ACHIEVING PERFORMANCE PORTABILITY ACROSS PARALLEL ACCELERATOR ARCHITECTURES

BY

STEPHEN M. KOFSKY

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Associate Professor Steven S. Lumetta
Parallel programming requires a significant amount of developer effort, and creating optimized parallel code is even more time-consuming. In the end, tuned parallel codes typically only perform well for a single architecture, or even microarchitecture. This thesis focuses on SPMD code written in CUDA, noting that programs must obey a number of constraints to achieve high performance on an NVIDIA GPU. Under such constraints, source-level optimizations can improve the performance of CUDA code on Rigel, a MIMD accelerator architecture currently under development. Source-level optimizations can produce code for Rigel that runs significantly faster than naïve translations. In some cases, benchmarks run nearly four times faster, rivaling the performance of hand-optimized code. Unlike a GPU, Rigel allows for a flexible execution model, making it difficult to extract performance information that can be leveraged to get good performance on other architectures. CUDA code written for Rigel performs poorly when executed on a GPU, and is significantly slower than optimized CUDA code tuned for GPUs.
# TABLE OF CONTENTS

LIST OF ABBREVIATIONS ........................................ iv

CHAPTER 1 INTRODUCTION ........................................ 1
  1.1 Motivation ............................................. 1
  1.2 Thesis Organization ................................... 2
  1.3 Rigel .................................................. 3

CHAPTER 2 BACKGROUND .......................................... 5
  2.1 CUDA .................................................. 5
  2.2 MCUDA ................................................ 7
  2.3 Rigel Task Model ....................................... 8
  2.4 Previous Work ......................................... 11

CHAPTER 3 RCUDA IMPLEMENTATION ......................... 16
  3.1 Source Code Transformations .......................... 16
  3.2 Runtime Library ....................................... 20

CHAPTER 4 OPTIMIZATIONS AND AUTOMATION ........... 29
  4.1 Kernel Transformations ................................ 29
  4.2 Runtime Optimizations ................................ 37
  4.3 Optimization Ordering ................................ 44
  4.4 Source Translation Automation ...................... 44

CHAPTER 5 EVALUATION ........................................... 46
  5.1 Simulation Infrastructure Methodology ............. 46
  5.2 Comparing RCUDA and RTM Performance ............. 46
  5.3 RCUDA Performance .................................... 51
  5.4 DMM Case Study of Performance Portability ....... 60

CHAPTER 6 CONCLUSION AND FUTURE WORK ............ 67

CHAPTER 7 REFERENCES .......................................... 69

APPENDIX A RCUDA CODE LISTING ......................... 72

APPENDIX B CUDA KERNEL CODE LISTING ............ 77
  B.1 DMM (CUDA-Rigel) .................................... 77
  B.2 SAXPY ............................................. 78
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CMP</td>
<td>Chip Multiprocessor</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating-point Unit</td>
</tr>
<tr>
<td>FLOPS</td>
<td>Floating-Point Operations Per Second</td>
</tr>
<tr>
<td>FFTW</td>
<td>Fastest Fourier Transform in the West</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>MCUDA</td>
<td>Multi-Core Compute Unified Device Architecture</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction Multiple Data</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PTX</td>
<td>Parallel Thread Execution</td>
</tr>
<tr>
<td>RCUDA</td>
<td>Rigel Compute Unified Device Architecture</td>
</tr>
<tr>
<td>RTM</td>
<td>Rigel Task Model</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multiprocessor</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multiprocessor</td>
</tr>
<tr>
<td>SPIRAL</td>
<td>Signal Processing Implementation Research for Adaptable Libraries</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single Program Multiple Data</td>
</tr>
<tr>
<td>ZPL</td>
<td>Z-level Programming Language</td>
</tr>
</tbody>
</table>
Parallel programming requires a significant amount of developer effort, and creating optimized parallel code is even more difficult and time-consuming. Programmers are forced to look beyond the speed of serial execution and must analyze and understand how different parallel components work together at a system level. In the end, optimized parallel codes are often tuned for a specific architecture or microarchitecture. Tuned parallel code typically requires significant modifications in order to perform well on other platforms.

This work focuses on SPMD code written in the CUDA [1] programming language. Tuned CUDA programs have common characteristics in terms of memory access patterns, specialized memory usage and the amount of parallelization. Optimizations that leverage these characteristics can be developed to maintain performance on another system, a characteristic known as performance portability. Code that has not been tuned may not exhibit the same characteristics, but these codes are of little concern since they have low performance to begin with. Only correct execution is required.

This thesis introduces RCUDA, a framework that allows CUDA code to be run on a MIMD accelerator. In addition to translating code, optimizations at the source-level are used to allow for performance portability. While most of the optimizations are currently applied by hand, the end goal is to automate the tuning process in the future.

1.1 Motivation

A sequential program tuned for one architecture typically performs reasonably well on most other sequential architectures. However, developing optimized code for multiple parallel systems is challenging for the reasons given above. It is advantageous to write a program once, tune it for a specific architecture and still get good performance on another system.
In addition to performance portability, this work allows different programming models to be compared for a MIMD architecture, and allows direct comparisons to be made to other platforms running the same code.

While the goal of this work is to create performance portable code, the approach is significantly different from previous research into performance portability. Instead of creating a new programming model so that a programmer must alter their development practices, RCUDA focuses on translating real world CUDA programs tuned for execution on a GPU, as shown in Figure 1.1. The source-to-source translation method gives RCUDA two key advantages: it works with code that already exists, and programmers do not need to specifically target the RCUDA framework.

The main contributions of this work include: (1) A source translation process that retains performance between parallel systems, (2) Source-level optimizations for translated CUDA code, (3) A high-performance software runtime library for executing SPMD codes on a MIMD architecture.

1.2 Thesis Organization

This chapter introduces performance portability, discusses why it is important and describes Rigel, a massively parallel accelerator architecture. Chapter 2 contains background information, including an overview of the CUDA programming model and the MCUDA framework along with previous work involving performance portability. Chapter 3 details the design and implementation of RCUDA, a framework that enables CUDA code to execute on Rigel. Chapter 4 describes the optimizations that can be used to improve
Figure 1.2: Diagram of the Rigel processor.

Table 1.1: Parameters for the baseline architecture.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>1024</td>
<td>–</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>192</td>
<td>GB/s</td>
</tr>
<tr>
<td>DRAM Channels</td>
<td>8</td>
<td>–</td>
</tr>
<tr>
<td>L1I Size</td>
<td>8</td>
<td>kB</td>
</tr>
<tr>
<td>L1D Size</td>
<td>2</td>
<td>kB</td>
</tr>
<tr>
<td>L2 Cluster Cache (Total)</td>
<td>8</td>
<td>MB</td>
</tr>
<tr>
<td>L3 Global Cache (Total)</td>
<td>4</td>
<td>MB</td>
</tr>
</tbody>
</table>

the performance of CUDA code on the Rigel architecture and discusses how the optimizations can be automated. Finally, Chapter 5 covers the results of the performance evaluations, and Chapter 6 contains the conclusion.

1.3 Rigel

Rigel [2], [3] is a 1024-core MIMD accelerator designed for task and data-parallel workloads. While Rigel is designed to be highly scalable, it uses a conventional programming model. Rigel is representative of future more generic accelerator architectures and shares similar characteristics with NVIDIA’s Fermi [4] architecture.
1.3.1 Architecture

The Rigel architecture is a hierarchy of cores, clusters and tiles, as shown in Figure 1.2. Table 1.1 provides additional architectural parameters. Each Rigel core is a dual-issue, in-order core that implements an ISA based on the MIPS ISA. Each core contains a single-precision FPU, a 2 kB L1 data cache and an 8 kB L1 instruction cache. Integer division is not supported in hardware. Rigel cores are grouped into clusters. A Rigel cluster contains eight cores with a shared L2 cache. The L2 cache is coherent so that data can be shared among cores in the same cluster. Clusters are grouped into tiles, with each tile containing 16 clusters and connected to 32 global cache banks.

1.3.2 Programming Model

Applications are developed for Rigel using a task-based API, called Rigel Task Model (RTM), where a task is mapped to one Rigel core. Tasks can vary in length and do not execute in lock-step. Task generation and distribution is dynamic and handled by software, the hardware only implements global and cluster-level atomic operations. Using a software approach allows a flexible execution model, which we leverage to map CUDA to the architecture.

1.3.3 Memory Model

Rigel is designed for a bulk synchronous execution model where there is little data sharing between tasks during execution. Rigel uses a hybrid memory model that utilizes both hardware coherence and software coherence as described in [5]. Cluster caches are coherent within the cluster, but not across different clusters. Values computed at the cluster-level need to be written back in order to be globally visible. At the global-level, Rigel has what can be thought of as opt-in cache coherence between different clusters. A programmer can explicitly use global memory loads and stores which go to a global cache visible to all clusters. Data in the global cache is coherent; however, global memory operations can be quite expensive and are designed to be used sparingly.
CHAPTER 2

BACKGROUND

This chapter describes the two different programming models that this work is derived from and provides an overview of previous research related to performance portability.

2.1 CUDA

CUDA is a programming model designed to execute fine-grained parallel code. A CUDA application uses host code to initialize and launch kernels on an accelerator device. As shown in Figure 2.1, the work size is defined by a one- or two-dimensional grid composed of thread blocks. Thread blocks themselves are one-, two- or three-dimensional sets of threads.

