OPTIMIZED SCHEDULING AND RESOURCE ALLOCATION
FOR THREAD PARALLEL ARCHITECTURES

A Dissertation
Presented to
The Academic Faculty

By

Sana Damani

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

May 2022

© Sana Damani 2022
OPTIMIZED SCHEDULING AND RESOURCE ALLOCATION
FOR THREAD PARALLEL ARCHITECTURES

Thesis committee:

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Thomas Conte
School of Computer Science
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and
Computer Engineering
Georgia Institute of Technology

Date approved: April 14, 2022
Ever Tried. Ever Failed.
No matter.
Try again. Fail again.
Fail better.
_Samuel Beckett_
ACKNOWLEDGMENTS

I would first like to thank my advisor Dr. Vivek Sarkar for his guidance over the last five years. I would also like to thank the members of my thesis committee for their insights in preparation of this dissertation. The work described in this dissertation was done in collaboration with many others, including my colleagues and co-authors in the Habanero Extreme Scale Software Research Lab and my mentors, Mark Stephenson, Daniel Johnson, Ram Rangan and Steve Keckler, who helped guide my research over the span of several internships at NVIDIA. Finally, I would like to thank my parents for supporting my education and for instilling in me the desire to learn.
# TABLE OF CONTENTS

Acknowledgments ................................................................. iv

List of Tables ................................................................. x

List of Figures ................................................................. xi

Summary ......................................................................... xiv

Chapter 1: Introduction ......................................................... 1
  1.1 Contributions .............................................................. 3
    1.1.1 Instruction Reordering .............................................. 3
    1.1.2 Thread Scheduling .................................................. 4
    1.1.3 Resource Allocation ............................................... 5
  1.2 Thesis Statement ........................................................ 6
  1.3 Organization ............................................................. 6

Chapter 2: Background ......................................................... 8
  2.1 SIMT Processors ........................................................ 8
    2.1.1 Handling Divergence .............................................. 9
  2.2 Migratory Thread Processors ....................................... 11
    2.2.1 Emu Programming Model ........................................ 12
Chapter 3: Common Subexpression Convergence

3.1 Motivation

3.2 Problem

3.3 Solution

3.3.1 Identifying Divergent Branches

3.3.2 Identifying Common Subexpression DAGs

3.3.3 Legality of Code Motion

3.3.4 Time Complexity

3.4 Evaluation

3.5 Discussion

3.6 Future Work

Chapter 4: Speculative Reconvergence

4.1 Motivation

4.2 Problem

4.3 Solution

4.3.1 Reconvergence point

4.3.2 Synchronization Algorithm

4.3.3 Deconfliction

4.3.4 Interprocedural Analysis

4.3.5 Automatic Detection of Reconvergence Point

4.3.6 Soft Barrier
LIST OF TABLES

3.1 Common Subexpression Convergence: Experimental Setup . . . . . . . . . . . 26

4.1 Speculative Reconvergence: Synchronization Primitives. . . . . . . . . . . . 39

4.2 Speculative Reconvergence: Benchmarks. . . . . . . . . . . . . . . . . . . 46

5.1 Subwarp Interleaving: Experimental Setup. . . . . . . . . . . . . . . . . . . 69

5.2 Subwarp Interleaving: Raytracing applications . . . . . . . . . . . . . . . . 70

5.3 Subwarp Interleaving: Microbenchmark Results . . . . . . . . . . . . . . 71

7.1 Memory Access Scheduling: Experimental Setup . . . . . . . . . . . . . . . . 102

7.2 Memory Access Scheduling: Explicitly-parallelized kernels with sources . . 102

7.3 Memory Access Scheduling: Impact on Compilation Time . . . . . . . . . 104
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Microprocessor Trend Data</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Turing-like SM architecture.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>GPU Thread Model</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Divergence Handling</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Data Allocation</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Emu Architecture</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Program Dependence Graph</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Analysis</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Optimizing Array Accesses</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Optimizing Predicated Instructions</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Common Subexpression Convergence: Microbenchmark Results</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Speculative Reconvergence: an illustration</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Pseudocode and execution pattern for RSBench</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Pseudocode for motivating use cases</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>CFG representation for Iteration Delay</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Deconfliction</td>
<td>42</td>
</tr>
<tr>
<td>4.6</td>
<td>SIMT efficiency</td>
<td>47</td>
</tr>
</tbody>
</table>
4.7 SIMT efficiency versus speedup. ....................................... 48
4.8 SIMT efficiency and speedup with soft barrier. .................. 49
4.9 Automatic Speculative Reconvergence. ............................ 50

5.1 Divergent Shader Execution ........................................... 54
5.2 Characteristics of RayTracing applications ......................... 55
5.3 Subwarp Interleaving .................................................... 56
5.4 Subwarp Interleaving SM architecture ............................... 59
5.5 Thread status state machine ........................................... 60
5.6 Thread Status Table ..................................................... 63
5.7 Stalls in Divergent Paths: An example ............................... 66
5.8 TST Operation ......................................................... 67
5.9 Microbenchmark ....................................................... 71
5.10 Subwarp Interleaving Results ....................................... 72
5.11 Sensitivity to L1 miss latency ....................................... 74
5.12 Sensitivity to number of warp slots. ................................. 75
5.13 Sensitivity to number of subwarps (at 32 peak warps). .......... 76

6.1 Impact of larger register file .......................................... 79
6.2 Raytracing megakernel .................................................. 80
6.3 Battlefield V ............................................................ 81
6.4 Raytracing kernel with register file sharing ......................... 82
6.5 Register File Sharing: Evaluation Methodology .................... 84
6.6 Speedup and slow path execution cycles for Battlefield V with RF sharing. 85
7.1 Thread migrations in SpMV ........................................... 89
7.2 Lattice for layout analysis ............................................. 92
7.3 Memory Access Scheduling as Sequential Ordering Problem .... 95
7.4 SPMV: Heuristic-based Scheduling. .............................. 100
7.5 Heuristic scheduler versus ILP scheduler. ...................... 101
7.6 Migrations and Speedup for explicitly-parallelized kernels. .... 103
7.7 Polybench Speedup ...................................................... 104
7.8 Migration Intensity for explicitly-parallelized benchmarks .... 106
While accelerators show significant performance improvements for applications with high data parallelism and regular memory accesses, they experience synchronization and memory access overheads in applications with irregular control flow and memory access patterns resulting in reduced efficiency [1]. Examples include graph applications, Monte Carlo simulations, ray tracing applications, and sparse matrix computations. This proposal focuses on identifying inefficiencies in executing irregular programs on thread-parallel architectures, and developing compiler transformations to address these inefficiencies. In particular, we describe instruction reordering, thread scheduling and resource allocation techniques that avoid serialization and reduce pipeline stalls on GPUs and minimize thread migrations on the EMU, thereby reducing overall program latency and improving processor utilization.
CHAPTER 1
INTRODUCTION

Moore’s law states that the number of transistors per chip doubles every two years[2]. This trend has continued as seen in the linear increase in the number of transistors per chip in figure Figure 1.1. On the other hand, Dennard scaling[3], which states that power decreases as the size of transistors decreases, ended in 2006 due to the power wall, resulting in a plateauing of single core frequency and performance. This combination of increasing transistor count and the end of Dennard scaling has triggered the rise of parallel architectures, including multicore CPUs, SIMT architectures such as the GPU, and migratory thread processors such as the EMU. While this trend towards parallel architectures promises continued performance scaling, writing efficient parallel code that effectively make use of hardware resources can be challenging for programmers. While compiler techniques, including automatic parallelization and thread reconvergence for GPUs, can help bridge this gap in performance, there is a lot of room for improvement. In particular, raytracing, monte carlo simulations and sparse applications have irregular program structures and fail to efficiently utilize parallel hardware. This dissertation describes compiler and architecture techniques for thread-parallel architectures aimed at increasing concurrency to improve the performance of irregular parallel programs executing on SIMT and migratory thread processors.

In particular, this dissertation focuses on optimizations for the GPU and EMU architectures. The GPU is a Single-Instruction Multiple-Thread or SIMT architecture. The GPU has thousands of threads running in parallel. It groups these threads into SIMT units called warps that execute the same instruction on different data in a lock-step manner. Originally designed for computer graphics, GPUs are widely used today for general purpose HPC applications and for machine learning. The EMU, on the other hand, is a highly-scalable PGAS system architecture aimed at handling irregular memory access patterns in
graph and sparse matrix applications by migrating threads to data instead of data to threads. Threads on the EMU system execute independently in a Multiply-Instruction Multiple-Data or MIMD manner. Whereas the GPU is programmed for HPC applications using CUDA, OpenCL or OpenMP with explicit host and device code regions, the EMU employs a modified Cilk programming model with no explicit thread migrations or data transfer. While performance on the GPU relies in part on data locality across threads in a warp, the EMU hardware specifically accelerates applications without data locality. The major sources of inefficiency on the GPU include thread divergence, memory divergence and low occupancy. On the EMU, the major source of inefficiency is suboptimal thread migration patterns. Finally, while the GPU aims at maximizing compute throughput, the EMU aims at minimizing memory bandwidth. We describe both these architectures in detail in the Chapter 2 and refer to them collectively as thread-parallel architectures.

---

1This microprocessor trend data plot is available under a Creative Commons Attribution 4.0 International Public License.
1.1 Contributions

This section provides a summary of the motivation, analysis and transformation of each optimization technique developed in our work.

1.1.1 Instruction Reordering

Instruction reordering techniques such as list scheduling, loop invariant code motion and partial redundancy elimination all work towards improving program performance in serial applications by eliminating redundant computations or reducing pipeline stalls. We propose two instruction reordering optimizations: Memory Access Scheduling to avoid redundant thread migrations, and Common Subexpression Elimination to increase SIMT efficiency.

Memory Access Scheduling

Memory Access Scheduling is an instruction scheduling optimization aimed at reducing the number of thread migrations on near memory processors. The analysis step identifies the data layout of memory access instructions and determines if accesses are co-located. We propose an interprocedural data-flow analysis for data layout identification and use this information to compute the distance between two memory accesses which gives us co-location information. The transformation step is a list scheduler that groups together co-located memory access instructions to minimize thread migrations. We propose both an optimal instruction linear programming approach and a more efficient, greedy heuristic-based scheduler and compare their impact on speedup and overall compile time.

Common Subexpression Convergence

Common Subexpression Convergence is an instruction reordering technique aimed at reducing the serialization impact of thread divergence on SIMT processors by moving common code across divergent branch paths to convergent regions to encourage parallel exe-
cution of common instructions. The analysis step identifies common subexpression trees across branch paths and uses a dynamic programming approach to compute the cost-benefit analysis of common subexpression convergence. The transformation step uses a program dependence graph to legally hoist or sink common subexpression trees to convergent paths before or after the branch respectively. If neither is possible, the pass splits the divergent branch into two and moves the common subexpression tree to the new intermediate convergent region.

1.1.2 Thread Scheduling

Thread scheduling techniques involve deciding an order of execution of multiple threads on limited resources. On SIMT processors such as GPUs, this involves warp scheduling to minimize pipeline stalls or divergent thread scheduling to avoid serialization. We propose two thread scheduling optimizations: Speculative Reconvergence to increase SIMT efficiency within expensive code regions, and Subwarp Interleaving to reduce pipeline stalls by interleaving the execution of divergent threads within a warp.

Speculative Reconvergence

Speculative Reconvergence is a thread reconvergence barrier placement technique for SIMT processors aimed at minimizing the run-time impact of thread divergence. Traditionally, compilers for SIMT processors place thread reconvergence barriers at the postdominator of divergent branches where all threads in a warp are guaranteed to arrive. Our proposal identifies early reconvergence points within loops where threads can reconverge, often across loop iterations, before executing common code in parallel. Effectively, speculative reconvergence reduces thread divergence in more expensive code regions at the cost of increased thread divergence in non-critical parts of the programming. The analysis step identifies the new reconvergence location as well as the candidate threads for reconvergence. We propose both a user-hint based and an automated static analysis to identify opportunity for
speculative reconvergence. The transformation step uses data-flow analysis to insert barrier instructions. We further extend speculative reconvergence with a soft barrier that allows for more balanced convergence across the program and reduces the cost of serialization on non-critical program paths.

Subwarp Interleaving

Subwarp Interleaving is a hardware thread scheduling technique that leverages thread divergence in SIMT processors to hide pipeline stalls in programs with low warp occupancy. While today’s GPUs serialize the execution of divergent paths in a program, subwarp interleaving wakes up sleeping threads in case of a pipeline stall if there are no other warps ready to execute, thereby interleaving the execution of diverged paths in the program and reducing stalls. We propose a hardware technique that identifies stalls in divergent code paths, identifies an appropriate subwarp to switch to and finally switches to the new subwarp. We study the performance impact of several scheduling heuristics including the use of an eager versus a just-in-time approach to switching subwarps, and varying the number of warp slots, the number of stalled warps and the number of active subwarps per warp.

1.1.3 Resource Allocation

We propose a dynamic register allocation technique to increase occupancy in applications with high variation in register usage on GPUs.

Software-Directed Register File Sharing

Software-Directed Register File Sharing is a software-hardware co-design approach that employs dynamic register allocation to time share registers across warps to increase overall warp occupancy. While the GPU compiler today allocates the maximum required registers to a warp at the onset and only deallocates them once thread block execution finishes. This approach ignores the observations that some applications only use maximum resources
for a short duration of their execution and for the remaining time, the allocated registers stay idle. In our proposal, the compiler only allocates a minimum number of registers to each warp at the onset, resulting in higher occupancy and better latency hiding. When the warp reaches a high resource utilization part of the program, it acquires registers and then releases them back to the free pool once they are not longer needed. We further propose a deadlock handling mechanism that employs a slow-path inserted by the compiler. We first characterise opportunity for register file sharing by studying register usage over time in Raytracing applications. We then study the impact of register file sharing on warp occupancy and slow path execution time.

1.2 Thesis Statement

*Compile-time approaches to optimized scheduling and resource allocation help irregular programs utilize thread-parallel architectures more efficiently.*

1.3 Organization

The rest of this document is organized as follows. We first provide background on architectures targeted by our transformations in Chapter 2. In Chapter 7 and Chapter 3, we describe our instruction reordering techniques for different optimization goals for different architectures. While Common Subexpression Convergence is a cross-block code motion transformation that identifies and moves common code from divergent paths with the aim of minimizing thread divergence on SIMT Processors, Memory Access Scheduling introduces a new kind of instruction scheduler that groups co-located memory accesses so as to reduce thread migrations on Migratory Thread Processors. In Chapter 4 and Chapter 5, we describe our thread scheduling techniques for SIMT processors. Whereas Speculative Reconvergence modifies thread convergence barriers to reduce thread divergence on GPUs, Subwarp Interleaving leverages thread divergence to reduce pipeline stalls. Chapter 6 details Software-Directed Register File Sharing, a dynamic register allocation technique that
also leverages thread divergence to increase occupancy on GPUs. Chapter 8 discusses related work aimed at increasing SIMT efficiency and occupancy, and reducing thread migrations. Finally, Chapter 9 summarizes our conclusions.
CHAPTER 2
BACKGROUND

2.1 SIMT Processors

GPUs are parallel, throughput oriented processors that exploit data parallelism to accelerate applications [4]. GPUs are known as Single Instruction Multiple Thread, or SIMT, processors, because they have a large number of threads grouped into blocks where each thread executes the same instruction in parallel on different data. Initially designed to accelerate graphics processing, researchers soon discovered their potential to accelerate high performance computing applications, resulting in the invention of the GPGPU and the CUDA programming language.

A CUDA kernel launches a grid of threads organized into thread blocks, each of which is further broken down into warps as shown in Figure 2.2. A warp is a unit of SIMT execution that consists of 32 threads that all fetch the same instruction and execute it in a lock-step manner on different data across threads. All threads in a kernel can access the same off-chip global memory. Threads within a block access faster on-chip shared memory that acts as a programmable scratchpad. Finally, threads access thread-local memory and registers. Since threads share registers, shared memory and warp slots, the number of threads that can run on an SM (or Streaming Multiprocessor) at a time is limited by per-thread resource utilization. GPU Occupancy refers to the ratio of warps active to the maximum number of warps supported on an SM.

Figure 2.1 shows a diagram of an Nvidia Turing-like SM architecture. The SM contains four processing blocks, each of which contains a front-end that feeds the processing block with instructions; an L0 instruction cache; 8 “warp slots” that hold warp-private scheduling data (e.g., program counters, scoreboard state, etc.) for the warps that map to the process-
2.1.1 Handling Divergence

When data-dependent conditional code is encountered on SIMD architectures, predication may be used to disable execution of certain data paths. SIMT architectures on the other hand allow each thread to branch independently depending on the evaluation of the branch condition. This thread divergence is supported by masking off the execution of a subset of threads where the hardware serializes the execution of the taken and not-taken paths of the branch (see Figure 2.3). When their paths merge, the compiler reconverges the threads in the warp so that they resume parallel execution. Performance in SIMT architectures relies on maximizing parallel execution of code by multiple threads within a warp, which we measure using a metric known as SIMT efficiency, the average percentage of active threads per warp.

While pre-Volta GPUs use a stack based mechanism to handle nested control divergence [6], Volta introduces the independent thread scheduling model which allows for finer grained synchronization patterns by maintaining per-thread state information. Further,
Figure 2.2: GPU Thread Model

Figure 2.3: Divergence Handling
Volta’s convergence optimizer maximizes SIMT efficiency by grouping together threads that execute the same code in parallel for maximum convergence.

The Volta ISA provides synchronization instructions that enable implementation of independent thread scheduling [7, 8, 9]: $BSSY$, $BSYNC$, and $BREAK$. Threads that encounter a $BSSY$ join a convergence barrier which is represented using a barrier register. $BSYNC$ synchronizes all threads that previously joined the corresponding convergence barrier. $BREAK$ removes a thread from the specified convergence barrier, a mechanism used to handle non-standard convergence points such as in loops or in short-circuit code.

As described in [7], the Volta GPU architecture guarantees forward-progress by ensuring that no thread is suspended indefinitely, to allow for arbitrary synchronization without resulting in a live-lock state. It achieves this by enabling the GPU to yield execution of threads. Furthermore, unlike barriers used for synchronization such as $syncthreads$, these barriers optimize performance by minimizing divergent execution of code and are not required for correctness of the program.

2.2 Migratory Thread Processors

**EMU**

The basic unit of processing in the Emu architecture is a *nodelet* shown in Figure 2.5b [10]. A nodelet consists of multiple *gossamer cores*, general-purpose pipelined processors that execute threads, a *nodelet queue manager* that handles thread scheduling and migration, a memory front-end that handles memory transactions, and a portion of the global address space.

A group of 8 nodelets along with a *migration engine*, a cross-bar that migrates threads from the source nodelet to destination nodelet, and a *stationary core* forms a *node* (see Figure 2.5a). An Emu system can have multiple connected nodes. The architecture has no memory hierarchy and all accesses reference a global memory address space.
Thread Migration

When a thread running on a nodelet executes a memory fetch instruction, the hardware first checks if the requested data is present on the current nodelet. If not, the hardware migrates the thread context to the nodelet that holds the requested data. Hence, threads migrate automatically without programmer intervention.

A thread context is compact, consisting of a thread status word and address and data registers [10]. The program itself is maintained in replicated memory and can be accessed from any nodelet, and does not need to be transferred with the thread context. Hardware thread migration on the Emu is a cheap operation relative to thread migration implementations in software, but it can still become a performance bottleneck.

2.2.1 Emu Programming Model

Emu uses the Cilk programming model with the following extensions to C [11]:

- `cilk_spawn` creates a new child thread that can execute in parallel with the parent. A parent thread can optionally spawn a child on a remote nodelet with a migrate hint.
• *cilk_sync* forces the parent to wait for all its spawned children to finish execution before proceeding.

