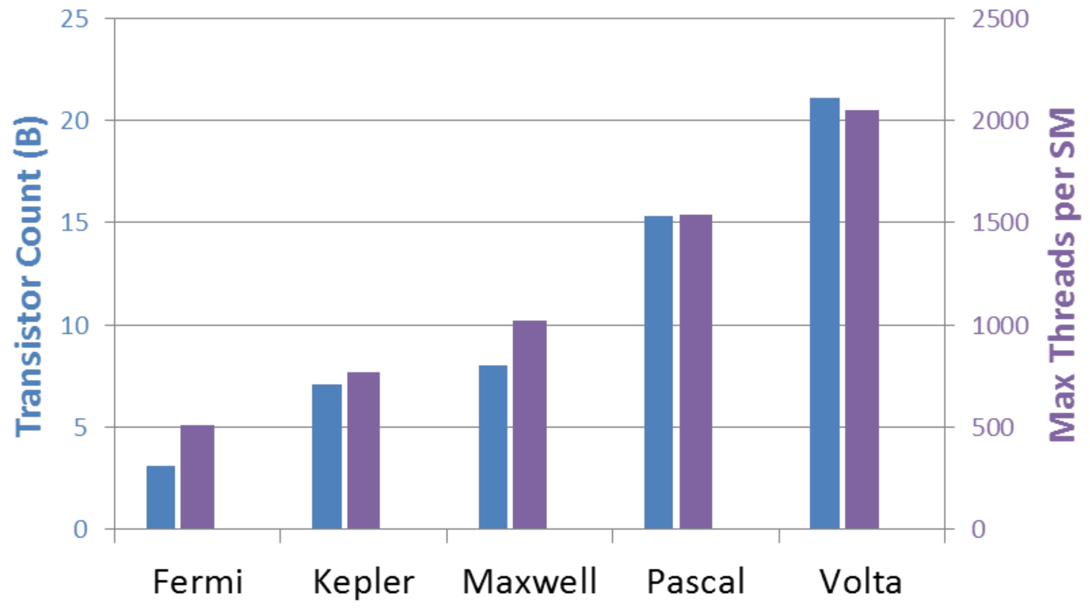


Figure 3.4: Increase of transistor account with each successive generation of NVIDIA GPU compared to the maximum number of threads supported per SM

These observations suggest that we can predict how much of a particular resource a pending thread block is likely to use in the near future based on how that resource was used by previous instances of the same thread block (i.e. have the same block index). By tracking the resource usage via performance counters already present on modern GPUs, we

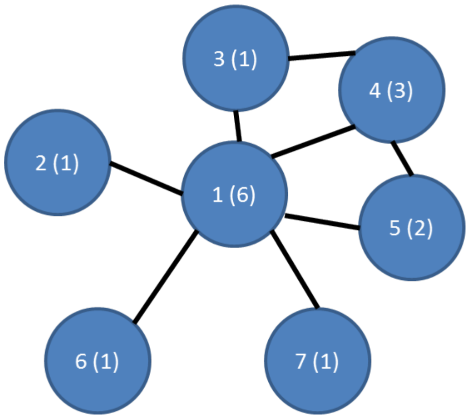


Figure 3.5: Sample graph with labels formatted as “Node Index (Number of Neighbors)”

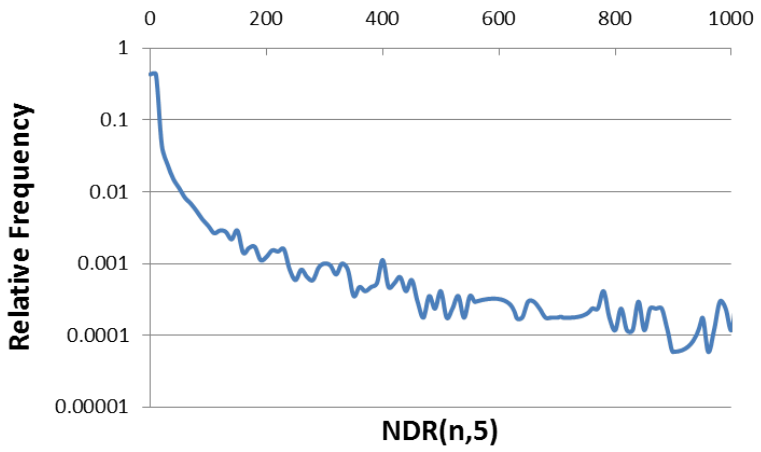


Figure 3.6: Relative frequencies of NDR values across 5-node diameter subgraphs in Netflix dataset

can feed this information to the thread block scheduler and more accurately place thread blocks on the ideal SM.

However, this is not a general solution. Table 3.2 lists several properties of kernel iterations in the LonestarGPU benchmarks when processing a fixed-size data set. One of these is the average execution latency, or how long each kernel instance runs on average before returning. DMR, MST, PTA, and SGD all invoke kernels several times across its execution, with the number of iterations depending on the size and structure of data. BFS and SSSP only invoke their respective kernels once, and as such their latencies make up most of the application’s run time.

The changing density patterns in the input data result in the emergence of “phases” where a thread block’s resource usage changes over time. For the purposes of this chapter, we define a phase shift to have occurred when a given metric of interest (namely, the rate of memory throttle stalls or utilization of the FPU) averaged across a window of the last

Application	**Grid / Block size**	**Regs per thread**	**SMem per block (B)**	**Total context size (KB)**	**Preemption latency (μs)**	**Execution latency (μs)**
BFS	240 / 128	32	0	16	3.0	56,000
DMR	112 / 512	32	0	64	11.8	9,080
MST	640 / 256	34	0	34	6.2	1,030
PTA	80 / 991	28	200	108	20.1	40
SSSP	160 / 256	32	0	32	5.9	23,000,000
SGD	900 / 16	26	68	2	0.3	10

Table 3.2: Average Iteration Metrics per Application

100 μs deviates by more than 5% (choosing a smaller window size for the running average is unlikely to be of use, since, as we will explain later in this section, there will be no way to effectively respond to this variances on this time scale). The exact property of these phases varies widely based on the benchmark as well as size and structure of the input data. We provide one example of the distribution of how long these phases last when executing the SGD benchmark in figure 3.7. There is not a clear upper bound on the length of these phases, but the majority last between 1 and 10 ms, with an average of about 5.4 ms. Across all benchmarks and several input data sets, the average phase length varies between 0.1 ms and 100 ms.

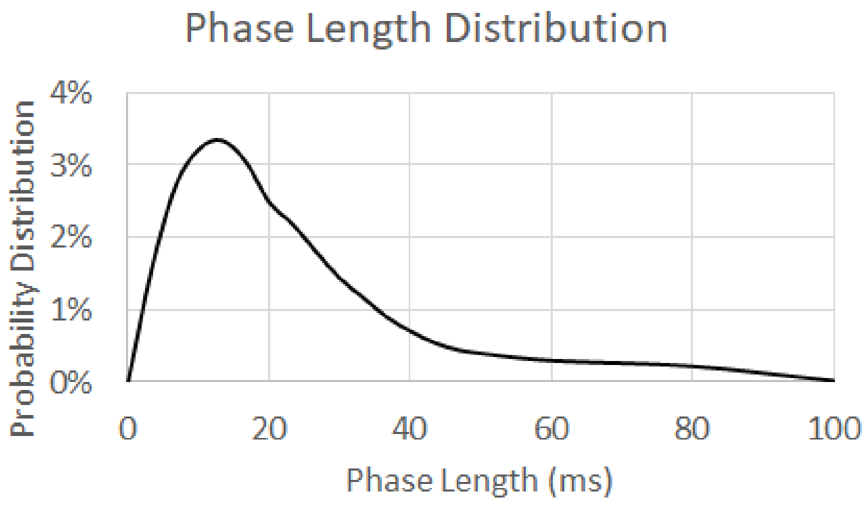


Figure 3.7: Distribution of phase lengths when running the SGD benchmark

The key observation is that while half of the benchmarks (SGD, PTA, and to a lesser extent MST) have average latencies on the order of or significantly less than the typical phase lengths we observe and can therefore be rescheduled at a fine-granularity at the next invocation of that thread block, the other half (DMR and to a much larger degree BFS and SSSP) have average latencies significantly higher than typical phase lengths. Waiting to make allocation decisions at thread-block dispatch may potentially miss several phase changes entirely. Indeed, since BFS and SSSP only invoke their kernel once, we have no hope of adjusting the allotted resources based on phase changes at kernel dispatch.

Thus, the second part of an ideal resource allocation strategy is to detect phase changes in real time, preempt thread blocks which are not ideally placed, and relocate them to a location that is better suited.

3.4 Design

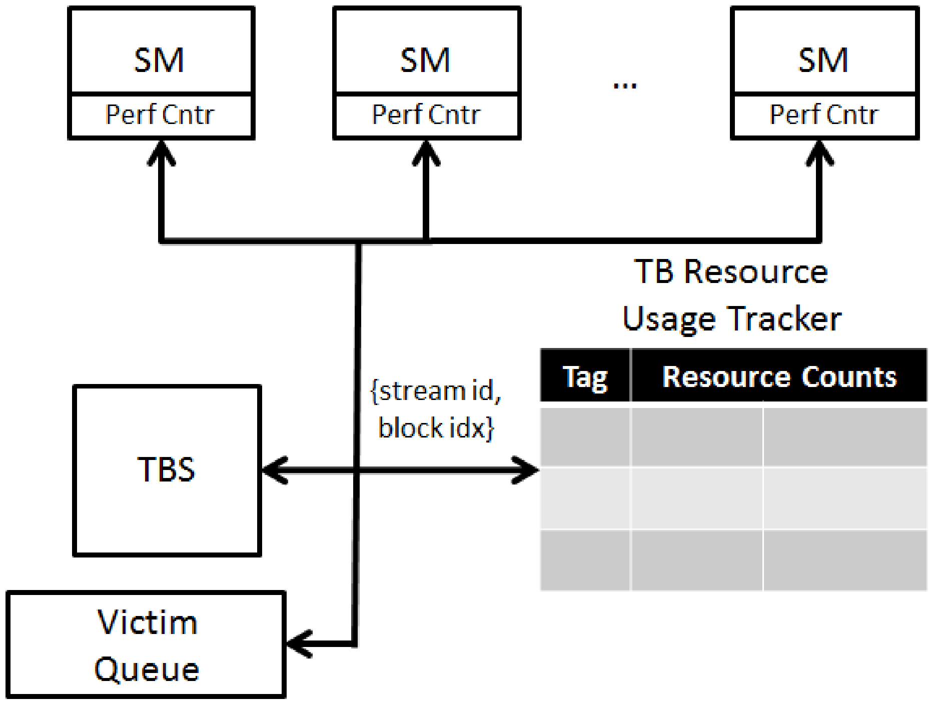


Figure 3.8: Diagram of the proposed system architecture with the SMs, thread block scheduler, and the proposed victim queue and thread block resource usage tracker

We use all of the discussed observations to design a dynamic thread block scheduling and preempting infrastructure to better approximate an ideal solution. The architecture is summarized in figure 3.8. In order to account for resource usage variance across thread blocks, we propose the inclusion of a *thread block resource tracker (RUT)* which is implemented as an SRAM table in hardware. The RUT is indexed with both the stream ID and block index and stores an N bit unsigned integer for each utilization metric to be tracked by the TBS. The reason why the stream ID is used instead of the kernel ID is that applications frequently use a pipeline of kernels with identical dimensions to process data in several steps, where each corresponding thread block with the same block index processes the same subset of data and therefore exhibit similar resource usage trends.

In addition to resources such as shared memory and register file usage, the TBS will also estimate different resource usages through the RUT and use these estimates as additional constraints when placing thread blocks.

When allocating a thread block, the TBS will:

- access the RUT with the stream ID and block index to receive estimates on key resources
- find the first SM (using a rotating round robin policy) that has enough register file, shared memory, and supplemental resources to accommodate the current thread block
- if no SMs meeting that criteria are available, thread block dispatch is stalled until one becomes available

Periodically, each SM will check the resource usage of of each resident thread block using preexisting performance counters. If any tracked resource has deviated over a certain threshold since the previous check, the SM informs the TBS, which then updates the appropriate RUT entry. Each entry will also be updated when the given thread block terminates. Care must be taken to choose the right frequencies and parameters so that the system is not overrun with traffic from updates.

Whenever the RUT is updated by a still-running thread block, the TBS will reevaluate the resource usage of the SM. If the estimated resource usage has increased such that the SM is predicted to be oversubscribed, the SM is instructed to preempt the thread block in question, the context is swapped to memory (see next subsection for details), and meta-data for the thread block is pushed onto a *victim queue* maintained by the TBS, if it is not full. If the victim queue is full, no thread blocks are preempted until space is made available.

If, upon update of the RUT, the estimated resource usage has decreased, the thread block searches through the victim queue in last-in-first-out order for a thread block that is not expected to oversubscribe the SM and context-swaps the thread block into the specified SM. If no thread blocks in the victim queue meet the criteria, the TBS checks to see if the next thread block which has not yet started execution will oversubscribe the SM, and issues if appropriate. The victim queue is searched first and in a LIFO order to take advantage of locality in the event that the thread block is issued to an SM which shares part of the cache hierarchy of the previous SM.

3.4.1 Thread Block Preemption

Preempting thread blocks on a GPU has been demonstrated as an effective way to ensure quality of service when running multiple kernels [32][33][34]. Preemption strategies usually fall into one of three classifications:

- Context switching: Due to the massive number of threads present on SMs, true context switching is generally avoided. It requires storing the register file and shared memory to a designated location in memory, which greatly increases the latency of preemption [34].
- Draining [32][34]: Draining allows all currently running thread blocks to complete until the SM is empty, and then swaps in a new set of thread blocks. As has already been established, waiting for thread blocks to finish is too costly for long running kernels such as BFS and SSSP to be a general solution.
- Flushing [34]: Flushing drops the currently running thread blocks and restarts them at the beginning of their execution on a new SM at a later time. This ensures low latency swapping, but requires certain properties of the kernel at the point of preemption: it must not have made any stores to memory that could affect the values of its earlier loads, and it cannot have executed atomic operations. Otherwise, the thread block may have different execution when rerun from the beginning. Because many irregular workloads are morph algorithms (e.g. DMR and MST) which alter the input data structures as they execute, they may not meet this criteria and flushing cannot be used.

Context switching is the only general solution and is what we use for our solution. Table 3.2 includes the average total context size for a thread block in each application, which is calculated as the amount of shared memory per block plus the thread block size multiplied by the number of registers per thread, multiplied by the register size (32 bits). The table also includes the average latency added per context switch, which was measured in our simulation infrastructure (see section 3.5 for details). As is apparent, the latency associated with preempting thread blocks in each benchmark is still an order of magnitude lower than the typical phase lengths we observe and can therefore be a reasonable cost to pay if thread blocks are better co-located and resource interference can be sufficiently reduced.

System	Configuration
SMs	80 SMs, 5120 Cores, 1.6GHz Max of 2K threads per SM Max of 1K threads per TB Max 64K registers per block Max 48 KB shared memory per block 128 KB L1
Memory Subsystem	6 MB L2 900 GB/s bandwidth
Thread Scheduling	GTO

Table 3.3: GPGPU-sim simulation parameters used for evaluation

3.5 Methodology

We evaluate our design using GPGPU-Sim version 3.2.2[35] whose configuration is described in table 3.3, meant to emulate the Tesla V100 we used in previously described experiments. We modified the simulator to enable multi-kernel execution and preemption with context switching, and co-ran pairs of applications from the LonestarGPU benchmark suite Version 3.0 [2]. Because the LonestarGPU benchmarks are predominately memory bound workloads, we also co-ran the benchmarks with samples from the Parboil benchmark suite [36] which includes a more diverse set of memory and compute bound workloads. For all pairs of workloads we measure the change in overall throughput (measured in instructions committed per cycle) of the LonestarGPU kernels. We compare our solution against two variants:

- A scheduler which has oracle knowledge of the average resource usage of the kernel as a whole for use in scheduling thread blocks, but does not track the dynamic behavior of the kernel and does not employ preemption. This provides a fair comparison to works such as Xu et al [37] which estimate kernels performance characteristics as a function of their input size.

- A scheduler which tracks the dynamic behavior of kernels as a whole and preempts when oversubscription is detected, but does not track individual thread blocks. This provides a fair comparisons to works such as Park et al [27].

For input, we used datasets synthesized using the Graph500 R-MAT generator [38] using its default parameters of (A=0.57, B=C=0.19) which were used in recent analytic works on power-law graphs [39][28]. The number of edges is set to 100,000 and the number of vertices is swept between 5,000 and 80,000.

As a heuristic, we have the SMs query the performance counters once every (context_size_in_KB / 1.1) μs, which roughly equates to five times the expected preemption latency. This information can be communicated via the TBS upon thread block dispatch. We set the size of each RUT entry to be 3 bits per metric of interest (rate of memory throttle stalls and FPU utilization), and set the total size of the table to be 16KB or about 21K entries, more than enough to ensure virtually no conflicts between different kernels. Any deviation detected with this metric over that time period will be communicated back to the TBS. We set the victim queue to be large enough to hold 1K entries.