Thread blocks can be executed in any order and are not guaranteed to run concurrently, limiting their interaction. However, CUDA does guarantee
that threads within a thread block are live concurrently, allowing the threads
to efficiently share data. CUDA uses textually-aligned barriers, such as those
of the Titanium language [6]. For example, it is illegal in CUDA to invoke a
barrier in both paths of an if-else construct when CUDA threads could take
different branches. Even though all threads reach a barrier, the barriers are
considered to be separate barriers, each requiring all threads to reach it.

CUDA differentiates between global, shared and local memory spaces.
Global memory persists across multiple kernel executions and can be accessed
by threads in any thread block. Shared memory is replicated for every thread
block and is accessible only to threads within the same thread block. Shared
memory may be file scoped or kernel scoped. If kernel scoped, the shared
memory is destroyed after the kernel executes. Finally, each thread has its
own local memory to store temporary variables. Additionally, the CUDA
runtime implements virtual memory addressing which allows the host and
device to share the same address space [7].

2.1.1 CUDA on a GPU

When CUDA code is run on a GPU, the host code is executed on the CPU
and kernels are executed on the GPU. The CUDA runtime implements vir-
tual memory addressing which allows the host and device to share the same
address space [7]. Even though there is a single virtual address space, the
CPU and GPU have separate memory physical spaces. Input to the kernels
needs to be copied to the device memory before kernel execution, and kernel
output is copied from device to host memory. In addition to handling the
input and output data, the host code is responsible for setting the grid and
thread block sizes.

During kernel execution, each thread block is mapped to a single Streaming
Multiprocessor (SM), however multiple thread blocks can be placed on the
same SM. Threads are executed concurrently in groups known as warps. The
size of a warp is device dependent. Current GPUs use a warp size of 32 [8].
Only a single warp is executed at a time, and a scheduler inside the SM
chooses which warp to execute. Each thread is allocated space in the SM’s
register file for local thread values. In addition, the register file also contains
shared memory for the thread block.
2.1.2 Optimized CUDA Code

CUDA is a restrictive programming model, requiring the programmer to express the application as a set of fine-grained tasks. Programmers are further limited when creating optimized CUDA programs. In order to effectively utilize all of the execution units a GPU has to offer, optimized CUDA programs use a large number of thread blocks containing many threads. The kernels themselves typically maximize global memory throughput and utilize shared memory [9]. Memory accesses are coalesced so that multiple accesses can be serviced at the same time. Shared memory is used to store variables that are accessed multiple times or shared across threads within the thread block. Since GPUs execute in a SIMD fashion, programmers take care to limit control divergence in warps.

2.2 MCUDA


2.2.1 Execution Model

With MCUDA, CUDA threads within a thread block are combined and serialized with a loop, creating code that iterates over the individual CUDA thread indices, as shown in Figure 2.2. MCUDA increases the work granularity by making thread blocks the smallest unit of work. During execution, thread blocks are mapped to separate OS threads so that they can be executed in parallel.

2.2.2 Limitations

Currently MCUDA does not support all features of the CUDA language. Missing features include texture support, C++ style template code and
atomic operations. In order to use MCUDA, programmers should replace any usage of textures with standard arrays.

2.3 Rigel Task Model

The Rigel Task Model (RTM) is a task-based programming API for the Rigel architecture that is designed for irregular, coarse-grained tasks.

2.3.1 Programmer’s View

RTM abstracts away the details of distributing tasks across the massively parallel processor and provides the programmer with an interface to enqueue and dequeue tasks. Figure 2.3 shows an example program using RTM. The programmer defines a task using a task descriptor structure (TQ_TaskDesc), which is simply a set of four 32-bit integers. The v1 and v2 fields of the task descriptor can be set to any value, while v3 and v4 are reserved for the runtime. Tasks themselves are independent of each other and are executed on a single Rigel core. RTM makes no guarantee of the execution order, nor on which core or cluster a task will execute.

Tasks can be enqueued individually by using the TQ_EnqueueTask() function while multiple tasks can be enqueued by calling the parallel enqueue function, TQParallelEnqueueLoop(). The TQParallelEnqueueLoop() function takes three parameters: the total number of iterations, the number of iterations per task and the task descriptor template. When the parallel enqueue function is called, one core from each cluster enqueues a portion of...
Figure 2.3: Example code using RTM. The variables X and Y can take any 32-bit integer value.

the tasks and sets the v3 and v4 fields of the task descriptor structure to the start and end iteration respectively. The code in Figure 2.3 enqueues a total of 128 tasks since there are 2048 iterations with 16 iterations per task.

The TQ_Dequeue() function attempts to dequeue a single task and either returns TQ_RET_SUCCESS, to indicate that the task descriptor was successfully dequeued, or TQ_RET_SYNC if there are no more tasks available. Internally, the TQ_Dequeue() function implements a barrier, and returns TQ_RET_SYNC only when all cores are finished executing tasks.

2.3.2 Implementation

RTM is implemented as a software library written in C and assembly. Codes using RTM are written in C and are linked to the software library. Internally, RTM uses hierarchal task queues to provide work distribution functionality while still retaining high performance. There is one global task queue, which is shared among all the clusters in addition to cluster-level task queues for each cluster.
Global Task Queue

The global task queue contains tasks that are visible to all clusters on the chip and enables tasks enqueued from one cluster to be executed by a core in another cluster. To maintain coherence between clusters, global atomic operations are used when adding or removing tasks from the global task queue. Since atomic operations are expensive, groups of up to eight tasks are enqueued or dequeued at a time in order to reduce the number of global task queue operations. Every cluster has a global task queue lock so that only one core from each cluster can access the global task queue at a time. Limiting access to the global task queue helps increase performance by reducing costly global memory operations.

Cluster Task Queues

Each cluster has a dedicated cluster-level task queue implemented as a linked list of task descriptors. Cluster task queues provide a fast access queue, but contain tasks that are only visible to cores inside the cluster. Tasks are inserted into the cluster task queue during enqueue or when a task group is dequeued from the global task queue. In the current implementation of RTM, tasks are never moved from the cluster task queue to the global task queue.

Task Enqueue

Tasks can be enqueued by a single Rigel core, or in parallel by one core in each cluster. When a task is enqueued, a unique task descriptor is generated and written to global memory. A core inserts a task directly into the local cluster task queue if it is operating in data parallel mode or if the number of tasks in the cluster task queue is fewer than eight. Otherwise, the core adds the tasks to the global task queue.

Task Dequeue

Tasks are only dequeued from the local cluster task queue. When dequeue is called, the core attempts to acquire the cluster queue lock. Once the lock
Table 2.1: Characteristics of CUDA and RTM programming models.

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>RTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Ordering</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>Work Load Granularity</td>
<td>Fine</td>
<td>Coarse</td>
</tr>
<tr>
<td>Index/Parameters</td>
<td>Uniform</td>
<td>Customizable</td>
</tr>
<tr>
<td>Task Grouping</td>
<td>Customizable</td>
<td>Fixed</td>
</tr>
<tr>
<td>Task Length</td>
<td>Uniform</td>
<td>Uniform or Irregular</td>
</tr>
<tr>
<td>Per-task Memory</td>
<td>Shared Memory</td>
<td>None</td>
</tr>
<tr>
<td>Global Data Structure</td>
<td>Grid</td>
<td>Global Task Queue</td>
</tr>
<tr>
<td>Cluster Data Structure</td>
<td>Thread Block</td>
<td>Cluster Task Queue</td>
</tr>
</tbody>
</table>

is acquired, the core removes a task descriptor entry from the cluster queue if the queue is not empty. If there are no available tasks in the cluster task queue, the core attempts to acquire a lock to the global task queue allowing it to fetch a task group. If the global task queue is empty, the core waits for new task groups to be added to the global task queue by cores from other clusters. Once all cores are waiting for new tasks, no additional tasks can be enqueued and the dequeue function returns TQ_RET_SYNC.

2.3.3 Comparison with CUDA

RTM has several differences with CUDA, as shown in Table 2.1. First of all, RTM is designed for tasks that are much longer than CUDA kernels. With RTM, tasks can be enqueued at any time, unlike CUDA which only allows new work to be added during a kernel launch. RTM tasks can execute any code, and task descriptors can contain whatever values the developer specifies. In CUDA, all CUDA threads use the same code and uniform index structure that cannot be changed. Unlike CUDA threads, RTM tasks are fetched independently and RTM does not guarantee that tasks from the same task group execute concurrently.

2.4 Previous Work

Performance portability has been actively researched for serial as well as parallel systems. Solutions range from developing programs in a different programming language that is able target multiple architectures, auto-tuning, or some combination of the two.
2.4.1 GPUocelot

GPUocelot [12] is a framework that allows CUDA applications to execute on a standard x86 multi-core processor without recompilation. The NVIDIA compiler compiles CUDA programs to PTX [13], a low-level ISA that can represent the semantics of a CUDA program. GPUocelot performs analysis on the PTX code and then converts the code to LLVM [14] IR. In addition to translating the code, threads are fused together to execute in a loop much like MCUDA. Also CUDA constructs such as the use of special purpose registers for the thread ID are replaced by a lookup to internal data structures.