• *cilk_for* spawns new threads for each iteration of a for loop, which can execute in parallel, and then waits for all the iterations to complete.

2.2.2 Data Allocation

Emu-Cilk provides the following functions for dynamic allocation of data structures distributed across nodelets [12].

• *mw_malloc(N)* allocates memory of size N on the local nodelet.

• *mw_localmalloc(N, ptr)* allocates memory of size N on the same nodelet as ptr.

• *mw_mallocrep(N)* allocates memory of size N on each nodelet [13]. A copy of a replicated data structure can be found on each nodelet.

• *mw_malloc1dlong(N)* allocates a 1D array of N long elements striped across nodelets in a round-robin manner (see Figure 2.4a).

• *mw_malloc2d(B, N)* allocates a 2D array of B blocks of N elements each. The blocks are striped across nodelets in a round-robin manner whereas elements within a block are co-located on a single nodelet (see Figure 2.4b for an example with B=4).

Other migratory thread processors

On Non-Uniform Memory-Access or NUMA systems, the latency of data access depends on where the data is located. NUMA-aware operating systems co-locate data with the threads that access them to minimize this latency. A variant of NUMA that migrates threads on remote access was shown by Li et al to be up to 2x faster than data shuffling[14].

In 1995, Rogers et al proposed using thread migrations as a tool for automatic parallelization of programs that use dynamic data structures to handle the problem of assigning
threads to processors when the layout is statically unknown [15]. In 1996, Jenks et al proposed the Nomadic Threads run-time system[16]. Similar to EMU threads, Nomadic threads transfer themselves to the processor that contains data that it requires instead of fetching data from the remote processor with the goal of reducing messages and exploiting locality of access within the target processor.
Figure 2.5: Emu Architecture [10]
CHAPTER 3
COMMON SUBEXPRESSSION CONVERGENCE

3.1 Motivation

We begin with Common Subexpression Convergence, a cross-block instruction scheduling technique that identifies common code in divergent paths and moves it to convergent regions to increase SIMT efficiency on GPUs[17].

In Chapter 2, we described SIMT processors such as GPUs that execute the same instruction for multiple threads in parallel such that each thread evaluates the same instruction differently based on thread-local data. Furthermore, as described in Section 2.1.1, GPUs handle divergent branches by serializing the execution of threads along diverged paths [18]. Increased divergence results in reduced SIMT efficiency, and common code along divergent paths increases redundancy. We propose identifying such common code in divergent paths and moving it to convergent regions to increase SIMT efficiency.

Our technique was inspired by Common Subexpression Elimination [19], a compiler technique that identifies common subexpressions that compute the same value for a single thread along the same path and replaces them with a single value to reduce redundant execution of code. Common Subexpression Convergence on the other hand identifies common subexpressions along divergent paths that may compute different values along each path for different threads and moves them to convergent code paths to avoid serialized execution of redundant instructions. Instead of values, we focus on identifying a common expression tree structure. Further, we propose an operand renaming pre-pass depicted in Listing 3.1.

We identify three cross-block code motion transformations as part of our Common Subexpression Convergence framework:

- **Hoist** moves common code to the nearest convergent control flow point before threads
diverge at the branch, as illustrated in Listing 3.1. This is only legal if all incoming
definitions are located at or before the branch, or can also be hoisted.

- **Sink** moves common code to the nearest convergent control flow point *after* threads
  reconverge at the postdominator, as illustrated in Listing 3.1. This is only legal if all
  uses of the instruction and redefinitions of its operands can also be moved to the join
  point.

- **Split** moves common code that can neither be hoisted nor sunk to an *intermediate*
  temporary reconvergence point within the divergent region, as illustrated in Listing
  3.1. The common code can be moved to this region before threads diverge once
  again. As with Hoist and Sink, this transformation is guaranteed to result in a SIMT
  efficiency that is greater than or equal to the prior SIMT efficiency. However, Split
  can sometimes introduce more overhead compared to the Hoist and Sink transforma-
  tions due to branch duplication and extra synchronization instructions. Note that
  Split is always legal.

<table>
<thead>
<tr>
<th>Listing 3.1: Before Hoist</th>
<th>Listing 3.2: After Hoist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b = ...</td>
</tr>
<tr>
<td>2</td>
<td>c = ...</td>
</tr>
<tr>
<td>3</td>
<td>if (threadId % 2) {</td>
</tr>
<tr>
<td>4</td>
<td>a = b * c</td>
</tr>
<tr>
<td>5</td>
<td>use a</td>
</tr>
<tr>
<td>6</td>
<td>} else {</td>
</tr>
<tr>
<td>7</td>
<td>a = b * c</td>
</tr>
<tr>
<td>8</td>
<td>}</td>
</tr>
</tbody>
</table>
Listing 3.3: Before Sink

```c
1   c = ...
2   if (threadId % 2) {
3       b = ...
4       a = b * c
5     } else {
6       a = b * c
7     }
8     use a
```

Listing 3.4: After Sink

```c
1   c = ...
2   if (threadId % 2) {
3       b = ...
4     } else { // sink
5       a = b * c
6     }
7     use a
```

Listing 3.5: Before Split

```c
1   b = ...
2   c = ...
3   if (threadId % 2) {
4       a = b * c
5     } else {
6       a = b * c
7     }
```

Listing 3.6: After Split

```c
1   b = ...
2   c = ...
3   if (!threadId % 2) // reconverge
4       b = ...
5     } if (threadId % 2) // split
6     a = b * c
7     use a
```

Listing 3.7: Before Renaming

```c
1   if (threadId % 2) {
2       a = b * c
3    } else {
4       // operand mismatch
5       a = d * c
6    }
7     use a
```

Listing 3.8: After Rename and Sink

```c
1   if (threadId % 2) {
2       x = b
3    } else {
4       x = d
5    }
6       a = x * c
7     use a
```
3.2 Problem

We summarize the problem statement for Common Subexpression Convergence as follows:

**Problem Statement.** Given a GPU program, identify the best combination of hoist, sink and split transformations to move common operations to a convergent region that maximizes the profit of common subexpression convergence and maintains dependencies.

3.3 Solution

We shall use Listing 3.3 as our running example to explain this algorithm.

Listing 3.9: Common Subexpression Convergence: Running Example

```plaintext
if (divergent_condition) {
    a = b * c
    d = a + c
    e = d + d
} else {
    a = b * c
    f = a / c
    e = f * f
}
```

We begin by building a program structure tree (PST) which is a hierarchical representation of single-entry single-exit regions of a control flow graph [20]. For any two single-entry single-exit regions in a program, the regions must either be disjoint, or one must be nested within the other, resulting in a tree structure. This approach can be extended to single-entry multiple-exit regions using the forward control dependence graph.

We traverse this PST from leaves to the root, thereby ensuring that the innermost regions are handled first. This helps optimize programs with nested conditionals and loops. At each region, we build a program dependence graph (PDG), a directed graph where nodes represent instructions in the region and edges between nodes represent control or data de-
dependence between instructions [21]. We use the control dependence edges to find operations that are control dependent on true and false edges of the same divergent branch, and data dependence edges to generate expression graphs and determine legality of code motion. Figure 3.1a shows the CFG for Listing 3.3 and Figure 3.2 and Figure 3.1b show the PDG and PST respectively.

3.3.1 Identifying Divergent Branches

Threads in a warp may diverge at a conditional branch if the branch condition depends on a thread-varying value. A thread-varying value is any value that is neither a compile time known constant nor a global read from a thread-uniform location. If any of the inputs to an instruction is thread-varying, the result of the instruction is also thread-varying.

We rely on use-def information to determine if a value is potentially thread-varying [22]. Any branch with a thread-varying condition is marked as divergent. Note that the branch may not in fact be divergent at run-time if all threads evaluate the condition identically, but at compile time we attempt to optimize all potentially divergent branches. We can use profile-guided compilation to improve the accuracy of divergence detection.
Algorithm 1: Common Subexpression Convergence: Hoist/Sink/Split

**Input**: Program \( P \), threshold

**Output**: Optimized Program \( P' \)

1. **Function** `CommonSubexpressionConvergence(P)`
   
   ```plaintext
   BuildPST();
   for each region \( R \) in the PST in bottom up order do
       BuildPDG(R);
       if `BranchIsDivergent(R.root)` then
           for each instruction \( I_1 \) backwards in \( R.child_0 \) do
               for each instruction \( I_2 \) backwards in \( R.child_1 \) do
                   \( D_1 = \text{BuildDAG}(I_1) \);
                   \( D_2 = \text{BuildDAG}(I_2) \);
                   \( \text{Profit} = \text{MatchProfit}(D_1, D_2) \);
                   if \( \text{Profit} < \text{threshold} \) then
                       continue;
                   if `LegalToHoist(D_1, D_2)` then
                       Hoist(D_1, D_2, R);
                   else if `LegalToSink(D_1, D_2)` then
                       Sink(D_1, D_2, R);
                   else
                       Split(D_1, D_2, R);
               end
           end
       end
   FlattenBranches();
   end
   ```
3.3.2 Identifying Common Subexpression DAGs

Given two expression DAGs that are control dependent on the same divergent branch, our goal is to detect the maximally profitable matching expression DAG.

• **Benefit:** Decrease in the number of divergently executed instructions. The benefit of moving a function call or loop into common code path may be much greater than the benefit of moving an arithmetic instruction, and we therefore introduce a weighted model for benefit computation, biased towards convergent execution of more expensive instructions. In particular, increase in coalesced memory accesses can have a large impact on performance.

• **Cost:** Increase in the number of copy instructions needed to handle operand renaming as shown in Listing 3.1. Note that some copy instructions may be eliminated by a later register coalescing pass that assigns the same register to both the original and renamed operands.

We formally define the expression DAG to be a directed graph $G$ such that leaves repre-
sent variables, interior nodes represent operators, and an edge from a to b indicates that b is an input to a. We assume an SSA representation so that each variable is defined only once. Our goal is to find a subgraph matching that maximizes $Profit = (Benefit - Cost)$.

**Algorithm 2:** Common Subexpression Convergence: A Dynamic Programming Solution to Finding Maximally Profitable Common Subgraphs

**Input:** Two Expression DAGs control dependent on the same divergent branch

**Output:** $T(i, j)$: Profit of matching expression DAGs with roots $i$ and $j$

1. **Initialization:** $T(i, i)$ where $i$ is a leaf node is trivially matched as having 0 profit while $T(i, j)$ where both $i$ and $j$ are leaf nodes is matched as having +1 profit required for the copy instruction.

2. **Recurrence:** We then define the recurrence to compute the profit of matching interior nodes, $T(i, j)$ as:
   - $T(i, j) = \text{cost of copy insertion, if } i \text{ and } j \text{ have an operator mismatch}$
   - $T(i, j) = \text{benefit of matching operator} + \text{sum(max(match subgraph, insert copy))}$ where, $i$ and $j$ belong to different expression DAGs.

Algorithm 2 shows the dynamic programming implementation that determines the profit of moving every combination of expression DAGs to the convergent region. We pick the pair of expression DAGs with the maximum profit, decide legality of hoist/sink/split and then perform code motion in accordance with Algorithm 1. We repeat this procedure in each divergent branch within the PST in a bottom-up manner. Figure 3.3 shows the expression DAGs and dynamic programming table for the running example in Listing 3.3.
Handling Nested Branches: After each region is processed, we flatten the branches within the region to handle common sub-regions nested within outer regions. Branch flattening or predication eliminates branches in the region and converts control dependencies to data dependencies. Our algorithm treats these flattened branches as regular expression trees when handling the parent region. See Listing 3.11 for an example where branch flattening enabled additional optimization.

Handling Loops in Divergent Branches: Branch flattening is insufficient for loops nested within regions which require special handling. When comparing two sub-regions within a region, we compare the loop bodies and iteration domains to detect common code and opportunity for optimization. If loop iteration domains are non-identical, we may use loop peeling or index-set splitting to generate identical loops that can be hoisted. If the loop bodies are only partially common, we use loop distribution to separate out the common operations before performing code motion.

Listing 3.10: Nested Common Code

```plaintext
1 if (threadId % 2) {
2    // common code
3    a = b * c;
4    use a;
5  } else {
6 if (threadId % 3) {
7    // common code
8    a = b * c;
9  } } else {
10  }
```

Listing 3.11: Branch Flattening

```plaintext
1 if (threadId % 2) {
2    p = true
3  } else {
4  p = threadId % 3;
5  }
6 if (threadId % 3) {
7    // flattened common code
8    (p) a = b * c;
9  } else {
10  }
```

Handling Array Accesses: The dynamic programming model can handle array accesses by including the array access operator as a node within the expression DAG with the base
address and index as inputs. The transformation step will require an alias analysis pre-pass to ensure all dependencies are maintained during code motion.

**Handling Predicated Instructions:** A divergent branch that is flattened can be optimized by including the predicate value as one of the inputs to the operation. The default predicate value associated with all operations is True.

```
if (condition) {
  p = ...;
  if p, a = b * c;
}
else {
  a = b * c;
}
```

```
if (condition) {
  p = ...;
  if p, a = b * c;
}
else {
  p = True;
}
```

**Figure 3.5: Optimizing Predicated Instructions**

### 3.3.3 Legality of Code Motion

A reordering transformation that preserves every dependence preserves the meaning of the program [23]. We show that our code motion transformation preserves all dependencies and is therefore legal.

- **True dependencies:** True or flow dependencies are read-after-write dependencies that may prevent hoisting of an instruction that reads an operand beyond the definition of any of its inputs. We only hoist code if (1) all input operand definitions can also be hoisted, or (2) there are no intervening definitions of any of its operands. Note
that splitting doesn’t change the order of operations and therefore has no dependence constraints. When sinking code, we must ensure that a definition is not moved past its use. To ensure this, we only sink code if (1) all its uses can also be sunk, or (2) there are no intervening uses.

- **False dependencies:** We eliminate anti- and output dependencies using operand renaming. Hence, our optimization does not violate false dependencies.

- **Control dependencies:** We only hoist code that is executed along all paths of a divergent branch. Moving this code to a block that is not control dependent on the branch is therefore legal.

### 3.3.4 Time Complexity

We present an approximate solution to optimal expression DAG matching using a heuristic-based dynamic programming approach. The two-dimensional table Profit takes $O(n^2)$ time to build, where $n$ is the number of instructions within an expression DAG. This procedure is repeated for each instruction within the divergent branch. Hence, for each region $R$, the algorithm takes $O(n^2)$ time in the worst case because the Profit table caches matching data, thereby ensuring that each pair of instructions within the divergent branch is only compared once. If the number of regions is $m$, the overall optimization takes $O(mn^2)$ time.

### 3.4 Evaluation

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Nvidia V100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>NVPTX code generator</td>
</tr>
<tr>
<td>Programming Language</td>
<td>CUDA</td>
</tr>
<tr>
<td>Performance Metrics</td>
<td>SIMT Efficiency, Speedup</td>
</tr>
</tbody>
</table>

Table 3.1 summarizes the experimental setup for our experiments. We use microbenchmarks to show the impact of the hoist, sink and split transformations on 2-way and 32-way
divergent (switch) use-cases. Figure 3.6 shows the improvement in SIMT efficiency with common subexpression convergence and the corresponding speedup on V100.

3.5 Discussion

Our optimization suffers from some legal restrictions as well as potential performance downsides if applied indiscriminately. Some of these are listed below:

- **Warp-level Instructions**: Warp-level instructions such as VOTE cannot legally be moved to convergent paths of the program.

- **Register Pressure**: Hoisting a definition away from its use may help decrease pipeline stalls but increases register live ranges and may result in spills.

- **Pipeline Stalls**: Sinking a definition close to its use can increase pipeline stalls, especially for high latency instructions such as LOAD.

- **Predication**: Predication of instructions increases predicate register pressure and may require insertion of additional instructions to handle nested divergence.

- **Smaller Blocks**: Split temporarily reconverges a diverged path which adds the cost of an additional branch and barrier instruction. It may also adversely impact basic-block level instruction scheduling and peephole transformations because of smaller basic blocks.

- **Cost Analysis**: Our dynamic programming solution doesn’t consider memory coalescing or instruction latency in the cost/benefit analysis.

- **Interprocedural Common Code Convergence**: In the example below, a function that is called from another function along multiple diverged paths results in serialized execution of the function body. Today, we rely on function inlining to detect and optimize such cases.
Listing 3.12: Interprocedural Example

```c
A() {
    B();
}
main() {
    if (cond) {
        A(); // B() is executed along both paths
    } else {
        B();
    }
}
```

- **Impact of loop distribution and index set splitting:** A loop that has a common body but different loop iteration counts may be modified into a candidate for transformation using index set splitting so that the two loops align and can be moved to a convergent region. Additionally, if two loops have the same loop body and different loop bounds that are unknown at compile time, they may be sunk to a convergent region after setting the upper and lower loop bounds conditionally. On the other hand, two loops with the same loop bounds but partially matching loop bodies may be transformed using loop distribution and the common code can be moved out.

### 3.6 Future Work

Our cost model currently takes into consideration the cost of additional MOV instructions. Future work should explore enhancing the cost model to also consider the cost of register pressure increase. We believe that further opportunities for CSC may be found when targeting multicore CPU code (written using OpenCL or OpenMP, for example) to run on GPUs. Unlike hand-written CUDA code that is optimized for execution on the GPU, automatically generated GPU code from multicore CPU code may be suboptimal since divergence does
not pose a performance penalty on multicore CPUs. Whereas common code across conditional branches is not redundant in CPU execution, the same code may perform poorly on SIMT architectures without the application of CSC. Additionally, the optimization can further be extended to interprocedural common subexpression convergence using analysis across function calls.
(a) Improvement in SIMT Efficiency

(b) Speedup

Figure 3.6: Common Subexpression Convergence: Microbenchmark Results
CHAPTER 4
SPECULATIVE RECONVERGENCE

4.1 Motivation

We now look at a different subset of GPU divergence problems where diverged threads within a warp might execute a given piece of code eventually, but not all threads may arrive at the same point in time. This pattern is commonly found in Monte Carlo simulations, which have a wide variety of real world applications including dose calculation for cancer treatment [24], financial option pricing [25], path tracing [26] and experimental particle physics [27]. Unlike Chapter 3 where we moved common code across diverged paths to a common convergent path, we now collect diverged threads at a synchronization barrier before executing expensive common code, with the objective of increasing SIMT efficiency along expensive code paths.

We focus on a set of specific divergence sub-problems, which can broadly be divided
RSBench_lookup_kernel() {
    while (true) {
        Prolog:
        material = get_random_material()
        Predict(L1)
        // num nuclides per material ranges from 4 to 321
        for (each nuclide in material) {
            // proposed reconvergence point
            L1:
            accumulate_neutron_cross_sections()
        }
        // original reconvergence point
        Epilog:
        post_processing()
    }
}

(a) RSBench: pseudocode [29].

Figure 4.2: Pseudocode and execution pattern for RSBench.

Figure 4.1(a) illustrates case (1). The loop body contains a divergent branch that potentially evaluates to true in a different iteration for each thread. As GPU compilers currently attempt reconvergence at the post-dominator (PDOM) of the divergent region, some threads execute `Expensive()`, while the remaining threads wait idle at the `then` part’s predecessor or post-dominator. The common code is therefore executed serially across threads as the cartoon execution diagram shows.

Instead of reconverging at the post-dominator of the branch where all threads are guaranteed to arrive before executing the next iteration, threads can wait at the `then` block for other threads to arrive before executing common code as in Figure 4.1(b). This technique, which we call `Iteration Delay`, enables convergent execution of the expensive common code by deferring the progress of some threads within the warp, and improves both the SIMT efficiency and the total runtime for the program in question[28].