3.6 Evaluation

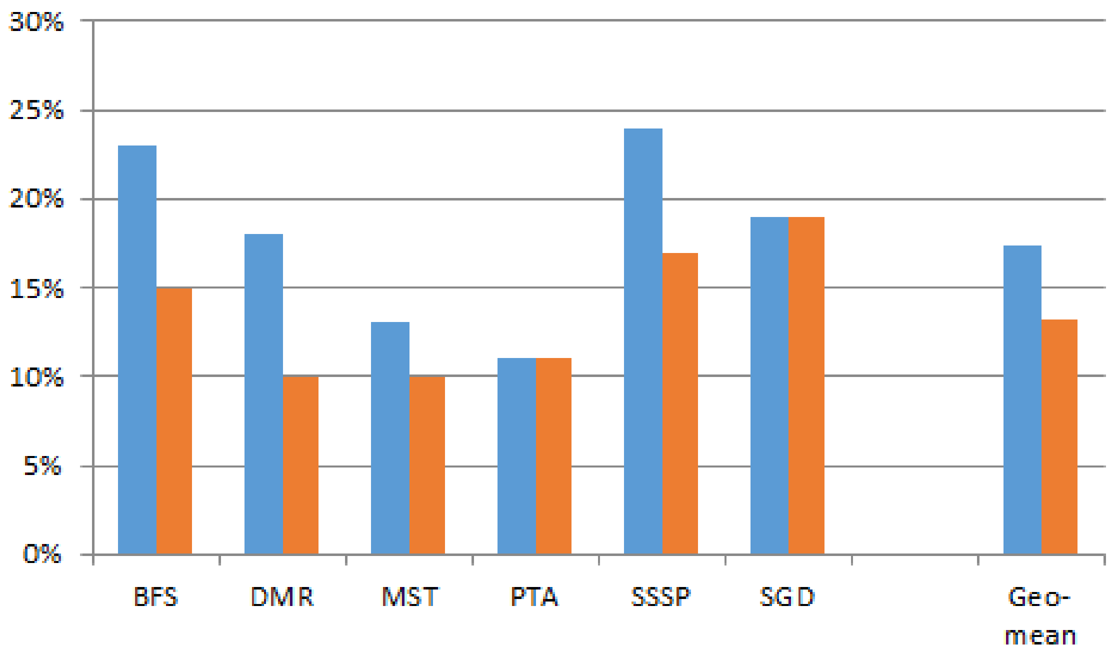


Figure 3.9: Throughput increases over oracle static classification (blue) and kernel granularity profiling (orange)

Figure 3.9 shows the throughput increase of our design over both the oracle static classification scheme and the kernel-wide monitoring solution described in section 3.5. Each result is the average increase over all pairings and input datasets involving that benchmark. Our design, across all benchmarks, achieves an average increase in throughput of 17.1% compared to the static scheduler and 12.9% compared to the kernel-wide profiler. The magnitude of the throughput increase correlates with the range of metric deviations reported in figures 3.2 and 3.3, as benchmarks with higher variance pose the greatest hazard for interference as the data patterns change. In most cases, we achieve a better improvement over the static scheduler than over the profiler. This is because the profiler, although it does not track individual thread block metrics, can still detect coarse-grained average changes in the data set across the whole kernel and can preempt the thread blocks in cases where the variances do not cancel each other out. The kernel profiler does not perform any better than the static scheduler for either PTA or SGD, as these benchmarks consist of kernels too short for thread block preemption to be effective. To better understand the efficacy of our design, we isolated several features.

3.6.1 Cataclysmic Thread Purging

Figure 3.10 shows a possible outcome of context swapping when the system is not calibrated properly. In this case, during the process of writing a thread block's state to memory, it is cached in the L1 and L2 levels of the memory hierarchy. This results in a generalized form of cache thrashing, where not just the contents of memory are brought in and out of the cache in an unstable manner, but also the contexts of the threads themselves.

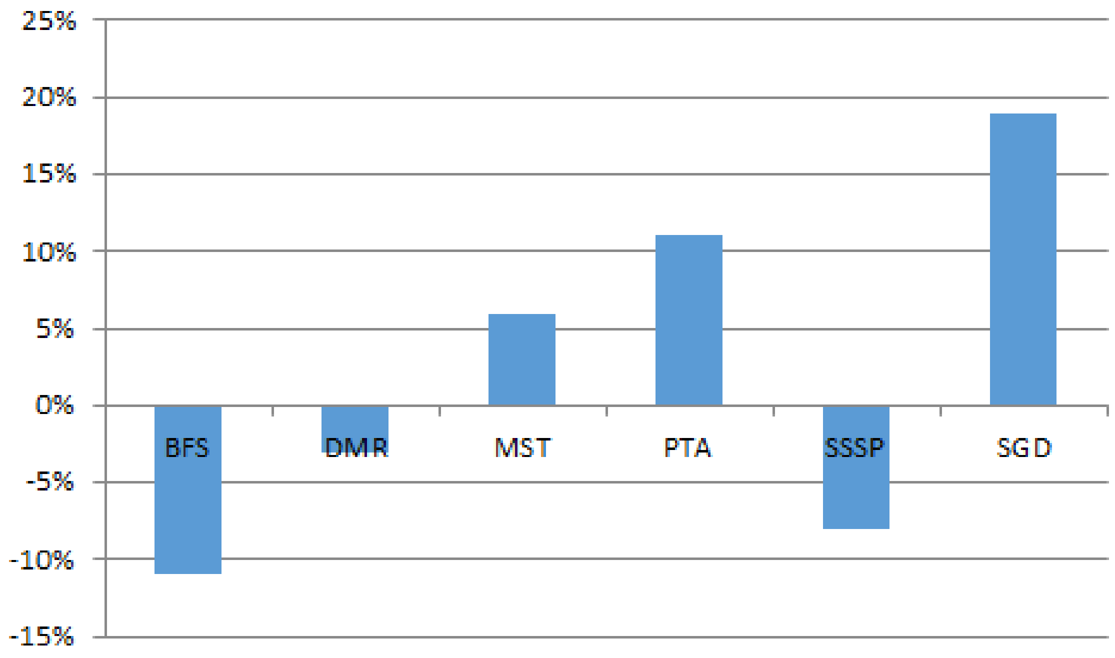


Figure 3.10: Negative impact of L1 and L2 caching during context swapping

As shown in figure 3.2, many of the kernels have context sizes which are a significant portion (10% or more) of the L1 cache. When cache entries are allocated as a result of saving or restoring a thread block's context, it evicts a significant portion of the cache. In memory-intense workloads, particularly BFS and SSSP, this has the effect of significantly increasing pressure on the load-store units and increasing the probability that further thread blocks are preempted as resources are saturated. As more thread blocks are evicted, more of the cache is displaced and still more pressure is placed on the memory system, resulting in a chain reaction where a large portion of the resident thread blocks are evicted. We refer to this phenomena as *cataclysmic thread purging*. Those benchmarks that do not make as much use of preemption (e.g. short running kernels such as PTA and SGD) see less effects, but the overall throughput improvement across benchmarks is reduced to just 2%.

As a result, our design marks context saving and restoring memory operations as "non-cacheable" to prevent such an event. We explore the possibility of relaxing this constraint in chapter 4.

3.6.2 Effectiveness of Predictive Scheduling versus Context Switching

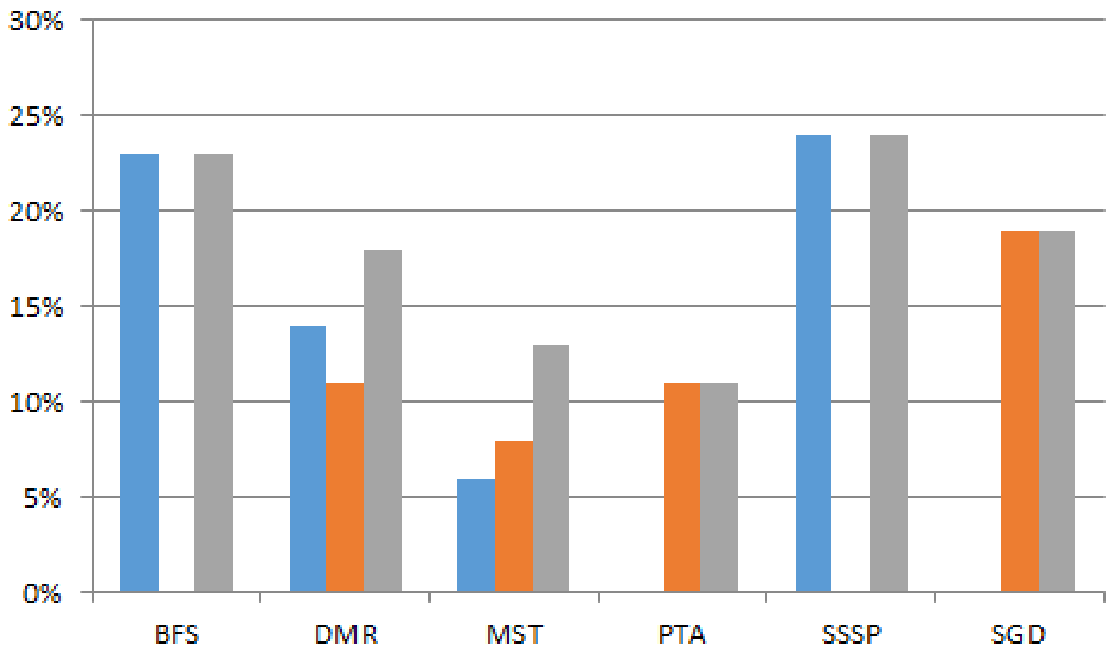


Figure 3.11: Throughput increases over static classification when employing only preemption (blue), only predictive scheduling (orange), and both (grey)

Figure 3.11 shows our design's performance when employing only the preemption or predictive scheduling strategies in isolation, and then using both together. BFS and SSSP achieve all of their throughput increases through preemption, as they only invoke their primary kernel once and so can't use the dynamic metrics to schedule future thread blocks more efficiently. Although the kernel profiling strategy is able to respond to global changes across the entire kernel, it cannot respond to deviations when individual thread blocks enter or exit dense regions of data. At the other extreme, PTA and SGD's kernels are

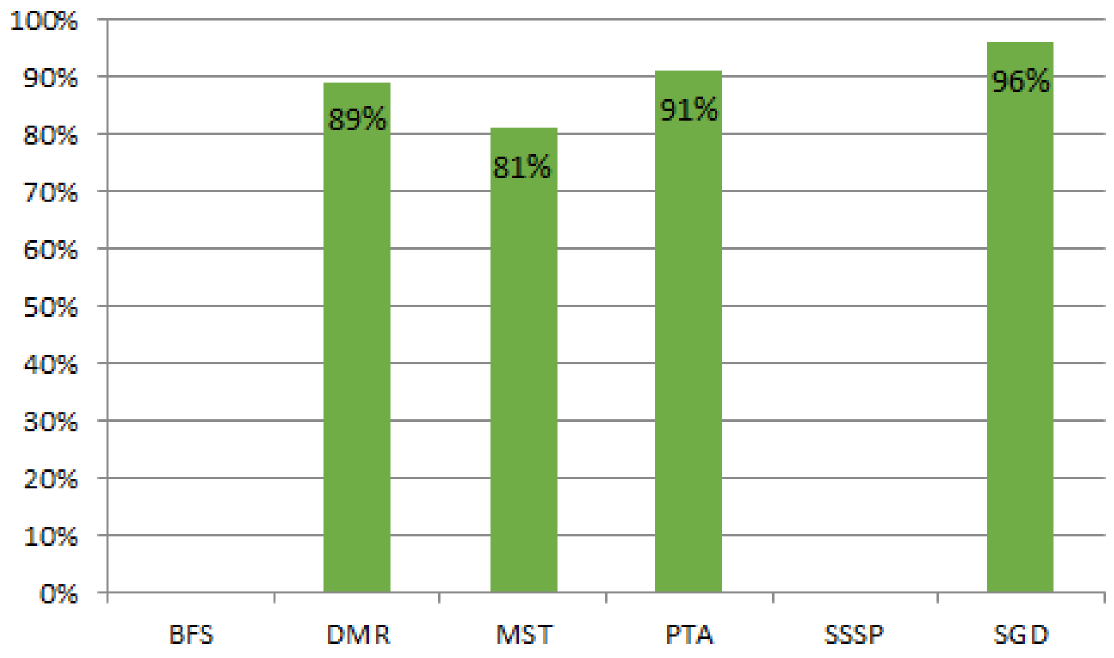


Figure 3.12: Measured correlation of average density processed by two thread blocks with the same block ID across subsequent kernel invocations, which indicate the effectiveness of predictive scheduling

too short to use preemption effectively and achieve all of their performance gains through scheduling enhancements. DMR and MST receive moderate speedups due to preemption, but are limited by their relatively short execution latencies and are less able to amortize the overhead of context switches.

To illustrate the effectiveness of predictive scheduling, figure 3.12 shows how well resource utilizations correlate across kernel invocations. Predictive scheduling relies on the assumption that thread blocks with the same block ID will enter similar regions of code, i.e. we expect these correlations to be high. BFS and SSSP of course have undefined correlations since they do not have multiple kernel invocations. SGD has a very high correlation of 96%, as it is a topology driven benchmark which is guaranteed to have the same nodes processed by the same threads each iteration. Small deviations occur as the graph is morphed over time, but slowly relative to the large number of pipelined, low-latency kernel invocations. DMR and PTA are also topology driven benchmarks, but morph the underlying data structures more rapidly relative to the frequency of kernel invocations, and such see a smaller but still substantive correlations at 89% and 91%, respectively. MST sees the lowest correlation at 81% as it is a data driven algorithm and thus see a greater variance in which thread blocks process which nodes. However, the use of distributed worklists keeps the correlation relatively high.

These correlation values illustrate why certain benchmarks are able to achieve speedups when using predictive scheduling in figure 3.11. The higher the correlation between of the input density across kernel invocations, the more accurate the resource predictor will be. SGD, with its high correlation, receives the most speedup. MST, with the smallest correlation, receives the least speedup. DMR and PTA receive moderate improvements, again in alignment with their correlation values.

Most benchmarks receive the vast majority of their speedup from one strategy or the other and don't receive much added benefit from using both. MST is the exception, as the two strategies are often able to "catch" opportunities to better schedule threads missed by the other. On occasions where a thread is not placed on an optimal SM due to variance from worklist distribution or otherwise, preemption can still relocate the thread block, albeit with higher overhead than the other benchmarks.

3.6.3 Memory Tracking versus Compute Tracking

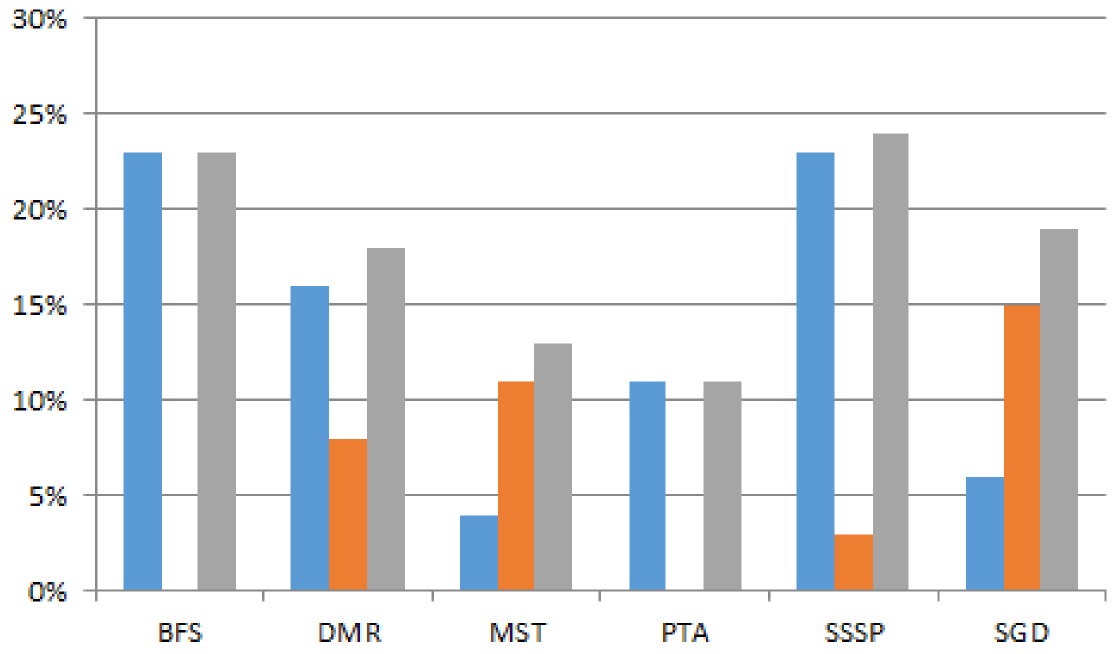


Figure 3.13: Throughput increases over static classification when only tracking memory throttle stalls (blue), only tracking FPU utilization (orange), and tracking both (grey)

Figure 3.13 shows the performance increase when only tracking memory stalls, FPU utilization, or both together. For most benchmarks, tracking memory stalls is much more profitable than tracking FPU utilization, which is expected as these are memory bound workloads. In particular, BFS and PTA have no floating point code. SGD and MST contain more complicated operators which make variable use of floating point operations depending on the input data structure and see more benefit from tracking FPU utilization. Although DMR and SSSP make use of floating point operations and the intensity does vary based on input structure, this variance is a largely a side effect of changing memory utilization: fewer memory stalls increases bandwidth, keeping the other functional units more occupied as a result and vice-versa. Accordingly, these see less benefit to tracking both metrics compared to other workloads, as tracking one implicitly gives information on the other.