2.4.2 ZPL

ZPL [15] is a parallel programming language designed to provide performance portable [16] code across various MIMD architectures. ZPL provides users with a global view of the program, making it unnecessary to write code for a specific processor. The ZPL compiler does the work of converting the ZPL code to C, which in turn can be compiled to a binary which runs on the device. A program written in ZPL can be significantly shorter than a program written in a lower level language such as C or Fortran. Snyder [15] found ZPL code to be performance portable across different MIMD systems including the Cray T3E and IBM SP-2.

2.4.3 Auto-Tuning

Auto-tuning has been used to achieve high performance on a single architecture and across architectures. Performance tuning of applications is a difficult process because of the number of different ways a kernel can be optimized. Ryoo et al. [17] have demonstrated how the optimization space can be pruned automatically and still generate high-performance code.

FFTW

FFTW [18] is a self-optimizing library to compute discrete Fourier transforms (DFTs). Computing DFTs is performance critical for many applications, and it is essential to choose a fast algorithm. Instead of trying to reduce floating
point operations, an optimization that is not effective at speeding up code on multiple systems, FFTW uses an auto-tuning approach to find the fastest DFT algorithm during runtime. FFTW generates a fast Fourier transform based on the Cooley-Tukey algorithm written in C. The code is tuned automatically by the library during runtime. The Executor contains composable blocks of code known as codelets. Codelets are small pieces of code that compute a portion of the transform and are used to implement special cases. The Codelet generator creates the codelets at compile time. The Planner chooses codelets and constructs a plan data structure that contains different codelets collectively called the plan. The plan is built using a dynamic programming with a divide and conquer approach, not an exhaustive search. Plans are chosen based on experimental results and are optimized to minimize runtime. The processor, memory system and compiler heavily influence the plan that is chosen. FFTW hides the implementation from the developer. A developer only needs to create a plan data structure for an input size and then can execute the plan to actually compute the DFT. Plans can be reused during runtime in order to reduce the overhead of FFTW. FFTW generates performance portable FFT (fast Fourier transform) algorithms that achieve higher levels of performance than other publicly available libraries, and even has comparable performance to vendor-optimized libraries tuned to a specific architecture.

ATLAS

Automatically Tuned Linear Algebra Software (ATLAS) [19], [20] is a specialized auto-tuning library for Basic Linear Algebra Subprograms (BLAS). The motivation behind ATLAS is to allow software to keep pace with rapidly improving hardware by automating the optimization process on new architectures. ATLAS gathers empirical data at install time and uses the results to generate high-performance BLAS codes.

BLAS can be broken down into three levels of increasing complexity. Level one contains vector-vector operations that are hard to optimize further than a compiler. Level two contains matrix-vector operations which have more room for optimization, but are still limited. Lastly, level three contains matrix-matrix operations which can be optimized by memory optimization techniques. ATLAS provides hand-tuned Level one BLAS due to the limited
optimization potential. For level two and three BLAS codes, ATLAS generates efficient compute kernels which are used be used to derive the rest of the BLAS. A matrix-vector multiplication kernel and a rank-1 update kernel generate all level two BLAS routines, while a matrix-matrix kernel is used to implement all of the level three BLAS codes.

During installation, ATLAS runs a variety of tests to determine the L1 cache size, number of registers, pipeline depth and the supported types of floating point operations for the target architecture. The results are then fed into a code generator which generates C code with optimal memory fetching, loop unrolling and floating point operation ordering. All of the BLAS code is generated at install time, and no further runtime auto-tuning is required.

The authors found that the auto-generated BLAS code from ATLAS performed similarly to implementations tuned to a specific architecture. ATLAS has been used to generate a complete set of BLAS and is widely used. While ATLAS performs well, it does have some drawbacks. For example, the optimizations given in the paper improve only improve single thread performance and do not take advantage of multiple cores, if available. Further, ATLAS depends on a C compiler to perform the low-level optimizations of the code generated during the auto-tuning process.

2.4.4 SPIRAL

SPIRAL [21], is an auto-tuning framework that optimizes digital signal processing (DSP) codes. Code tends to outlast the hardware system it was tuned for and newer systems require hardware to be optimized all over again. SPIRAL performs auto-tuning at install time, not compile time as is common with other auto-tuning work. Also, the code that is auto-tuned must be written in SPL (Signal Processing Language). SPL is a high-level language that is essentially a mathematical equation. The advantage is that the SPL code can be broken down into simpler transforms. SPIRAL itself has three components. The first is the Formula Generator, which generates the possible algorithms for auto-tuning. Next, the Formula Translator translates the algorithm into another language that can be compiled. Finally, the Search Engine searches for the best algorithm to use either by using an exhaustive, dynamic or random search. Based on the results of the searches, new can-
didate algorithms are generated. SPIRAL can successfully find much faster DSP transforms by using auto-tuning. Also performance is similar to FFTW, which only supports fast Fourier transforms.
CHAPTER 3

RCUDA IMPLEMENTATION

RCUDA is a framework that allows CUDA code to be executed on Rigel. RCUDA consists of a CUDA-to-C translation engine and a software runtime library that implements CUDA built-in functions as well as load balancing. Appendix A contains the complete RCUDA runtime source code. Figure 3.1 shows the translation process performed on CUDA code by RCUDA.

3.1 Source Code Transformations

This section details the CUDA-to-C conversion process which enables CUDA code to run on Rigel. The source code transformations for kernel code are automatic, while host code must be hand-edited.

![Figure 3.1: Translation steps performed on CUDA code by the RCUDA framework. CUDA code is first translated to C code, compiled and finally linked to the RCUDA runtime.](image)
1: __global__ void transpose(float *odata, float *idata, int width, int height)  
2: {  
3: __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];  
4:  
5: // read the matrix tile into shared memory  
6: unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;  
7: unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;  
8: if((xIndex < width) && (yIndex < height))  
9: {  
10: unsigned int index_in = yIndex * width + xIndex;  
11: block[threadIdx.y][threadIdx.x] = idata[index_in];  
12: }  
13:  
14: __syncthreads();  
15:  
16: // write the transposed matrix tile to global memory  
17: xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;  
18: yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;  
19: if((xIndex < height) && (yIndex < width))  
20: {  
21: unsigned int index_out = yIndex * height + xIndex;  
22: odata[index_out] = block[threadIdx.x][threadIdx.y];  
23: }  
24: }

Figure 3.2: Transpose kernel from the CUDA SDK that uses shared memory.

3.1.1 Kernel Code

CUDA kernel source must be translated because the RCUDA runtime maps individual thread blocks to a single Rigel cluster during execution. Within a cluster, threads can be distributed dynamically or statically to individual cores, and threads execute in a serial loop between synchronization points.

Shared variables are stored in a per-cluster data structure so that each core can access the shared data through the cluster cache. Most local variables are stored in a cluster-level data structure, allowing threads to migrate to another core within the cluster after a synchronization point. Local thread variables that are produced and consumed before a synchronization point do not have to be replicated in a cluster-level data structure since they are not needed when the thread moves to another core. Figure 3.2 shows a transpose kernel from the NVIDIA GPU Computing SDK [22], and Figure 3.3 shows the code after translation.

Data Structures

When a kernel is translated, three data structures are generated as shown in Figure 3.3. A local_state data structure contains all of the local thread vari-
```c
struct transpose_params {
    float * odata, * idata;
    int width, height;
};

struct local_state {
};

struct block_state {
    local_state[512];
    float block[BLOCK_DIM][BLOCK_DIM+1];
};

transpose_params params;
block_state transpose_block_state[MAX_CLUSTERS];

void transpose(dim3 blockIdx, dim3 blockDim, dim3 gridDim) {
    block_state * bs = (&transpose_block_state[getSMID()]);
    dim3 threadIdx;
    int __threadIndex;
    float * odata = params.odata;
    float * idata = params.idata;
    int width = params.width;
    int height = params.height;
    int xIndex;
    int yIndex;
    while((__threadIndex = atomic_get_next_tid_2d(&threadIdx, blockDim.x)) >= 0) {
        xIndex=((blockIdx.x*BLOCK_DIM)+threadIdx.x);
        yIndex=((blockIdx.y*BLOCK_DIM)+threadIdx.y);
        if (xIndex<width && yIndex<height) {
            unsigned int index_in = yIndex*width+xIndex;
            bs->block[threadIdx.y][threadIdx.x]=idata[index_in];
        }
    }
    __rigel_sync_threads();
    while((__threadIndex = atomic_get_next_tid_2d(&threadIdx, blockDim.x)) >= 0) {
        xIndex=((blockIdx.y*BLOCK_DIM)+threadIdx.x);
        yIndex=((blockIdx.x*BLOCK_DIM)+threadIdx.y);
        if (xIndex<height && yIndex<width) {
            unsigned int index_out = yIndex*height+xIndex;
            odata[index_out]=bs->block[threadIdx.x][threadIdx.y];
        }
    }
}
```

Figure 3.3: Transpose kernel after the RCUDA source-to-source translation process.
ables (not shared memory) with values generated before a `__syncthreads()` call but consumed after the barrier. In the case of the transpose kernel, the values assigned to `xIndex` and `yIndex` before the `__syncthreads()` call are not used again after the synchronization point, so the `local_state` data structure is empty. The second data structure, `block_state`, is replicated for each cluster and contains a `local_state` structure for each CUDA thread in the block as well as any shared memory. Since the thread block dimensions are chosen only when the kernel executes, the local thread variables are stored in an array with a length of 512-equal to the CUDA defined maximum number of threads in a block. The `transpose_params` structure contains all of the parameters to the original kernel function. There is only one instance of the `transpose_params` structure, and it is initialized before the kernel function executes.