As an example of case (2), we describe RSBench, a mini-app that represents the packed-data multipole macroscopic cross section lookup kernel of the Monte Carlo neutron transport algorithm [29, 30]. Figure 4.2 depicts the high-level pseudocode for RSBench. Traditional divergence handling tries to reconverge at the earliest possible point where all threads

(b) Sample execution pattern for RS-Bench.
are guaranteed to arrive, i.e. at the inner loop post-tail, which means that the warp executes some iterations of the inner loop divergently.

Instead, we execute the inner loop convergently by collecting threads across iterations of the outer loop, especially if the inner loop body is expensive, a technique we call Loop Merge. However, as shown in Figure 4.2(b), while this alternative execution sequence ensures parallel execution of expensive common code, it changes the convergence properties of the code outside the divergent region. The prolog and epilog regions of the loop, which were previously executed in a convergent manner, are now executed divergently. Hence, the profitability of our solution depends on program properties, such as the cost of the common code and the cost of executing the prolog and epilog divergently.

As our examples show, while postdominator-based synchronization minimizes the static region within which threads diverge in the program, it fails at times to achieve optimal warp execution efficiency because it conservatively synchronizes a warp only at locations where all threads are guaranteed to arrive. Our approach attempts to speculatively reconverge all threads that may execute an expensive code path.

4.2 Problem

There are two broad categories of control flow divergence: divergent threads may execute disjoint sequences of code with no commonality, or divergent threads may eventually execute some common code sequences, just not aligned in time. For this work, we focus on the second category: periods of divergent execution where threads within the same warp might eventually execute common sequences of code, but are not able to exploit this commonality today. Broadly, two subclasses of patterns exhibit this behavior:

- Code that is common across diverged paths, including common function calls, common instruction subsequences, or irregular control flow.

- Divergently executed code that is common across loop iterations, such as a divergent
<table>
<thead>
<tr>
<th>Case</th>
<th>Pseudocode</th>
</tr>
</thead>
</table>
| (a) Divergent condition within loop | ```
for (i = 0; i < N; i++)
{
    Prolog()
    if (cond())
    {
        Expensive()
    }
    RECONVERGE
    Epilog()
}
``` |
| (b) Iteration delay | ```
for (i = 0; i < N; i++)
{
    Prolog()
    if (cond())
    {
        RECONVERGE
        Expensive()
    }
    Epilog()
}
``` |
| (a) Loop trip count divergence | ```
for (i = 0; i < N; i++)
{
    Prolog()
    for (j = 0; cond(); j++)
    {
        Expensive()
    }
    RECONVERGE
    Epilog()
}
``` |
| (b) Loop Merge | ```
for (i = 0; i < N; i++)
{
    Prolog()
    for (j = 0; cond(); j++)
    {
        RECONVERGE
        Expensive()
    }
    Epilog()
}
``` |
| (a) Common function call | ```
foo()
{
    Expensive()
}
main()
{
    if (cond())
    {
        // common code
        a = b * c
    } else
    {
        // common code
        a = b * c
    }
    RECONVERGE
}
``` |
| (b) Reconverge in Function | ```
foo()
{
    RECONVERGE
    a = b * c
}
main()
{
    if (cond())
    {
        foo();
    } else
    {
        foo();
    }
}
``` |
| (a) Common Code | ```
main()
{
    if (cond())
    {
        // common code
        a = b * c
    } else
    {
        // common code
        a = b * c
    }
    RECONVERGE
}
``` |
| (b) Function Outlining | ```
foo()
{
    RECONVERGE
    a = b * c
}
main()
{
    if (cond())
    {
        foo();
    } else
    {
        foo();
    }
}
``` |
branch or an imbalanced loop nested inside a loop.

Figure 4.3 illustrates several example code patterns which exhibit regions of divergent control flow that execute common code. Typical compiler-inserted reconvergence points located at post-dominators do not necessarily maximize run-time convergence for these cases. Instead of waiting for guaranteed reconvergence at post-dominators, threads could instead reconverge opportunistically at earlier points of execution. In each example, an alternative synchronization pattern can improve runtime convergence and performance. Our goal is to find a way to enable these common code sections to execute convergently to improve execution efficiency.

Divergent Conditions in Loops. Figure 4.3-1(a) shows a loop containing a divergent branch. In this example, multiple threads eventually execute the expensive code in the then part but do so at different iterations of the loop. The standard reconvergence logic collects all threads at the post-dominator of the divergent region before executing the next loop iteration, and the execution of the expensive conditional code remains serialized across loop iterations, resulting in poor GPU utilization.

Figure 4.3-1(b) shows an alternative execution that employs a different synchronization pattern that collects multiple threads across different iterations of the loop before executing conditional code inside the divergent path. Threads wait at the start of the expensive region for all threads to arrive before executing convergently, resulting in improved SIMT efficiency within the conditional and minimal thread idle time. In this example, the condition has no else part to execute. In cases where multiple paths exist, we may prefer to reconverge for expensive paths while allowing diverged execution of less expensive paths. We call this alternative synchronization approach Iteration Delay.

Loop Trip Count Divergence. Figure 4.3-2(a) shows a variation of unnecessary serialization where a loop contains a nested loop with a divergent trip count, i.e. threads exit the inner loop after different numbers of iterations. Threads that exit the inner loop early wait for all threads to arrive before executing the epilog and beginning the next iteration of
the outer loop, where threads once again begin to execute the inner loop in a convergent manner. The inner loop body is therefore common across iterations of the outer loop. Figure 4.3-2(b) shows an alternative synchronization point within the inner loop which collects threads across iterations before executing the expensive inner loop convergently. We call this approach *Loop Merge*.

Programs that have a non-nested divergent loop may be modified using thread coarsening, i.e. combining work from multiple threads into a single thread by converting a loop into nested loops which can then be optimized as described above. This situation is common in CUDA applications where, unlike in software-driven work distribution, users specify many thousands of independent tasks, and the GPU scheduler distributes work across SMs.

**Common Function Call.** Figure 4.3-3(a) shows an example of common code across diverged paths of a non-uniform branch. Both the taken and not taken paths of the branch call `foo()`, i.e., the function call body is common across all paths of the divergent branch. All threads in the warp will eventually execute `foo()`, but not together. The compiler today fails to reconverge at the function body because the calls to `foo()` are made from different locations in the program, and existing post-dominator based analysis fails to recognize this function body as a potential reconvergence point. To improve SIMT efficiency within the function body, we ensure that all threads reconverge at the start of the function and execute the body in parallel as in Figure 4.3-3(b). Unlike previous examples, reconvergence within the function body does not conflict with the compiler inserted reconvergence point at the post-dominator, nor does it affect convergence properties of the code outside the function body.

**Common Code.** Figure 4.3-4(a) shows another example of common code across diverged paths of a non-uniform branch. Both the taken and not taken paths of the branch execute \( a = b \times c \). Common Subexpression Convergence (described in Chapter 2) finds this common code and moves it to a convergent region using hoist/sink/split operations. However, this code motion can result in increased register pressure. We propose a func-
tion outlining solution in Figure 4.3-4(b) which then allows for speculative reconvergence within the new function so that diverged threads execute common code in a convergent manner.

4.3 Solution

Our solution uses Speculative Reconvergence to improve SIMT efficiency by collecting threads across loop iterations, or across divergent code paths, before executing common code paths.

Our approach operates in two steps: (1) find the region and program point where threads should reconverge (Section 4.3.1); and (2) insert synchronization instructions that achieve convergence before this point and allow exiting threads to withdraw from the barrier (Section 4.3.2).

4.3.1 Reconvergence point

To specify reconvergence hints, we require two pieces of information that the compiler uses to automatically place convergence synchronization:

1. The predicted location for reconvergence.

2. The region of code where this prediction should apply.

The first element allows a potential reconvergence point to be specified. There are two ways to determine the reconvergence point: user-directed and compiler-detected. For user-directed reconvergence, this location is specified by the user via a label in the source code. The second element describes which threads are considered to be candidates for reconvergence at the predicted location. Threads that enter the region will attempt to honor the predicted reconvergence point, and threads that leave the region are no longer considered candidates for reconvergence.
Listing 4.1: User Inserted Reconvergence Points.

```c
Predict(L1)
for (i = 0; i < N; i++)
{
    Prolog()
    if (divergent_condition())
    {
        // User-specified reconvergence point
        L1:
        Expensive()
    }
    // Original reconvergence point
    Epilog()
}
```

For this work, we focus primarily on user-directed convergence, but describe automatic compiler heuristics in Section 4.3.5. Listing 4.1 shows programming directives introduced to enable the user-directed approach. `Predict(L1)` marks the start of the prediction region. The region ends where all threads are no longer able to reach the label. We use this information to disambiguate between predictions at different loop nesting levels. Label `L1` marks the user-provided reconvergence point where all threads wait before executing the code that follows.

These two pieces of information help convey the following objectives to the compiler:

- All threads within the region should reconverge, when possible, at the predicted reconvergence point.
- Threads that leave the region should withdraw from the barrier at the predicted location, so other threads do not wait forever.
- The user-specified convergence hints should receive priority over any standard GPU convergence synchronization that might conflict, such as compiler-inserted reconver-
Table 4.1: Speculative Reconvergence: Synchronization Primitives.

<table>
<thead>
<tr>
<th>Synchronization Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JoinBarrier</td>
<td>Threads that enter the barrier expect to wait on the barrier at a subsequent convergence point. Implemented using BSSY.</td>
</tr>
<tr>
<td>WaitBarrier</td>
<td>Threads wait on all participating threads to arrive before clearing the barrier and continuing execution. Implemented using BSYNC.</td>
</tr>
<tr>
<td>CancelBarrier</td>
<td>Threads that exit the region without waiting on the barrier must clear the barrier so other threads may continue execution. Implemented using BREAK.</td>
</tr>
<tr>
<td>RejoinBarrier</td>
<td>We introduce a new primitive to denote the location where threads that have cleared a barrier but expect to wait on the barrier again must rejoin the barrier. This is seen primarily in the case of loops. Implemented using BSSY.</td>
</tr>
</tbody>
</table>

4.3.2 Synchronization Algorithm

Figure 4.4 shows the Control Flow Graph (CFG) for Listing 4.1. The user-defined region start and reconvergence points are at blocks BB0 and BB3, respectively. Compiler primitives are defined in Table 4.1. We now describe an algorithm that inserts these synchronization primitives.

The insertion of JoinBarrier and WaitBarrier primitives is trivially achieved using user-defined or compiler-detected region start and reconvergence points described in Section 4.3.1. The compiler preserves this information through any control-flow altering optimizations prior to the synchronization pass. Figure 4.4(a) shows the insertion of JoinBarrier and WaitBarrier at BB0 and BB3 respectively.

The compiler must ensure that a thread that waits on a barrier that it has already cleared rejoins the barrier and a thread that exits the region without waiting on a barrier that it has joined withdraws from the barrier. In the next section, we detail program analyses that we
Dataflow Analysis

We need two pieces of information to decide the placement of CancelBarrier and RejoinBarrier primitives: (1) whether a thread is part of an uncleared barrier, and (2) whether a thread expects to wait on the barrier. Figures Figure 4.4(b) and (c) depict the data flow analyses the compiler uses to decide ideal placement of synchronization primitives, and Figure 4.4(d) shows the final synchronization. We insert a RejoinBarrier in BB3 where the barrier $b0$ was cleared by a WaitBarrier, and a CancelBarrier in BB5 where threads that joined barrier $b0$ may escape without clearing the barrier. Finally, to ensure that all threads reconverge at the region exit, we introduce an orthogonal barrier register and insert a pair of JoinBarrier and WaitBarrier primitives at the dominator (BB0) and post-dominator (BB5) of the region respectively.

**Joined Barrier Analysis.** A barrier register has been joined at a program point P if at least one path from program start to P contains a JoinBarrier primitive not followed by a WaitBarrier primitive. Equation (4.1) depicts the dataflow equations for joined barrier
analysis. In Figure 4.4(b), the barrier at BB3 is joined at BB0 and cleared at BB3.

\[ \text{Gen}(BB) = \text{JoinBarrier} \]
\[ \text{Kill}(BB) = \text{WaitBarrier} \]

\[ IN(BB) = \cup \{\text{OUT}(p) \mid \forall p \in \text{predecessors}(BB)\} \]
\[ OUT(BB) = (IN(BB) - \text{Kill}(BB)) \cup \text{Gen}(BB) \]

**Barrier Live Range Analysis.** Next, we perform standard backward liveness analysis on barrier registers to compute the set of blocks where the barrier is live. Barrier *liveness* is analogous to register liveness, i.e. a barrier is *live* at a program point P if there is a *WaitBarrier* along some path from P to the end of the program. Otherwise it is considered *dead*, i.e. it has no further uses. We generate the liveness of the barrier register at a use of the barrier (*WaitBarrier*), and kill the liveness of the barrier register at the *JoinBarrier*. Equation (4.2) depicts the dataflow equations for live barrier analysis. In Figure 4.4(c), the barrier \( b0 \) is dead at BB5 and BB0.

\[ \text{Gen}(BB) = \text{WaitBarrier} \]
\[ \text{Kill}(BB) = \text{JoinBarrier} \]

\[ IN(BB) = (OUT(BB) - \text{Kill}(BB)) \cup \text{Gen}(BB) \]
\[ OUT(BB) = \cup \{\text{IN}(s) \mid \forall s \in \text{successors}(BB)\} \]

Note that *CancelBarrier* and *RejoinBarrier* primitives can affect the dataflow analyses described above, but these primitives are not yet present in the code and are therefore ignored in our equations.

### 4.3.3 Deconfliction

Two barriers are said to be conflicting if their live ranges overlap in a non-inclusive manner, i.e. neither one is a complete subset of the other. If a region has conflicting barriers, threads may wait for each other at two different places within the region resulting in unpredictable behavior. The compiler inserts synchronization primitives that may conflict with the compiler-inserted synchronization at the postdominator of the divergent region. To break this conflict and avoid unpredictable behavior, we implement a *deconfliction pass*. 

41
A barrier live range extends from the moment threads join the barrier until the barrier is cleared either by waiting or exiting threads. In Figure 4.5(a), the blue arrows represent the two live intervals for barrier b0 which was inserted by our optimization, while the red arrow represents the live interval for barrier b1, inserted by the compiler at the region postdominator. Barrier b0 and b1 conflict because their respective live intervals overlap in a non-inclusive manner.

We implemented two deconfliction strategies: static deconfliction and dynamic deconfliction. In static deconfliction, if two barriers are found to conflict, the compiler eliminates barrier operations corresponding to the PDOM barrier. If the newly defined convergence point is never, or rarely, entered, static deconfliction may result in poor performance. Figure 4.5(b) shows static deconfliction as applied to our example. The synchronization primitives for b1 are removed thereby eliminating the conflict. In dynamic deconfliction on the other hand, no instructions are deleted. Instead, threads waiting on one barrier exit out of the conflicting barrier, thereby removing conflicts only in cases where the convergence point is in fact executed at run time. In our example, Figure 4.5(c), threads that enter block

Figure 4.5: Deconfliction: (a) Conflict analysis; (b) Static deconfliction by deleting conflicting barrier; (c) Dynamic deconfliction by exiting conflicting barrier.
BB3 exit barrier b1 before waiting on barrier b0, thus effectively eliminating the conflict at runtime.

Static deconfliction has an advantage over dynamic deconfliction in terms of number of instructions executed and barrier registers used. However, if a conditional branch is rarely executed, and the prolog/epilog sections are expensive, dynamic deconfliction performs better because it retains the original synchronization points.

4.3.4 Interprocedural Analysis

As an extension to our optimization, we propose an interprocedural variant that handles a function body eventually executed by all threads in a warp. In Figure 4.3(c), foo() is ultimately called from both paths of the divergent branch, but serially. To indicate the desired early reconvergence point at the entry of the function, the user interface now uses the function name instead of a label name to indicate the PC at which all threads must wait before executing in parallel.

Speculatively reconverging within the divergent function call rather than at the post-dominator block of the divergent condition does not adversely affect performance because there are no prolog/epilog sections and hence no increase in divergent execution. The only cost associated with the optimization is the insertion of additional barrier instructions.

Finally, the programmer or the compiler must move calls to extern functions into a wrapper function body which acts as the required reconvergence point. The wrapper function may also be used for functions that are called from within multiple independent regions of the program.

To perform this optimization across functions, we introduce an interprocedural analysis that propagates barrier information upwards through the call graph from the callee to the call site. After this first stage propagation is complete, the barrier operations are inserted by the compiler as before.
4.3.5 Automatic Detection of Reconvergence Point

In some cases, such as programs generated by parallelizing compilers, manual modification may not be possible. In this case, automatic detection of some common code patterns may be useful. Our approach looks for opportunities within common CFG patterns in code such as divergent branches or loops with non-warp-uniform iteration counts nested inside an outer loop.

We use three metrics to approximate the cost-benefit analysis of changing convergence patterns. First, we compute instruction count in prolog/epilog sections, weighted by loop trip count, nest depth, and instruction latency. Static analysis is limited by its inability to predict dynamic loop counts and caching behavior, rendering it too conservative. Profile information may help improve the accuracy of our profitability tests. Next, memory access patterns, such as convergent memory accesses may suffer if previously convergent accesses become divergent due to our optimization. Heuristics may choose to account for the cost of making memory accesses divergent. Finally, synchronization requirements within the program including barriers and warp synchronous instructions may affect the correctness of modifying the convergence properties of the region.

Automatic Speculative Reconvergence is a challenging problem for the compiler to solve unaided, particularly when there are conflicting locations for Speculative Reconvergence such as in the case of triply nested loops. Further, the profitability of modifying convergence points depends on the relative cost of the common code, its divergence properties, and the prolog/epilog regions of the code. Incorrect Speculative Reconvergence may result in large performance degradations, depending on the run time behavior of the program, which is why we rely on the user’s understanding of application behavior to decide the best reconvergence point and provide a compiler technique to correctly insert the necessary synchronization primitives.
4.3.6 Soft Barrier

As described so far, we wait for all threads to arrive at a predicted convergence point before proceeding, thereby maximizing reconvergence at the specified location. However, maximizing convergence may not always be the best choice. As we increase convergence at the specified location, we can also increase serialized executions of other portions of code (e.g., of the prolog and epilog code sections in Figure 4.2). In the case of Loop Merge, depending on the relative cost of these sections with the inner loop, this alternate synchronization pattern may not be profitable. Earlier work such as [31, 32] have made similar observations about the utility of a tuning parameter to balance idle lanes versus the cost of refilling with work. To help address this tradeoff, we provide an optional threshold parameter to ensure that at least some minimum set of threads arrive at the reconvergence point before proceeding, but also allow threads to proceed without having to wait for all possible participants. This approach guarantees some minimum degree of convergence at the specified location, while ensuring that newly serialized code regions have their executions amortized across more threads.