3.6.4 Overheads

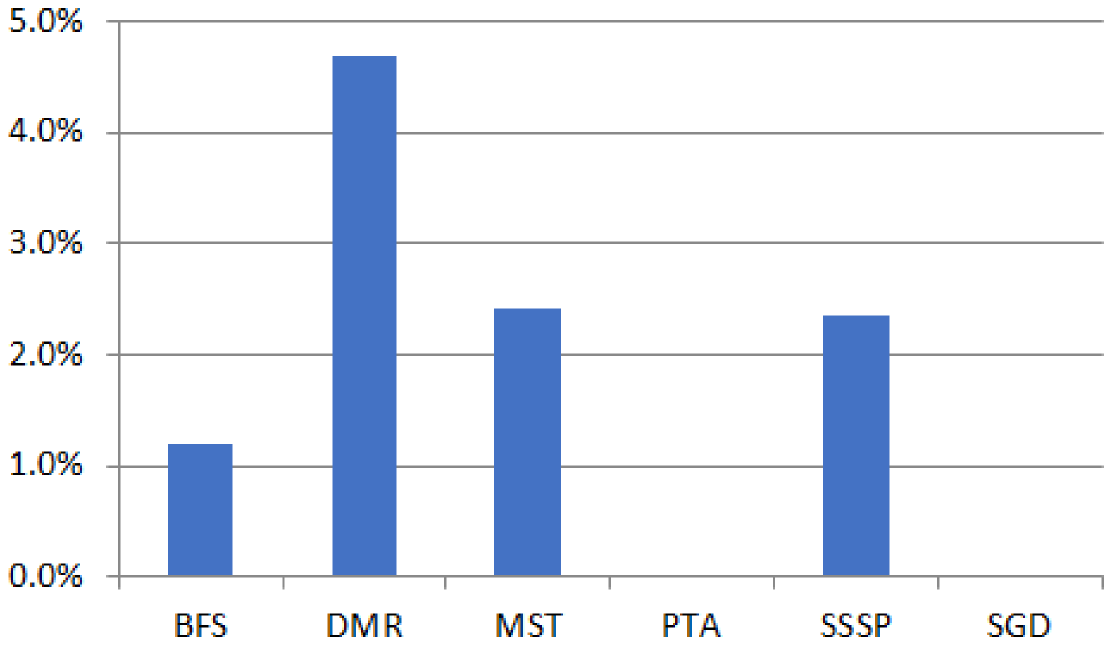


Figure 3.14: Throughput overhead of preempting threads

Figure 3.14 shows the throughput overhead of supporting preemption and context migration. Excluding benchmarks which do not make use of it, we see an average overhead of 2.7%. This is smaller than the overheads observed by Park et al [34] on the order of 10% because:

1. The amount of register and shared space allocated by these benchmarks is smaller than the benchmarks explored in their work

2. Preemptions in their work occur on the order of $10 - 100\times$ more frequently than in this work due to typical phase lengths

By choosing appropriate parameters for our design, we have ensured significant performance improvements with small overhead. The increased memory traffic associated with updating the RUT and context switching across the whole GPU is 13 KB/s and 1 GB/s, respectively, which accounts for less than 1% overhead on the total available throughput of the system. The RUT and victim queue together account for 20 KB of extra SRAM shared across all SMs, which adds under 0.5% the overhead of the L2 cache.

3.7 Related Work

Thread block preemption has been recognized as a powerful solution towards better resource management. Tanasic et al [32] suggest hardware extensions to make preemption on GPUs more efficient. Park et al [34] demonstrate how combining several preemption techniques and dynamically choosing between them can reduce latency and improve throughput in multiprogrammed workloads.

Many works have explored the problem of resource contention in GPUs and how to prevent it. Kayiran et al [40] explored how changing the number of thread blocks issued by compute versus memory intensive workloads reduces resource contention. Pai et al [26] and Zhong et al [20] investigate alternative static and dynamic methods of modifying kernel's execution properties to enable better utilization.

This work followed the lead of many others in exploring more sophisticated schedulers. Li et al [41] propose a co-design between the thread scheduler and cache allocation scheme to avoid cache contention without underutilizing other resources. Sethia et al [42] propose augmentations to the scheduler to prioritize memory requests from irregular, memory intensive workloads to reduce stalls. Zhao et al [43], Xu et al [37] and Jiao et al [19] investigate classification of kernels to enable better thread block placement. Park et al [27] develop a strategy to dynamically monitor a kernel's execution and preempt thread blocks when appropriate. Wu et al [44] and Chen et al [45] investigate solutions at the program and runtime level to enable better thread scheduling and preemption. To our knowledge we are the first to analyze how thread scheduling is impacted by the issues of irregular algorithms, and the first to develop a solution of tracking resource usage at the thread block level.

3.8 Conclusion

Processing diverse data patterns in irregular workloads results in significant resource usage variance across different threads. However, gradual changes prevalent in practical data sets make it possible to respond dynamically to these changes and modify scheduling decisions. By adding a hardware table to track metrics for each thread block as well as a queue to hold preempted thread blocks, we can effectively reorganize threads in response to changing data patterns. By increasing SRAM storage requirements by less than 0.5%, we can gain 17% and 13% improvements of throughput over previously proposed static and dynamic scheduling strategies, respectively.

CHAPTER 4

Block Shuffling with Reconfigurable Hardware

4.1 Introduction

Different workloads map better to different configurations of architectures. We have seen in previous chapters how programs that exhibit regular parallelism map well to SIMD architectures, while programs with irregular parallelism struggle to maintain performance on the same platform. Out-of-order CPUs excel with serial code or workloads that require low-latency execution, and custom accelerators can be designed for other, more specific workloads. Heterogeneous systems containing several of these different architectures are often used with explicit control transfers between them to handle different execution cases. However, these control transfers are costly, requiring long latency transmissions over system buses which may take on the order of milliseconds to complete. These transfers are therefore done at a coarse level, and execution phases lasting on shorter scales may not benefit from such heterogeneity.

An alternative approach is the design of reconfigurable architectures: architectures that, when triggered either by software or hardware events, can activate, deactivate, or otherwise repartition resources across compute units, allowing workloads to achieve better performance without the need to transfer data between different processors or accelerators. We have explored such a design which organizes processing elements and memory nodes into tiles whose connections can be reconfigured via a high-speed crossbar. In this chapter, we demonstrate how such a design can be used to achieve significant ($8\times$) speedups when executing sparse matrix multiplications, a workload which exhibits both regular and highly data intensive phases, as well as irregular and memory intensive phases.

We then go on to demonstrate that the same mechanisms within the architecture can allow for very low latency transfer of threads between compute elements in a process we call *block shuffling*. This process improves transfer latency by 98% compared to naïve strategies. This can be used to augment the strategies described in chapter 3 and create a highly flexible architecture which can respond to any change in execution conditions with very low latency. We demonstrate a 24% increase in throughput when using block shuffling over other schedulers on the same architecture.

Section 4.2 lays out the details of the reconfigurable architecture we have considered for accelerating more complex workloads. In section 4.3, we analyse a case study of mapping sparse matrix-matrix multiplication to this architecture and how reconfiguring resources to changes in the workloads' properties can increase utilization and power efficiency. Finally, section 4.4 describes how our block shuffling algorithm works in tandem with context compression techniques to achieve sizable speedups on irregular workloads.

4.2 Architecture

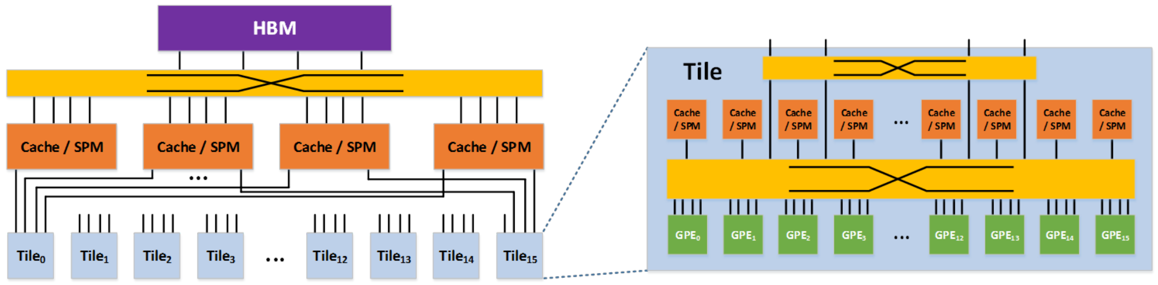


Figure 4.1: Overview of the reconfigurable architecture

The reconfigurable architecture considered in this chapter, shown in figure 4.1, consists of a series of tiles connected to a memory hierarchy. Each tile consists of a set of General-Purpose Processing Elements (GPEs), alternatively referred to as simply Processing Elements (PEs), a tile manager to distribute work (not shown), a set of memory modules that can be reassigned as caches or scratchpad memories, and a pair of reconfigurable crossbars.

4.2.1 Processing Elements

Each GPE is a simple, in-order core. It receives an instruction stream to execute from the tile manager. Each contains a integer and floating-point unit, as well as load and store pipelines. These can be individually power-gated by the tile manager to save power during low compute phases of computation if they are not expected to be fully utilized.

4.2.2 Caches

Each memory bank can be reconfigured by the tile manager to behave as a cache or a scratchpad memory (SPM) by deactivating the tag banks, allowing software to explicitly address certain elements. The crossbar connecting the GPEs to the memory banks can be configured to either allow each GPE to access its own, private memory, or a shared pool amongst other GPEs in the tile. This allows the architecture to easily switch between a shared memory architecture and a non-shared one.

4.2.3 Crossbar

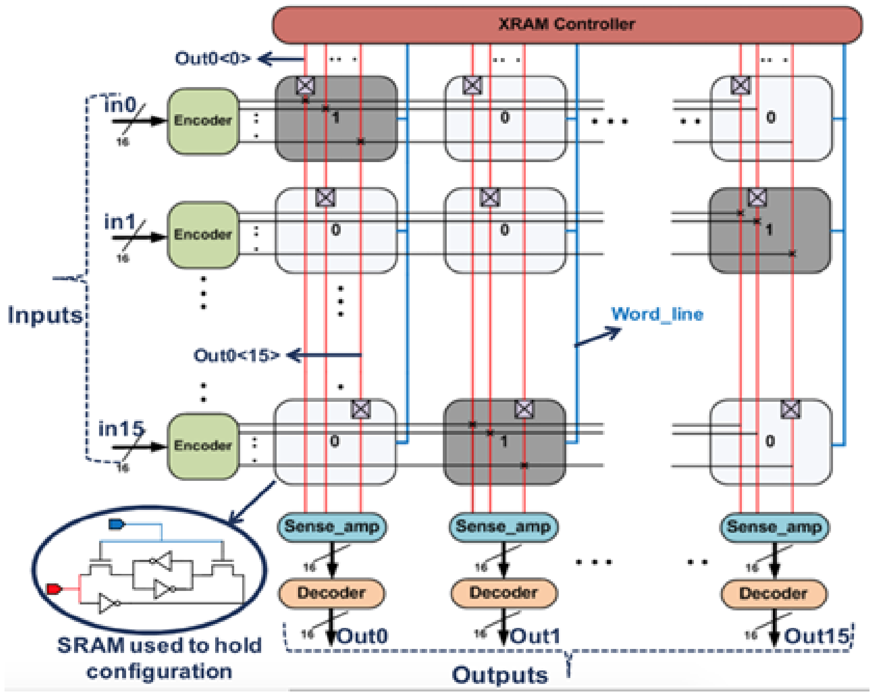


Figure 4.2: Crossbar design [3]

The crossbar is modeled after a design proposed by Sewell et al [3] and is shown in figure 4.2. It contains a programmable controller which allows for arbitrary connections between input ports and output ports. Common configurations include a simple passthrough scheme where input i is connected to output i, or broadcast where each input is connected to every output. The tile manager can reconfigure the controller within 10 cycles.

4.2.4 Memory Hierarchy

The architecture contains two levels of reconfigurable caches. The first level consisting of isolated partitions within each tile and the second level shared across all tiles. A reconfigurable crossbar exists in between the GPEs and first level cache, as well as in between the first and second level caches. The crossbar in between the caches allows for data sharing between tiles when necessary without requiring main memory to be accessed. A standard, non-reconfigurable crossbar is placed between the second level cache and off-chip main memory.

4.3 Case Study: Sparse Matrix-Matrix Multiplication

As a primary demonstration of the benefit of this architecture, we present the performance of a workload well suited for its reconfigurability: sparse matrix-matrix multiplication based on outer product summations. Sparse matrix-matrix multiplication is a key algorithm in many data analytic workloads [46] [47] [48] [49]. The proliferation of power-law graphs in big-data has resulted in typical inputs consisting of small, dense regions and larger, sparse regions (see chapter 3).

Matrix-matrix multiplication is traditionally implemented as a series of inner products between rows of the first matrix (**A**) and columns of the second matrix (**B**) to generate a new matrix (**C**):

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{k,j}$$

Such an algorithm consists of regular memory accesses as the matrices are streamed in an entirely deterministic fashion and exhibit simple control flow and high data reuse. As such, matrix multiplication based on this model is one of the most characteristic workloads at which data parallel processors excel.

Unfortunately, this approach has significant disadvantages when operating on sparse matrices. Because most of a sparse matrix's elements are zero, compressed formats are used to only store non-zero values and coordinates to identify the value's location in the matrix. Thus, the regular memory accesses and simple compute operators (multiply-accumulate operations) on the data which map traditional matrix-multiplication so well to architectures like GPUs are replaced by irregular memory accesses and control flow irregularities. Current GPU sparse-matrix libraries achieve less than 1 GFLOPS in throughput when density drops below 0.1% despite a peak theoretical throughput of over 4 TFLOPS [50].

An alternative approach to sparse matrix-matrix multiplication is an algorithm based on outer products.

4.3.1 Outer Product Algorithm

In this algorithm, which is illustrated in figure 4.3, partial products ($\mathbf{C_i}$) are generated by calculating the outer product between the i^{th} column of the first matrix ($\mathbf{a_i}$) with the i^{th} row of the second matrix ($\mathbf{b_i}$), and summing all of the partial products together (**C**).

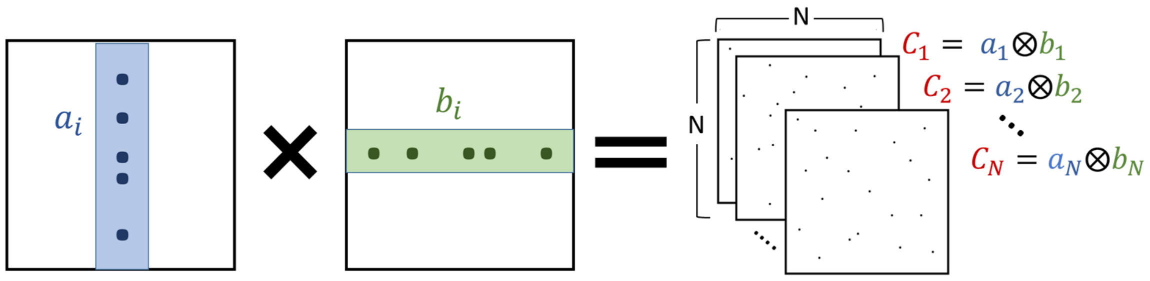


Figure 4.3: Outer Product-based Multiplication

$$\mathbf{C} = \sum_{i=0}^{N-1} \mathbf{C}_i = \sum_{i=0}^{N-1} \mathbf{a}_i \mathbf{b}_i$$

We label the process of generating the outer products as the “multiply” phase, and the process of summing them together as the “merge” phase. The implementation details of the outer product algorithm on a system with two processing units (P_0 and P_1) are shown more explicitly in figure 4.4. Partial products can be represented in either a compressed row (CR) format or compressed column (CC) format. In the CR format, a linked list is maintained for every row of the output, with each node of the lists corresponding to a contiguous row of a partial product generated in the multiply phase. For the CC format, a linked list is maintained for every column of the output. These lists are generated in the multiply phase and then sorted and merged together by their indices in the merge phase.

This approach has the theoretical advantage over inner product methods of minimizing redundant data accesses and eliminating index-matching. During the multiply phase, each data element is loaded exactly once and multiplication is performed unconditionally on every element. During the merge phase, assuming there is enough local storage to hold all the partial product rows for a given output row simultaneously, each piece of data again only needs to be streamed from main memory once. The merge phase does exhibit less data-parallelism, as each row being sorted experiences control divergence depending on the exact layout of the data.

4.3.2 Performance on Traditional Hardware

Despite these theoretical advantages, the outer product algorithm performs poorly on traditional hardware. Figure 4.5 shows how our implementation running on an Intel Xeon processor with 6 threads compares against the Intel MKL SpGEMM library.

The runtime of MKL drops significantly as input density and resulting product size decreases. However, the outer product’s data structures shown in figure 4.4 become less efficient as density decreases, requiring a larger ratio of bookkeeping pointers to matrix data and increasing the data bandwidth.

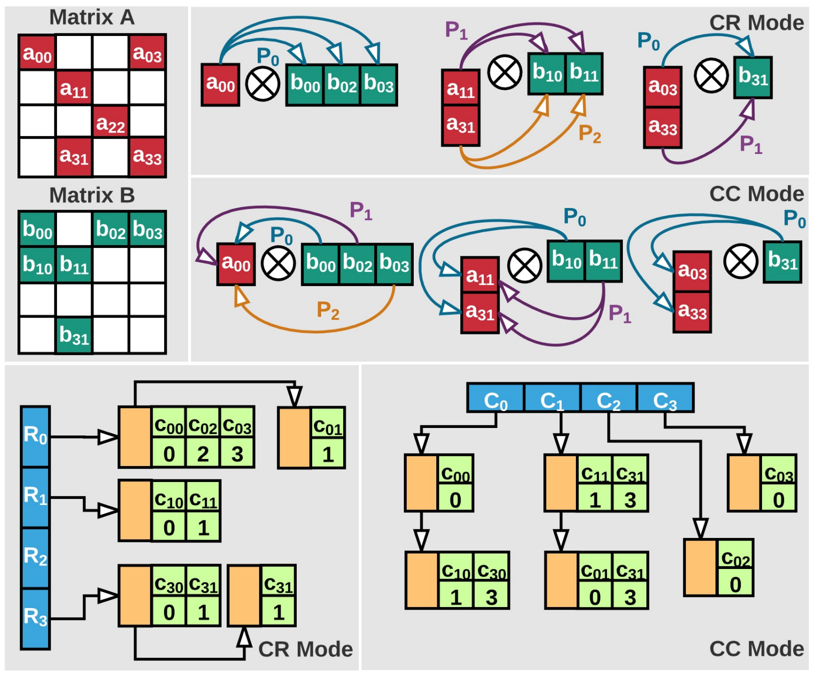


Figure 4.4: Outer Product Mapping

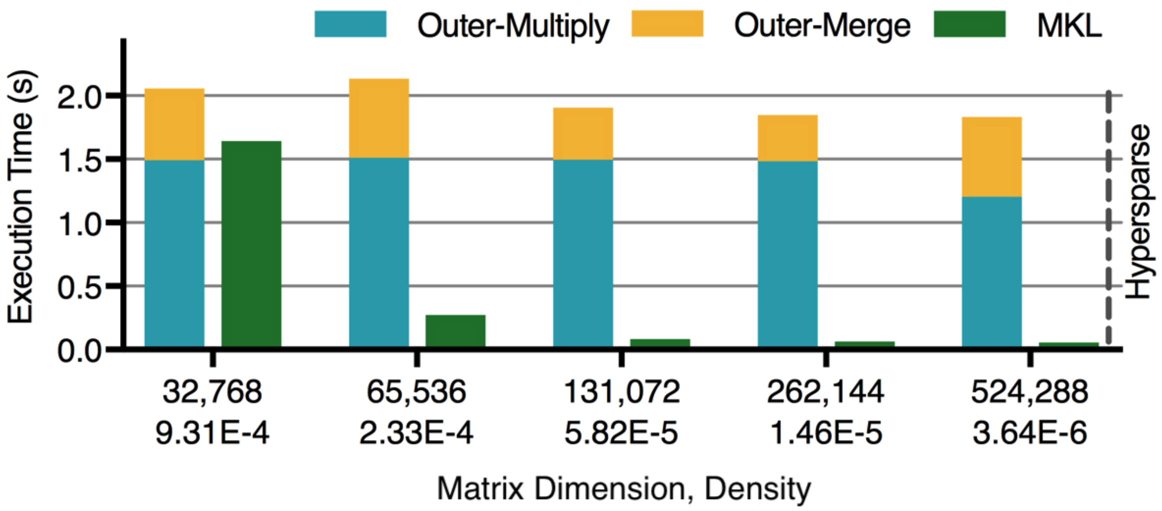


Figure 4.5: Outer product implementation versus MKL implementation on CPU processing uniformly random matrices with 1 million non-zeros

We also compared a CUDA implementation of the outer product algorithm against the CUSP implementation of SpMM provided by NVIDIA when running on an Tesla K40 GPU. The results are shown in figure 4.6.