Code Transformation

Every translated kernel function has the same three input parameters: `blockIdx`, the block index; `blockDim`, the thread block dimensions; and `gridDim`, the size of the grid. Before the actual kernel code is executed, the pointer to the cluster’s `block_state` structure is set, as shown on line 18 in Figure 3.3. There is one `block_state` structure defined for each cluster and the `getSMID()` function simply returns the Rigel cluster number. Next, the input parameters are copied locally to stack variables, which occurs on lines 23 and 24 in the translated transpose kernel. Lines 27 to 36 correspond to lines 6 through 12 or the original CUDA code. Differences in spacing and use of parentheses arise from the source-to-source transformation. Thread indices are generated by the RCUDA runtime inside the `atomic_get_next_tid_2d()` function. The `__rigel_sync_threads()` function call replaces the CUDA `__syncthreads()` call. Lines 40 through 49 contain the translated code after the synchronization point.

3.1.2 Host Code

CUDA host code must be hand-edited for Rigel since it is executed on the accelerator itself in the same memory space rather than on a separate host processor. Required changes to the host code include combining separate host
and device memory allocations and removing copying that is not necessary with Rigel’s single address space. Unlike a GPU, or multi-core, on Rigel every core executes the host code as shown in Figure 3.4. An example of host code is shown in Figure 3.5.

The RCUDA source translation process does a few automatic transformations. RCUDA removes the CUDA-specific constructs of kernel calls from the host code and converts the calls to standard C function calls. The kernel call on line 10 of Figure 3.5 is translated to the code in Figure 3.6. The parameters are stored in a global structure and updated by core 0 when the kernel is launched. All cores read from the parameter structure inside the kernel code.

3.2 Runtime Library

The second major component of the RCUDA framework is the software runtime library that is linked to the translated source code and provides an implementation of CUDA built-in functions in addition to load balancing control code.

The runtime library uses a combination of global and cluster-level data
```c
float * idata;
float * odata;
int width, height;
dim3 blockDim, gridDim;
... 
__rcuda_init(); // RCUDA runtime initialization
if (RigelGetThreadNum() == 0) { // Core 0 performs initialization
    ReadInput(idata, &width, &height); // Initialize idata, width and height
    gridDim.x = width/16; gridDim.y = height/16; gridDim.z = 1;
}
blockDim.x = blockDim.y = 16; blockDim.z = 1;
... 
// All other cores immediately jump to kernel launch
// even though the setup is not complete.
// The kernel launch will wait for core 0 and
// will use the values set by core 0.
transpose<<blockDim, gridDim>>(odata, idata, width, height);
// All cores will exit the CUDA kernel when all the work has been completed.
```

Figure 3.5: Rigel host code for transpose before RCUDA translation.

```c
struct transpose_params {
    float * odata, * idata;
    int width, height;
};
... 
__rcuda_init(); // RCUDA runtime initialization
if (RigelGetThreadNum()==0) { // Core 0 performs initialization
    ReadInput(idata, &width, &height); // Initialize idata, width and height
    gridDim.x = width/16; gridDim.y = height/16; gridDim.z = 1;
    blockDim.x = blockDim.y = 16; blockDim.z = 1;
    params.odata=odata; params.idata=idata;
    params.width=width; params.height=height;
}
__rcuda_kernel_launch(transpose, gridDim, blockDim);
// All cores will exit the CUDA kernel when all the work has been completed.
```

Figure 3.6: Transpose host code after RCUDA translation. Kernel input parameters are written to a global data structure by core 0.
structures to coordinate kernel execution across the chip. The thread block and grid dimensions are stored globally along with a count of the remaining thread blocks left to execute, as shown in Figure 3.7. Each cluster has a copy of variables used to keep track of thread block execution, as shown in Figure 3.8. Cluster-level variables include a count of the total threads in the thread block, the number of threads remaining to execute and synchronization variables.

### 3.2.1 CUDA Built-In Functions

CUDA built-in functions supported by the RCUDA framework include a cluster-level barrier and atomic operations. In RCUDA the \_syncthreads() call is implemented with \_rigel\_sync\_threads(). Once all cores in the cluster enter \_rigel\_sync\_threads(), the number of CUDA threads left to execute (ThreadCount) is reset so that the next section of the kernel can be executed. CUDA atomic functions are implemented with Rigel atomic operations.

### 3.2.2 Work Distribution

RCUDA handles work distribution hierarchically, at both the global chip-level and the local cluster-level. CUDA uses a grid of thread blocks to define the scope of work. Thread blocks are mapped to individual Rigel clusters.
Figure 3.9: CUDA thread block mapped to Rigel cluster using RCUDA with one synchronization call.

and executed simultaneously. Threads are executed concurrently by cores in the same Rigel cluster, allowing for synchronization across threads in a block. With the RCUDA runtime, a Rigel cluster can only execute one thread block at a time. The first core in a cluster that runs out of work attempts to fetch a new thread block for the cluster to be executed after all the other cores in the cluster are finished with the current thread block. Fetching blocks on demand allows the thread blocks to be dynamically allocated to clusters. A cluster only fetches one block at a time, which improves load balance at the cost of requiring more fetches. Locally, at the cluster-level, RCUDA control code handles work distribution by dividing up the threads among the cores in the cluster as shown in Figure 3.9. CUDA threads can either be mapped statically, with each core executing a fixed portion of the threads, or dynamically, with cores being assigned threads on demand for improved load balance at the expense of more frequent and potentially contended dequeue operations.

3.2.3 Stages of Kernel Execution on Rigel

Kernel execution begins when the host code invokes a kernel by calling the `__rcuda_kernel_launch()` function. Currently, only one kernel can be executed at a time, and the runtime requires the availability of all the clusters on the chip to execute a kernel. These requirements are a design choice and
not due to a fundamental limitation of either the architecture or the RCUDA framework. With the RCUDA framework, kernel execution can be broken down into four stages, as shown in Figure 3.10: Kernel Launch, Thread Block Fetch, Thread Block Execution, and Global Kernel Completion Barrier.

**Kernel Launch**

When a CUDA kernel is called from the host code, the cores enter the kernel launch stage. During the kernel launch stage, core 0 writes the kernel function parameters to global memory and initializes the RCUDA runtime global variables shown in Figure 3.7. Both `GlobalGridDim` and `GlobalBlockDim` are updated with the thread block and grid dimensions respectively. Additionally, core 0 saves the total number of blocks to `GlobalBlockCount` and the total number of threads per block to `GlobalThreadCount`. At the end of initialization, core 0 increments the `GlobalKernelLaunchCount` variable. By updating `GlobalKernelLaunchCount`, core 0 signals that the global variables are initialized and the other cores can now perform work.

While core 0 configures the global variables, one core from each cluster increments the per-cluster `KernelLaunchCount` variable. Cores only proceed to the thread block fetch stage when the `GlobalKernelLaunchCount` variable is greater than or equal to the cluster’s `KernelLaunchCount`.

---

**Figure 3.10: Stages of kernel execution with the RCUDA runtime.**
1: threadVal = ATOMIC_DEC(GlobalThreadCount);
2: threadIdx.y = threadVal / GlobalBlockDim.x;
3: threadIdx.x = threadVal - GlobalBlockDim.x * threadIdx.y
4: threadIdx.z = 0;

Figure 3.11: Code used to convert an integer to a two-dimensional thread index.

Thread Block Fetch

Each core enters the thread block fetch stage after kernel launch and whenever there are no threads left to execute in the current thread block. The first core to enter the thread block fetch stage fetches a thread block identifier (ID) from a global counter. To fetch a thread block, a core atomically decrements the number of thread blocks left, which is stored in the GlobalBlockCount variable. Next, a cluster-level barrier is used to synchronize the cores in the cluster since only one thread block can be executed at a time within a cluster. The core that performed the fetch waits for all cores in the cluster to enter the thread block fetch stage before updating the shared variables. Once all cores in the cluster enter the barrier, the fetching core updates the cluster’s CurrentBlock variable with the new thread block ID and resets the number of threads left to execute, which is stored in the ThreadCount variable. If CurrentBlock is nonnegative, then each core in the cluster enters the thread block execution stage. However, if a negative thread block ID is fetched, no work remains and the cores in the cluster proceed to the kernel completion barrier stage.

Thread Block Execution

During the thread block execution stage, the CUDA kernel executes. The CUDA threads within the thread block are dynamically or statically distributed across the cores in the cluster by RCUDA control code. When the threads are dynamically distributed, the CUDA thread index is generated from ThreadCount, the variable containing the number of threads left to execute. A core atomically decrements the number of threads left and the result is converted into a CUDA thread index. The method of conversion differs depending on the dimensions of the thread block. If a one-dimensional thread block is being executed, then the thread index is simply the result, (ThreadCount-1, 0, 0). However, for two-dimensional thread blocks, the
thread index is calculated using the thread block dimensions, as shown in Figure 3.11. Thread indices for three-dimensional thread blocks can be calculated in a similar manner. The CUDA thread indices are regenerated for each thread after every \_rigel\_sync\_threads() call as shown in Figure 3.3. Once a core has finished executing its portion of the thread block, it moves back to the thread block fetch stage and attempts to fetch a new thread block for the cluster.