4.4 Evaluation

We modified the NVIDIA production GPU compiler to evaluate our proposed Speculative Reconvergence technique. We implemented two variants of this optimization: (1) a user-guided approach for marking reconvergence points (Section 4.3.1) and (2) compiler heuristics to detect divergence patterns (Section 4.3.5). Further, our implementation uses the dynamic deconfliction strategy. Our results were collected on NVIDIA’s Volta V100 GPU using the nvprof profiler [33].
Table 4.2: Speculative Reconvergence: Benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsbench</td>
<td>A nuclear reactor simulation mini-application that optimizes Monte Carlo neutron transport [30]. The main kernel in RSBench has a loop with a divergent trip count. We apply thread coarsening to increase work per thread.</td>
</tr>
<tr>
<td>xsbench</td>
<td>[34] simulates a problem similar to RSBench, but is memory bound rather than compute bound. In particular, we find that the nested divergent loop in the XSBench kernel has both an expensive inner loop and an expensive epilog.</td>
</tr>
<tr>
<td>mcb</td>
<td>A Monte Carlo benchmark used to test performance of parallel architectures. Simulates a simplified variant of the heuristic transport equation [35].</td>
</tr>
<tr>
<td>pathtracer</td>
<td>A simple CUDA-based microbenchmark that renders a sample scene composed of spheres in a Cornell box. Has loop trip count divergence.</td>
</tr>
<tr>
<td>mc-gpu</td>
<td>A GPU-accelerated Monte Carlo simulation used to model radiation transport of x-rays for CT scans of the human anatomy [36].</td>
</tr>
<tr>
<td>mummer</td>
<td>A parallel sequence alignment kernel used for genome sequencing [37].</td>
</tr>
<tr>
<td>MeiyaMD5</td>
<td>Performs Message-Digest algorithm 5 (MD5) hash reverses [38].</td>
</tr>
<tr>
<td>Optix</td>
<td>NVIDIA’s ray tracing engine optimized to achieve high performance for ray tracing based algorithms on parallel architectures. [31].</td>
</tr>
<tr>
<td>gpu-mcml</td>
<td>A benchmark that simulates photon transport [39].</td>
</tr>
</tbody>
</table>

4.4.1 Benchmarks

We evaluate our proposal on a set of divergent workloads that exhibit the patterns described in Section 4.2. Many of these are Monte Carlo applications that model processes with variable and unpredictable length, leading to poor SIMT efficiency and reduced performance due to divergent code in loops. Table 4.2 describes the set of applications we evaluate. We did not find any applications that exhibit the common function call pattern described in Figure 4.3(c); instead, we validated this pattern using microbenchmarks.

4.4.2 Results: Programmer-Annotated Applications

For our evaluation, we manually insert reconvergence points and mark the prediction region as described in Section 4.3.1. The primary metric we influence in this work is SIMT efficiency, a measure of convergent execution of threads in a warp. Figure 4.6 shows SIMT
efficiency before and after our transformations. Many of these applications exhibit relatively low SIMT efficiency in their default state, indicating potential for improvement. After modifying reconvergence points to user-specified alternate locations, we see significant increases in SIMT efficiency. Applications such as gpu-mcml, pathtracer, and rsbench have highly variable inner loop trip counts, resulting in significant divergence with default synchronization; all threads within a warp must wait for any straggler threads still executing the inner loop before they can proceed. After marking a reconvergence point inside the loop instead of at the loop post-dominator, idle threads are able to diverge and acquire new work before rejoining the inner loop. SIMT efficiency is improved most when threads have a relatively high degree of compute inside their loops compared with the cost of newly-serialized code before and after the loop to refill empty threads.

While the primary metric that our transform attempts to improve is SIMT efficiency, our end goal is improved performance. Figure 4.7 shows our relative improvement in SIMT efficiency as well as application speedup. As expected, we see that improvement in SIMT
efficiency leads to improvement in performance. However, there are many aspects that affect runtime performance beyond SIMT efficiency. While we improve overall SIMT efficiency, especially in the compute-intensive portions of code, the tradeoff is that we must now pay for any additional runtime overhead where we execute other regions of code (e.g., prolog or epilog) more divergently and more frequently. For example, each RSbench thread processes materials of different cost in nuclide count, leading to unpredictable loop iterations, between four and 321 iterations per thread. PathTracer simulates Monte Carlo light transport using Russian Roulette to randomly terminate paths, with each sample running one or more bounces up to some maximum limit. In these cases, applying Loop Merge enables better SIMT efficiency of the inner loop, and the performance gains due to higher SIMT efficiency outweigh the cost of added serialized execution to refill empty threads.

In most cases, it is reasonable to expect that SIMT efficiency improvement serves roughly as an upper bound on speedup. Applications with smaller improvements in SIMT
efficiency and lower speedups can have lower variance in iteration counts and/or higher serialization cost in the prolog and epilog sections of the code, limiting improvement.

4.4.3 Results: Soft Barrier

Our approach above aims to improve performance by maximizing SIMT efficiency. However, the best performance is not necessarily always achieved at maximal convergence. To balance the tradeoff between lower SIMT efficiency when threads are idle and the cost of refilling idle threads with new work, we implemented soft barrier (see Section 4.3.6), an extension to our Speculative Reconvergence optimization which allows for partial reconvergence before execution of common code. Our implementation of the soft barrier employs a user-defined threshold and waits for enough threads to arrive at the reconvergence point before proceeding rather than waiting for all threads. Equivalently, it allows the program to continue execution until the number of active threads drops below some threshold and refilling idle threads becomes worth the cost.
Figure 4.8 shows the results of using soft barriers with varying threshold values for two applications: PathTracer and XSBench. PathTracer executes fastest when all threads reconverge before executing; the cost of filling an idle thread with new work is low enough, relative to the main computation, that it is best to immediately refill any idle thread. On the other hand, XSBench reaches peak performance when waiting much longer to refill empty threads. An expensive process is required when a thread wants a new task, and executing this process every time one or a few threads become idle is not profitable. For the particular configuration shown, performance is best when executing the inner loop until as few as four threads are participating. We leave the problem of automatically discovering the ideal threshold parameter for a particular problem to future work.

4.4.4 Results: Automatic Speculative Reconvergence

While the primary focus of this work is enabling user-directed reconvergence, we also experimented with automatic selection of alternate reconvergence points via compiler heuristics (see Section 4.3.5). We examined a large database for opportunities to apply both Loop Merge and Iteration Delay synchronization patterns during backend compilation from PTX to machine code.

Figure 4.9 shows upside potential for a set of applications which were automatically
discovered to have opportunity for Speculative Reconvergence. Of the traces detected to have opportunity, several candidates for Loop Merge and Iteration Delay make use of the OptiX raytracing engine [31], an application space known for divergence. Another application, MeiyaMD5, contains a load-imbalanced, compute-heavy inner loop making it the ideal candidate for Loop Merge. For brevity, we restrict results here to cases with significant upside potential. However, many examples with compiler-detected opportunity see no change or even regression, limiting automatic application of this technique without better heuristics, profile guidance, or user input. Finally, automatic Speculative Reconvergence performs the same as programmer-annotated variants of the benchmarks from Table 4.2.

4.5 Discussion and Future Work

This section discusses limitations of our optimization, interactions with other compiler passes, as well as future work.

Interaction with loop optimizations. If a loop is completely unrolled, Iteration Delay and Loop Merge cannot be applied. On the other hand, if the inner loop of a loop nest is partially unrolled by a factor of N, Loop Merge may be still applied. Reconvergence is needed only once per N iterations of the inner loop body, which may reduce the overhead of synchronization for reconvergence. Loop interchange may make the inner loop divergent, thereby introducing opportunity for Loop Merge. Additionally, loop fusion inside an outer loop could help improve performance of Loop Merge and reduce divergence. Loop fission may help break the outer loop into two separate loops, one with the expensive inner loop and the other with the expensive prologue/epilogue section, which could help reduce the effect of increased divergence at the prologue/epilogue regions of the outer loop. These optimizations may also inhibit Loop Merge in some cases.

Interaction with function inlining and code refactoring. If a function call that is common across divergent paths is inlined, we can no longer reconverge threads at a common PC, which inhibits the applicability of our optimization. On the other hand, common
code across divergent paths may be refactored into a single method. This operation is the inverse of the function inlining optimization above and introduces opportunity for reconvergence.

**Interaction with warp synchronous operations.** Our technique changes the convergence properties of a program which may change behavior of warp synchronous operations. However, CUDA 9.0 introduced the requirement that instructions that require inter-thread communication cannot implicitly assume convergence and must use `warpsync` for correctness, which would inhibit automatic Speculative Reconvergence.

**Interaction with scalar datapaths.** Techniques such as Loop Merge change the scalar nature of loop indices across threads. For devices with scalar pipelines, the ability to exploit scalar data to reduce register count may be reduced if previously scalar data must be stored in thread-private registers. Additionally, loops with deterministic iteration count could previously be considered non-divergent. However, if we modify convergence synchronization, previously convergent regions of loops may become divergent, again limiting the use of scalar datapaths.

**Multiple concurrent predictions.** Speculative Reconvergence works at all levels of nesting: within a loop, within a function call, and within nested conditional statements. Our method can also support multiple concurrent predictions within a region. If these predictions are exclusive, they can be supported using deconfliction. If they are not exclusive, soft barriers may be utilized to allow for limited convergence at each point. A detailed study of concurrent overlapping predictions is left to future work.
CHAPTER 5
SUBWARP INTERLEAVING

5.1 Motivation

GPUs hide stalls by concurrently scheduling among many active warps [40]. If there are insufficient warps available due to high per-warp resource utilization, the warp scheduler is unable to schedule a different warp in case of a pipeline stalls, which results in higher overall application latency. In this chapter, we focus on long latency stalls in divergent code paths using Raytracing applications as our posterchild.

Raytracing kernels are latency-sensitive, divergent, and contain relatively few active warps. Previous chapters such as Common Subexpress focused on reducing divergent execution. However, in this chapter, we describe an approach that improves latency tolerance by leveraging the divergence inherent in raytracing to effectively increase the scheduler’s ability to hide stalls.

Figure 5.1 illustrates warp divergence in the control flow graph of a raytracing megakernel, where the divergent blocks labeled “Shader A” and “Shader B” contain pedagogical single-block shader programs. Notice that after the point labeled ①, a warp of threads splits into two subwarps, which we define as PC-aligned subsets of a warp’s threads. After this point, one subwarp, S0, runs “Shader A” and the other, S1, runs “Shader B”. Modern GPUs serialize the execution of these shader programs within a warp. A scheduling policy chooses one subwarp to execute first, and when that subwarp runs to a statically designated point of convergence, only then will the GPU execute the other subwarp. For example, if subwarp S0 runs first, then it must run to completion before S1 can begin execution. This approach is inefficient for workloads in which there are insufficient active warps to effectively hide the workload’s memory requests.
To characterize the opportunity for better latency tolerance in raytracing applications, we define *exposed* long-latency or load-to-use stalls as cycles when no active warp in an SM is able to issue, and at least one active warp is stalled on an outstanding memory load operation. The lower this metric, the better the latency tolerance. Figure 5.2 shows total exposed load-to-use stalls and exposed load-to-use stalls in divergent code blocks, both normalized to respective kernel runtimes, across our suite of raytracing kernels. These applications are often stalled waiting for memory, and a significant percentage of those stalls are in divergent code regions.

### 5.2 Problem

We propose *Subwarp Interleaving* to allow the GPU’s scheduler to interleave subwarps instead of serializing them, enabling long-latency operations from divergent code paths to overlap in time and reducing average exposed memory latency [41].

Figure 5.3 compares today’s subwarp serialization (labeled “Baseline”) to Subwarp Interleaving on the example in Figure 5.1, using only two subwarps for clarity. In the baseline case (Figure 5.3a), the load-to-use stalls due to memory accesses in diverged sub-
warps $S_0$ and $S_1$ cannot be overlapped because of the serialized execution of divergent threads. In Figure 5.3b, Subwarp Interleaving switches between the subwarps in a warp when the program would otherwise stall. With Subwarp Interleaving, instead of stalling at point 3, a subwarp scheduler recognizes that $S_1$ is ready and makes $S_1$ the active subwarp. Subwarp $S_1$ executes instructions until it stalls at 6 waiting for the result of the texture operation, at which point the subwarp scheduler can swap $S_0$ back in and successfully hide the long-latency load. Subwarp Interleaving increases the perceived occupancy of the GPU which allows the architecture to better tolerate long-latency operations.

Figure 5.3 compares today’s subwarp serialization (labeled “Baseline”) to Subwarp Interleaving on the example in Figure 5.1, using only two subwarps for clarity. In the baseline case (Figure 5.3a), the load-to-use stalls due to memory accesses in diverged subwarps $S_0$ and $S_1$ cannot be overlapped because of the serialized execution of divergent threads. In Figure 5.3b, Subwarp Interleaving switches between the subwarps in a warp when the program would otherwise stall. With Subwarp Interleaving, instead of stalling at point 3, a subwarp scheduler recognizes that $S_1$ is ready and makes $S_1$ the active subwarp. Subwarp $S_1$ executes instructions until it stalls at 6 waiting for the result of the
Figure 5.3: Subwarp Interleaving concept. In GPUs today, the scheduler serializes the execution of subwarps $S_0$ and $S_1$, and is unable to hide load-to-use stalls in each shader. The Subwarp Scheduler interleaves the execution of $S_0$ and $S_1$ to hide load-to-use stalls, thereby reducing overall execution time of the program.

texture operation, at which point the subwarp scheduler can swap $S_0$ back in and successfully hide the long-latency load. Subwarp Interleaving increases the perceived occupancy of the GPU which allows the architecture to better tolerate long-latency operations.

The goal of subwarp interleaving is to (a) identify true or potential long stalls in divergent code paths, (b) identify an appropriate subwarp to switch to, (c) save current subwarp state and (d) switch to a new subwarp.

5.3 Solution

We propose both a software-driven and hardware-based subwarp interleaving solution.
5.3.1 Software Subwarp Interleaving

We first describe a software-directed implementation of Subwarp Interleaving on a GeForce RTX™2080 Ti. NVIDIA departed from its previous approach to divergence handling when it introduced the Volta Instruction Set Architecture (ISA) [42]. Instead of implementing a divergence stack to handle divergent execution [40], Volta’s ISA introduced convergence barriers and an associated set of instructions that the compiler uses to more flexibly schedule threads [28]. We developed a prototype compiler that uses Turing’s independent thread scheduling support to automatically transform shader programs to perform Subwarp Interleaving. While this approach was ultimately a failure, we discuss it because it uncovered several challenges and motivated hardware changes.

Independent thread scheduling is a SIMT paradigm that enables fine-grain synchronization and execution between parallel threads in a program [42]. Independent thread scheduling provides a mechanism, which we call YIELD, that provides forward progress guarantees by explicitly switching execution between a warp’s subwarps. This mechanism prevents deadlock in starvation-free algorithms because it ensures that all threads will eventually execute, independent of thread ordering, and therefore all threads have an opportunity to access a contended resource [42].

Conceptually, YIELDs performs an explicit subwarp switch, which we can leverage to implement Subwarp Interleaving. We modified a production-quality compiler to place explicit YIELDs calls before possible long-latency instructions. In Figure 5.1, for example, our compiler places YIELDs calls immediately before the uses of the long-latency memory operations at points ③ and ⑥. Before the hardware stalls waiting for ③ in program S2, for instance, YIELDs switches execution to the subwarp executing program s3. We experimented with various policies for inserting YIELD, all of which appealed to the intuition that switching execution to another subwarp before the warp stalls would allow more long-latency instructions to overlap in time.

The data in Figure 5.2, which shows that a significant fraction of cycles is stalled
waiting for memory in divergent code, motivated us to invest significant effort experiment-
ing with Subwarp Interleaving strategies that work on contemporary hardware. While we
did manage in some cases to reduce load-to-use stalling, our approach increased stalling
for other reasons and ultimately decreased performance. Our efforts did provide valuable
insights regarding the disconnect between our intuition and our poor results, in particular:

- YIELDs can change an application’s convergence properties, which is useful for
  maintaining algorithm correctness, but can adversely affect performance if misused.
  In many cases our approach allowed subwarps to remain diverged well past static
  reconvergence points.

- A compiler cannot statically know where to place YIELDs because predicting which
  loads might stall (i.e., miss in caches) is intractable.

- YIELDs does not prevent switching to a subwarp that is currently stalled. In fact,
given that we cannot statically know which load-to-use dependencies are going to
stall, we could even switch from a subwarp that will not stall to a subwarp that is
stalled.

- The warp’s dependence tracking mechanism is shared among a warp’s subwarps and
therefore Subwarp Interleaving can introduce false dependencies. While scoreboard
aliasing does not introduce functional issues, it increases the likelihood that YIELDs
switches to a subwarp that is stalled.

5.3.2 Hardware Subwarp Interleaving

At a high-level Hardware Subwarp Interleaving (SI) treats subwarps of warps, defined as
a maximal group of threads at a given PC, as the primitive scheduling unit. As a result,
SI exploits warp divergence to allow warps that would otherwise stall to gainfully occupy
a scheduling slot. An SI architecture demotes stalled subwarps of a warp so that other
subwarps of the same warp get a chance to execute.
Figure 5.4: Subwarp Interleaving SM architecture. Our Subwarp Scheduler replaces the divergence handling mechanism and uses subwarp wakeup and selection logic described in Section 5.3.2 to select a new subwarp.

As Figure 5.4 shows, we replace the baseline architecture’s divergence handling unit with a subwarp scheduler unit, which contains a thread status table and subwarp wakeup and selection logic that chooses which of a warp’s subwarps should execute when the warp scheduler selects the warp for execution on a core. Importantly, SI does not change the baseline architecture’s warp scheduler. The rest of this section describes the components of an SI architecture and how they interact with a baseline Turing-like architecture.

Divergence Handling for SIMT

Subwarp Interleaving leverages divergence and must therefore closely interact with the underlying SIMT execution model, examples of which include SIMT stacks [40] and modern convergence barrier models [42, 43]. While SI can be applied to different SIMT execution models, we examine it in the context of NVIDIA’s contemporary convergence barrier
architectures.

When a warp’s SIMT execution diverges, for example because of a conditional branch, the warp will splinter into two or more subwarps and the underlying architecture must keep track of each subwarp’s status. The architecture must provision resources to track each thread to handle the degenerate, fully divergent case where each thread operates independent of the others.

Figure 5.5 describes a possible state machine that tracks the status of a single thread. Baseline state transitions are in black-and-white and SI-specific additions to the state machine are highlighted in color. Every thread starts in the INACTIVE state. On program entry, threads transition to the ACTIVE state. Thereafter, when a BSSY $Bx$ operation [44] executes, all active threads register themselves by setting a per-thread bit in the associated barrier register, $Bx$.

On a divergent branch a divergence handling unit serializes the execution of a warp’s subwarps. The unit chooses one subwarp for active execution by leaving that subwarp’s threads in the ACTIVE state and transitioning the threads of the other subwarps to the READY state.

Figure 5.5: Thread status state machine with Subwarp Interleaving. A divergence handling unit in a Turing-like architecture maintains per-thread state to decide which threads of the warp execute at a given time. Subwarp Interleaving adds the STALLED state to demote subwarps that have suffered long-latency stalls. When a stalled subwarp’s outstanding dependences complete, the subwarp becomes eligible for scheduling again.
On encountering a BSYNC Bx operation, a thread can remain in the ACTIVE state (labeled “successful BSYNC” in the state transition diagram) if all threads participating in barrier register Bx are either in the BLOCKED or INACTIVE states. Otherwise, the thread moves to the BLOCKED state.

When an active subwarp’s threads transition to the BLOCKED state, the divergence handling unit examines the statuses of all threads and selects, via an action we call subwarp-select, a subwarp to move from the READY state to the ACTIVE state. In a Turing-like SIMT model, the divergence handling unit triggers the subwarp-select action based on thread states and per-thread program counters (PCs) stored in dedicated registers.

Subwarp Interleaving SIMT Operation

Subwarp Interleaving demotes subwarps that have suffered load-to-use stalls. SI builds on the baseline architecture’s thread status state machine by adding a new state, STALLED, and three new types of transitions, namely, subwarp-stall (shown in red), subwarp-wakeup (in green), and subwarp-yield (in dashed blue). While the subwarp-stall and subwarp-wakeup transitions are functionally required for long-latency tolerance, subwarp-yield is optional but may provide extra Subwarp Interleaving flexibility.