The multiply phase, which shows high data-level-parallelism ideally suited for a GPU, scales well with density. Both phases achieve high L1 hit rates (>80%) and low data dependency stalls (<5%), but the merge phase suffers from a much lower total throughput. This is a result of numerous conditional branches within the code to handle different relative

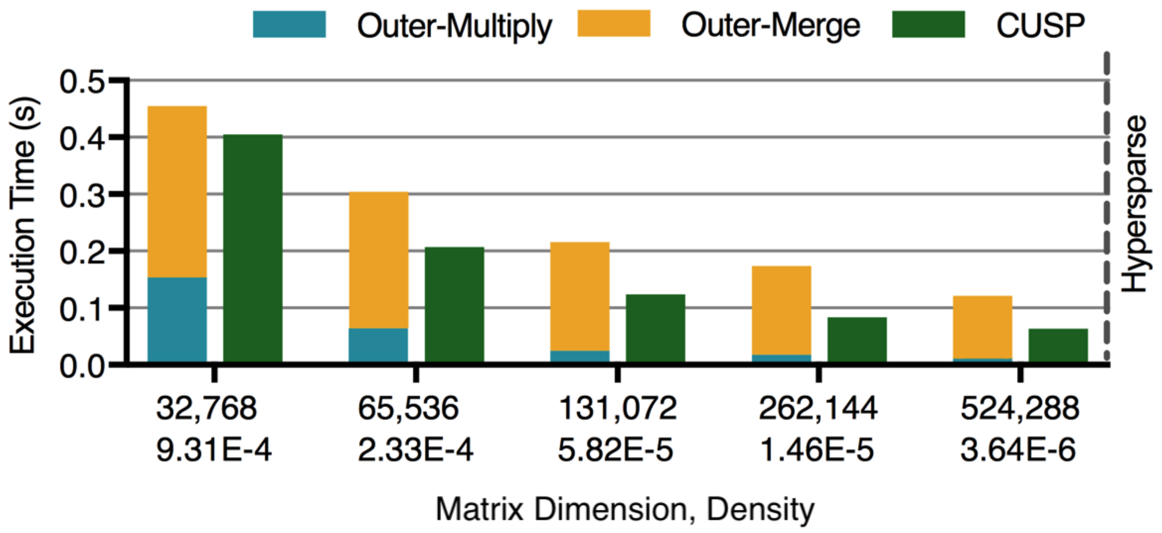


Figure 4.6: Outer product implementation versus MKL implementation on GPU

column indices as they are read in and sorted. Because there is little correlation between adjacent threads as they process these branches, many threads within a given warp diverge and must be executed serially. Thus, while the high degree of parallelism available is attractive, the SIMD nature of the GPU's processing elements prevent an overall improvement of the algorithm over traditional libraries.

4.3.3 Mapping the Outer Product Algorithm

As traditional architectures have been shown to have shortcomings with one or both of the phases of the algorithm, we instead demonstrate how configuring our custom architecture for each phase of the algorithm can provide benefit.

Within the multiply phase, each tile is responsible for generating one partial product. Crossbars connecting the GPEs to the L1 cache are set to broadcast so that individual GPEs can share the values of a row from the second matrix, and each GPE loads its own value from a column of the first matrix, as shown in figure 4.4. Partial products are stored as linked lists, with each row referenced by a pointer from the set R_i. The architectural configuration is shown in figure 4.7.

For the merge phase, multiple GPEs cooperate to merge all the partial product elements that contribute to a particular row of the final matrix. To minimize the number of memory accesses, we merge the lists using the following algorithm (assuming rN rows containing rN elements each to merge, where r and N represent the density and dimension of the matrix, respectively):

1. Fetch the head of each row and sort by column index into a linked list ($\mathcal{O}(r^2N^2)$

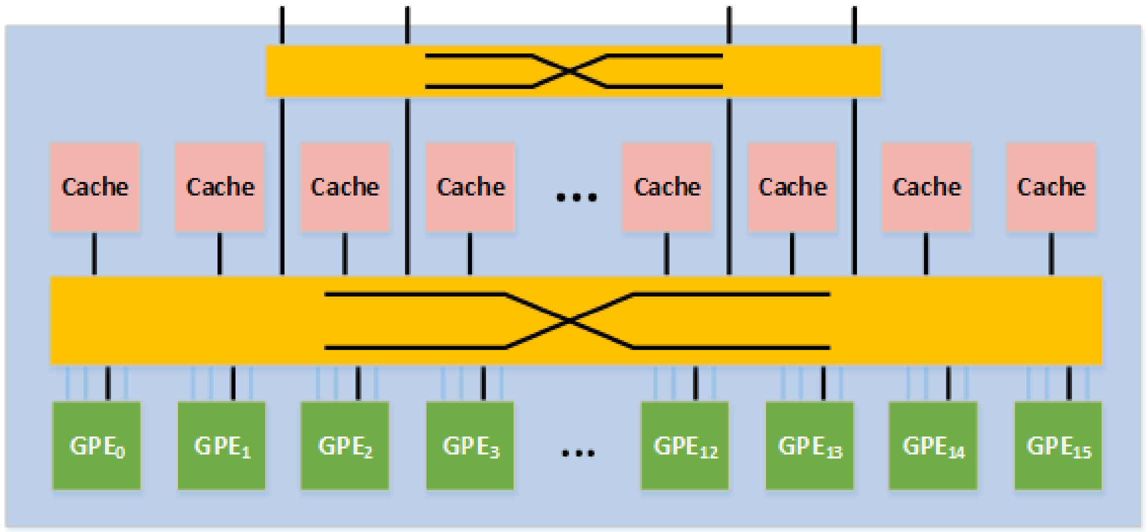


Figure 4.7: Architecture configuration for multiplication phase with global caches, and point-to-point crossbars

operations)

2. Store the smallest-indexed element from the list into the final location, loading the next element from the corresponding row and sorting it into the list ($\mathcal{O}(rN)$ operations)

3. Repeat 2 until all elements of each row have been sorted and shipped to memory (r^2N^2 iterations)

The overall complexity is $\mathcal{O}(r^3N^3)$. While less efficient algorithmically then say, a merge sort, the number of elements stored in local memory is only on the order of rN.

Since each GPE is operating independently with no data sharing, the L1 caches are reconfigured as private SPMs. Batches of GPEs are disabled to throttle bandwidth and save power since the latency is bottlenecked by the ability to read in the partial products. The configuration is shown if figure 4.8.

4.3.4 Evaluation

We simulate the architecture running our outer product implementation using the gem5 simulator [51] with the configurations specified in table 4.1. Work is assigned to the GPEs using a greedy algorithm. Half the GPEs are disabled for the merge phase. Memory traces were generated in a C++ implementation and fed into the gem5 simulator.

We compare against Intel MKL (Version 2017 Initial Release) on the CPU and NVIDIA cuSPARSE (Version 8.0) and CUSP (Version 0.5.1) on the GPU specified in table 4.2. We input sparse matrices provided by the University of Florida Suite Sparse [52] and Stanford

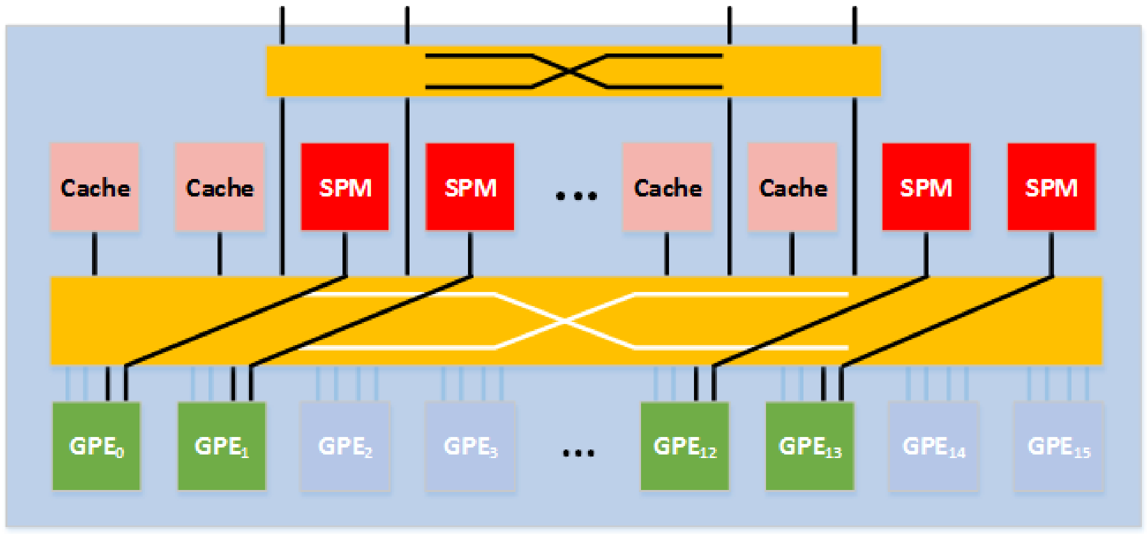


Figure 4.8: Architecture configuration for merge phase with private SPMs, and fixed crossbar connections, and deactivated cores

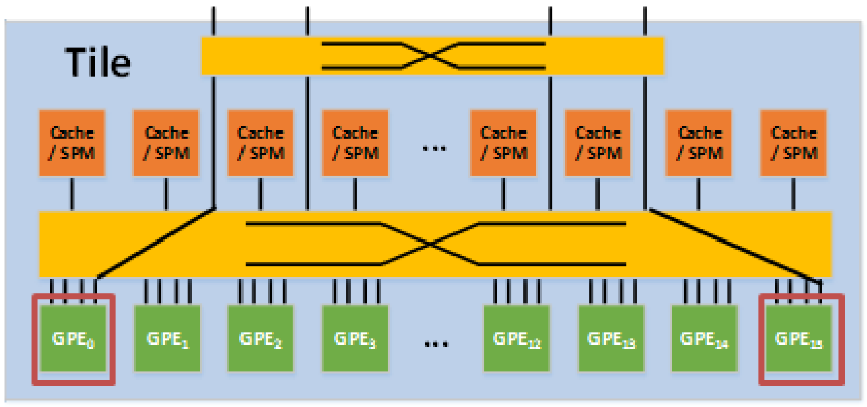


Figure 4.9: Alternate configuration for enabling point-to-point connections. In this example, between GPE_0 and GPE_{15}

Network Analysis Project [53], as well as synthetic matrices generated by the Graph500 R-MAT data generator [38]. We used the default values (A=0.57, B=C=0.19) to generate power-law graphs with the number of edges equal to 100,000 and the number of vertices swept between 5,000 and 80,000, as well as uniformly random matrices with the same dimensions and average density.

Figure 4.10 compares the performance of the reconfigurable architecture against the CPU and GPU libraries on the generated matrices. The reconfigurable architecture outperforms both CPU and GPU implementations, but performs comparatively better for non-uniformly distributed matrices.

Figure 4.11 shows the speedups of the custom architecture over the CPU and GPU

Processing Element	1.5 GHz clock speed, 64-entry outstanding requests queue, 1 kB scratchpad memory *Multiply* phase: All 16 PEs per tile active *Merge* phase: 8 PEs per tile active, rest disabled
L0 cache/ scratchpad	*Multiply* phase: 16 kB, 4-way set-associative, 16-ported, shared, non-coherent cache with 32 MSHRs and 64 B block size per tile *Merge* phase: 2 kB, 4-way set-associative, single-ported, private cache with 8 MSHRs and 64 B block size + 2 kB scratchpad per active PE-pair
L1 cache	4 kB, 2-way set-associative, 16-ported, shared, non-coherent with 32 MSHRs and 64 B blocks
Crossbar	16×16 & 4×4 non-coherent, swizzle-switch based
Main Memory	HBM 2.0 with 16 64-bit pseudo-channels each @ 8000 MB/s with 80-150 ns average access latency

Table 4.1: Simulation parameters of architecture

CPU	3.6 GHz Intel Xeon E5-1650V4, 6 cores/12 threads 128 GB RAM, solid state drives
GPU	NVIDIA Tesla K40, 2880 CUDA cores @ 745 MHz, 12 GB GDDR5 at 288 GB/s

Table 4.2: CPU and GPU configurations

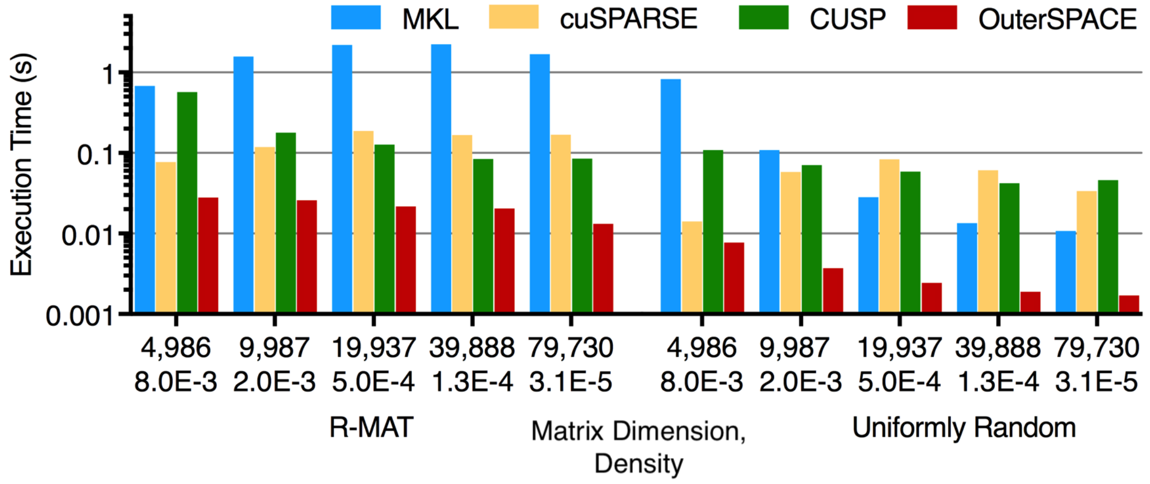


Figure 4.10: Speedups of architecture over the CPU running Intel MKL and the GPU running cuSPARSE and CUSP on synthetic workloads

implementations on real data sets. The reconfigurable architecture outperforms for all the inputs, with an average speedup of $7.9\times$ over MKL, $13.0\times$ over cuSPARSE and $14.0\times$ over CUSP.

Running the *filter3D* and *roadNet-CA* inputs achieve relatively smaller speedups compared to the other matrices, as these are closer to diagonal matrices for which the competing libraries are optimized towards and use fewer comparisons while multiplying two regular matrices. MKL performs poorly on *email-Enron*, a real-world email dataset with the characteristics of a power-law graph [54]. The reconfigurable architecture achieves the highest speedups over cuSPARSE for matrices that have a more smeared (irregular) distribution of non-zeros, such as *ca-CondMat*, *cit-Patents*, *p2p-Gnutella31* and *web-Google*.

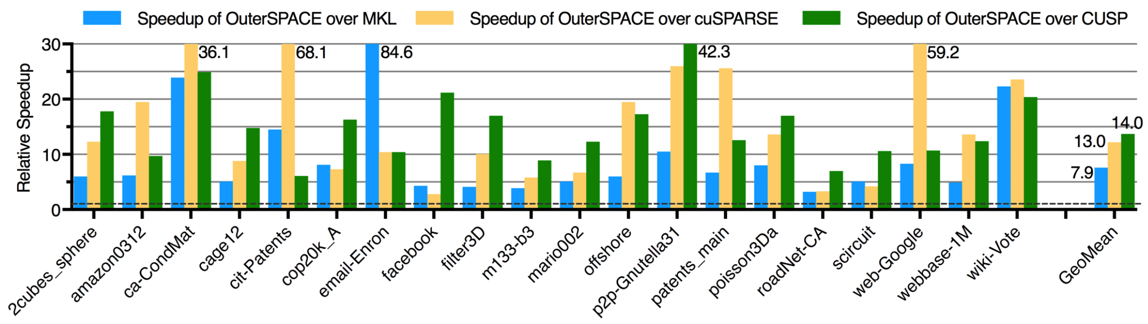


Figure 4.11: Speedups of architecture on real world workloads

4.4 Block Shuffling

In addition to activating and deactivating resources based on the algorithm and input data, we can also make use of the reconfigurable memory hierarchy to quickly move data to different locations. This allows us to accelerate preemption and context transfers discussed in chapter 3 by keeping all data transfers on-chip and reducing buffer space requirements. We call this new technique *block shuffling*.