**Global Kernel Completion Barrier**

The host code on Rigel is written with the assumption that the kernel execution is complete before any cores return from the kernel function. The sole purpose of the kernel completion barrier stage is to wait until all clusters are finished executing thread blocks before returning to user code. When all of the cores in a cluster have completed executing a thread block, and no more thread blocks are available, the cores enter the barrier. Once in the barrier, the cores in the cluster wait for all the other cores on the chip to enter the barrier. After all cores enter the barrier, the cores return to the host code and are able to execute additional kernels.

### 3.2.4 Comparing RCUDA and RTM

RCUDA and RTM both provide an interface for software task distribution while hiding the complexities from the developer. Externally, from the point of view of the programmer, RCUDA can be used to execute uniform SPMD code while RTM has the flexibility of running irregular tasks in MIMD fashion. Internally, RCUDA and RTM use different methods to handle workload distribution and task mapping.

**Programmer’s Perspective**

RTM is comprised of a runtime library that provides functions to distribute work across Rigel. When using RTM, a programmer must explicitly add initialization, task enqueue and task dequeue functionality. In RCUDA code, a programmer must only call an initialization function and a CUDA kernel; workload distribution code is automatically added by the source translation process. RCUDA’s load balancing code is completely transparent to the programmer; which greatly simplifies the user level code.
Table 3.1: Memory overhead of internal variables used by RCUDA and RTM runtimes. Both use global and cluster-level data structures while RTM requires additional per-core and per-task variables.

<table>
<thead>
<tr>
<th></th>
<th>RCUDA</th>
<th>RTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Variables</td>
<td>72 bytes</td>
<td>128 bytes</td>
</tr>
<tr>
<td>Per-Cluster Variables</td>
<td>32 bytes</td>
<td>256 bytes</td>
</tr>
<tr>
<td>Per-Core Variables</td>
<td>0 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Per-Task Variables</td>
<td>0 bytes</td>
<td>32 bytes</td>
</tr>
</tbody>
</table>

Programming Model

RCUDA is designed to efficiently run SPMD CUDA code on a MIMD architecture, and, as a result, RCUDA directly inherits CUDA’s rigid programming model. All CUDA threads within a grid use the same code and variables; only the values of the thread block and thread indices are different. The CUDA thread indices in RCUDA are uniform and can be generated easily using the equation shown in Figure 3.11. RCUDA’s uniform indexing strategy is advantageous because it reduces the number of data structures required to implement programming model as demonstrated by Table 3.1. Threads within a thread block are assumed to be uniform in length and work imbalance will occur if the threads are irregular since all threads must complete before fetching new work.

RTM is generally more flexible than RCUDA in that it allows any core to add work to the task queue at any time; in addition, RTM can effectively load balance irregular tasks. The flexibility comes at the cost of increased complexity of the runtime system. For example, the four task parameters in RTM can contain any value, so a copy of the variables must be kept for each task. Additionally, the global barrier code is more complex in RTM because any core can insert more work at any time during execution, which may require that other cores start working again after having reached a barrier.

Load Balancing

In terms of load balancing, each runtime has advantages. Both RCUDA and RTM distribute work on a global and cluster-level. For RCUDA, thread blocks are distributed among clusters and threads within the thread blocks
are mapped to different cores. With RTM, task groups are dynamically allocated to clusters though the global task queue, and tasks are distributed to different cores by the cluster task queue.

At the cluster-level, RCUDA is optimized for common CUDA code, where the execution time for each thread is uniform, and each thread block performs a similar amount of work. The RCUDA runtime code is designed with the assumption that all threads have the same cost. RTM handles both uniform code and irregular codes well through the use of the global and cluster task queues.

Task Mapping

A major advantage of RCUDA is that the developer can write code knowing that all threads within a thread block are live at the same time and execute on the same cluster. Using thread blocks allows programmers, both at the application and library level, to exploit data sharing through Rigel’s shared cluster cache. RTM has a notion of task groups, but the construct is not exposed to the library user, and RTM makes no guarantees that tasks in the same task group are executed simultaneously.

RCUDA executes consecutive thread blocks in order, which leads to implicit blocking and can improve the cache usage in many kernels. RTM has a more complicated ordering; for example, RTM does not execute consecutive tasks in order if the tasks are enqueued from different clusters. Programmers must explicitly handle blocking in RTM programs.

Portability

CUDA kernel code written for RCUDA can be compiled for GPUs and multicore CPUs, allowing programmers to write code for Rigel and use the same code on other architectures. Using RCUDA, programmers can also use existing CUDA kernel code on Rigel.
CHAPTER 4
OPTIMIZATIONS AND AUTOMATION

This chapter introduces two classes of optimizations that can be used to improve performance of CUDA kernels on Rigel and discusses how the source transformations could be applied automatically. The first class of optimizations are \textit{Kernel Transformations} which modify kernel code and are used to efficiently map CUDA constructs, including shared memory and thread synchronization, to Rigel. \textit{Runtime Optimizations}, the second class of optimizations, change the manner in which the kernel code is executed, including execution ordering and load distribution. The last part of the chapter describes how to automate the source-to-source translations that are currently performed manually.

4.1 Kernel Transformations

Some CUDA constructs do not map well to Rigel. The first is shared memory; unlike a GPU, Rigel uses a cached single address space without any specialized memories. Second is thread synchronization; while the \_\_\texttt{syncthreads()}\_\_ call is a low-latency operation on a GPU, on Rigel it must be implemented in software using cluster-level atomics. Kernel transformations can be used to remove shared memory accesses and thread synchronization from some CUDA kernels, resulting in improved performance on Rigel.

4.1.1 Shared Memory Removal

Programmers can avoid memory bandwidth bottlenecks by caching data in a shared memory scratchpad to leverage temporal locality and overcome the limited cache and prefetch capabilities of a GPU.

Shared memory can be utilized in a variety of ways, but the most common shared memory usage patterns in high-performance CUDA code are shown in
Figure 4.1: Common shared memory access patterns. Shared memory is shown in white, global memory in gray, and black represents local memory.

\[
\text{shared}_\text{mem}[f(\text{threadIdx}.x)][g(\text{threadIdx}.y)] = \text{global}_\text{mem}[h(\text{threadIdx}.x, \text{threadIdx}.y)]
\]

Figure 4.2: Shared memory example with indexing expressions \(f\), \(g\), and \(h\).

Figure 4.1. Shared memory is used to store a local copy of global data in the cache case. Delayed write is the reverse of caching, where shared memory values are written to global memory locations. Using delayed write, programmers can structure memory accesses allowing for improved throughput. Scratchpad read and scratchpad write are used when threads read or write local variables to shared memory.

Many CUDA kernels use a combination of caching and delayed write to simply store a local copy of global data. The kernel first populates a shared memory array with global data, then the kernel accesses the shared memory for all computations before writing the results back to global memory. Typically, the shared memory is accessed uniformly across all threads, meaning that the threads use the same indexing expressions to calculate the offsets into the global and shared memory arrays. An example of shared memory usage is shown in Figure 4.2. In the example, the indexing expressions \(f\) and \(g\) are used to generate the offset into the shared memory array while \(h\) determines the offset into the global memory array.

Using shared memory as a fast access cache works well on the GPU, but on
Figure 4.3: Examples of shared memory usage that cannot be replaced with a mapping function.

Rigel it creates a redundant copy of values with the same access time. Using a mapping function, CUDA kernels can bypass the shared memory array and access global memory directly. A shared memory array can be replaced by a mapping function if the kernel meets the following requirements:

- The shared memory array must only be used to store global data, and never used to store values generated during the thread block execution. Shared memory values are commonly read multiple times, so restricting shared memory to a copy of global data avoids computing shared memory values more than once.

- When accessing the shared memory array, each thread must use the same indexing expressions. By using the same indexing expressions, the mapping function does not need conditional statements, which can severely degrade performance.

- The indexing expressions must be invertible. Shared memory indexing expressions must only depend on one dimension of the thread index and other variables that remain constant during the thread block’s execution. The indexing expressions for the global memory array share the same restrictions as shared memory indexing expressions, but can only depend on the thread index dimensions used in the shared memory indexing expressions.

While restrictive, many CUDA kernels meet these requirements, allowing shared memory to be replaced with a mapping function.

Figure 4.3 shows three examples of shared memory usage that cannot be replaced with a mapping function. In the first case, the threads with

```c
1: if (threadIdx.x == 0)
2:     sharedMem[threadIdx.x] = globalMem[threadIdx.x];
3: else
4:     sharedMem[threadIdx.x] = globalMem[threadIdx.x*20];
```

```c
1: sharedMem[threadIdx.x + threadIdx.y*blockDim.y] =
2:     globalMem[threadIdx.x][threadIdx.y];
```

```c
1: sharedMem[threadIdx.x] = globalMem[threadIdx.x] * inputVal;
```
1: __global__ void transpose(float *odata, float *idata, int width, int height)
2: {
3: __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
4: 
5: // read the matrix tile into shared memory
6: unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
7: unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
8: if((xIndex < width) && (yIndex < height))
9: {
10:    unsigned int index_in = yIndex * width + xIndex;
11:    block[threadIdx.y][threadIdx.x] = idata[index_in];
12: }
13: __syncthreads();
14: 
15: // write the transposed matrix tile to global memory
16: xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
17: yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
18: if((xIndex < height) && (yIndex < width))
19: {
20:    unsigned int index_out = yIndex * height + xIndex;
21:    odata[index_out] = block[threadIdx.x][threadIdx.y];
22: }
23: }
24: }

Figure 4.4: Transpose kernel from the CUDA SDK that uses shared memory.

threadIdx.x equal to 0 use a different global memory indexing expression than the other threads. Shared memory removal should not be performed because the mapping function would require a conditional statement. The second example contains indexing expressions that cannot be inverted because the shared memory indexing expression depends on both X and Y components of the thread index. Shared memory indexing expressions can only depend on one of the thread index dimensions, so shared memory removal cannot be performed. In the third case, the shared memory array is not populated with the global values, but rather values generated during thread block execution. While shared memory removal is possible, the shared memory values would have to be computed each time they are read. To avoid potentially degrading performance, the shared memory should not be removed.