The blow-up of the Subwarp Scheduler unit in Figure 5.4 shows the logical blocks responsible for subwarp-stall (Subwarp Stall), subwarp-wakeup (Subwarp Wakeup), and subwarp-select (Subwarp Selection), which we discuss in turn.

subwarp-stall: Subwarps that suffer load-to-use stalls on long latency operations cannot make forward progress until their memory lookups return. Rather than occupy a warp’s scheduling slots, simple combinational logic in the Subwarp Stall unit transitions threads of stalled subwarps to the STALLED state. If all threads of a warp are STALLED, the warp scheduler will not select the warp for execution. However, by transitioning a subwarp that suffers a stall to STALL, SI makes other READY subwarps of the warp eligible for scheduling.
subwarp-wakeup: A dedicated unit, called Subwarp Wakeup, continually polls specific scoreboards for successful completion of outstanding long-latency operations from threads of stalled subwarps, and transitions threads upon completion to the READY state. The scoreboards the Subwarp Wakeup module polls can be textbook, per-register completion trackers or count-based scoreboards, which are low-complexity dependency trackers that leverage in-order completion of memory operations to infer bulk completion of one or more operations based on outstanding counts. We assume the latter in this paper.

subwarp-select: After a subwarp-stall or subwarp-wakeup transition, the hardware must find a new subwarp to move to the ACTIVE state. The Subwarp Selection module consults thread statuses and per-thread PCs to transition subwarps in the READY state to ACTIVE.

Optionally an SI architecture can support eager transitions from the ACTIVE state. The subwarp-yield transition allows a subwarp to relinquish its scheduling slot to another subwarp of the same warp. The transition can be achieved either through an explicit software instruction, encoded as a scheduling hint in the instruction stream, or via fixed hardware policies. An example of a fixed hardware policy is to yield after issuing a configurable threshold of long-latency operations (such as texture lookups or global memory loads) or operations to core-level shared functional units, such as those executing transcendental operations. Subwarps that subwarp-yield transition to the READY state allowing the Subwarp Selection module to select them if another READY subwarp cannot be found.

Subwarp Interleaving Microarchitecture

We now fill in the details of the blocks presented in Figure 5.4. The Subwarp Scheduler unit tracks the status of a warp’s threads, and wakes up threads as and when their data dependencies are satisfied. Figure 5.6 shows the hardware structures SI uses for both tasks.
Thread Status Table  At the heart of the Subwarp Scheduler unit is a per-warp 32-entry Thread Status Table (TST), which integrates the scoreboards and information needed to perform both the subwarp-wakeup and subwarp-select operations. Minimally, the TST must have dedicated storage for each thread’s current state (3b) and scoreboard information, namely, scoreboard ID (s bits) and current count (t bits). The quantity s depends on the number of counted scoreboard trackers a design uses, and t depends on the maximum number of outstanding operations a single scoreboard counter can track.

![Thread Status Table](image)

Figure 5.6: Thread Status Table Operation. The TST holds thread active and ready states as well as scoreboard counters. On writeback, the wakeup logic updates the corresponding scoreboard for each thread and marks a STALLED thread as READY if all its scoreboard counters are 0.

Subwarp Wakeup  A count-based scoreboard, typically used with a variable latency operation such as a memory lookup, is incremented when an associated operation is issued and decremented when the associated operation writes back [45]. A dependent consumer blocks until an appropriate dynamically or statically tagged scoreboard counts down to a desired value. If there are ordering guarantees for the producer operation, a non-zero scoreboard count may be specified. If not, the dependent operation will simply wait until the scoreboard counts down to zero. This guarantees that all register writes guarded by this
scoreboard have completed and the corresponding registers are safe for consumption.

A subwarp-stall transition occurs in response to an instruction checking if a particular scoreboard, $SB$, has counted down to a desired value, and failing that check. As part of the subwarp-stall transition, the TST records $SB$ and its current counter value, in the scoreboard ID and scoreboard count fields, respectively, for all applicable threads. Thereafter, every time a subwarp writes back a scoreboard-protected operand to the vector register file, in addition to updating the main set of per-thread scoreboards located in the warp scheduler, those scoreboard IDs get broadcast to the TST as well. Each TST entry has associated wakeup logic that compares the broadcast scoreboard IDs with the recorded ID in the entry. If any of the broadcast IDs match with the recorded entry, the recorded count is decremented by one. Otherwise, the earlier value for the count field is retained. When a TST entry’s scoreboard count field reaches zero and its current state is the STALLED state, the entry transitions to the READY state.

Figure 5.6b shows the wakeup logic for a single entry of the TST. To handle the two writeback paths for long-latency operations in our design, one from the texture units and another from the load/store units, the TST will need to support a total of 64 $s$-wide comparators for wakeup (i.e. 32 entries x 2 comparators per entry), where $s$ is the bitwidth of the scoreboard ID. Typically, $s$ is less than four. Each TST entry needs a subtractor to decrement the scoreboard count on a successful scoreboard ID match, for a total of 32 subtractors per TST. Additionally, the logic requires 64 2:1 muxes (32 entries $\times$ 2 2:1 muxes per entry) to correctly update TST fields. Supporting per-thread wakeup logic is an extreme point in the space that allows the Subwarp Scheduler unit to support up to 32 independent subwarps, but comes with a sizable cost in terms of extra logic and state that warps must maintain. In the evaluation section we show that we can retain much of the benefits of per-thread tracking by binning threads of a subwarp into a limited number of TST entries. For example, a TST with two entries could efficiently support two subwarps per warp.
**Subwarp Selection** Recall that when there is no active subwarp, the baseline relies on hardware support to identify an eligible subwarp in the READY state and transitions that subwarp of threads to ACTIVE. In our SI-enabled architecture, we implement this functionality in dedicated logic called Subwarp Selection that operates on information available in the TST.

There are many potential policies to guide Subwarp Selection. During our research we considered many policies and converged to a single heuristic with two configuration knobs that operates on stalled warps that contain at least one READY subwarp. One knob configures *when* to trigger subwarp-select to make a READY subwarp ACTIVE. In this paper we consider three points in this knob’s space: triggering subwarp-select when 1) at least one warp (per processing block) is stalled, 2) when at least half of the warps are stalled, and 3) when all warps are stalled. The other policy knob configures *where* to switch subwarps, and in this paper we consider two points in this knob’s space: 1) triggering subwarp-select on all stalled warps with subwarps in the READY state, and 2) triggering subwarp-select on only the lowest-numbered stalled warp with a subwarp in the READY state. For warps with multiple READY subwarps, Subwarp Selection selects the next READY subwarp in a round-robin manner to transition to ACTIVE. If no ready subwarp is available, the current subwarp transitions back to ACTIVE.

**Operation** The above Subwarp Interleaving support works as follows to better tolerate long latency stalls. The currently active subwarp transitions to STALLED upon suffering a long-latency stall or moves to BLOCKED upon hitting a BSYNC operation.

The Subwarp Selection heuristic described above selects and transitions a ready subwarp, if available, to the ACTIVE state. The newly-active subwarp might issue long latency operations from its code region, which could cause the subwarp to transition to the STALLED state. Even if the code region does not stall, the subwarp will eventually transition to the BLOCKED state. At the point where all subwarps are in the STALLED or
1. BSSY B0, syncPoint
2. @P0 BRA Else // P0 is 1 for t0, 0 for t1
3. TLD R2, R0, R1; &wr=sb5 // incr. scoreboard 5
4. FMUL R10, R5, c[1][16];
5. FMUL R2, R2, R10; &req=sb5 // load-to-use stall
6. BRA syncPoint;
   Else:
7. TEX R1, R8, R9; &wr=sb2 // incr. scoreboard 2
8. FADD R1, R1, R3; &req=sb2 // load-to-use stall
9. BRA syncPoint;
syncPoint:
10. BSYNC B0;

Figure 5.7: Simple code example that shows a divergent if-then-else branch with load-to-use stalls along both paths.

BLOCKED states, memory lookups from all applicable subwarps of a warp will successfully overlap in time, thus achieving improved latency tolerance compared to a baseline execution that serializes execution of divergent subwarps. Eventually, as memory lookups return to the SM, STALLED subwarps move to the READY state, thus making them eligible for selection again.

On a system that supports the subwarp-yield transition, after issuing a group of independent long latency texture lookups or global memory loads, the issuing subwarp may optionally attempt to yield its scheduling slot (through the use of explicit software instructions or through hardware based programmable thresholds) and eagerly move from ACTIVE to the READY state. Theoretically, subwarp-yield provides more flexibility to Subwarp Interleaving and enables maximal memory latency overlap. However, eager approaches like subwarp-yield lead to more switching on average in a subwarp’s lifetime. Frequent switching among subwarps can hurt performance by thrashing a processing block’s L0 instruction cache, and incurring higher overall switching penalties, since each switch costs a fixed latency.

Figure 5.8a illustrates the operation of the TST and the state machine described earlier on the toy example from Figure 5.7. We trace the execution of the toy example in steps. Each step may take one or more cycles. Though we show at most one state change per step (across subwarps) for clarity, in practice, since the subwarp-wake transition happens asynchronously as a result of register writeback, STALLED-to-READY state change can
Figure 5.8: TST operation with two 1-thread subwarps for the example code in Figure 5.7. Initially, all threads are active. When threads arrive at a divergent branch, the election logic selects one thread to execute (ACTIVE) and moves the other to a READY state. In (a), when the ACTIVE thread encounters a load-to-use stall at the use instruction, it moves to a STALLED state and the scheduler activates the READY thread. In (b), an ACTIVE thread yields execution to the READY thread as soon as it issues a long latency instruction and does not wait for a stall in the pipeline.
happen concurrently with other state changes brought about by the baseline SIMT model itself. In step 1, control flow diverges at the branch at PC 2. Prior to this, not shown in the figure, the \textit{BSSY B0, syncPoint} operation is executed by both threads, t0 and t1, and the convergence barrier mask in \textit{B0} gets set to \texttt{0x0b11}. After control flow diverges, thread 0 (t0) goes down the “else” path and its TST entry moves to the \texttt{READY} state. At step 2, thread 1 (t1) issues its long latency texture operation at PC 3 and remains in the \texttt{ACTIVE} state. At step 3, t1 successfully executes an independent math operation at PC 4. At step 4, t1 suffers a load-to-use stall and moves to the \texttt{STALLED} state. Thread 0 (t0) is selected in step 6 and issues a long-latency operation (PC 7) at step 5. It suffers a load-to-use stall in step 7. Meanwhile, in the background, t1’s scoreboard stall condition cleared and it is now eligible for selection. Thread t1 becomes \texttt{ACTIVE} in step 8, attempts to execute the \textit{BSYNC} at step 9, and moves to the \texttt{BLOCKED} state in step 10.

In a similar vein, Figure 5.8b illustrates Subwarp Interleaving on a system supporting the \textit{subwarp-yield} transition. They key difference is that the \textit{subwarp-yield} transition at step 2, after t1 has issued its long-latency texture operation, enables t0 to become \texttt{ACTIVE} and issue its long-latency texture operation much earlier in the overall schedule (at step 4) compared to Figure 5.8a, where t1 is able to issue its long-latency lookup only in step 6. Thus, \textit{subwarp-yield} allows a system to maximize memory level parallelism, which is key to lowering average memory access latencies.

5.4 Evaluation

5.4.1 Methodology

\textit{Simulator}

To collect the results in this paper, we extended an execution-driven, proprietary simulator that guides NVIDIA GPU product designs. This bare metal simulator (i.e. not full-stack) initializes data and instruction memory from traces. We configure the simulator to simulate
Table 5.1: Subwarp Interleaving: Experimental Setup.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td># Streaming Multiprocessors</td>
<td>2</td>
</tr>
<tr>
<td>Processing blocks per SM</td>
<td>4</td>
</tr>
<tr>
<td>Warp slots per processing block</td>
<td>{2, 4, 8}</td>
</tr>
<tr>
<td>Warp slots per SM</td>
<td>{8, 16, 32}</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>L1 data cache size</td>
<td>128KB</td>
</tr>
<tr>
<td>L1 instruction cache size</td>
<td>{64KB}</td>
</tr>
<tr>
<td>L0 instruction cache size</td>
<td>{16KB}</td>
</tr>
<tr>
<td>L1 miss latency</td>
<td>{300, 600, 900} cycles</td>
</tr>
<tr>
<td>Subwarp switch latency</td>
<td>6 cycles</td>
</tr>
</tbody>
</table>

a Turing-like SM in a cycle-accurate fashion, including processing blocks, L1 caches, texture units, and the RT-core raytracing accelerator. Because our target applications are not memory bandwidth limited, which we verified by examining performance counters from silicon runs on an NVIDIA GeForce Titan RTX, we do not model a complete GPU memory system, choosing instead to model memory with a simple fixed-latency stub model. Our stub model allows us to flexibly sweep through a range of memory latency values relatively quickly, often 100× faster than a full GPU simulator with a memory system. We collected several traces of gameplay, based on recent game and NVIDIA driver builds. Table 5.1 lists the simulation parameters that we use to collect the results in this paper. We test the sensitivity of SI to simulation parameters, such as the number of warp slots per processing block and SM. As mentioned above, we do not model the memory system beyond the SM, but instead study sensitivity to a variety of memory latencies, ranging from optimistically fast to pessimistically slow. To evaluate Subwarp Interleaving we modified our simulator to execute the policies described in Section 5.3.2.

**Application Traces**

Our work focuses on raytracing applications because they tend to be highly divergent with unpredictable memory behavior. Table 5.2 shows the applications we use to evaluate SI. Full-frame, trace-based execution is intractable so instead we collected traces of key raytracing kernel calls from the associated DXR applications. While our results do not
Table 5.2: Real-time graphics applications. AO for Ambient Occlusion, R for Reflection, and M for multiple raytracing effects represented in a given trace.

<table>
<thead>
<tr>
<th>Application</th>
<th>Name</th>
<th>RT effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArchViz Interior</td>
<td>AV1</td>
<td>GI-D</td>
<td>Architectural rendering [46]</td>
</tr>
<tr>
<td>ArchViz Interior</td>
<td>AV2</td>
<td>AO</td>
<td>Architectural rendering</td>
</tr>
<tr>
<td>Battlefield V scene 1</td>
<td>BFV1</td>
<td>R</td>
<td>Game [47]</td>
</tr>
<tr>
<td>Battlefield V scene 2</td>
<td>BFV2</td>
<td>R</td>
<td>Game</td>
</tr>
<tr>
<td>Control</td>
<td>Ctrl</td>
<td>M</td>
<td>Game</td>
</tr>
<tr>
<td>Collage</td>
<td>Coll1</td>
<td></td>
<td>Demo</td>
</tr>
<tr>
<td>Collage</td>
<td>Coll2</td>
<td></td>
<td>Demo</td>
</tr>
<tr>
<td>DDGI Villa</td>
<td>DDGI</td>
<td></td>
<td>Demo</td>
</tr>
<tr>
<td>Mechwarrior</td>
<td>Mech</td>
<td>M</td>
<td>Game</td>
</tr>
<tr>
<td>Minecraft</td>
<td>MC</td>
<td>M</td>
<td>Game</td>
</tr>
</tbody>
</table>

reflect current frame-level opportunity, we predict that raytracing will become the de facto real-time rendering standard. Thus we use the performance of these kernels as a proxy for the future opportunity of our approach. We include two traces for ArchViz and Battlefield V, which represent different raytracing techniques or scenes of their respective applications.

5.4.2 Results

In this section we first evaluate the scaling potential of SI using a microbenchmark. We then evaluate the performance of SI on raytracing kernels from interactive graphics applications.

CUDA Microbenchmark

Figure 5.9 shows the core of a simple CUDA microbenchmark that we designed to mimic a megakernel’s structure while allowing us to test the upper bounds of SI’s performance. We can configure the benchmark to splinter a warp into anywhere from two to 32 subwarps (via SUBWARP_SIZE). Each subwarp calls the gen ld to use stalls function, which performs a reduction of a slice of data, and we have ensured that each subwarp will suffer compulsory data cache misses when executed. Table 5.3 shows the performance potential of Subwarp Interleaving on the above microbenchmark. We vary the divergence factor as 2, 4, 8, 16, and 32, by varying SUBWARP_SIZE as 16, 8, 4, 2, and 1, respectively. SI delivers almost linear speedups until about 16-way divergence before
__global__ void subwarps(int *data, int *result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int warp_tid = tid % WARP_SIZE;
    int sub warp_id = warp_tid / SUBWARP_SIZE;
    int subwarp_offset = subwarp_id * NUM_ACCESS ES_PER_SUBWARP;
    for (int it = 0; it < ITERATIONS; it++) {
        switch (subwarp_id) {
            case 0:
                result[tid] = gen_ld_to_use_stalls(data, subwarp_offset, subwarp_id);
                break;
            case 1:
                result[tid] = gen_ld_to_use_stalls(data, subwarp_offset, subwarp_id);
                break;
            ...
            case 31:
                result[tid] = gen_ld_to_use_stalls(data, subwarp_offset, subwarp_id);
                break;
        }
        __synce warp();
        subwarp_offset += L2_CACHE_LINE;
    }
}

Figure 5.9: A CUDA microbenchmark that mimics a RT megakernel’s structure but allows
us to selectively generate up to 32 subwarps with guaranteed exposed load-to-use stalls.

Table 5.3: Subwarp Interleaving performance on the microbenchmark with an L1 miss
latency of 600 cycles.

<table>
<thead>
<tr>
<th>SUBWARP_SIZE</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divergence factor</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Speedup(×)</td>
<td>1.98</td>
<td>3.95</td>
<td>7.84</td>
<td>15.22</td>
<td>12.66</td>
</tr>
</tbody>
</table>

tapering off. Performance plateaus at high divergence factors (above 16) because with in-
creasing active subwarps, the number of active instruction fetch streams increases as well,
leading to L0 and L1 instruction cache thrashing. With 32-way divergence, we see load-to-
use stalls decrease to 0 with Subwarp Interleaving, but instruction fetch stalls rise sharply,
leading to diminishing returns.

5.4.3 Raytracing Kernels

We next describe the performance of Subwarp Interleaving on real-time raytracing kernels.
Unlike the microbenchmark scaling study, our goal here is to evaluate the effectiveness
of SI in exploiting these kernels’ natural divergence to effectively tolerate memory stalls.
We study the performance of our technique with an exhaustive sweep over three different
Figure 5.10: Subwarp Interleaving versus baseline at a fixed L1 miss latency of 600 cycles.
parameters: subwarp selection trigger policy, L1 miss latency, and Subwarp Interleaving with and without the subwarp-yield transition in our state machine. We trigger subwarp selection based on how many warps are currently stalled in a given processing block. We pick three points in that axis: $N = 1$ to represent all active warps having stalled, $N \geq 0.5$ to represent at least half of the active warps having stalled, and $N > 0$ to represent any active warp having stalled. For L1 miss latency, we sweep across values of 300, 600, and 900 cycles. For the last parameter, we report just the basic Subwarp Interleaving design as SOS (for switch-on-stall) and SOS plus subwarp-yield after long-latency instructions as Both.

Figure 5.10a shows the performance of subwarp interleaving at a fixed L1 miss latency of 600 cycles. In addition to showing the performance of individual settings, the graph shows a BestOf bar that captures the best performance for each application across all SI configurations. The single best performing setting is Both, $N \geq 0.5$, achieving an average speedup of 6.3%. Average BestOf speedup across all settings is 6.6%.