For greater consistency with previous chapters in this section we consider an architecture closer to that of a GPU, using multithreaded SIMT cores rather than SPMD processing elements within a tile but keep the same reconfigurable memory hierarchy. We will make use of NVIDIA's terminology of streaming multiprocessor and thread block scheduler instead of tile and tile manager, respectively. When performing a block shuffle, the first level crossbars are configured to directly connect the SMs to the L2 cache to prevent interference with other threads, corresponding to the configuration shown in figure 4.9. The second level crossbars connect the L2 cache specified by the TBS for the buffer space.

The crossbars between the L1 and L2 cache are configured such that each SM can store its context in a specified buffer space determined by the TBS. The context is then read by a paired SM which will restore the thread block's state.

4.4.1 Context-Size Reduction

As shown in chapter 3, displacing too many cache entries during context migration can result in a chain reaction of thread evictions called *cataclysmic thread eviction*. While avoiding the L1 cache and storing context data directly in the L2 reduces this likelihood of this significantly, we seek to reduce this possibility by reducing the size of the thread block's contexts. Throughput machines have not been designed in the past with context swapping in mind, and there are several opportunities to reduce the massive context sizes previously discussed including eliminating dead values, compressing common register patterns, and detecting which scratchpad entries have been written to.

Detecting when registers have "dead" values, that is, values that will never be read again, is simple to do during compilation. A value is dead in the region of code before an instruction that writes to that register and after the most recent instruction that has read from it. To reduce register size, this can be communicated to the runtime by including an extra bit in the instruction's encoding for each source register, indicating whether that register's value is dead or not after execution. SMs will maintain a bit vector for every active warp, with each bit indicating whether the corresponding register contains a live value. Upon a block shuffle, dead values need not be transferred.

Register compression is a technique that has been used in the past to recognize common value patterns in registers written to and read by warps to compress their size and save space or energy [55] [56] [57]. A hardware structure called a compressor/decompressor is placed by the register file to check for five common patterns within a warp:

1. Constant: Every register holds the same value
2. Stride one: Every register holds the value of the previous register plus one
3. Stride four: Every register holds the value of the previous register plus four
4. Stride one - half warp: Same as stride one, but each half warp has its own starting value
5. Stride four - half warp: Same as stride four, but each half warp has its own starting value

A compressed register needs to hold potentially two 4-byte values for the half warp cases, plus 3 bits to indicate which pattern it matches, reducing the storage requirements by a factor of 15. Every register write will pass through the compressor to see if it matches one of the patterns, and if so, the compressed data is stored in a *compressed register cache* maintained by the SM. An additional bit vector is also maintained for each warp to indicate whether each register has an entry in the register cache or not.

Although the register file is the dominant source of state on an SM, shared memory still accounts for up to several hundred KBs in modern systems. To mitigate this overhead, the design will take advantage of two properties of how shared memory is typically managed by the programmer.

One immediate optimization is based on the observation that not all shared memory locations will be in use throughout the execution. If a block shuffle occurs near the beginning of a thread block's execution, it will likely not have stored much data in shared memory. For example, figure 4.12 shows when data is written to shared memory over time during an example thread block's execution. If a block shuffle occurs before 300,000 cycles in, this example would not need to transfer any data from shared memory to ensure correctness.

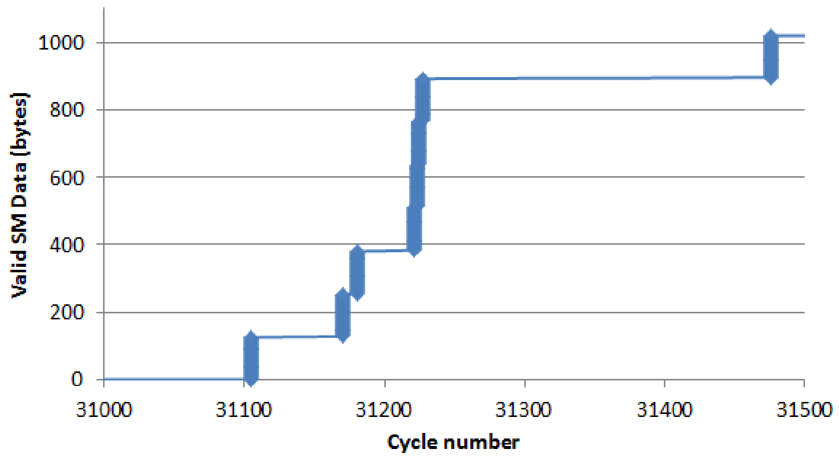
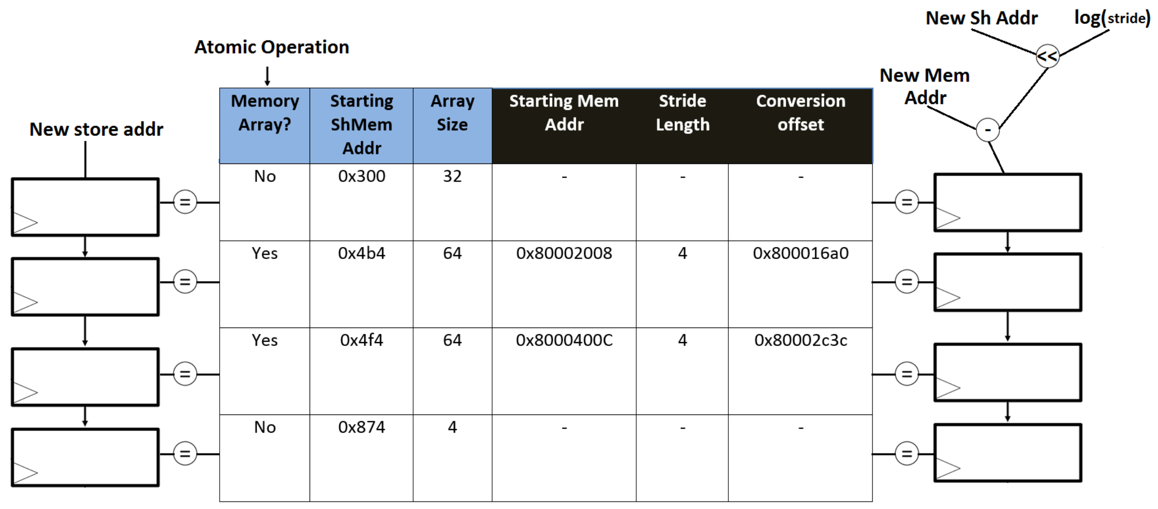


Figure 4.12: Trace of how much of a thread block's shared memory space contains valid data over time

We can track what data in the shared memory is valid through use of an auxiliary hardware table referred to as an *SM Array Table*. Each entry of the baseline SM Array Table will contain four fields: a valid bit, the starting index of a contiguous piece of data in shared memory, the size of the contiguous array, and a thread block ID. After a thread shuffle is signaled, each entry of the SM Array Table can be read out, and the respective shared memory entries can be written to memory. The state of the SM Array Table should be written to memory as well, so that only the necessary entries are read back when the thread-block is resumed. However, since warps will typically write to adjacent shared memory addresses to maximize bandwidth, large contiguous chucks of shared memory should be easily encapsulated in a few entries of the table and not add much overhead.

The second optimization to make is that many kernels use shared memory to store values directly from memory without modifying them. This is important because if the memory address a shared memory entry is associated with can be tracked, and it can be guaranteed that the memory location has not been modified, then the core does not need to offload its shared data on a thread block swap. Instead, it can simply offload meta-



Memory Array?	Starting ShMem Addr	Array Size	Starting Mem Addr	Stride Length	Conversion offset
No	0x300	32	-	-	-
Yes	0x4b4	64	0x80002008	4	0x800016a0
Yes	0x4f4	64	0x8000400C	4	0x80002c3c
No	0x874	4	-	-	-

Figure 4.13: High level design of the SM Array Table, which keeps track of which entries in the shared memory are valid, and if they can be found in memory

data containing the location of data in memory, and those addresses can be read when the thread block is restarted. This can be implemented by adding four new fields to the SM Array Table: a single bit identifying whether the entry is associating data with a memory location, the starting address of the array in memory, the stride between elements of the array, and a “conversion offset” which makes it easier to determine if a new store to shared memory “aligns” with a particular array. The conversion offset is defined as the memory address minus the shared index multiplied by the stride (alternatively, it is the y-intercept on a graph relating shared memory indexes to memory addresses). The final SM Array Table design, shown in figure 4.13 operates as follows:

- When a load is issued to memory, a static hint generated at compile time will signal if the contents of this load will be stored in shared memory. Comparison logic will check if the addresses in each lane of the warp have a constant, power of 2 stride. If so, the first lane’s address will be buffered along with the stride.

- If and when the data returned by the load is stored into shared memory, the shared memory address from the first lane will multiplied with the buffered stride (which is a simple shift operation, since it is guaranteed to be a power of two), and then subtracted from the memory address. This result is the “conversion offset” of the data, and will be compared against entries in the cache.

- On each cycle, the new conversion offset is compared against stored offsets in a particular entry of the SM Array Table. This process is pipelined, so successive writes to shared memory can proceed without stalling.

- If the conversion offsets, thread block IDs, and strides match, then the values can be appended / prepended to the array by incrementing the array size, and modifying the starting addresses, if necessary. If some but not all values match, the “memory array” bit is cleared.
- Whenever a store is issued, the store address is propagated down the pipeline on the left of figure 4.13. If the address to any of the entries match, the bit marking the entry as a memory array is cleared, indicating that the data in shared memory must now be offloaded to memory on a thread-block swap.
- If an atomic memory operation is ever issued from the thread-block, all entries of the SM Array Table must clear their ”memory array” bits, since there can no longer be any guarantee that their memory counterparts haven’t been modified.
- In the event of a block shuffle, the SM Array Table is read one entry at a time. Data corresponding to non-memory arrays are written to memory as before, but any data corresponding to memory-arrays need not be touched. Only the meta-data needs to be exported to memory, so that the values can be re-read when the block is resumed.

4.4.2 Block Shuffling Procedure

When a block shuffle is triggered, the following occurs:

1. TBS signals an individual SM that it should migrate its context to a separate SM with a particular L2 ID, which is used by the SM to calculate a set of memory addresses corresponding to that buffer space
2. SM writes all non-storage related context information, such as program counter, stack pointer, and added microarchitectural state such as register file bit masks to buffer space
3. SM writes register file contents one by one (by iterating through its RF bitmask) to buffer space
    - If the bits indicate that the register is live and compressed, the value is read from the compressed register cache
    - If the bits indicate that the register is live and uncompressed, the value is read from the register file as normal
    - If the bits indicate the register is dead, no value is read

4. SM writes scratchpad memory contents line by line to buffer space
    - Starting with line 0 and progressing until the line pointed to by the next entry in the SM Array Table, the SM reads the line and writes to the buffer space
    - If the next line corresponds to an entry in the SM Array Table, the table entry is transferred instead, and the SM proceeds to the line at the end of the table entry
5. Once all of the context has been transferred, the SM notifies the TBS, which then signals the receiving SM with the buffer space ID
6. The receiving SM follows the same process, but reading from the buffer space instead of writing
7. When reading a compressed register value, it is passed through the decompressor before stored in the RF
8. When reading an SM Array Table entry, the value is either expanded or read from main memory, accordingly

4.4.3 Methodology

We model block shuffling using the same configuration of GPGPU-Sim from chapter 3 with added support for a reconfigurable memory hierarchy and auxiliary hardware structures for block shuffling, such as the SM Array Table, compressor/decompressor, and compressed register cache. The simulators use the configurations listed in table 4.3.

System	Configuration
SMs	80 SMs, 5120 Cores, 1.6GHz Max of 2K threads per SM Max of 1K threads per TB Max 64K registers per block Max 48 KB shared memory per block 128 KB L1
Memory Subsystem	6 MB L2 900 GB/s bandwidth
Thread Scheduling	GTO
Compressor	one read / write per cycle
Compressed Register Cache	2 KB per SM
SM Array Table	128 Entries per SM

Table 4.3: GPGPU-sim simulation parameters used for evaluation

4.4.4 Results

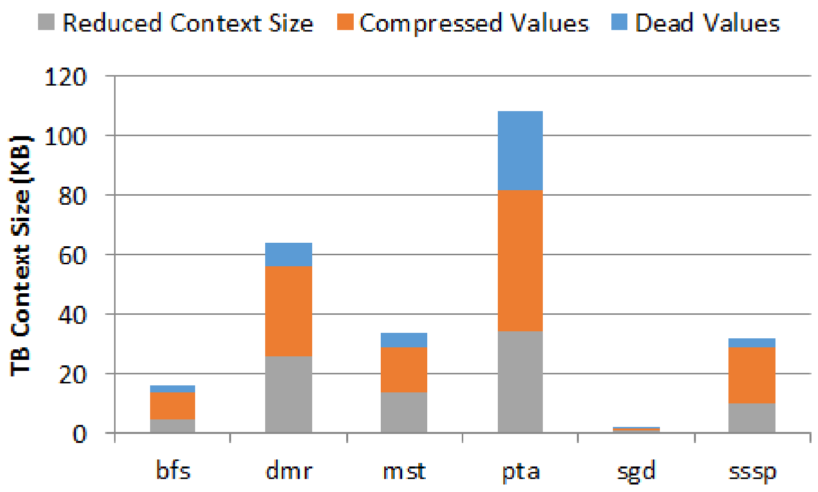


Figure 4.14: Reduction of thread block context sizes

Figure 4.16 shows the effectiveness of compression techniques on the thread block contexts. On average, contexts are reduced to 37.0% of their original value, with the new state size shown in grey. Most of the reduction comes from compressing common values. Although irregular algorithms tend to operate with pointers that will be random and difficult to compress, these are usually limited to a few registers only. The remainder of the registers often share the same value or exhibit the other properties described, and thus can be reduced significantly. Eliminating dead values provides some benefit but not nearly as much; as discussed in chapter 3, these kernels tend to use a small number of registers as it is.

Although the benchmarks from the LonestarGPU suite do not make much use of shared memory, we wanted to evaluate how our shared memory compression scheme performs to ensure that our design is generalizable to programs that may make use of it. Figure 4.15 shows the context reduction of benchmarks using shared memory from the Rodinia benchmark suite [58]. On average, the context sizes are reduced by 73% for offload: 55% from register compression, 8% from tracking valid entries in shared memory, and 10% from tracking constant arrays in shared memory. The reload context sizes are reduced by an average of 63%.

Figure 4.4 shows the latency of thread block shuffling compared with the latency of traditional context swapping, and the latency of executing individual iterations of each kernel. On average, the latency of thread block shuffling is 90% less than the latency of context switching. BFS, SSSP, and DMR see moderate improvements of 2-4% as block shuffling can be performed more frequently than preemption and so some smaller phase changes can be handled. SGD, PTA, and MST get the most benefit as their execution latencies were too

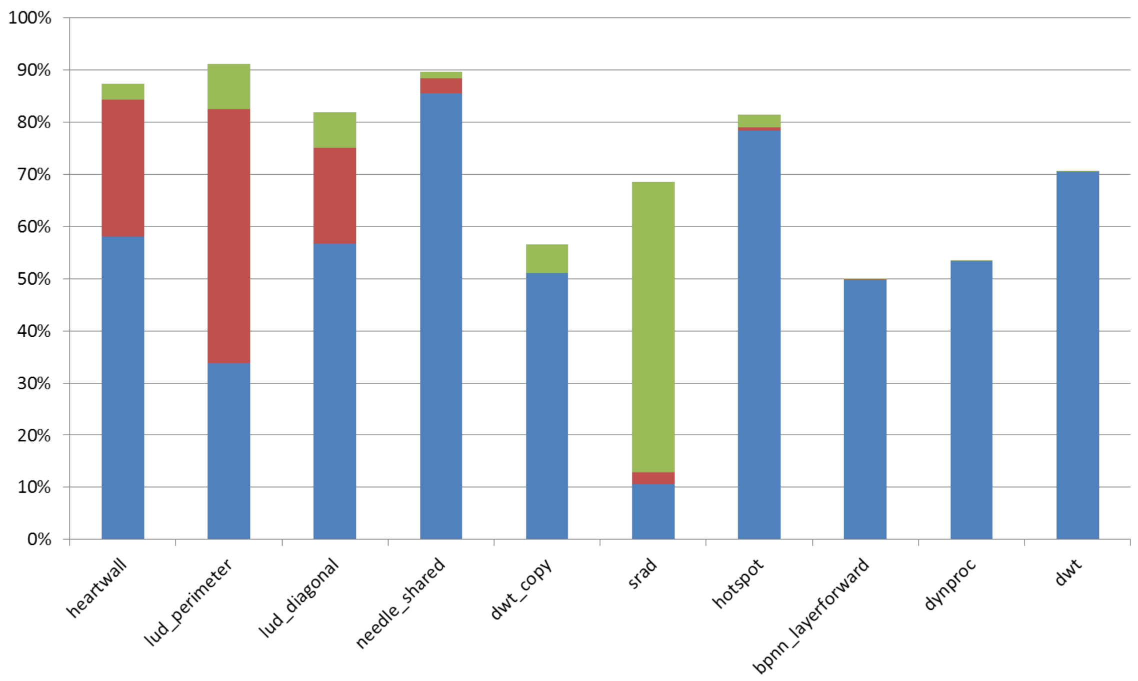


Figure 4.15: Context reduction on Rodinia benchmarks from register file (blue), reloading shared memory (red), offloading shared memory (green)

Application	**Preemption latency (μs)**	**Block shuffle latency (μs)**	**Execution latency (μs)**
BFS	3.0	0.06	56,000
DMR	11.8	0.25	9,080
MST	6.2	0.13	1,030
PTA	20.1	0.49	40
SSSP	5.9	0.01	23,000,000
SGD	0.3	0.13	10

Table 4.4: Block shuffle latency compared to preemption latency and iteration latency

small to make good use of preemption with out incurring significant overheads of transferring the full context sizes to main memory and back. By using thread block shuffling, all kernels can feasibly have their thread blocks migrated in response to shifting resource usages, allowing for a much finer grained architecture for processing mixed workloads.

The compressor/decompressor and SM Array Table have both been designed in Verilog and synthesized into a 28 nm technology netlist using Synopsys' Design Compiler.