Shared Memory Removal and Transpose Kernel

The transpose kernel shown in Figure 4.4 meets the requirements for shared memory removal. Transpose uses the caching access pattern, and on line 11, the shared array block is populated with global data. The indexing expressions used for the shared memory array accesses only depend on one
#define BLOCK(_a,_b) idata[(blockIdx.y * BLOCK_DIM + _a) * \n    width + blockIdx.x * BLOCK_DIM + _b]

Figure 4.5: The preprocessor macro that can be used to replace shared memory in the transpose kernel.

component of the thread index, while the global memory indexing expression is based on a compile-time constant (BLOCK_DIM), the block index and the thread index. The kernel pads the shared memory, so not all values in the shared memory array are initialized. Padding, discussed further in Section 4.1.1, can be ignored since the kernel does not read the uninitialized shared memory values. In the case of the transpose kernel, the shared memory is initialized within a conditional block, meaning that some threads might not write to the shared memory at all. The conditional statement before the __syncthreads() call can be ignored since subsequent reads to the shared memory are assumed to use initialized values. The second conditional statement on line 19 must be kept, since global memory writes occur in dependent statements. On line 22, the global array odata is filled by the shared array using delayed write. Since the global data from idata is cached and then written to the global array odata using delayed write, it is possible to remove the shared memory in the transpose kernel. The shared memory read on line 22 of the transpose kernel in Figure 4.4 can be replaced by the preprocessor macro shown in Figure 4.5, which combines lines 6, 7, and 10 so that global memory is read on line 22. The macro maps the shared memory reads to global memory reads, so the transpose kernel can directly populate the output array odata from idata without any intermediate copying to shared memory.

Automation of Shared Memory Removal

In order to automate shared memory removal, the source-to-source translator must first determine the shared memory usage pattern for the kernel. In both the cache and delayed write cases, shared memory can be removed if the kernel adheres to the constraints given previously. Additionally, shared memory can be removed from a kernel that uses both cache and delayed write as long as the input global array is not the same as the output global array.

If the kernel is found only to be using cache and delayed write memory...
assess patterns, the next step is to determine if the indexing expressions meet
the requirements of shared memory removal. The source-to-source transla-
tor must analyze the shared memory accesses within the kernel to validate
that the same indexing expressions are used by all threads and that the
variables used in the indexing expressions obey the shared memory removal
constraints.

Once the indexing expressions are validated, a mapping function from
shared memory array indices into global data arrays can be constructed
by modifying the global indexing expressions to depend on the index into
the shared memory array, and not the thread index directly. For example,
consider the following shared memory access:

\[
\text{shared\_mem[threadIdx.x+4][threadIdx.y*BLOCK\_SIZE]}
\]

= \text{global\_mem[threadIdx.x*inputWidth + threadIdx.y]}

For the shared memory accesses, there are two shared memory indexing ex-
pressions, one dependent on \( \text{threadIdx.x} \), and the other on \( \text{threadIdx.y} \):

\[
f(x) = x + 4 \\
g(y) = y \times \text{BLOCK\_SIZE}
\]

Similarly, the global memory indexing expression \( h \) can be written as follows:

\[
h(x, y) = x \times \text{inputWidth} + y
\]

To find the mapping function, the indexing expression \( h \) must be redefined
in terms of the shared memory indexing expressions. Let the variables \( a \) and
\( b \) equal the indices of the shared memory array. Both \( a \) and \( b \) can be written
in terms of \( x \) and \( y \) as shown below:

\[
a = x + 4 \\
b = y \times \text{BLOCK\_SIZE}
\]

Now, substituting in \( a \) and \( b \) for \( x \) and \( y \) in the global indexing expression \( h \),
yields the following equation:

\[
h(a, b) = (a - 4) \times \text{inputWidth} + b/\text{BLOCK\_SIZE}
\]
The result is the mapping function, and wherever the shared memory array is used in the kernel, it can be replaced with the following preprocessor macro:

```
#define MAPPING_FUNC(a,b) \ 
    global_mem[(a-4) * inputWidth + b/BLOCK_SIZE]
```

It is common for CUDA code to pad variables in shared memory to reduce bank conflicts. With padding, the kernel code does not initialize the entire shared memory array. Shared memory can still be replaced because the pad locations should not be accessed if they are not initialized earlier in the execution. The transpose kernel in Figure 4.4 uses padding to optimize shared memory accesses. On line 3, the shared memory array defined is larger than required, the second dimension is defined as \( \text{BLOCK}_\text{DIM} + 1 \) when only indices ranging from 0 to \( \text{BLOCK}_\text{DIM} - 1 \) are used.

Since shared memory is being replaced by global memory, some errors that were previously transparent to the programmer may become more apparent. A shared memory location could be read that has not been written, or the mapping function may generate an invalid index that would be out of range of the global memory array. For example, consider the mapping function shown above. If a kernel read the value for `shared_mem[0][0]`, it would be using an undefined value, but it would not crash. After replacing the shared memory access with the mapping function, it is possible that the mapping function would generate a negative value into the global array, potentially causing a fatal error at runtime. While exceeding the limits of the global memory array could crash the program, it is not a concern because it is the result of buggy CUDA code. CUDA does not define what shared memory values are initialized to at the beginning of a thread block’s execution, so in correct code shared memory values must be set before being read.

Shared memory removal can degrade performance if the mapping function is complex and used often in the translated kernel code. The additional overhead of evaluating the mapping function can outweigh the benefits of removing redundant memory copies in some cases. To minimize the risk of reducing performance, the source-to-source translator must compare the complexity of the mapping function verses the indexing expressions in the original CUDA kernel. Shared memory should not be removed if the mapping function is used repeatedly and is more complex than the indexing expressions.
1: globalMem[threadIdx.x] = threadIdx.x;
2: __syncthreads(); // Can be removed
3: int value = globalMem[threadIdx.x];
1: globalMem[threadIdx.x] = threadIdx.x;
2: __syncthreads(); // Cannot be removed
3: int value = globalMem[blockDim.x-threadIdx.x];
1: atomicAdd(&sharedValue,1);
2: __syncthreads(); // Cannot be removed
3: if (atomicCAS(&sharedValue,16,0) == 16)
4: DoSomething();
5: else if (atomicCAS(&sharedValue,8,0) == 8)
6: DoSomethingElse();

Figure 4.6: Three examples of synchronization usage.

4.1.2 Synchronization Removal

Thread block synchronization is expensive on Rigel for two reasons: the barrier code itself is implemented in software using cluster-level atomics, and RCUDA needs to regenerate CUDA thread indices after each __syncthreads() call. Synchronization removal can be performed on a kernel when there are no memory access conflicts across the barrier, meaning that no thread initializes shared or global memory before the barrier that is subsequently consumed after the barrier by another thread.

Figure 4.6 contains three examples of kernel code using synchronization. In the first example, the __syncthreads() call can be safely removed because the global memory value initialized before the barrier is accessed by the same thread after the barrier. The second case is identical to the first except that the global memory value initialized before the barrier is consumed by a different thread after the __syncthreads() call. The __syncthreads() call cannot be removed due to the memory access conflict. In the last example, all threads use atomics to alter the value of the variable sharedValue before the synchronization point. Then after the barrier, all threads use compare and set methods to read, and possibly alter, sharedValue. Even though only atomic operations are used, there is still a memory access conflict: all threads alter the value of sharedValue before the barrier and subsequently read the value after the barrier. For a block size of 16, the conditional statement on line 3 will always be true for one thread, and the conditional statement on line 5 will never be true. Without the barrier, it is possible that the variable sharedValue would have a value of 8 which would cause the conditional statement on line 5 to be true. Removing the __syncthreads() call would
change the behavior, so the call must remain.

Synchronization removal is usually ineffective by itself because programmers do not typically add extra barriers, especially in tuned CUDA code. However, if the shared memory removal optimization can remove the shared memory accesses across the barrier, synchronization removal can be used to eliminate the barrier if there are no other conflicts.

Synchronization Removal and Transpose Kernel

In the case of the transpose kernel in Figure 4.4, the barrier can be removed, but only after shared memory removal is performed. The shared memory removal process removes all shared memory usage, which eliminates memory access conflicts across the barrier. Figure 4.7 shows the transpose kernel with the barrier removed.