Figure 5.10b shows the reduction in total exposed load-to-use stalls ($T$) and exposed load-to-use stalls in divergent code blocks ($D$) in the Subwarp Interleaving run with respect to the corresponding baseline metrics. Exposed stalls in divergent code blocks, $D$, dropped by 26.5% on average (yellow bars). However, more than half of our traces see small reductions for $D$. There are two primary reasons that warps may be divergent yet present few interleaving opportunities. First, SI cannot generate two independent subwarps for “if-then” divergence because the subwarp that executes the then code must finish before the two subwarps rejoin and continue executing together. SI can only exploit divergence that leads to multiple, independent subwarps. Second, even for branches that generate two or more independent subwarps, which we expect the megakernel’s primary branch to do, SI is still at the mercy of the subwarp selection order. Ideally subwarps with loads execute before compute-heavy subwarps that will never trigger a subwarp-stall or subwarp-yield. Thus, while the $D$ metric is indicative of opportunity, it provides only a loose approximation.
Figure 5.11: Average speedups of Subwarp Interleaving over baseline across different L1 miss latency settings.

For applications with significant load-to-use stalls where most of the stalls are in divergent code blocks, SI is likely to help (BFV1, BFV2). For applications where most load-to-use stalls are in convergent code, performance is unlikely to improve commensurately with the reduction in $D$ (Coll1, Coll2).

Sensitivity Studies

L1 miss latency We summarize the sensitivity to latency by reporting the average performance of the various configurations in Figure 5.11. As expected, Subwarp Interleaving performs better with increasing L1 miss latencies, demonstrating its ability to effectively tolerate memory latency stalls. The BestOf speedups across L1 miss latency settings of 300, 600, and 900 cycles are 4.2%, 6.6%, and 7.6%, respectively.

Peak warp slots per SM As applications increasingly look to exploit task-level parallelism with modern graphics API support for Asynchronous Compute queues (“Async” queue) [48, 49], there will be contention for limited available warp slots since tasks from multiple queues will overlap in time. This trend may prevent an approach like DWS [50], which relies on forking new warps at divergence points, from finding enough free warp slots to achieve maximal latency tolerance. To study the effect of limited warp slots on performance of SI, we sweep through different maximum warp count settings and com-
Subwarps per warp  We study the sensitivity of Subwarp Interleaving to the number of subwarps per warp supported in the SI design. The sensitivity we present allows an architect to choose an appropriately sized Thread Status Table (TST) to balance performance with power and area concerns. Figure 5.13 presents speedup data from sweeping subwarp count from 2, 4, 6, to unlimited (max 32). Even with support for as little as 2 subwarps per warp, Subwarp Interleaving is able to achieve an average speedup of 4.2%, with speedups increasing sub-linearly with more subwarps per warp. The 4 subwarps per warp configuration only requires a 4-entry TST and uses one eighth the TST and Sub warp Wakeup logic of the unlimited configuration. Yet the 4 subwar p configuration achieves a 5.2% speedup, capturing 82% of the unlimited configuration’s average upside. Per the 80-20 Pareto principle,
the 4 subwarp configuration may be an appropriate design point for SI.

**Instruction cache sizing** Finally, our baseline configuration upsizes the L0 and L1 instruction caches (16KB and 64KB respectively) to better cater to the needs of Subwarp Interleaving. An experiment with $4 \times$ smaller L0 and L1 instruction caches (to mimic shipping GPUs) yielded a 4.5% average speedup, which is about 70% of our single best configuration speedup of 6.3%.

### 5.5 Discussion

Our primary objective with SI was to reduce exposed memory load-to-use stalls. Our evaluation on a cycle-level simulator shows that Subwarp Interleaving does reduce exposed long-latency stalls, sometimes significantly. However, productization of SI is challenged by several factors. First, the reduction in exposed long-latency stalls due to SI often comes with an increase in instruction fetch stalls due to frequent switching among instruction streams. These stalls can be mitigated with larger caches, albeit at an area cost. In an area-limited chip design, spending additional area on one feature typically requires involves reducing area for or eliminating another feature. These decisions require broad perfor-
mance per $mm^2$ analyses across many candidate features in a product design, with the ones producing the biggest bang for the buck winning out.

Second, Amdahl’s Law limits speedups for some of the raytracing kernels. Recall that the invocation of various hit or miss shaders in the raytracing megakernel depends on the outcome of a ray traversal operation, which the RT-core unit performs. While Subwarp Interleaving reduces exposed load-to-use stalls in divergent hit/miss shader execution for all of our raytracing kernels, the latency of ray traversal operations is often the dominant factor.

Third, the order in which a processing block encounters subwarps is important. For example, in a warp with two subwarps $A$ and $B$, if only subwarp $B$ contains load-to-use stalls, execution order matters. Unless $B$ executes first, SI will be of no value since $A$ will never switch. When $A$ completes and $B$ begins executing, no other subwarp is available to switch to on its memory stalls.

Fourth, while Subwarp Interleaving broadly applies to raytracing kernels, and we argue that raytracing is an extremely important application category, SI currently applies to a limited set of code. SI can only improve application performance when there are long stalls within divergent code, and too few active warps to hide the latency. As skilled SIMT programmers strive to remove divergence from code, we did not expect SI to be universally applicable. We profiled a broad suite of more than 400 non-raytracing CUDA and Direct3D compute kernels and found only 11 that feature long stalls in divergent code, and none benefited beyond the margin of noise from SI.

In addition, current RT game titles are not fully raytraced, but also include general compute and traditional rasterization-based graphics, which dilute SI’s gains at the frame level. While Subwarp Interleaving is not readily applicable to today’s workloads, we remain optimistic that as GPU applications and technologies evolve, it could be a boon for a future GPU design. As researchers explore novel uses of NVIDIA’s raytracing cores to accelerate complex tree data structure traversals [51, 52, 53, 54], we expect that more applications
with unpredictable control flow and dependent memory fetches will begin running efficiently on modern GPUs, and a first class hardware feature like Subwarp Interleaving that exploits divergence to achieve better latency tolerance will find broader benefit. Though sophisticated software techniques can help improve convergence rates and memory level parallelism, a well-designed hardware technique will avoid the runtime overheads that software techniques impose and will also relieve application developers of the burden of devising strategies to overcome performance problems and allow them to focus on delivering richer functionality.

5.6 Future Work

Future work may implement a combined software-hardware approach to subwarp interleaving where the compiler provides a priority to each subwarp which the hardware uses for better branch target selection. This approach helps ensure that memory-heavy subwarps are scheduled first so that math-heavy subwarps are available for interleaving in case of a stall.
Chapter 6
Software-Directed Register File Sharing

6.1 Motivation

As described in section 5.1, Raytracing kernels are latency-sensitive, divergent, and contain relatively few active warps. This low occupancy is a result of register-limitedness in Raytracing applications. Figure 6.1 shows the improvement in performance for Raytracing applications when the size of the register file is doubled.

![Speedup for RT apps with 2x RF](image)

Figure 6.1: Speedup for Raytracing applications with increased occupancy after doubling the register file size.

In chapter 5, we described Subwarp Interleaving, a hardware approach that leverages GPU thread divergence to increase effective occupancy, resulting in improved latency tolerance in Raytracing kernels. However, Subwarp Interleaving was limited in its impact to stalls in divergent code regions. In this chapter, we describe a software-hardware co-design approach that uses register file sharing to increase the number of active warps in Raytracing kernels, to improve latency hiding in both divergent and convergent code regions.

Raytracing algorithms employ a megakernel implementation on GPUs which has a di-
vergent branch that calls a different shader depending on the material of the object that the ray intersects and the compiler may allocate each such shader a different number of registers. The warp then receives the maximum registers that it may require. Figure 6.2 shows an example megakernel with four shaders.

![Figure 6.2: Raytracing megakernel](image)

Occupancy is the ratio of active warps to the maximum number of warps the SM supports. The GPU warp scheduler interleaves the execution of active warps to achieve latency hiding. High occupancy allows for better latency hiding while low occupancy may result in stalls. Occupancy is limited by the hardware resources used by each thread, including registers and shared memory[55].

Since the compiler allocates the maximum registers to Raytracing kernels, these applications tend to have low occupancy. However, our experiments show that these warps do not actually use all the allocated registers for the entire duration of their execution. Figure Figure 6.3 shows a histogram of register usage for Battlefield V over time. The x-axis shows the registers allocated to the currently executing shader. The y-axis shows the total simulation cycles spent executing the corresponding shader. While the compiler assigns each warp 128 registers, the kernel spends most of its time executing shaders that use 64 registers, leaving most of the allocated registers idle.
6.2 Solution

We propose Software-Directed Register File Sharing, a software-hardware co-design technique that allows warps to time-share registers to increase occupancy. The compiler selects an initial register target which decides the occupancy of the kernel. Next, it inserts register acquire and release instructions at the start and end of each shader that uses more than the initial number of registers. The compiler also selects a lower bound on the number of registers required to run the application.

Figure 6.4 shows the example Raytracing megakernel with acquire and release instructions inserted. Each warp is initially assigned 46 registers, which allows for more warps to be active. When a warp invokes a shader that uses more registers, such as shader 1, it tries to acquire the required registers from the free pool before executing the shader. Once the shader execution finishes, the warp releases the registers back to the free pool. Hence, with Register File Sharing, each warp only acquires additional registers when it needs them and then releases them back when they are not needed, thereby allowing other warps to acquire them rather than leaving these registers idle.
6.2.1 Slow path execution

If the free register pool has insufficient registers, the acquire() request may fail. To handle this, we use a `try_acquire()` instruction which returns a boolean indicating whether the acquire was successful. If the acquire was successful, the shader executes like normal. However, in the case that the acquire returns false, the compiler inserts a slow, fall-back path which uses fewer registers but may have higher latency due to the insertion of spill instructions, rematerialization or changes to instruction-level parallelism employed by the compiler to generate code for the shader using fewer registers.

6.2.2 Deadlock Prevention

Note that there is a potential for deadlock if there are multiple warps that need to acquire registers to make progress and there are insufficient registers in the free pool to service any of the acquire requests. This gives rise to a situation where multiple warps are waiting on each other to release registers before they can make progress, resulting in a deadlock. However, with our slow path solution, a warp can always make forward progress even if it fails to acquire registers, and there can be no deadlock.
6.2.3 Compiler Algorithm

- **Fast path compilation:** First, the compiler separately compiles each of the raytracing shaders and outputs the optimal and minimum register targets for each.

- **Initial register target selection:** Next, we select an initial register target $R$ such that $R$ is greater than the minimum registers required by any shader in the application to make progress. This selection can be done either statically or using profile information. We employ a profile-based feedback loop that uses runtime performance information to update $R$.

- **Slow path compilation:** The compiler then recompiles each shader $S$ which used more than $R$ registers to generate a slow version $S'$ by setting maxregcount=$R$.

- **Acquire and Release:** Finally, we insert the acquire and release instructions and conditionally call the fast and slow versions of each shader based on whether the acquire request succeeded. Listing 6.1 shows the modified shader with `try_acquire()`, `release()` and slow and fast path execution.

Listing 6.1: Slow Path Execution

```c
shader()
{
    if (try_acquire(N)) {
        fast_version()
        release(N)
    } else {
        slow_version()
    }
}
```
6.3 Evaluation

For our experiments, we used a GPU simulator and ran a sweep across varying register launch targets. A lower launch target results in higher occupancy but also a higher rate of failure to acquire registers. On the other hand, a higher launch target avoids slow path execution at the cost of occupancy. Our experiments aim at finding the sweet spot that improves occupancy without spending a lot of cycles in slow path execution. Figure 6.5 summarizes our evaluation methodology.

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Ampere-based simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
<td>RT traces</td>
</tr>
<tr>
<td>Register File</td>
<td>Increase effective RF size based on launch target</td>
</tr>
<tr>
<td>Metrics</td>
<td>Speedup*, slow path rate</td>
</tr>
<tr>
<td>Slow path avoidance</td>
<td>Retry loop</td>
</tr>
</tbody>
</table>

- *Does not simulate cost of slow path execution

Figure 6.5: Register File Sharing: Evaluation Methodology

Figure 6.6 shows the impact of varying initial register targets and register file sharing on Battlefield V. Our observations show that a launch target of 48 gives the maximum speedup from improvement in occupancy but also results in a 100% failure to acquire registers. On the other hand, a launch target of 80 results in a 20% speedup and 5% cycles spent in slow path execution. Note that our current experiments do not take into consideration the slowdown due to slow path execution.
6.4 Discussion

**Launch target:** The compiler selects an initial register target to allocate to each warp on launch. This target determines the occupancy of the application. Our proposal uses a profile-guided approach to identify the ideal launch target.

**Minimum register count:** The compiler selects a minimum register count below which the warp cannot release registers. This minimum count is required to ensure that the warp is able to execute at least the slow path and guarantee forward progress. Further, certain shaders may be "hot" and we may want to always execute them in fast path mode, which affects the minimum register count.

**Cache locality:** Register file sharing increases the number of active warps which might increase the number of cache misses especially in Raytracing kernels where different warps may be executing different shaders.

**Cost of acquire:** Acquire instructions may result in pipeline stalls, whereas release instructions may execute asynchronously and do not affect overall warp latency. The cost of acquires may be mitigated by the increase in occupancy which provides the warp scheduler
other work to schedule while the acquire request is pending. Instruction level parallelism is another technique that may help reduce this cost. Listing 6.2 shows how we can hoist code to increase instruction level parallelism and hide acquire latency.

Listing 6.2: Code Hoisting

```c
p = try_acquire(N);
// common code using R registers
if (p) {
    // fast path execution using N + R registers
    release(N)
} else {
    // slow path execution using R registers
}
```

6.5 Future Work

In this chapter, we described software-directed Register File Sharing to improve occupancy on GPUs by time-sharing registers across warps. To handle potential deadlock situations, we proposed slow-path execution where the compiler creates a copy of the code using fewer registers. To do this, the compiler inserts spill or rematerialization code or modifies the instruction schedule to reduce register pressure. This slow-path is executed conditionally if register acquire fails because there are insufficient registers available in the free pool. However, slow path execution is suboptimal, and future work may focus on tactics that avoid or reduce the impact of slow path execution, which may include:

- **Partial acquire**: Instead of an all-or-nothing approach where a warp executes the slow path unless it gets all the registers that it needs, the compiler might create multiple slow paths using different register targets. If the warp acquires only a subset of required registers, it can execute a better performing slow path. Partial acquire thereby reduces the impact of slow path execution when acquire fails.
• **Retry loop:** Instead of immediately assuming a deadlock and executing the slow path when acquire() fails, the compiler might insert a back-off loop that attempts to acquire multiple times before executing the slow path.

• **Register aware warp scheduling:** If the compiler inserts scheduling hints, the warp scheduler can select a warp to schedule based on the number of registers available in the free pool. If a warp needs more registers than available in the free pool, the warp scheduler selects a different warp to issue until the free pool has more registers available. This approach helps avoid slow path execution by delaying acquire of registers until the free pool has sufficient registers.

• **Register aware subwarp scheduling:** As described in chapter 2, when threads in a warp diverge at a branch as in a raytracing kernel, the GPU serializes the execution of the subwarps. In branch target selection, as with register aware warp scheduling, the divergence handling mechanism may be modified to take into consideration the number of registers available in the free pool. At the branch, the divergence handling hardware checks the register pool to select the appropriate subwarp based on compiler inserted register acquire hints to avoid slow path execution.

Furthermore, while our solution is limited to acquire and release at a function or shader level, future work may look into register file sharing at a finer granularity such as at a basic-block level.
CHAPTER 7
MEMORY ACCESS SCHEDULING

7.1 Motivation

Migratory thread architectures reduce memory bandwidth, latency, and energy utilization by migrating lightweight thread contexts to lightweight near-memory processors (called "nodelets") instead of transferring blocks of data to a requesting processor [10]. This approach works well for data-intensive applications with weak locality, such as graph analysis, data analytics, and machine learning [56]. However, while transporting thread contexts is cheaper than moving data, thread migrations still incur energy and bandwidth overheads and can be particularly expensive if threads frequently migrate in a ping-pong manner between processors due to poor locality of access.

To study the cost of thread migration, we wrote two microbenchmarks with a single thread that reads data from (1) a replicated array, and (2) a distributed array. We found that the first microbenchmark with no migrations ran $15.7 \times$ faster on an EMU processor than the second with 100,000 migrations. The cost per-migration is approximately 272 cycles, which can be a significant overhead if incurred for a large number of read operations.

We use the sparse matrix-vector multiplication (SpMV) code from Listing 7.1 as a running example. The function `mw_malloc1dlong` distributes arrays across nodelets in a round-robin manner starting from Nodelet 0. Hence, `values[j]` and `colind[j]` are always on the same nodelet. The location of $x[colind[j]]$ cannot be statically determined, so we conservatively assume that it will result in a migration.
Figure 7.1: Thread migrations in SpMV

### Listing 7.1: Sparse Matrix Vector Multiply

```c
1 long *values = mw_malloc1dlong(N);
2 long *x = mw_malloc1dlong(N);
3 long *colind = mw_malloc1dlong(N);
4 for (i = 1 to N) {
5     sum = 0;
6         for (j = rowptr[i] to rowptr[i+1]) {
7             sum += values[j] * x[colind[j]];
8         }
9     y[i] = sum;
10 }
```

Listing 7.2 shows the code generated for line 9 and Figure 7.1(a) illustrates the execution of a single iteration of the SpMV loop. We observe that there are (at most) three migra-
tions in each iteration of the inner loop resulting in $3N^2$ total migrations. Listing 7.3 shows an alternative code sequence which groups together accesses from the same nodelet as much as possible, before executing a migration-inducing instruction. The modified schedule has a maximum of two migrations per iteration of the inner loop resulting in $2N^2$ total migrations, a reduction of 33%. Simulator studies show that total migrations for the SpMV application reduced from 2,190,497 to 1,317,771 migrations with the modified schedule.

Listing 7.2: 3 Migrations

```
1 S1: R0 = load col ind [j]; // migrate to nodelet 0
2 S2: R1 = load x[R0];     // migrate to nodelet 1
3 S3: R2 = load values[j]; // migrate to nodelet 0
4 S4: R3 = mul R2, R1;
5 S5: sum = add sum, R3;
```

Listing 7.3: 2 Migrations

```
1 S1: R0 = load col ind [j]; // migrate to nodelet 0
2 S3: R2 = load values[j]; // migrate to nodelet 0
3 S2: R1 = load x[R0];     // migrate to nodelet 1
4 S4: R3 = mul R2, R1;
5 S5: sum = add sum, R3;
```

7.2 Problem

We define co-located memory accesses as accesses to addresses that are statically known to reside on the same nodelet and would not cause a thread migration when issued consecutively. This includes accesses to replicated memory which never result in a migration, accesses to multiple elements of the same block of a 2D array allocated by `mw.malloc2d()`, and accesses to a 1D array `array` allocated by `mw.malloc1dlong()` with a stride that equals the number of nodelets. Co-located memory accesses represent a different form of locality than traditional cache-based spatial or temporal locality. Co-located accesses may not reside in the same block of memory and may never be reused. However, grouping together co-located data accesses reduces thread migration. Note that co-location is a symmetric relation because the order of the memory references does not impact the result.
Problem Statement. Identify co-located memory accesses and reorder them to minimize the number of thread migrations while respecting data and control dependencies. We call this the Memory Access Scheduling problem[57].

7.3 Solution

We propose a three-step solution to this problem. First, we identify and propagate data layout information for each memory access through a dataflow analysis we call Layout Analysis described in Section 7.3.1. Next, we use data layout and array index information to identify co-located memory accesses by a process we call Stride Analysis (see Section 7.3.2). Finally, we reorder instructions to minimize thread migrations by grouping together co-located accesses. We propose both an ILP-based (Section 7.3.3) and a heuristic-based instruction scheduler (Section 7.3.4).

7.3.1 Layout Analysis

The Layout Analysis pass determines how consecutive elements of an array are allocated across the nodelets. For our analysis, we consider the following kinds of data allocation:

- **Local**: The array is allocated on the local memory of the nodelet where the requesting thread is running.
- **Co-located**: The array is allocated on the same nodelet as another pointer using `mw_localmalloc()`.
- **1D**: The array is striped across nodelets.
- **2D**: The array is block striped across nodelets.
- **Replicated**: A copy of the data element resides on each nodelet.