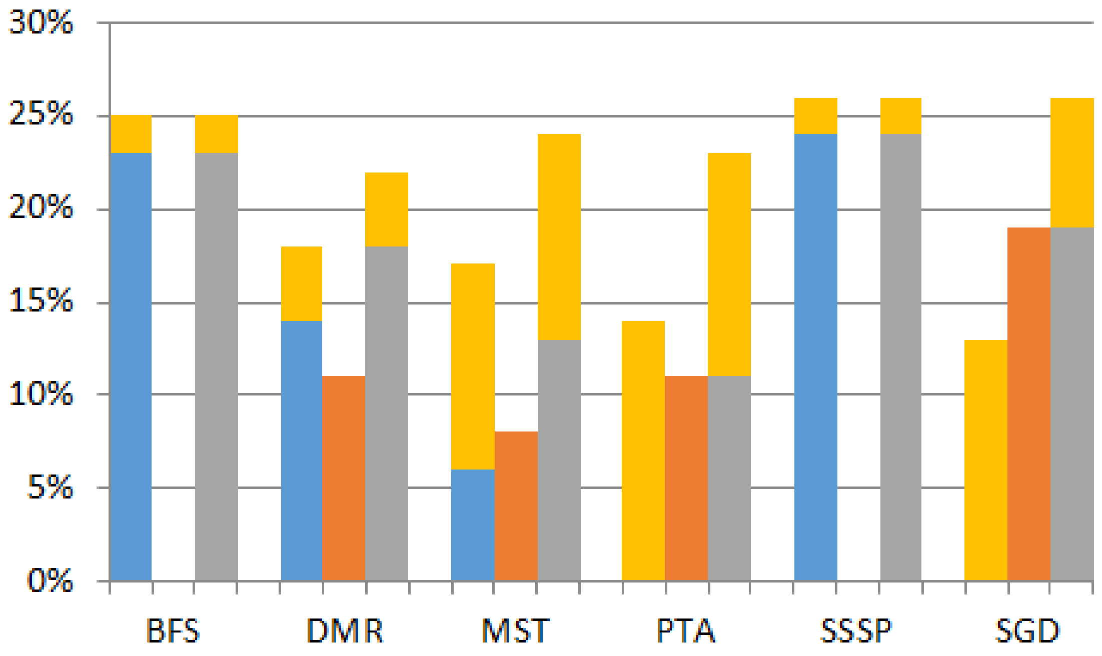


Figure 4.16: Throughput increases when employing only preemption (blue), only predictive scheduling (orange), and both (grey), with the throughput gains provided by block shuffling over traditional preemption annotated (yellow)

Number of entries	Area (μm^2)	Power (mW)
16	17902	18.5
32	36236	36
64	72789	74
128	165310	149

Table 4.5: Synthesis results of different sized SM Array Table

4.5 Related Work

Yavits et al [59] codesign a sparse matrix-matrix multiplication algorithm and hardware based on custom RAM tables. Lin et al [60] used FPGAs to build reconfigurable compute engines in order to accelerate sparse matrix-matrix computation but demonstrate less scalability for full system throughputs.

Register compression has been used by Gebhart et al [61] [62] [63] via smaller register caches accessed before the register file. Jeon et al [64] has used dead value analysis to reduce the size of the register file. Pekhimenko et al [65], Lee et al [66], Stephenson et al [67] and Kloosterman et al [57] have all identified common register value trends across warps that were used to compress the size of register files.

These works have all been motivated by reducing power and energy consumption. We are the first to our knowledge who have looked at using these techniques to quickly repartition resources among contexts.

CHAPTER 5

Enabling Context Migration in Scalable Vector Cores

5.1 Introduction

Although less powerful than a GPU, SIMD enabled processors are a popular choice for high performance computing due to the ease of programming. Rather than writing separate kernels and explicitly managing memory transfers between host and device, programmers need only annotate "for" loops with independent loops in source code, and the loop contents will be parallelized without control or memory transfers. To facilitate a greater spectrum of workloads, Arm introduced the Scalable Vector Extension (SVE) which allows for implementation specific hardware vector sizes as well as several other enhancements over its previous Advanced SIMD instruction set. This enables seamless hardware improvements without requiring code to be recompiled for successive generations with different vector sizes.

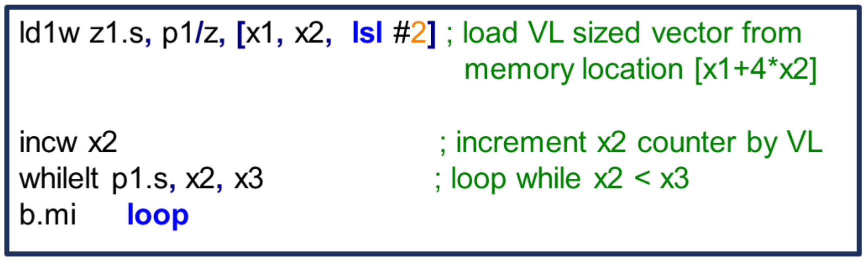
An additional benefit is that heterogeneous systems using different vector lengths can be used for workloads which exhibit different levels of parallelism over time, such as we would see with irregular workloads or workloads containing a diverse set of parallelizable sections. Code sections which are not able to fully utilize a given vector length can be moved to a core with a smaller vector length to conserve power, while more parallelizable loops can be run on larger vector cores however the OS sees fit. However, this strategy poses practical issues, as once vector code has begun execution, correctness cannot be ensured if the thread is migrated to a core with a smaller vector length. This greatly limits how heterogeneous architectures can be designed to take advantage of variable parallelism within designs.

In this chapter, we will demonstrate the value of larger vector lengths across a variety of regular and irregular workloads and motivate the design of heterogeneous systems with different vector lengths. We will then describe our solution for the correctness problem with a small set of instructions to the ISA which allows for flexibility in writing new code as well as assuring correctness for legacy code.

5.2 Background

5.2.1 SVE

Single-Instruction Multiple-Data (SIMD) extensions to processors allow programmers to take advantage of wide arithmetic functional units when processing smaller units of data by executing multiple instances of the same instruction simultaneously. For example, Arm's Advanced SIMD instruction set (more commonly known as Neon)[68] makes use of 128-bit functional units. Programmers can make use of Neon instructions to process 2x64-bit elements, 4x32-bit elements, 8x16-bit elements, or 16x8-bit elements simultaneously. The program explicitly indicates which of these sizes should be used, and the programmer therefore has knowledge over how many elements are being executed simultaneously. Neon is well designed for DSP applications, processing media codecs, et cetera. It only enables simple control flow and processing of regular contiguous data structures.

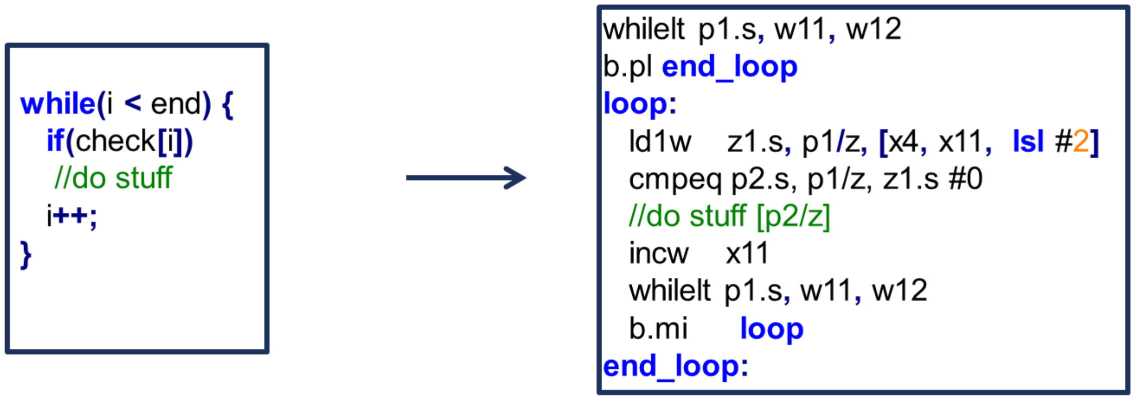


```
ld1w z1.s, p1/z, [x1, x2,  lsl #2] ; load VL sized vector from
                                     memory location [x1+4*x2]

incw x2                             ; increment x2 counter by VL
whilelt p1.s, x2, x3                ; loop while x2 < x3
b.mi     loop
```

Figure 5.1: Example loop in SVE

SVE, on the other hand, does not specify the hardware vector length (VL), instead allowing sizes between 128 and 2048 bits. Programmers write "vector-length-agnostic" (VLA) code which specifies the subelement size, but treats the VL as an unknown variable that is resolved dynamically at runtime. Figure 5.1 shows an example of one of these such loops. The programmer uses the *incw* instruction to increment a counter variable (stored in *x2* for this example) by the VL and loops while the counter is below some threshold. A larger VL will result in fewer iterations and vice-versa, so the programmer is not aware of how many iterations will occur for a given input, but the code can be written such that it performs correctly regardless of the hardware's VL. To handle input sizes that are not an integer multiple of the VL as well as supporting "if" statements within the code, SVE provides a set of predicate registers which deactivate lanes from executing their respective instructions when necessary. An example is shown in figure 5.2, where the loop body is predicated by register *p2* which is set by the *compeq* instruction.



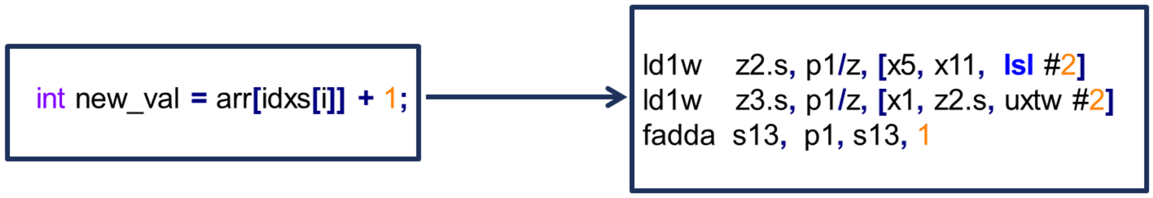
```
while(i < end) {
   if(check[i])
    //do stuff
   i++;
}
```

```
whilelt p1.s, w11, w12
b.pl end_loop
loop:
  ld1w    z1.s, p1/z, [x4, x11,  lsl #2]
  cmpeq p2.s, p1/z, z1.s #0
  //do stuff [p2/z]
  incw    x11
  whilelt p1.s, w11, w12
  b.mi      loop
end_loop:
```

Figure 5.2: Predication

An additional enhancement of SVE is the support for gather-load and scatter-store instructions. This is useful for HPC applications, which may use complex data structures consisting of pointers. Gather-load and scatter-store instruction enable indirect access to non-contiguous memory arrays within a single instruction. This is shown in figure 5.3, where the values for *idxs* are stored in vector register *z2*, which is used by the subsequent *ld1w* load instruction to indirectly access several scattered memory locations.

SVE also adds horizontal reduction instructions so that dependencies between elements in a vector can be resolved. SVE supports summation, minimum, maximum, and logical reductions within single instructions. Figure 5.4 shows all of these features in action, where *z3* is filled with non-zero elements of a sparse matrix using gather-loads, and the *fadda* reduction instruction sums the product of these values with vector elements to calculate a partial dot-product, all the while being predicated by register *p1* to transparently manage the number of loop iterations.



```
int new_val = arr[idxs[i]] + 1;
```

```
ld1w    z2.s, p1/z, [x5, x11,  lsl #2]
ld1w    z3.s, p1/z, [x1, z2.s, uxtw #2]
fadda  s13,  p1, s13, 1
```

Figure 5.3: Vector load-stores

```
scvtf s13, xzr                                ; sum = 0
loop:
  ld1w   z1.s, p1/z, [x4, x11,  lsl #2]     ; ld vals[i]
  ld1w   z2.s, p1/z, [x5, x11,  lsl #2]     ; ld cols[i]

  ld1w   z3.s, p1/z, [x1, z2.s, uxtw #2]  ; ld vec[cols[i]]
  fmul   z1.s, p1/m, z1.s, z3.s             ; vals[i] * vec[cols[i]]
  fadda  s13,  p1, s13, z1.s                ; sum += sum(z1)
  incw   x11
  whilelt p1.s, w11, w12
  b.mi      loop
end_loop:
```

Figure 5.4: Sparse Matrix-Vector Multiplication

5.2.2 Heterogeneous Systems

Heterogeneous multi-core systems are those where the individual cores are distinct from one another, having different performance, power, and area metrics. An example is Arm's big.Little architecture [69] which has been implemented to dynamically trading off performance and power, primarily in mobile settings and other power-constrained devices. In this architecture, a larger, faster, and more power-hungry core is paired with a slower, more power-efficient one. During periods when latency is determined to be non-critical (as determined by the operating system), an application is run on the little core to limit battery use. During periods when high performance or low latency is needed, the context can be switched to the big core to meet the computation needs before being migrated back to the small core. Figure 5.5 shows an example of a big.LITTLE architecture where the "big" core has twice the issue width and VL of the "LITTLE" core.

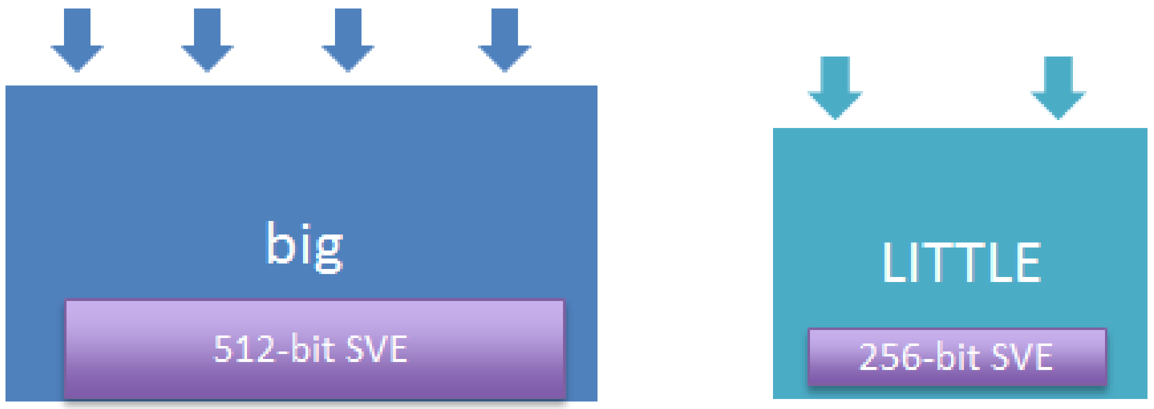


Figure 5.5: big.LITTLE architecture consisting of different issue widths and vector lengths

5.3 Motivation

There has been a clear increase in the use of SIMD extensions in the use of HPC applications. Figure 5.6 shows how the number of Arm processors sold has increased over the past decade according to the World Semiconductor Trade Statistics. Several large scale systems have been implemented using SIMD extensions. Fujitsu's Post-K Supercomputer is intended to be employed using Armv8 cores with SVE enabled, and SenWei, Bull Atos Technologies, and the European Processor Initiative have all announced supercomputer designs based on Arm cores with the possibility of included SVE.

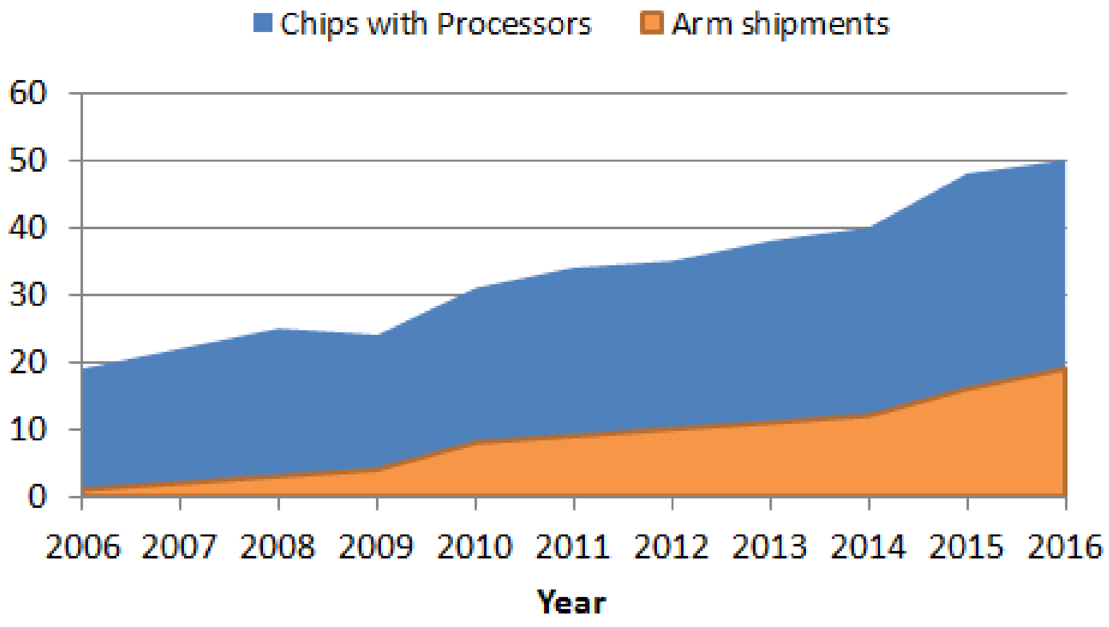


Figure 5.6: Number of processors (in billions) sold according to WSTS

There has yet to be systems designed which make explicit use of multiple SVE-enabled cores with different VLs. There is little public data on how workloads scale with different VLs to justify this hardware decision. In addition it is not immediately clear how such a system would operate with guaranteed correctness. Although VLA code can be written to guarantee that the same results are generate for every valid VL, there is no such guarantee when the VL is changed part way through execution. Correctness can be trivially guaranteed when moving to a larger VL by masking all previously unused lanes, but migration to a smaller VL means that certain data must be discarded and, worse-yet, information regarding the VL can be leaked via instructions like *incw* and cause inconsistent behavior after the context switch.

5.3.1 Scalability Study

We first provide evidence for the value of including larger VLs in systems by showing how performance is improved by increasing the width of the vector pipeline. We ported a subset of GNU's glibc string library [70] to make use of SVE instructions. Their descriptions are given in table 5.1 and have wide use in text parsing.