Automation of Synchronization Removal

Synchronization removal should always be performed if possible since it removes unnecessary code in the kernel, and more importantly, reduces the number of times that thread indices are generated by the runtime. It is safe to remove a `__syncthreads()` call if no thread before the barrier accesses memory locations (either shared or global) read by another thread after the barrier. The source-to-source translator must first analyze memory accesses before a barrier. If no threads access shared or global memory before the barrier, then the `__syncthreads()` call can be safely removed. In the case where threads access either shared or global memory, then further analysis is required. The `__syncthreads()` call can be removed only if the shared or global values accessed before the barrier are not consumed by other threads after the barrier.

4.2 Runtime Optimizations

Unlike GPUs, Rigel uses software to handle the work distribution of CUDA kernels. Using software is advantageous because not every kernel need be executed the same way. Both static work partitioning and thread fusing can help achieve better performance on Rigel.
1: #define BLOCK(_y,_x) idata[(blockIdx.y * BLOCK_DIM + _y) * \n2: width + blockIdx.x * BLOCK_DIM + _x]  
3: 
4: void transpose(dim3 blockIdx, dim3 blockDim, dim3 gridDim) {  
5:   dim3 threadIdx;  
6:   int __threadIndex;  
7:   float * odata = transpose_mc_params.odata;  
8:   float * idata = transpose_mc_params.idata;  
9:   int width = transpose_mc_params.width;  
10:  int height = transpose_mc_params.height;  
11:  while((__threadIndex = atomic_get_next_tid_2d(&threadIdx,blockDim.x)) >= 0){  
12:    int xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;  
13:    int yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;  
14:    if (xIndex < height && yIndex < width) {  
15:      unsigned int index_out = yIndex * height + xIndex;  
16:      odata[index_out] = BLOCK(threadIdx.x, threadIdx.y);  
17:    }  
18:  }  
19:  }

Figure 4.7: Transpose kernel after synchronization and shared memory removal.

4.2.1 Static Work Partitioning

The RCUDA runtime supports load balancing at the cluster-level by allowing individual cores to fetch threads on demand. Dynamic load balancing requires that thread indices be generated at runtime, which can be expensive for short threads or threads with many synchronization points. An optimization is to statically assign work to each Rigel core so that each core executes a fixed number of threads within a thread block. For static work assignment to improve performance on Rigel, the CUDA threads must perform similar amounts of work, and the number of threads in a thread block should be divisible by eight, the number of cores in a cluster, to avoid load imbalance. Since static work partitioning does not change the kernel code, it can be applied to any kernel without the risk of generating incorrect code.

Static Work Partitioning and Transpose Kernel

Figure 4.8 shows the transpose kernel from Figure 4.4 with static work partitioning. When static work partitioning is applied, no local state cluster-level data structure is generated because CUDA threads do not migrate between cores. Instead, the local thread state is stored on the stack. In the case of the transpose kernel, two arrays are defined on the stack: xIndex and yIndex. Both of the arrays have a length of 64 because the maximum number of CUDA threads in a thread block is 512, which divided by eight
struct block_state {
    float block[BLOCK_DIM][BLOCK_DIM+1];
};

void transpose(dim3 blockIdx, dim3 blockDim, dim3 gridDim,
    dim3 startIdx, dim3 endIdx) {
    block_state * bs = ( & transpose_mc_vars[getSMID()]);
    dim3 threadIdx;
    int __threadIndex;
    float * odata = params.odata;
    float * idata = params.idata;
    int width = params.width;
    int height = params.height;
    unsigned int xIndex[64];
    unsigned int yIndex[64];
    for (threadIdx.y=startIdx.y,__threadIndex=0;threadIdx.y<endIdx.y;threadIdx.y++) { 
        for (threadIdx.x=startIdx.x;threadIdx.x<endIdx.x; threadIdx.x++,__threadIndex++) {
            xIndex[__threadIndex]=((blockIdx.x*BLOCK_DIM)+threadIdx.x);
            yIndex[__threadIndex]=((blockIdx.y*BLOCK_DIM)+threadIdx.y);
            if (((xIndex[__threadIndex]<width)&&(yIndex[__threadIndex]<height))) {
                unsigned int index_in =
                ((yIndex[__threadIndex]*width)+xIndex[__threadIndex]);
                bs->block[threadIdx.y][threadIdx.x]=idata[index_in];
            }
        }
    }
}
(the number of Rigel cores in a cluster) equals 64. While moving the local state from a cluster-level variable to the stack seems trivial, it creates code that is easier for the compiler to optimize. When using file scoped variables, the compiler has no way to determine that the values are not needed after the execution of the kernel function. Using the stack memory, the compiler can easily identify the scope of the local thread variables, and thus optimize away any unnecessary writebacks.

Automation of Static Work Partitioning

While static work partitioning helps reduce runtime overhead, there is more potential for load imbalance because CUDA threads are not dynamically distributed to the Rigel cores during execution. It is important to only apply the static work partitioning optimization when the kernel code performs a similar amount of work across all threads, and when the number of threads is divisible by the number of Rigel cores. In order to meet these requirements, the thread block dimensions must be known at compile time and the source-to-source translator must analyze the code to ensure that there are not large blocks of code that execute conditionally based on the thread index value (threadIdx). In addition to compiler analysis, another way to ensure a performance improvement is to use auto-tuning [23]. Two versions of the kernel can be generated during the source translation stage, one that uses static work partitioning, and another that distributes work dynamically. Auto-tuning can be used to find the best performing kernel at runtime.

4.2.2 Thread Fusing

Thread fusing is a source-level transformation that allows threads to execute as a group in parallel through software pipelining. Thread fusing is similar to loop unrolling: just as a loop can be unrolled to reduce the number of iterations, threads can be fused to decrease the granularity of work. For example, a CUDA thread performs the computation for one element of the thread block (X, Y), while a thread using fusing can perform the computation of the threads (X,Y) to (X,Y+N) or (X,Y) to (X+N, Y) where N is the number of threads combined together. Thread fusing has a few key advantages: first it can be used as a way to optimize memory accesses by enforcing execution
ordering. Secondly, thread fusing reduces runtime overhead, due to the decreased work granularity. Finally, thread fusing can make the kernel code more efficient by eliminating identical work performed by the fused threads.

For some kernels it is advantageous to enforce an execution order as a way to optimize memory accesses. In CUDA code with a two-dimensional thread block, it is common to see an indexing expression based off of the thread index, for example:

$$(\text{threadIdx.y} \times \text{BLOCK.\_SIZE}) + \text{threadIdx.x}$$

Here, the $Y$ component of the thread index is multiplied by a constant factor, and the $X$ component is used as an offset. On Rigel, an optimal grouping would have threads with the same $Y$ index value combined together so that the code accesses sequential memory accesses, and thus hits the same cache line. Thread fusing also reduces false sharing when unfused threads with varying $X$ values compete for cache lines.

Thread fusing reduces runtime overhead by reducing the granularity of the work. The CUDA thread index only needs to be fetched once for a group of fused threads, which means the thread index generating code is executed less often. Additionally, redundant work can be eliminated since variables that are independent of the thread index only need to be computed once in a group of fused threads. Merging multiple threads together also allows the compiler to optimize a group of threads instead of just a single thread, allowing it to generate faster, more efficient code.

Thread Fusing and Transpose Kernel

Figure 4.9 shows the transpose kernel with thread fusing applied. The transpose kernel is executed using 16x16 thread blocks, so threads can be fused to combine multiple threads with the same $X$ index, or the same $Y$ index. In the case of transpose, it is advantageous to combine threads with the same $Y$ index so that the fused threads access consecutive memory locations. After thread fusing, the modified transpose kernel performs the work of 16 CUDA threads at once, executing threads $(0, Y)$ to $(15, Y)$. The code fetches the $Y$ component of the thread index dynamically, and executes the logic of 16 threads, each with a different $X$ component.
struct block_state {
    float block[BLOCK_DIM][(BLOCK_DIM+1)];
};

void transpose(dim3 blockIdx, dim3 blockDim, dim3 gridDim) {
    block_state * bs = (&transpose_mc_vars[getSMID()]);
    dim3 threadIdx;
    int __threadIndex;
    float * odata = params.odata;
    float * idata = params.idata;
    int width = params.width;
    int height = params.height;
    int xIndex;
    int yIndex;
    while ((__threadIndex = atomic_get_next_y(&threadIdx, blockDim.x)) >= 0) {
        xIndex = ((blockIdx.x*BLOCK_DIM) + 0);
        yIndex = ((blockIdx.y*BLOCK_DIM)+threadIdx.y);
        if (yIndex < height) {
            unsigned int index_in = yIndex*width+(blockIdx.x*BLOCK_DIM);
            if (xIndex + 0 < width)
                bs->block[threadIdx.y][0] = idata[index_in + 0];
            if (xIndex + 1 < width)
                bs->block[threadIdx.y][1] = idata[index_in + 1];
            if (xIndex + 2 < width)
                bs->block[threadIdx.y][2] = idata[index_in + 2];
            ...
            if (xIndex + 15 < width)
                bs->block[threadIdx.y][15] = idata[index_in + 15];
        }
        __rigel_sync_threads();
        while ((__threadIndex = atomic_get_next_y(&threadIdx, blockDim.x)) >= 0) {
            yIndex = (blockIdx.x*BLOCK_DIM)+threadIdx.y;
            if (yIndex < width) {
                int index_out = yIndex*height + (blockIdx.y*BLOCK_DIM);
                if (xIndex + 0 < height)
                    odata[index_out + 0] = bs->block[0][threadIdx.y];
                if (xIndex + 1 < height)
                    odata[index_out + 1] = bs->block[1][threadIdx.y];
                if (xIndex + 2 < height)
                    odata[index_out + 2] = bs->block[2][threadIdx.y];
                ...
                if (xIndex + 15 < height)
                    odata[index_out + 15] = bs->block[15][threadIdx.y];
            }
        }
    }
}

Figure 4.9: Translated transpose CUDA kernel with thread fusing applied.
In addition to optimizing the memory accesses, thread fusing simplifies code by removing duplicate variables. In the resulting code for the transpose kernel, all threads set the \texttt{yIndex} to the same value, so its value only has to be computed once for every 16 threads. Redundant statements can also be removed; for example, the conditional statement of the original code in Figure 4.4 on line 8 depends on \texttt{yIndex} and \texttt{xIndex}. After thread fusing, all combined threads share the same \texttt{yIndex} value, so the code only needs to compare the value of \texttt{yIndex} once.