The Layout Analysis pass is an interprocedural dataflow analysis pass that first traverses all functions in the module to propagate the layout of each pointer or array variable within
Figure 7.2: Lattice for layout analysis

Algorithm 3: Intraprocedural Layout Analysis: DFA

1. **Domain:** \{local, colocated, 1D, 2D, replicated\}
2. **Direction:** forward
3. $\top$ = Memory layout not yet determined. This is the initial value, i.e., the empty set $\{\}$.  
4. $\bot$ = Memory layout cannot be determined. This is the universal set.
5. **Meet Operation:** $\land$ on the Lattice as shown in Figure 7.2
6. **Transfer function:** given instruction $dst = Instr(op1, op2)$,
   
   $Layout(dst) = Layout(op1) \land Layout(op2)$

7. $\forall x$
   
   - $x \land \top = x$
   - $x \land \bot = \bot$
   - $c1 \land c2 = \bot$, if $c1 \neq c2$

the function and then propagates the layouts across functions. Algorithm 3 describes our dataflow analysis over the lattice shown in Figure 7.2.

Once we know the layout of each variable based on the intraprocedural analysis, we propagate this information across function calls using a flow-sensitive and context-insensitive interprocedural analysis. At each of the call sites, if the layout of the arguments in the caller function is determined by the intraprocedural analysis, then we propagate the layout to the callee function parameters, by applying the Meet($\land$) operator, $Layout(param) = Layout(param) \land Layout(arg)$. On the other hand, if only the layout of the parameters of
the callee function is determined by the intraprocedural analysis and the argument layout is unknown, then \( \text{Layout}(\text{arg}) = \text{Layout}(\text{param}) \). The analysis also propagates the layout of the return value is across the def-use chain of the result of the call statement in the caller function. The interprocedural analysis iterates until convergence, that is, until there are no more updates to the layout information. Convergence is guaranteed by the monotonicity of the transfer function. Finally the layout analysis infers the memory layout for each variable in the program.

7.3.2 Stride Analysis

Next, we use layout and array index information to determine whether a pair of migration-inducing memory accesses is co-located. For our purposes, we define a migration-inducing memory access as one that could potentially require a thread migration. This includes memory reads and atomics from non-replicated data structures. We propose Stride Analysis, a pass that determines the distance between two memory accesses depending on the distribution of the array in memory and the offset into the array. Our analysis relies on the basic assumption that the base address of an array is on nodelet 0 in node 0. The offset from the base address, therefore, decides the nodelet of each element.

Given two memory references, we define \( \text{stride} \) as the relative distance between two migration-inducing memory accesses in terms of nodelets.

- **Case 1**: Both accesses have a 1D layout.

  Consecutive elements are located on adjacent nodelets and striped across all nodelets in a round-robin fashion. Hence, the stride between the two accesses is simply the absolute difference between their offsets:

  \[
  \text{stride}(A[i_1], B[i_2]) = |i_1 - i_2|
  \]

- **Case 2**: Both accesses have a 2D layout.
Consecutive rows are on adjacent nodelets so that all elements of a row are on the same nodelet. Hence, the stride is computed using the difference between the row offsets and is independent of the column offset:

\[ \text{stride}(A[i_1][j_1], B[i_2][j_2]) = |i_1 - i_2| \]

- **Case 3**: A has a 1D layout and B has a 2D layout.

\[ \text{stride}(A[i_1], B[i_2][j_2]) = |i_1 - i_2| \]

- **Case 4**: They were allocated on the same nodelet using `mw_localmalloc()`.

\[ \text{stride}(a, b) = 0 \]

- **Otherwise**:

\[ \text{stride}(a, b) = \text{unknown} \]

Finally, we use stride information to determine if any pair of memory references \( \langle M_1, M_2 \rangle \) is co-located. Two non-replicated memory accesses \( \langle M_1, M_2 \rangle \) are co-located if their addresses are separated by some multiple of total nodelets, i.e.,

\[ \text{stride}(M_1, M_2) = \text{num_nodelets} \times k \]

for some \( k \in \mathbb{Z} \).

### 7.3.3 ILP-based Scheduler

Given a directed acyclic graph \( G = (V, E) \) with a set of nodes \( V \), edges \( E \) that represent precedence relations between the nodes, and a distance matrix, a sequential ordering of the
Figure 7.3: SpMV: Memory Access Scheduling as Sequential Ordering Problem. Vertices shaded the same color are co-located memory accesses. White vertices represent non migration-inducing instructions. The solid arrows represent data dependences while the dotted arrows represent augmented dependences added by the ILP solver. The undirected weighted edges illustrated by dashed lines represent the cost of migration between nodes.

A sequential ordering of the resultant graph, called the Migration Dependence Graph, generates a schedule that minimizes thread migrations and maintains data and control dependences. While layout and stride analysis were performed globally and interprocedurally, our scheduler generates a separate migratory dependence graph for each basic block.

\[
Distance(I_1, I_2) = \begin{cases} 
0, & \text{if } I_1 \text{ and } I_2 \text{ are co-located} \\
1, & \text{if } I_1 \text{ and } I_2 \text{ are non-co-located}
\end{cases}
\]
We envision that our approach can be extended to more global scopes in the future. Figure 7.3(b) shows the migration dependence graph corresponding to the SpMV example from Listing 7.1. Algorithm 4 shows our proposed ILP formulation. Our formulation has \(O(n^2)\) variables and \(O(n^2)\) constraints.

Algorithm 4: ILP Formulation

1 Parameters

- \(n\) = number of migration-inducing instructions
- \(\text{Distance}_{ij} = \begin{cases} 0, & \text{if } i, j \text{ are on the same nodelet} \\ 1, & \text{otherwise} \end{cases}\)
- \(\text{Dependence}_{ij} = \begin{cases} 1, & \text{if } j \text{ depends on } i \\ 0, & \text{otherwise} \end{cases}\)

Decision Variables

- \(T_i\), time at which instruction \(i\) is scheduled

Objective

Minimize number of migrations,

\[
\sum_{i=0}^{n} \sum_{j=0}^{n} (T_i - T_j - 1) \times \text{Distance}_{ij}
\]

Constraints

- All instructions must be scheduled, i.e.,

  \[ \forall i, 1 \leq T_i \leq n \]

- Data dependences must be maintained, i.e.,

  \[ \text{Dependence}_{ij} = 1 \implies T_i \leq T_j \]

- Only one instruction may be scheduled at a given time, i.e.,

  \[ i \neq j \implies T_i \neq T_j \]

Theorem. Memory Access Scheduling is NP hard

Proof. To prove that Memory Access Scheduling is NP-hard, we reduce the Sequential Ordering Problem to memory access scheduling. The sequential ordering problem takes
as input a directed acyclic graph $G = (V, E)$ and a distance matrix, $D$, and generates a minimum cost path that respects precedence constraints.

Let $B$ be an empty basic block. For each node $n \in V$, we add a new migration-inducing instruction to $B$. Next, we convert the precedence constraint edges to data dependencies between the instructions. Finally, for each pair of nodes $n_1$ and $n_2$, we add a migration cost of $D[n_1][n_2]$ between the corresponding instructions in the basic block. Then, a minimum-migration schedule of instructions in the basic block that maintains data dependencies between instructions generates a sequential ordering of $G$, i.e., a minimum-length path in $G$ that respects precedence constraints.

We now describe the overall instruction scheduling algorithm to minimize migrations using the sequential ordering problem (see Algorithm 5).

As a first step, we build a Migration Dependence Graph with nodes representing migration-inducing instructions and directed edges between nodes representing direct or indirect dependencies. Next, we use layout and stride analysis (described in Section 7.3.1 and Section 7.3.2) to determine distances between each pair of nodes in the Migration Dependence Graph. We then pass this Migration Dependence Graph as an input to an ILP solver for the sequential ordering problem described in Algorithm 5. The solver returns an optimal ordering for the memory access instructions in the Migration Dependence Graph. To respect the sequential ordering generated by the solver in the final schedule, we augment the original dependence graph of the program with false dependencies. If instruction $I_1$ was scheduled before $I_2$ in the sequential ordering, we add an edge from $I_1 \rightarrow I_2$ in the Augmented Dependence Graph. Finally, we perform a topological sort on the instructions in the augmented dependence graph, which optimally orders memory accesses while respecting original dependencies in the program.

Figure 7.3(c) shows the augmented dependence graph for SpMV generated using the ILPScheduler() detailed in Algorithm 5. A topological ordering of the graph in Figure 7.3(c) results in the optimized schedule from Listing 7.3.
Correctness: Our instruction scheduler does not move instructions across sync/lock instructions. Further, the topological ordering of instructions maintains all data dependences. Hence, reordering instructions within a thread does not introduce data race conditions or affect the correctness of the program.

Theorem. The augmented dependence graph is a DAG, i.e., there are no cycles in the augmented dependence graph, and the program has a valid schedule.

Proof. We know that the original dependence graph has no cycles. Assume that the augmented graph does contain a cycle due to a newly added edge from node A to B. For a cycle to exist, there must be a path from node B to A. This path was either an actual dependence from B to A or an augmented edge introduced by the ILP solver.

If there were a dependence from B to A in the original dependence graph, the sequential ordering would respect precedence and schedule B before A in the final schedule by definition. If, on the other hand, B to A contains an augmented edge, then the sequential ordering contains a cycle. In either case, a cycle results in a contradiction.

Algorithm 5: ILPScheduler: An ILP solver that takes as input the dependence graph and distance information and outputs a minimum cost sequential ordering of memory access instructions, which is used to augment the dependence graph. A topological sort over the resultant dependence graph generates the final schedule.

```
Input: NodeletMap, DependenceGraph
Output: Schedule of instructions
1 Dependencies ← CalcMemDeps(DependenceGraph);
2 Distances ← CalcMemDistances(NodeletMap);
3 T ← SequentialOrdering(Dependencies, Distances);
4 AugmentDependenceGraph(T, DependenceGraph);
5 TopologicalSort(DependenceGraph);
```

7.3.4 Heuristic Scheduler

We now describe a heuristic-based list scheduling algorithm (Algorithm 6) that aims to reduce the number of migrations in the program in polynomial time.
Algorithm 6: Heuristic Scheduler: A list scheduler that performs a topological sort on the dependence graph and groups co-located memory access instructions together.

**Input:** NodeletMap, DependenceGraph

**Output:** Schedule of instructions

```plaintext
1 \( current \leftarrow -1; \)
2 \( \text{while some instructions not scheduled do} \)
3 \( \quad \text{scheduled} \leftarrow \text{False}; \)
4 \( \quad // \text{schedule non-migrating instructions} \)
5 \( \quad \text{for } \text{inst in all instructions do} \)
6 \( \quad \quad \text{if inst.ready()} \land (\neg \text{inst.memoryAccess()} \lor \text{inst.nodelet} = \text{current}) \text{ then} \)
7 \( \quad \quad \text{schedule(inst);} \)
8 \( \quad \quad \text{scheduled} \leftarrow \text{True}; \)
9 \( \quad // \text{migrate to new nodelet} \)
10 \( \quad \text{if } \neg \text{scheduled} \text{ then} \)
11 \( \quad \quad \text{current} \leftarrow \text{Migrate();} \)
```

As a first step, we build a dependence graph for the program where nodes represent instructions and edges represent dependencies. Next, we use layout and stride analysis (described in Section 7.3.1 and Section 7.3.2) to group memory access instructions on the same nodelet into supernodes, thereby forming a Dependence Supergraph. Finally, we perform a topological sort of the dependence supergraph, which tries to schedule all instructions from one nodelet before migrating to a new nodelet. Arithmetic instructions and remote or replicated memory accesses do not induce migrations and may be scheduled whenever their operands are ready. Note that this supergraph may have cycles if none of the nodelet groups have all instructions ready to schedule, in which case we select a ready instruction at random.

Figure 7.4 shows how the instructions in our SPMV program from Listing 7.1 are scheduled using the HeuristicScheduler() detailed in Algorithm 6. Once again, it can be verified that a topological ordering of the supergraph in Figure 7.4(b) results in the minimal-migration schedule from Listing 7.3.

Figure 7.5 compares the output of the heuristic scheduler and the ILP-based scheduler for a synthetic example. The heuristic scheduler finds a cycle in the dependence supergraph.
Figure 7.4: SPMV: Heuristic-based Scheduling.

(Figure 7.5b) that must be broken. The number of effective migrations depends on which
nodelet it schedules first: selecting group 1 results in four migrations, whereas group 2
yields three migrations. However, the ILP solver always generates the optimal schedule
with three migrations (see Figure 7.5c).

7.4 Evaluation

This section describes the experimental setup and benchmarks used, followed by a detailed
analysis of the impact of memory access scheduling on thread migrations, speedup, and
compile time. Table 7.1 summarizes the Emu Chick hardware setup used in our exper-
iments. As mentioned in Chapter 2, this hardware supports the Emu Cilk programming
model.

All benchmarks discussed below were ported to Emu Cilk. Each benchmark was com-
piled with and without memory access scheduling enabled, and profiled on the Emu sim-
ulator v20.06 [12] to count the number of thread migrations. We also measured the actual
performance impact on real hardware to obtain the average kernel execution time with and
without memory access scheduling over three runs.
Performance Results

Table 7.2 lists the 7 explicitly-parallel kernels used in our evaluation, along with their sources. These kernels used all three Emu Cilk primitives: \textit{cilk\_spawn}, \textit{cilk\_sync}, and \textit{cilk\_for}. Figure 7.6(a) shows the percentage reduction in thread migrations for each of the benchmarks, measured on the simulator for both the ILP scheduler and the heuristic scheduler. Figure 7.6(b) shows the corresponding speedup on actual hardware with a single Emu node. The baseline for all our experiments is the EMU LLVM-based compiler, using the default compilation flags. We see that memory access scheduling reduces migrations by up to $1.99\times$ based on the simulator, and resulted in a speedup on real hardware of up to $1.87\times$ with a geometric mean speedup of $1.23\times$.

We also evaluated our approach on a collection of 15 automatically-parallelized benchmarks from the Polybench suite [64]. Automatic parallelization was performed using the
Table 7.1: Memory Access Scheduling: Experimental Setup

<table>
<thead>
<tr>
<th>Configuration</th>
<th>1 Emu Node (8 Nodelets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>64 GB</td>
</tr>
<tr>
<td>Per-Thread Registers</td>
<td>16</td>
</tr>
<tr>
<td>Storage</td>
<td>1TB Solid State Disks</td>
</tr>
<tr>
<td>Compute</td>
<td>12 Gossamer Cores</td>
</tr>
<tr>
<td></td>
<td>1 Stationary Core</td>
</tr>
<tr>
<td>Gossamer Core Clock Speed</td>
<td>150 MHz</td>
</tr>
<tr>
<td>System Interconnect</td>
<td>Serial RapidIO (SRIO)</td>
</tr>
</tbody>
</table>

Table 7.2: Memory Access Scheduling: Explicitly-parallelized kernels with sources

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpMV CSR</td>
<td>CSR Sparse Matrix-Vector Multiply [59]</td>
</tr>
<tr>
<td>SpMV COO</td>
<td>COO Sparse Matrix-Vector Multiply [59]</td>
</tr>
<tr>
<td>SpMM</td>
<td>Sparse Matrix-Dense Matrix Multiply [60]</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Simulation of thermal transmission [61]</td>
</tr>
<tr>
<td>Array Sort</td>
<td>Quick Sort an array of elements [62]</td>
</tr>
<tr>
<td>Matrix Sort</td>
<td>Sort non-zero elements in sparse matrix [62]</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>3D MRI reconstruction [63]</td>
</tr>
</tbody>
</table>

PPCG compiler [65] to obtain OpenMP code, which was converted to Emu Cilk by replacing OpenMP parallel for loops by `cilk_for` loops. We used the loop unroll pragma to enable loop unrolling by a factor of 2 in kernels that had an inner loop over columns of a 2D array. The 15 benchmarks chosen were those for which PPCG successfully generated parallel code in our evaluation.

Figure 7.7 shows the speedup for the automatically parallelized Polybench benchmarks using memory access scheduling on a single node configuration of the Emu hardware. Our results show that memory access scheduling improves run-time by up to $1.43 \times$ for these applications with a geometric mean of $1.10 \times$.

7.4.2 Analysis

Impact of Register Pressure. Some benchmarks showed unexpected slowdowns, or lack of improvement, with memory access scheduling enabled. Upon further investigation, a major contributor to this anomaly was the impact of instruction reordering on register pres-
Figure 7.6: Migrations and Speedup for explicitly-parallelized kernels.

Sure, which in some cases resulted in increased run time due to spill related migrations. Note that a register spill that is inserted between co-located memory accesses can result in two additional migrations (to the spill location and back) when there would be none in the absence of a spill. The examples with the largest slowdowns were lu and syr2k in Figure 7.7, with ILP Scheduler speedups of 0.87× and 0.94× respectively. Looking at the generated code, we saw a a 40% and 6% increase in static spill-related instructions after memory access scheduling respectively. We also studied the dynamic number of spill related migrations at run time for lu and found that the percentage of spill related migrations increased from 2% to 40% with memory access scheduling. These results show that memory access scheduling is more effective for programs with low register pressure or architectures with more thread registers. It also motivates future work on integrating memory
Table 7.3: Memory Access Scheduling: Impact on Compilation Time

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base (s)</th>
<th>ILP (s)</th>
<th>Heuristic (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpMV CSR</td>
<td>1.11</td>
<td>1.47</td>
<td>1.437</td>
</tr>
<tr>
<td>SpMV COO</td>
<td>1.141</td>
<td>9.647</td>
<td>1.39</td>
</tr>
<tr>
<td>SpMM</td>
<td>1.265</td>
<td>1.336</td>
<td>1.303</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.230</td>
<td>1.533</td>
<td>1.271</td>
</tr>
<tr>
<td>Array Sort</td>
<td>1.247</td>
<td>1.356</td>
<td>1.284</td>
</tr>
<tr>
<td>Matrix Sort</td>
<td>1.768</td>
<td>1.983</td>
<td>1.804</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>2.244</td>
<td>28.893</td>
<td>2.282</td>
</tr>
</tbody>
</table>

Impact of Loop Unrolling. To illustrate how loop unrolling can have an impact on memory access scheduling, Listing 7.4 and Listing 7.5 show the difference between memory access scheduling of the innermost loop of a 2D Jacobi kernel without and with a $2 \times 2$ loop unrolling. The unrolled loop provides a larger window of instructions to the scheduler which is able to group co-located accesses across the two unrolled loop iterations. We see that the unrolled version has approximately $1.5N^2$ migrations compared to the original loop for which the memory access scheduler generated code with $3N^2$ migrations. Loop unrolling by a larger factor would allow for grouping more co-located accesses across loop iterations, thereby reducing the number of migrations further. For our experiments, we empirically selected an unroll factor of 2. While a larger unroll factor may expose more opportunity for grouping co-located memory accesses across iterations, this increase in opportunity comes at the cost of increased register pressure resulting in register spills which,
as discussed above, can increase thread migrations rather than reducing them.