Workload	*Description*
strchr	finds first occurrence of a character in a string
strcmp	compares two given strings
strcpy	copies a string to a new location
strlen	returns length of a given string
strstr	searches for a substring within another string

Table 5.1: Description of workloads

These workloads stream through their inputs at regular intervals using direct indexing. No scatter-gather operations are used, and lanes are typically only predicated in the relatively rare case of passing past the end character. As such, these are highly regular workloads, and give us a glimpse into how programs with predictable structure behave as the VL is scaled up.

μArch Parameter	*Value*
Frequency	2.8 GHz
Memory bus width	512 bits
L1-I	64 KB
L1-D	32 KB
L2	1 MB
Issue Width	4
Decode Depth	6
ROB Entries	128

Table 5.2: Baseline core configuration

To evaluate these trends, we simulated the execution of these programs using an in-house simulator modeling an ARMv8-A core with enabled SVE, who's baseline properties are described in table 5.2. Hardware vector lengths of 128, 256, 512, and 1024 bits were tested. 1 million invocations of each function were simulated passing in a random subset of dictionary words. Their speedups, normalized to a 128 bit vector length, are shown in figure 5.7.

Ideally, we would observe speedups of 1x, 2x, 4x, and 8x, respectively across these sweeps in correspondence with the increase in vector length. In practice, such speedups are rarely seen as a result of Ahmdal's law and sections of non-parallelizable code. In the

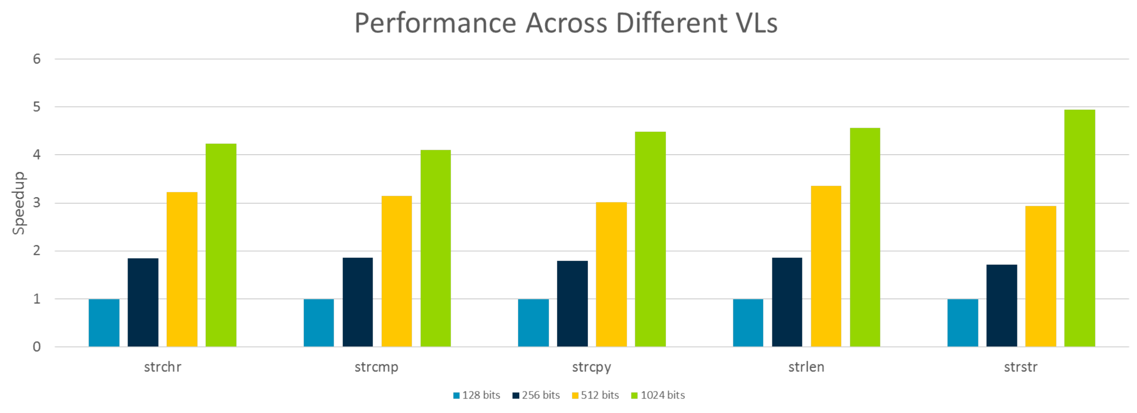


Figure 5.7: Speedups of running benchmarks on different vector length machines

case of these benchmarks, although the inner bodies of the loops are highly parallel, the memory system does not scale with the vector length. As a result, vector loads take longer to complete on average. Figure 5.8 shows how the number of stalls as a result of memory misalignments (in other words, loads that spill over the memory word boundary and require subsequent load operations) when running strchr. When the vector length is 128 bits, the data fetched by a load instruction is always aligned properly and can be moved from the cache to the pipeline in a single cycle. Assuming data exists in the cache (a reasonable assumption if using a prefetcher for these data streaming algorithms), the number of instructions completed per cycle (IPC) should be near the maximum possible of 4 (since this is a four-wide machine). However, as the vector length increases, more load instructions cross cache line boundaries and require extra cycles to bring in the full data. Figure 5.9 shows how the IPC drops as a result of these increased stalls, resulting in the sublinear speedup we observe.

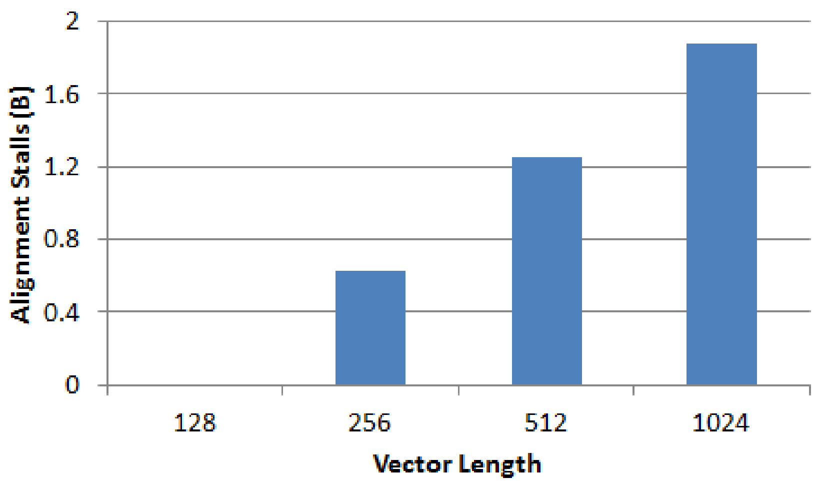


Figure 5.8: Number of stalls (in billions) as a result of load misalignments as VL increases

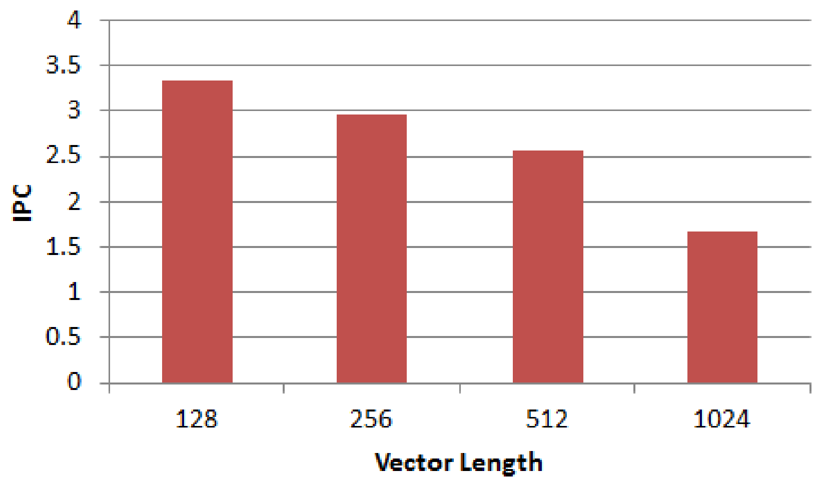


Figure 5.9: IPC as VL increases

To better quantify the value of larger vectors, we compare this speedup to other microarchitectural enhancements. Table 5.3 describes the parameters that we analyzed by modifying the baseline architecture. We measured the speedups when varying the pipeline's width and depth and the number of reorder buffer (ROB) entries on the benchmarks. Of these, only the issue width had any non-negligible performance impact. We show the speedups of strchr when issue width is varied given a vector length of 128 bits as well as 512 bits, as compared to the speedup of increasing the vector length in figure 5.10.

μArch Parameter	*Range*
Issue Width	1-8
Decode Depth	4-8
ROB Entries	32-256
VL	128-1024

Table 5.3: Sweep Values

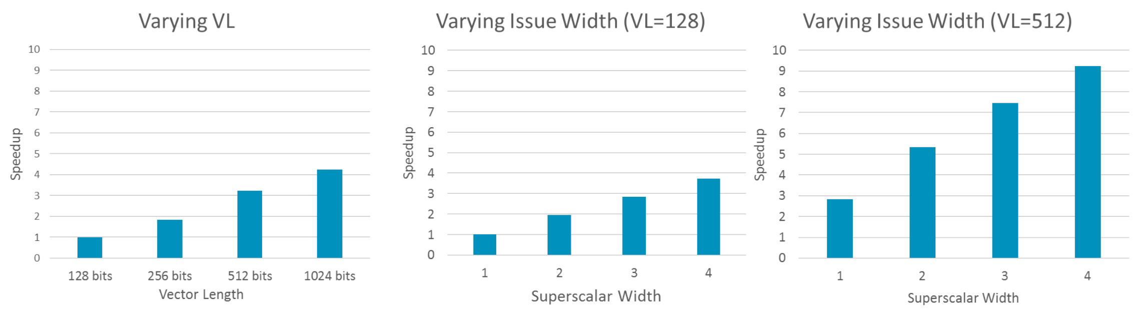


Figure 5.10: Speedups of strchr when varying issue width and vector length

We learn several things from these results:

- These workloads exhibit significant amounts of parallelism between loop iterations such that ROB size and pipeline depth, which seek to reduce the number of stalls due to inter-instruction dependencies, have little effect

- For such workloads, increasing the vector length provides speedups comparable to increasing the issue width

- Increasing either the issue width or the vector length does not preclude the benefits of either. The two techniques can be combined.

To evaluate irregular workloads, we ported the SSSP benchmark to SVE. We compared the speedups when processing a dense graph (density=10%) versus a sparse one (density=0.001%), which are shown in figure 5.11. As explained in chapter 3, irregular workloads have different execution behavior when processing differently structured data. In the case of SSSP, the benchmark has a significantly improved cache hit rate when processing dense data sets, resulting in greater throughput. We observe the same tendencies here, with sparse data sets scaling much more poorly with vector length.

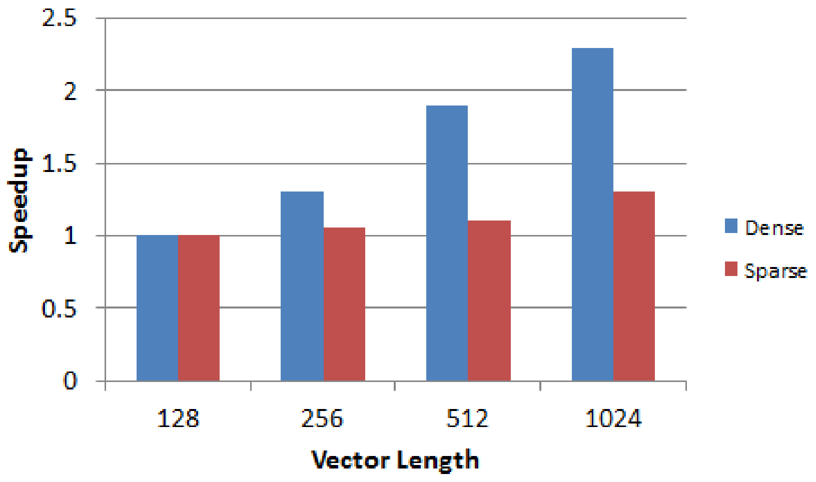


Figure 5.11: Speedups for dense and sparse workloads SSSP

The conclusion we draw is that the combination of the specific programs being, run, the structure of the input data, and the desired power and performance goals significantly impact the ideal vector length to run a thread on. Including multiple cores in a system offers the runtime greater flexibility in maximizing performance when processing a diverse set of workloads.

5.4 ISA Extensions

We now present our solution to the correctness problem of changing the VL part way through execution. It is built on the observation that incorrect results only occur when the same values are used before and after a context migration. If we can identify regions of code that are isolated in their usage of vector data, then we can limit migrations to different sized vector cores to those points in the program that are guaranteed not to use any previously written vector values. This is difficult to do dynamically in hardware, as it requires knowledge of the program and algorithm as a whole. However, the programmer and/or compiler can, if desired, explicitly annotate these regions. We therefore design our solution around the following goals:

- Allow for identification of migratory regions without significant change to programming model
- Enable nested function calls to vector code
- Ensure correct functionality of legacy code
- Prioritize correctness and simplicity

Requiring support for legacy code implies that we must assume that a thread can't be migrated unless the runtime is certain there won't be correctness issues, which is not likely to include many cases besides the trivial one where no vector code has been executed.

We propose the addition of two instructions as well as an architectural register to the ISA. These three added features will be used by the runtime to determine if a running thread is currently *pinned* to a particular vector length (meaning the thread can only be migrated to cores with a vector length greater than or equal to the current one), or if the thread is *unpinned* and can be migrated anywhere. We describe the ISA additions in detail here:

- *PIN_LEVEL* is the added register. A value of 0 indicates that the current thread is not pinned to a particular VL and can be migrated anywhere the runtime chooses without impacting correctness. Any other value indicates that the current thread is pinned to the current VL and cannot be migrated to cores with smaller VLs.
- *PIN_VL* is an added instruction. It increments *PIN_LEVEL* by 1 and indicates to the runtime that the thread should be pinned to the current VL. The programmer must place this instruction before the first vector instruction, or the whole function (and potentially the entire thread) will be permanently pinned to the current VL (the reasoning for this is to allow correct execution of legacy code and is described below).

- *UNPIN_VL* is the second added instruction. If *PIN_LEVEL* is 0 when this instruction is executed, it triggers an exception. Otherwise, it decrements *PIN_LEVEL* by 1, possibly (though not necessarily) indicating to the runtime that it is safe to unpin the current thread.

In the typical usage, the programmer places the *PIN_VL* instruction before any vector code, and places *UNPIN_VL* after all the relevant vector data is dead and the programmer can guarantee that no correctness issues will occur if the thread is migrated to a different sized core. This situation is shown in figure 5.12. A green segment of code indicates that the thread can be migrated anywhere, whereas a red segment of code indicates that the thread must stay on cores that can operate using the current vector length.

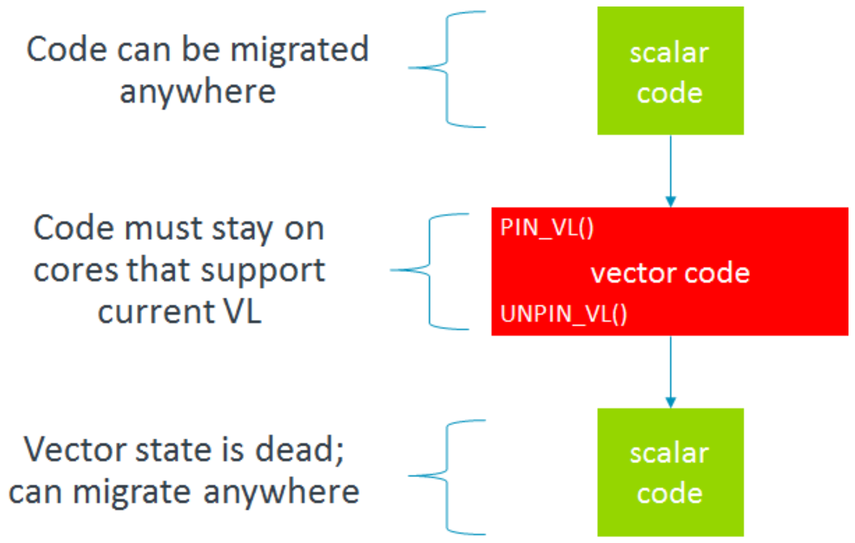

Code can be migrated anywhere
scalar code
Code must stay on cores that support current VL
PIN_VL()
vector code
UNPIN_VL()
Vector state is dead; can migrate anywhere
scalar code

Figure 5.12: Typical use of *PIN_VL*

It may seem redundant to include a *PIN_VL* instruction as the runtime could implicitly pin the thread at the first vector instruction, but it is needed to properly support legacy code. Figure 5.14 demonstrates how the runtime handles legacy code where vector instructions are executed before a *PIN_VL* instruction is seen. In this case, the runtime implicitly pins the thread before the first vector instruction. However, because there is no way for the runtime to determine dynamically if and when all vector state is dead and will not be used in the future, it must pessimistically assume that the thread must be tied to the current VL until it is terminated. Even if later vector code is executed that includes a *PIN_VL / UNPIN_VL* pair, there is no way to determine if vector state generated before that code segment will be used. Thus, although legacy code not making use of these new instructions will not be able to benefit from having multiple VLs in a system, we can guarantee that it will execute correctly, as it will behave as if there is only one available VL.

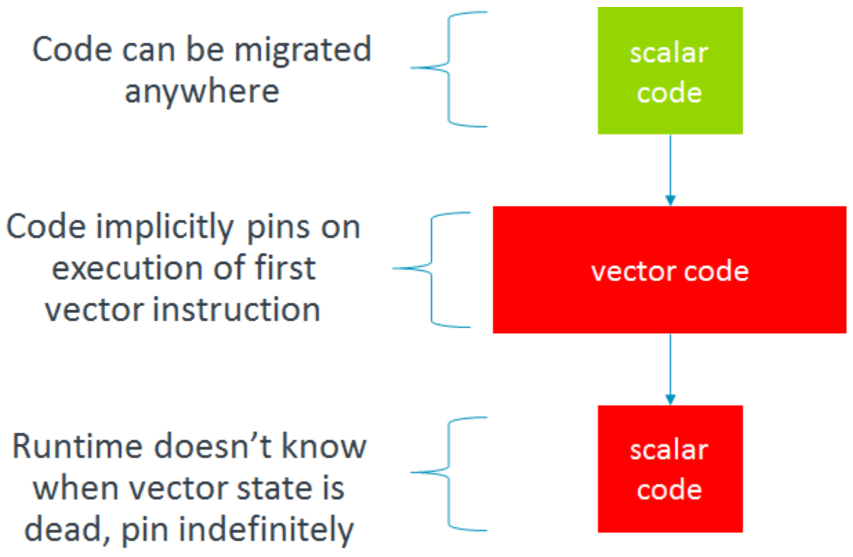

Code can be migrated anywhere
scalar code
Code implicitly pins on execution of first vector instruction
vector code
Runtime doesn't know when vector state is dead, pin indefinitely
scalar code

Figure 5.13: Using *PIN_VL* with legacy code

The *PIN_LEVEL* register is needed to handle the case where multiple *PIN_VL* instructions are executed in sequence, as would occur if vector code calls other vector subroutines, as shown in figure 5.14. In this case, the thread should not be unpinned after the nested function calls *UNPIN_VL*, because there is still vector state from the calling function that is active. *UNPIN_VL* should only take effect if there has been a matching *UNPIN_VL* for every previous *PIN_VL*. This is the purpose for *PIN_LEVEL*. A positive value in that register means there are unmatched *PIN_VL* instructions and the thread should remain pinned. As *UNPIN_VL* instructions should never be seen before a corresponding *PIN_VL* instruction, the execution of *UNPIN_VL* while *PIN_LEVEL* has a value of 0 represents undefined behavior and generates an exception.

There are a few additional challenges that arise as a result of the inclusion of these new instructions and architectural register. The first is that care must be taken to include the *PIN_LEVEL* register in any sort of state management. An example where this is necessary is when executing the Linux functions setjmp [71] or longjmp [72]. These instructions allow for exceptional control transfer into or out of nested functional calls or coroutines by saving and later restoring the current context and environment. In any situations like these, the current *PIN_LEVEL* value must be saved as well.

An additional concern is that on many heterogeneous systems, the OS is allowed to indefinitely disable cores, usually for power or performance considerations. A plausible scenario is that the OS moves the system into a power saving mode, disables the "big" core, and migrates all code to the "LITTLE" core. If a thread is currently pinned to the larger VL, it will not be able to continue execution. Therefore, it is necessary to augment

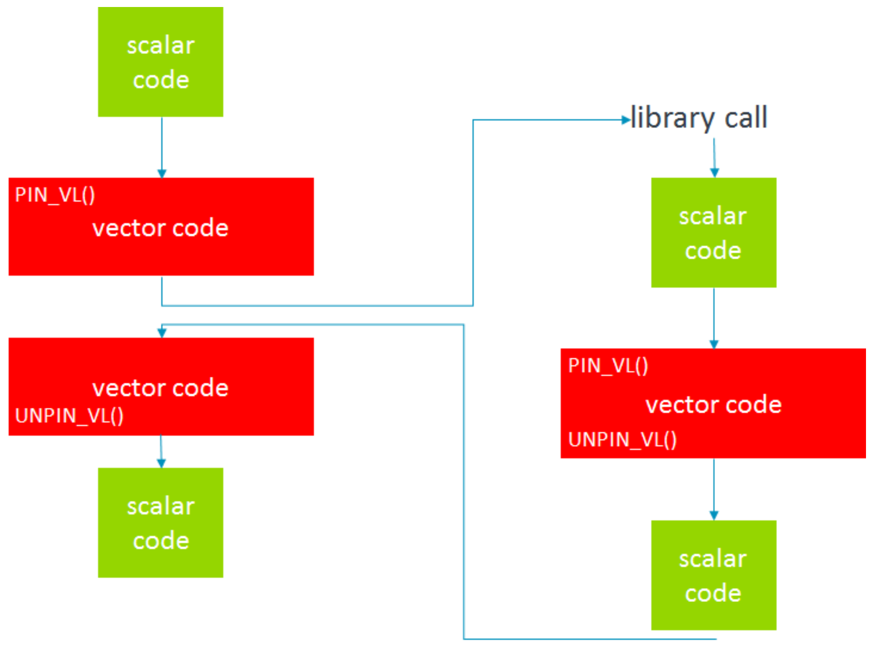


Figure 5.14: Ambiguity which arises during function calls

OS schedulers on these systems to be aware of this new restriction and prevent starvation.

5.5 Conclusion

In this chapter, we have demonstrated the value of including multiple sized vector cores within a single system. Different workloads as well as diverse possible inputs to those workloads have a wide range of benefits when running on smaller or larger vector pipelines. System designers will have greater flexibility in navigating the complex tradeoffs between power, performance, and area, if they have the option to migrate threads between different vectors cores dynamically based on execution behavior. To enable this behavior while ensuring correctness as the hardware vector length changes during execution, we have shown how the inclusion of two new instructions and one architectural register can be effective. These additions to the ISA are simple to use, and allow legacy code to continue functioning correctly, although the code will not benefit from having access to multiple vector lengths. These extensions can be combined with strategies discussed in earlier chapters to better facilitate the migration of large contexts between cores.

CHAPTER 6

Conclusion

This dissertation motivates the need to augment data parallel processors to better handle the dynamic needs associated with more general workloads exhibiting irregular parallelism. As Moore's law slows down, more and more complicated tasks will be parallelized in an attempt to maintain performance expectations, but the memory and control irregularities manifest in these workloads make scheduling work items and partitioning resources to those items more challenging. Application-specific integrated circuits may find increased use for applications of key importance, but to maintain the prevalence of general application development and deployment, general purpose architectures like GPUs and SIMD processors must be augmented. This dissertation has focused on the goal of generalizing and replacing context management schemes to better accommodate the large number of threads present in throughput architectures.

Chapter 3 outlined the problem of irregularities in data structures manifesting themselves as variances in hardware resource usages, causing interference and slow downs between threads. However, these shifts happen in a smooth manner and allows the opportunity for the runtime to react dynamically. Augmenting the thread block scheduler to track performance metrics allows for better allocation heuristics at thread block dispatch, as well as appropriate times to preempt running thread blocks to better handle changing execution conditions. This allows for up to a 17% increase in throughput over other scheduling strategies.

Chapter 4 extended these ideas by considering the inclusion of reconfigurable resources in throughput architectures. By incorporating programmable crossbars and enabling/disabling different compute and memory units, systems can be better balanced for a variety of workloads. This also enables a more sophisticated version of preemption and thread block migration called block shuffling, which can be done entirely on chip with 90% reduced buffer space and 24% increased throughput.

Finally chapter, 5 showed how extending these ideas to heterogeneous systems with different sized vector pipelines introduces undefined behavior and limits the flexibility of system designers to account for a greater variety of workload types. We motivated the need for more general systems by illustrating the difference in available parallelism across different workloads, and then provided an extension to the ISA which ensures correctness while allowing for greater scalability of future systems.

BIBLIOGRAPHY

[1] Nvidia, "Fermi Architecture Whitepaper," http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, Accessed: May 2015.

[2] Burtscher, M., Nasre, R., and Pingali, K., "A quantitative study of irregular programs on GPUs," *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, IEEE, 2012, pp. 141–151.

[3] Sewell, K., Dreslinski, R. G., Manville, T., Satpathy, S., Pinckney, N., Blake, G., Cieslak, M., Das, R., Wenisch, T. F., Sylvester, D., et al., "Swizzle-switch networks for many-core systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, Vol. 2, No. 2, 2012, pp. 278–294.

[4] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ACM, 2008, pp. 73–82.

[5] Fatahalian, K., Sugerman, J., and Hanrahan, P., "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, 2004, pp. 133–137.

[6] Barnes, J. and Hut, P., "A hierarchical O (N log N) force-calculation algorithm," *nature*, Vol. 324, No. 6096, 1986, pp. 446.

[7] Braunstein, A., Mézard, M., and Zecchina, R., "Survey propagation: An algorithm for satisfiability," *Random Structures & Algorithms*, Vol. 27, No. 2, 2005, pp. 201–226.

[8] Harish, P. and Narayanan, P., "Accelerating large graph algorithms on the GPU using CUDA," *International conference on high-performance computing*, Springer, 2007, pp. 197–208.

[9] Méndez-Lojo, M., Nguyen, D., Prountzos, D., Sui, X., Hassaan, M. A., Kulkarni, M., Burtscher, M., and Pingali, K., "Structure-driven optimizations for amorphous data-parallel programs," *ACM Sigplan Notices*, Vol. 45, ACM, 2010, pp. 3–14.

[10] Kim, J. and Batten, C., "Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists," *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2014, pp. 75–87.

[11] Nasre, R., Burtscher, M., and Pingali, K., "Data-driven versus topology-driven irregular computations on gpus," *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, IEEE, 2013, pp. 463–474.

[12] Luo, L., Wong, M., and Hwu, W.-m., "An effective GPU implementation of breadth-first search," *Proceedings of the 47th design automation conference*, ACM, 2010, pp. 52–55.

[13] Nasre, R., Burtscher, M., and Pingali, K., "Morph algorithms on GPUs," *ACM SIGPLAN Notices*, Vol. 48, ACM, 2013, pp. 147–156.

[14] Mendez-Lojo, M., Burtscher, M., and Pingali, K., "A GPU implementation of inclusion-based points-to analysis," *ACM SIGPLAN Notices*, Vol. 47, No. 8, 2012, pp. 107–116.

[15] Jones, S., "Introduction to dynamic parallelism," *GPU Technology Conference Presentation S*, Vol. 338, 2012, p. 2012.

[16] "Unified Memory for CUDA Beginners," `https://devblogs.nvidia.com/unified-memory-cuda-beginners/`, Accessed: 2018-03-15.

[17] AMD, A., "Accelerated parallel processing: OpenCL programming guide," *URL http://developer. amd. com/sdks/AMDAPPSDK/documentation*, 2011.

[18] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al., "In-datacenter performance analysis of a tensor processing unit," *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, 2017, pp. 1–12.

[19] Jiao, Q., Lu, M., Huynh, H. P., and Mitra, T., "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS," *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, IEEE, 2015, pp. 1–11.

[20] Zhong, J. and He, B., "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 25, No. 6, 2014, pp. 1522–1532.

[21] Liang, Y., Huynh, H. P., Rupnow, K., Goh, R. S. M., and Chen, D., "Efficient GPU spatial-temporal multitasking," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 26, No. 3, 2015, pp. 748–760.

[22] Nvidia, "Kepler Architecture Whitepaper," `.www.nvidia.com/content/PDF/NVIDIAKeplerGK110ArchitectureWhitepaper.pdf`, Accessed: July 2019.

[23] Nvidia, “Multi-process service,” .http://docs.nvidia.com/deploy/mps/index.html, Accessed: July 2019.

[24] Adriaens, J. T., Compton, K., Kim, N. S., and Schulte, M. J., “The case for GPGPU spatial multitasking,” *IEEE International Symposium on High-Performance Comp Architecture*, IEEE, 2012, pp. 1–12.

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

[26] Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R., “Improving GPGPU concurrency with elastic kernels,” *ACM SIGPLAN Notices*, Vol. 48, ACM, 2013, pp. 407–418.

[27] Park, J. J. K., Park, Y., and Mahlke, S., “Dynamic resource management for efficient utilization of multitasking GPUs,” *ACM SIGOPS Operating Systems Review*, Vol. 51, No. 2, 2017, pp. 527–540.

[28] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Anderson, M. J., Vadlamudi, S. G., Das, D., and Dubey, P., “Graphmat: High performance graph analytics made productive,” *Proceedings of the VLDB Endowment*, Vol. 8, No. 11, 2015, pp. 1214–1225.

[29] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., and Dubey, P., “Navigating the maze of graph analytics frameworks using massive graph datasets,” *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 979–990.

[30] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J., “NVIDIA Tesla: A unified graphics and computing architecture,” *IEEE micro*, Vol. 28, No. 2, 2008, pp. 39–55.

[31] Nvidia, C., “NVIDIAs Next Generation CUDA Compute Architecture: FERMI,” *Comput. Syst*, Vol. 26, 2009, pp. 63–72.

[32] Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., and Valero, M., “Enabling preemptive multiprogramming on GPUs,” *ACM SIGARCH Computer Architecture News*, Vol. 42, IEEE Press, 2014, pp. 193–204.

[33] Menon, J., De Kruijf, M., and Sankaralingam, K., “iGPU: exception support and speculative execution on GPUs,” *ACM SIGARCH Computer Architecture News*, Vol. 40, IEEE Computer Society, 2012, pp. 72–83.

[34] Park, J. J. K., Park, Y., and Mahlke, S., “Chimera: Collaborative Preemption for Multitasking on a Shared GPU,” *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2015, pp. 593–606.

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

[36] Stratton, J. A. et al., "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign.

[37] Xu, Q., Jeon, H., Kim, K., Ro, W. W., and Annavaram, M., "Warped-slicer: efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 230–242.

[38] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A., "Introducing the graph 500," *Cray Users Group (CUG)*, Vol. 19, 2010, pp. 45–74.

[39] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., and Dubey, P., "Navigating the maze of graph analytics frameworks using massive graph datasets," *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 979–990.

[40] Kayıran, O., Jog, A., Kandemir, M. T., and Das, C. R., "Neither more nor less: optimizing thread-level parallelism for GPGPUs," *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, IEEE Press, 2013, pp. 157–166.

[41] Li, D., Rhu, M., Johnson, D. R., O'Connor, M., Erez, M., Burger, D., Fussell, D. S., and Redder, S. W., "Priority-based cache allocation in throughput processors," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2015, pp. 89–100.

[42] Sethia, A., Jamshidi, D., and Mahlke, S., "Mascar: Speeding up GPU warps by reducing memory pitstops," *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, Feb 2015, pp. 174–185.

[43] Zhao, X., Wang, Z., and Eeckhout, L., "Classification-Driven Search for Effective SM Partitioning in Multitasking GPUs," *Proceedings of the 2018 International Conference on Supercomputing*, ACM, 2018, pp. 65–75.

[44] Wu, B., Chen, G., Li, D., Shen, X., and Vetter, J., "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," *Proceedings of the 29th ACM on International Conference on Supercomputing*, ACM, 2015, pp. 119–130.

[45] Chen, G., Zhao, Y., Shen, X., and Zhou, H., "EffiSha: A software framework for enabling effficient preemptive scheduling of GPU," *ACM SIGPLAN Notices*, Vol. 52, ACM, 2017, pp. 3–16.

[46] Buluç, A. and Gilbert, J. R., "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, Vol. 25, No. 4, 2011, pp. 496–509.

[47] Gilbert, J. R., Reinhardt, S., and Shah, V. B., "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, Vol. 10, No. 2, 2008, pp. 20–25.

[48] Van Dongen, S. M., *Graph clustering by flow simulation*, Ph.D. thesis, 2000.

[49] Duff, I. S., Heroux, M. A., and Pozo, R., "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum," *ACM Transactions on Mathematical Software (TOMS)*, Vol. 28, No. 2, 2002, pp. 239–267.

[50] Tech, A., "NVIDIA Launches Tesla K40," `http://www.anandtech.com/show/7521/nvidia-launches-tesla-k40`, 2013.

[51] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al., "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, Vol. 39, No. 2, 2011, pp. 1–7.

[52] Davis, T. A. and Hu, Y., "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, Vol. 38, No. 1, 2011, pp. 1.

[53] Leskovec, J. and Krevl, A., "SNAP Datasets: Stanford Large Network Dataset Collection," `http://snap.stanford.edu/data`, June 2014.

[54] Chapanond, A., Krishnamoorthy, M. S., and Yener, B., "Graph Theoretic and Spectral Analysis of Enron Email Data," *Computational & Mathematical Organization Theory*, Vol. 11, No. 3, Oct 2005, pp. 265–281.

[55] Lee, S., Kim, K., Koo, G., Jeon, H., Ro, W. W., and Annavaram, M., "Warped-compression: enabling power efficient GPUs through register compression," *ACM SIGARCH Computer Architecture News*, Vol. 43, ACM, 2015, pp. 502–514.

[56] Liu, Z., Gilani, S., Annavaram, M., and Kim, N. S., "G-Scalar: Cost-effective generalized scalar execution architecture for power-efficient GPUs," *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2017, pp. 601–612.

[57] Kloosterman, J., Beaumont, J., Jamshidi, D. A., Bailey, J., Mudge, T., and Mahlke, S., "Regless: just-in-time operand staging for GPUs," *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2017, pp. 151–164.

[58] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K., "Rodinia: A Benchmark Suite for Heterogeneous Computing," *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 44–54.

[59] Yavits, L. and Ginosar, R., "Sparse Matrix Multiplication on CAM Based Accelerator," *CoRR*, Vol. abs/1705.09937, 2017.

[60] Lin, C. Y., Zhang, Z., Wong, N., and So, H. K. H., "Design space exploration for sparse matrix-matrix multiplication on FPGAs," *2010 Int'l Conference on Field-Programmable Technology*, Dec 2010, pp. 369–372.

[61] Gebhart, M., Johnson, D. R., Tarjan, D., Keckler, S. W., Dally, W. J., Lindholm, E., and Skadron, K., "Energy-efficient mechanisms for managing thread context in throughput processors," *ACM SIGARCH Computer Architecture News*, Vol. 39, ACM, 2011, pp. 235–246.

[62] Gebhart, M., Johnson, D. R., Tarjan, D., Keckler, S. W., Dally, W. J., Lindholm, E., and Skadron, K., "A hierarchical thread scheduler and register file for energy-efficient throughput processors," *ACM Transactions on Computer Systems (TOCS)*, Vol. 30, No. 2, 2012, pp. 8.

[63] Gebhart, M., Keckler, S. W., and Dally, W. J., "A compile-time managed multi-level register file hierarchy," *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2011, pp. 465–476.

[64] Jeon, H., Ravi, G. S., Kim, N. S., and Annavaram, M., "GPU register file virtualization," *Proceedings of the 48th International Symposium on Microarchitecture*, ACM, 2015, pp. 420–432.

[65] Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C., "Base-delta-immediate compression: practical data compression for on-chip caches," *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ACM, 2012, pp. 377–388.

[66] Lee, S., Kim, K., Koo, G., Jeon, H., Ro, W. W., and Annavaram, M., "Warped-compression: enabling power efficient GPUs through register compression," *ACM SIGARCH Computer Architecture News*, Vol. 43, ACM, 2015, pp. 502–514.

[67] Stephenson, M., Sastry Hari, S. K., Lee, Y., Ebrahimi, E., Johnson, D. R., Nellans, D., O'Connor, M., and Keckler, S. W., "Flexible software profiling of gpu architectures," *ACM SIGARCH Computer Architecture News*, Vol. 43, ACM, 2015, pp. 185–197.

[68] Grisenthwaite, R., "Armv8 technology preview," *IEEE Conference*, 2011.

[69] Greenhalgh, P., "Big. little processing with arm cortex-a15 & cortex-a7," *ARM White paper*, Vol. 17, 2011.

[70] GNU, "GNU String Library," https://www.gnu.org/software/libc/manual/html_node/String-and-Array-Utilities.html, Accessed: July 2019.

[71] Kerrisk, M., "Linux Programmer's Manual: setjmp," http://man7.org/linux/man-pages/man3/setjmp.3.html, Accessed: July 2019.

[72] Kerrisk, M., “Linux Programmer’s Manual: longjmp,” http://man7.org/linux/man-pages/man3/longjmp.3p.html, Accessed: July 2019.