Listing 7.4: Jacobi: $3N^2$ migrations

```
1 Iteration 0:
2   // migrate
3   ld a[i][j];
4   ld a[i][j-1];
5   ld a[i][j+1];
6   // migrate
7   ld a[i-1][j];
8   // migrate
9   ld a[i+1][j];
10 Iteration 1:
11   // migrate
12   ld a[i][j+1];
13   ld a[i][j];
14   ld a[i][j+2];
15   // migrate
16   ld a[i-1][j+1];
17   // migrate
18   ld a[i+1][j+1];
```

Listing 7.5: Jacobi with loop unroll: $1.5N^2$ migrations

```
1 Iteration 0:
2   // migrate
3   ld a[i][j];
4   ld a[i][j-1];
5   ld a[i][j+1];
6   ld a[i][j+1];
7   ld a[i][j];
8   ld a[i][j+2];
9   // migrate
10  ld a[i-1][j];
11  ld a[i-1][j+1];
12  // migrate
13  ld a[i+1][j];
14  ld a[i+1][j+1];
```

**Impact of Heuristic scheduler vs. ILP scheduler.** While we expect the ILP scheduler to generate an optimal schedule for memory access instructions within a basic block, there are cases where the heuristic scheduler outperforms the ILP scheduler in practice (e.g., for CSR SPMV and gesummv) because of differences in spill instruction interleaving which are not modeled by either algorithm. This also motivates why we developed two schedulers, and recommend using whichever one delivers better performance for a given application.

**Optimization Opportunity.** As with other compiler optimizations, the impact of memory access scheduling depends on the extent to which the underlying program is bottlenecked by the overhead targeted by the optimization, viz., thread migrations in this case.
To better understand optimization opportunity, we computed the ratio of number of migrations to execution in the original program as a migration intensity metric. Figure 7.8 shows this metric for the explicitly-parallelized benchmarks. The three benchmarks with the largest migration intensity are Jacobi, SPMM and CSR SPMV. Among these, memory access scheduling showed large speedups of $1.87\times$ and $1.71\times$ for SPMM and CSR SPMV respectively. However, the improvement for Jacobi was not as large, though still respectable at $1.13\times$. We believe that this is due to other overheads in Jacobi due to which a large ($1.48\times$) reduction in migrations did not result in a proportional improvement in execution time.

7.4.3 Compilation Time

We evaluated the compile time impact of our pass by measuring the overall compilation time for each of the benchmarks using the base compiler, the ILP scheduler, and the heuristic scheduler. Table 7.3 shows that while the ILP scheduler increases compile time by a factor of two for COO SPMV and a factor of seven for the MRI-Q benchmark, the compile-time for the remaining benchmarks is comparable to that of the baseline compiler (within $1.5\times$). The heuristic scheduler has compile-time within $1.2\times$ of the baseline compiler.
for all benchmarks we studied. For larger blocks or higher unroll factors, we expect the heuristic scheduler to significantly reduce compile time.

7.5 Discussion

**Register Pressure and Phase Ordering:** In our current implementation, our memory access scheduler executes before the register allocation pass. Hence, it does not see migration-inducing instructions generated by register spills. It would be interesting to explore at least two directions to address spill-induced migrations: (1) scheduler heuristics that reduce register pressure in addition to reducing migrations; and (2) reordering spill instructions using a second memory access scheduling pass after register allocation.

**Alias Analysis:** Our work focuses on array accesses but we believe that it can be extended to general memory access instructions. Our findings suggest that the accuracy of co-location information and applicability of memory access scheduling may improve with the help of run-time profile data, user-provided layout information, or multi-version compilation. In addition, a more accurate alias analysis may allow for more aggressive grouping of co-located accesses.

**Global Scheduling:** We implemented memory access scheduling at the basic block-level because that was sufficient for the benchmarks we studied. However cross-block memory access scheduling may help improve performance for other applications.

**Loop Unrolling:** We found that loop unrolling increases the scheduler window to identify more co-located memory accesses and thereby reduces migrations across loop iterations. A future implementation could use a cost-based approach to identify the optimal loop unroll factor for our optimization. Finally, it would be interesting to study the impact of other loop transformations, such as thread coarsening, loop permutation, and loop tiling, on memory access scheduling.
7.6 Future Work

In this chapter, we described Memory Access Scheduling, an instruction scheduling approach to reducing thread migrations on migratory thread architectures. One of the problems we observed with memory access scheduling is its impact on register pressure and spill code. Future work may involve efforts to avoid such migrations, either by reducing spills, reordering spills, rematerializing values or using a helper thread to carry the spill stack. Another improvement may be to use coherent replicated memory as spill stack so that refills never result in a thread migration.

While we have only addressed the issue of migration frequency, a different issue may be the size of thread state. The EMU compiler today performs liveness analysis at migration points and only migrates live registers to reduce the size of the migrating thread state. Instruction scheduling may be used to further minimize live state at migration points by reordering migration inducing instructions after the end of live ranges of registers not used at destination processors.
CHAPTER 8
RELATED WORK

8.1 Redundancy Elimination

Common Subexpression Elimination is a classical compiler optimization that identifies and eliminates redundant expressions. An expression computed in basic block BB is redundant if it is available at the entry of the block and its computation is not preceded by a redefinition of any of its operands within BB [66]. Expression $b \times c$ is redundant in line 6 in the example code from Listing 8.1 because it has already been evaluated along all paths.

Partial Redundancy Elimination identifies and eliminates expressions that are redundant along some but not all paths [66]. As an example, Listing 8.1 shows a program where $b \times c$ is redundant along one path in the program but not along the other.

<table>
<thead>
<tr>
<th>Listing 8.1: Before CSE</th>
<th>Listing 8.2: After CSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if (condition) {</td>
<td>1 if (condition) {</td>
</tr>
<tr>
<td>2 // b*c computed here</td>
<td>2 t = b * c</td>
</tr>
<tr>
<td>3 a = b * c + d</td>
<td>3 a = t + d</td>
</tr>
<tr>
<td>4 } else {</td>
<td>4 } else {</td>
</tr>
<tr>
<td>5 // b*c computed here</td>
<td>5 t = b * c</td>
</tr>
<tr>
<td>6 e = b * c</td>
<td>6 e = t</td>
</tr>
<tr>
<td>7 }</td>
<td>7 }</td>
</tr>
<tr>
<td>8 // b*c recomputed here</td>
<td>8 // remove recomputation</td>
</tr>
<tr>
<td>9 f = b * c</td>
<td>9 f = t</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 8.3: Before PRE</th>
<th>Listing 8.4: After PRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if (condition) {</td>
<td>1 if (condition) {</td>
</tr>
<tr>
<td>2 // b*c computed here</td>
<td>2 t = b * c;</td>
</tr>
<tr>
<td>3 a = b * c + d</td>
<td>3 a = t + d</td>
</tr>
<tr>
<td>4 } else {</td>
<td>4 } else {</td>
</tr>
<tr>
<td>5 e = d</td>
<td>5 t = b * c</td>
</tr>
<tr>
<td>6 }</td>
<td>6 e = d;</td>
</tr>
<tr>
<td>7 // b*c recomputed here</td>
<td>7 }</td>
</tr>
<tr>
<td>8 f = b * c</td>
<td>8 f = t</td>
</tr>
</tbody>
</table>
8.2 Instruction Scheduling

In 1986, Gibbons and Muchnick proposed an $O(N^2)$ per-basic block instruction scheduling algorithm that used a DAG representation to avoid scheduling deadlocks and heuristics to reduce the number of pipeline stalls [67]. Algorithm 7 describes the list scheduling algorithm in detail. The heuristics they propose include

- Stall inducing instructions.
- Number of successors.
- Length of the path from node to leaf.

In Chapter 7, we describe a basic block level instruction scheduling algorithm to minimize thread migrations on migratory thread architectures.

Fisher proposed Trace Scheduling, a global instruction scheduling algorithm that uses block frequencies to construct a trace through the program [68]. It then uses a greedy list scheduling approach to reorder instructions within this trace. They use code replication to preserve code semantics on off-trace paths. This is a profile guided approach ideal for programs with skewed branch probabilities at runtime.

In 1991, Bernstein proposed a global instruction scheduling algorithm that used data and control dependences to build a Program Dependence Graph and reorder instructions across basic blocks [69]. A Program Dependence Graph (PDG) explicitly represents both data dependences and control dependences in the program [21], thereby listing all ordering constraints between instructions. Instructions within basic blocks with the same set of control dependences can be reordered. Similar to the basic block scheduling algorithm described above, the global instruction scheduler performs a topological sort on the PDG and maintains a set of candidate basic blocks from which the best instruction is selected.
Algorithm 7: Instruction Scheduling Algorithm

Input: Basic Block BB
Output: BB after instruction reordering

Function Schedule(BB)

1. DAG = BuildDAG();
2. for each instruction I in the DAG do
   3. if I has no predecessors then
      4. Add I to candidateSet;
   end
3. end
4. best = NULL;
5. while candidateSet is not empty do
   6. for cand in candidateSet do
      7. if bestI is NULL or cand is better than bestI then
         8. best = cand;
      end
   end
7. Emit(best);
8. Remove best from candidateSet;
9. for each successor succ of bestI do
   10. if succ has no predecessors then
        11. Add succ to candidateSet;
   end
12. end
end

8.2.1 Instruction Scheduling and Register Allocation

The compiler is responsible for reducing memory references using register allocation and reducing pipeline stalls using instruction reordering techniques. However, these goals may be at odds with each other and the phase ordering problem between instruction scheduling and register allocation is well known [70].

If we perform register allocation before instruction scheduling, the allocator may assign the same physical register to independent virtual registers, thereby creating new WAW and WAR dependences that did not exist in the original program. This limits the instruction scheduler’s ability to overlap independent instructions to minimize pipeline stalls.

On the other hand, if we perform instruction scheduling before register allocation, the scheduler may overlap too many instructions and increase register pressure, or the number of live registers beyond what the hardware supports. The register allocator must then insert spill code to handle the higher register pressure which increases overall runtime of the program. Further, to hide the pipeline stalls introduced by spill code, we may need an additional instruction scheduling pass.

Modern compilers solve this problem either by accounting for register pressure during instruction scheduling, using multiple scheduling passes or by combining the scheduling and allocation passes.

8.3 SIMT Efficiency on GPUs

8.3.1 Compiler Solutions

Coutinho et al. describe branch fusion, a divergence optimization that uses a variation of sequence alignment, called instruction alignment, to identify common code across diverged paths [71] and merge divergent code using the Split transformation. They do not consider the use of code hoisting or sinking as a means to avoid the introduction of additional branch instructions. Further, their instruction alignment algorithm is restricted to common code
discovered in a specific order.

Code compaction is a code factoring technique used to reduce program size in embedded devices with limited memory space. In their work, Debray and Evans detect common code by identifying common CFG sub-graphs and identical def-use chains within basic blocks. In addition, they employ register renaming to handle non-identical operands in common basic blocks [72].

Han and Abdelrahman propose branch distribution [73], an optimization to reduce branch divergence by factoring out structurally similar code from diverged paths thereby reducing the total number of dynamic instructions executed in the divergent region.

In Chapter 3 we extend branch distribution using a cost model to minimize transformation overhead and adds support for loops, functions and nested conditionals, making it a global transformation.

Von Hanxleden and Kennedy introduce loop flattening [74]. Their work provided early proof that executing SIMD threads across iterations could improve efficiency. Similarly, Han and Abdelrahman propose Loop Merge for GPUs [75], which generically merges arbitrary nested loops via a loop transformation. They show that their optimization, which developers can direct via a special pragma, can significantly improve SIMT utilization and performance. However, the transformation can lead to code bloat because it duplicates the epilogue and prologue of the inner loop.

Han and Abdelrahman also present Iteration Delay in separate work [32]. As with Loop Merge, their compiler applies a transformation that introduces a vote to determine which path of an if-then-else to execute on each iteration, which is exactly what performance-conscious programmers do for such code patterns [76], as in the case of the Optix ray tracing engine [31].

Diamos et al. propose the concept of a thread frontier, and show the benefits of optimistically reconverging execution before immediate post-dominator blocks [77]. Their work, which is primarily concerned with irregular control flow, relies on a mechanism for
the compiler to dictate the order in which threads traverse basic blocks during execution. By programmatically allowing the insertion and deletion of threads from barriers, our work allows the architecture to use independent traversal mechanisms, and also necessitates an alternative compiler algorithm to manage barriers. Fung and Aamodt describe a mechanism for defining a likely convergence point to allow divergent threads to reconverge earlier than the immediate post-dominator [78].

In Chapter 4, we describe a compiler framework that speculatively reconverges threads to improve SIMD efficiency along expensive divergent paths for a broad range of control flow patterns.

8.3.2 Hardware Solutions

Dynamic Warp Formation dynamically regroups diverged threads across warps executing the same instruction, thereby increasing SIMT efficiency [79]. This regrouping relies on the warp scheduler and needs all participating warps to arrive at the divergent branch at the same time. If some threads fall behind, they may be unable to regroup resulting in lower SIMT efficiency. An improvement over DWF is thread block compaction [78], where divergence is handled at a block level rather than at a warp level. At a divergent branch, warps within a block synchronize and regroup into new convergent warps until the next branch or the reconvergence point at which warps synchronize once more. At the reconvergence point, threads are regrouped into their original warps. This approach retains the increased SIMT efficiency of DWF which avoiding its pitfalls.

Variable warp sizing [80] reduces the penalty of serialization due to divergence and helps interleave execution of instructions across subwarps, but the technique requires complex hardware to handle warp grouping and it can exacerbate divergence in cases where a subwarp’s threads are randomly scattered across lanes (and therefore base warps are still likely to contain threads from more than one subwarp). Finally, Keckler et al. [81] explore temporal SIMT, where convergent threads execute in SIMD fashion but diverged threads
may continue executing in parallel in MIMD fashion.

### 8.4 Thread Scheduling on GPUs

Warp scheduling heuristics aim at improving data locality [82] [83], reducing busy wait code execution [84], reducing barrier induced stalls [85], reducing power consumption [86] in addition to reducing stalls by interleaving warp execution [40].

Dynamic Warp Subdivision splits up a single warp into multiple scheduling slots on a divergent branch and allows divergent threads to interleave their execution thereby improving latency hiding within divergent paths. In Chapter 5, we extend dynamic warp subdivision for modern GPU architectures using an automatic approach that retains the latency hiding benefits but allows for unlimited warp subdivision.

### 8.5 Register File Sharing

GPU register file virtualization is a technique that uses register file sharing between warps to reduce the register file size and power utilization[87]. To avoid deadlock, they allow only one warp to make progress and throttle all remaining warps. If there are insufficient registers for one warp, they perform hardware register spilling. They use hardware register renaming to map architected registers to the smaller set of physical registers. Their technique does not work with thread divergence.

Voitsechov et al. propose eager register release to increase occupancy in register-limited applications[88]. Instead of waiting for the entire thread block to finish execution before releasing all registers, warps eagerly deallocate dead registers, i.e., registers that have no future uses. Since they only release registers and never acquire them, there is no potential for deadlock. While this technique works well for applications that ..., it does not work for applications that ... or applications like raytracing which have thread divergence.

RegMutex is a software-hardware technique that allows warps to time share a subset of registers[89]. Khorasani et al. propose a division of the register file into a base register
set and an extended register set which is shared across warps. The compiler is responsible for identifying acquire and release points using liveness information. Since RegMutex supports both acquire and release, there is a potential for deadlock. To avoid deadlock, they ensure that the extended set always has sufficient registers to allow at least one warp to make progress. Finally, to handle register renaming, they perform a compiler pass that inserts copy instructions. RegMutex only performs acquire and release in convergent paths and cannot handle Raytracing applications.

8.6 Thread Migrations

8.6.1 Reducing Thread Migrations on EMU

Belviranli et al. proposed algorithm-level optimizations using improved thread creation strategies and better data distribution at the program level to reduce thread migrations on the Emu architecture [90]. Rolinger et al. addressed the problem of thread migrations and load balancing in SpMV by optimizing data layouts to allocate data blocks across nodelets, so that memory access latency is reduced [91] [92]. Hein et al. studied the impact of replication, remote writes, and data layout transformations on reducing thread migrations [93]. Page proposed a prefiltering mechanism to avoid redundant migrations [56]. These approaches rely on changing the algorithm or data layout.

Another data layout optimization to reduce thread migrations in bitonic sort was proposed by Velusamy et al. [94]. They use a dynamic data remapping strategy instead of a static data layout to avoid remote memory accesses in their work. However, this run-time remapping operation has an associated overhead. Chatarasi et al. used traditional compiler loop transformations to increase memory access locality, such as loop fusion, to reduce thread migrations in graph applications on the Emu architecture [95].

In Chapter 7, we describe an instruction scheduling technique that automatically reorders co-located memory accesses and can reduce migrations across applications and across multiple access patterns on the same data structures without any run-time overhead.
and without changes to the algorithm.

8.6.2 Optimizing Migrations on Network Processors

Migration cost becomes a bottleneck in mobile computing in the case of application migration. To minimize the latency overhead of migrations, Zhang et al implemented a compiler and runtime solution that determines which parts of the application to migrate and when using reachability and liveness analysis[96]. Further, Zhuang et al used compiler techniques to minimize the frequency and size of migrations of mobile agents on a distributed system[97]. Similar to Memory Access Scheduling on EMU, their work adds edges to the program dependence graph to minimize migrations.
CHAPTER 9
CONCLUSIONS

We have described scheduling and resource allocation techniques that help improve performance on thread-parallel architectures such as the GPU and the EMU. These optimizations include Common Subexpression Convergence and Speculative Reconvergence, aimed at increasing SIMT efficiency in GPU programs that have high thread divergence; Subwarp Interleaving and Register File Sharing, aimed at better latency hiding and increased occupancy in GPU programs with high divergence; and Memory Access Scheduling, aimed at reducing the overhead of redundant thread migrations in EMU programs. We have shown that compile-time approaches, in conjunction with architecture changes, can help achieve better utilization in otherwise irregular programs running on thread-parallel architectures.

Our scheduling techniques such as Subwarp Interleaving and Speculative Reconvergence which reduce and leverage thread divergence may enable future GPU programmers to write divergent algorithms which perform better or are more intuitive than techniques that try to avoid thread divergence due to the cost of serialization.

Dynamic resource allocation techniques that use slow-path compilation to ensure forward progress may be extended to allocate other resources such as GPU shared memory. Further, our proposed time-sharing techniques may be applied to large scale data-centers where the goal is to use minimal resources to service multiple applications that have different resource and latency requirements.

The EMU architecture presents new challenges to compiler developers. With changes in memory hierarchy, traditional compiler optimizations must be reconsidered to generate the best performing code for migratory thread architectures. Memory access scheduling is a first step towards achieving this goal.
REFERENCES


learning algorithm performance on the lucata computer,” in 2020 IEEE High Perfor-
mance Extreme Computing Conference (HPEC), 2020.

The case of data shuffling,” in CIDR, 2013.


to remote memory accesses in multiprocessors,” in Proceedings of the 1996 Confer-
ence on Parallel Architectures and Compilation Technique, 1996, pp. 2–11.

mization for simt processors,” in Languages and Compilers for Parallel Computing,
978-3-030-72789-5.


sium on Compiler Optimization, Association for Computing Machinery, 1970.

control regions in linear time,” in Proceedings of the ACM SIGPLAN 1994 Confer-


and optimizations,” in 2011 International Conference on Parallel Architectures and
Compilation Techniques, 2011.

[23] R. Allen and K. Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-

and S. Brain, “A Monte Carlo Dose Calculation Tool for Radiotherapy Treatment


[46] Archviz interior rendering, Epic Games Inc.


[94] M. Endler, Bitonic Sort on CUDA.


Sana Damani was born in Mumbai, India on May 19, 1990. She obtained her Bachelor’s degree in Computer Engineering from the University of Pune in 2012. She then worked as an engineer in the industry for five years before entering the Georgia Institute of Technology to work on compiler research under the tutelage of Dr. Vivek Sarkar. Her research focuses on the use of machine-dependent optimizations, including scheduling and resource allocation, to improve the performance of applications running on parallel architectures. She is also a recipient of the 2021 Nvidia graduate fellowship award. Sana received her PhD from the Georgia Institute of Technology in 2022.