Automation of Thread Fusing

Thread fusing is achieved by putting copies of the thread code back-to-back with implicit index generation. The actual source translation is relatively straightforward; the complex part is determining the number of threads to fuse together and which threads to combine. The number of threads to combine together depends on the code complexity. For small kernels, it is advantageous to fuse many threads together since runtime overhead could be significant overhead. The benefits of thread fusing diminish when the thread size increases, not only because the runtime overhead is less of a concern, but also due to increased register pressure. For optimal performance, thread fusing should not create code that causes the compiler to spill registers to the stack. Usually the thread block size is defined as a compile time constant, so the source-to-source translator can determine a grouping size so that the number of combined threads is divisible by eight, the number of Rigel cores in a cluster. However, if the thread block size is determined at runtime, another approach is required. Instead, multiple kernel versions can be generated, each with a different number of threads combined. Auto-tuning can be used to determine which kernel performs the best at runtime.

The other consideration of thread fusion is how to combine the threads together. For two-dimensional thread blocks, most CUDA threads use one component of the thread index as an offset into a global memory array. Sequential memory accesses patterns offer the highest performance on Rigel, so it is advantageous to group threads together that access memory sequentially. For example, if the \texttt{X} component of the thread index is used as an offset, multiple threads sharing the same \texttt{Y} value should be combined so that fused threads access sequential addresses.
4.3 Optimization Ordering

The optimizations need to be applied in the correct sequence in order to make analysis easier and to achieve the highest possible performance. Kernel transformations should be performed first, so that the kernel code is as simple as possible to make the runtime optimization analysis easier. First, shared memory removal should be performed on the kernel in order to eliminate any shared memory accesses that are not required. Synchronization removal should be applied next, because the shared memory removal optimization could remove memory conflicts if shared memory accesses were optimized away. After the kernel transformations, thread fusing can be used to combine CUDA threads into larger units of work. Static work partitioning should be applied after thread fusing since fusing threads will change the work granularity.

4.4 Source Translation Automation

In addition to optimizations, some CUDA-to-C source translations are currently done by hand. Manual source transformations include software coherence actions and atomics.

4.4.1 Software Coherence Actions

Rigel provides a shared cluster cache so that all cores within a cluster can efficiently share data. The cluster caches are not coherent between clusters and programmers are required to insert software coherence actions if coherence is necessary.

CUDA provides a rigid programming model with explicit global values, temporary shared values, and local values. Software coherence actions must be used when a kernel writes to a global variable. In order to make the value globally visible the result must be written back by flushing the cache line before the kernel execution is complete. To automate the software coherence actions, the source-to-source translator must be able to detect when a globally defined variable is updated by the kernel code. After the variable is updated, the source-to-source translator must add a call to flush the cache line.
Unlike global variables, CUDA shared and local thread variables are replicated for each cluster because a thread block only executes on one cluster. No software coherence actions are required for shared and local variables because they will never be accessed by another cluster. However, to improve performance, shared and local variables can be invalidated at the end a thread block’s execution in order to eliminate unnecessary writebacks.

4.4.2 Atomics

CUDA atomic operations can be used on device memory or shared memory. On Rigel, device memory is accessible by all clusters so a CUDA atomic operation acting on device memory can be converted to a Rigel global-level atomic operation. Since shared memory can only be accessed by a single cluster, atomic operations performed on shared memory can be translated to a Rigel cluster-level atomic operation.
CHAPTER 5
EVALUATION

This chapter describes the simulation and evaluation methodology, presents the benchmarks and examines the results. The evaluation is broken down into three parts. First is a comparison of RCUDA and RTM performance across a set of benchmarks to test how each runtime handles load balancing and scalability. Second is an analysis of RCUDA performance using benchmarks representing typical GPU workloads, with and without optimizations applied. Finally, various implementations of dense-matrix multiplication are executed on a GPU and Rigel with a discussion of the performance portability of the different algorithms.

5.1 Simulation Infrastructure Methodology

All performance results for the Rigel accelerator design are produced using a cycle-accurate execution driven simulator that models cores, caches, interconnects, and memory controllers [2]. GDDR5 memory timing parameters are used for the DRAM model. Benchmark and library codes are compiled with LLVM 2.5 using a custom backend and are run in the simulator. All RCUDA and RTM code was written in C, while inline assembly was used for global and cluster-level atomic operations. RCUDA optimizations have yet to be fully automated in the framework, and thus were applied by hand-editing translated CUDA kernels. Results for CUDA on GPU were gathered on a Tesla [24] T10 4-GPU server using one GPU.

5.2 Comparing RCUDA and RTM Performance

The software runtime libraries RCUDA and RTM have significant differences, as discussed in Section 3.2.4. This section compares these runtimes in terms of workload distribution and scalability.
5.2.1 Workload Distribution

RCUDA and RTM both handle workload distribution across Rigel, but divide the work up in different ways. RCUDA distributes thread blocks to clusters, then cores within the cluster execute threads. The programmer specifies the total number of thread blocks and the number of threads contained within a thread block. On the other hand, RTM uses hierarchical task queues to distribute tasks. Fixed size task groups are distributed to clusters and cores within the cluster dequeue and execute individual tasks on demand.

Three benchmarks, each with an RCUDA and RTM version, are used to test how the two runtimes perform with different workloads. For each benchmark, the RCUDA version executes a total of 8192 CUDA threads and the RTM version executes 8192 tasks, meaning that the amount of work is the same, only the method of task distribution is different.

The first benchmark, Short, represents a workload with uniform, fine-grained tasks. The RCUDA version of Short executes a total of 1024 thread blocks with eight threads each. Each thread’s work is comprised of a single instruction. A thread block size of 8 is used because it matches the task group size of RTM. The RTM configuration of Short executes a total of 8192 tasks with one instruction each.

Large is the second benchmark, and it represents a workload with uniform, coarse-grained tasks. The Large benchmark is identical to Short except that the RCUDA threads and RTM tasks are both 8192 instructions long.

The last benchmark, Irregular, simulates a non-uniform workload. Like the Short and Long benchmarks, the Irregular RCUDA version executes a total of 8192 threads and the RTM version executes 8192 tasks. In the RCUDA version, the length of the threads in each thread block range from 0 to 7168 instructions, using the formula below:

\[
\text{Thread Instruction Count} = \text{threadIdx.x} \times 1024
\]

Having threads of various lengths in the same thread block represents a worst case scenario for RCUDA because cores in the cluster wait for the longest thread to finish before proceeding. The RTM version of Irregular executes tasks that range from 0 to 7168 instructions in length, using the following formula:
Figure 5.1: RCUDA performance relative to RTM for workloads with short, long and irregular tasks.

\[
\text{Task Instruction Count} = (\text{TaskNum}\%8) \times 1024
\]

Using the above formula, each task group (group of eight tasks), contains the same distribution of instruction lengths as a thread block in the RCUDA version.

The results are shown in Figure 5.1. RCUDA outperforms RTM in the Short and Long benchmarks. For the Short benchmark the RCUDA version executes almost 60% faster than the RTM version. However, for Long, the RCUDA version is only about 6% faster than the RTM version. These results demonstrate that RCUDA is efficient for shorter tasks, but as the task length increases the advantages of RCUDA diminish since the execution time is dominated by task execution and not runtime overhead. For the Irregular benchmark, RCUDA takes over 40% longer to execute than the RTM version. The performance difference is due to RCUDA’s inability to effectively load balance irregular work at the cluster-level. With RCUDA, cores do not start executing new work until the current thread block finishes execution. On the other hand, with RTM, a core is able to fetch a new task once it completes execution of its current task; it does not have to wait for any other core to finish before proceeding.
5.2.2 Workload Scalability

Here, both runtimes are compared on how well their performance scales with an increase in workload size. Four benchmarks are used with various workload sizes to evaluate scalability. The RTM benchmark is implemented using the RTM runtime and executes tasks with a single instruction each. The remaining three benchmarks all use RCUDA: RCUDA-8 Threads, RCUDA-16 Threads, RCUDA-32 Threads use thread blocks with 8, 16 and 32 threads respectively.

As Figure 5.2 shows, execution time rises with the amount of work, except for some RCUDA configurations with less than 8192 threads. With more tasks or threads in the system, the runtime code is simply executed more often leading to a longer execution time. The results indicate that every configuration of RCUDA executes in less time than the RTM benchmark due to the lightweight and uniform nature of the workload. Unlike the RTM benchmark, the RCUDA benchmarks sometimes speed up with an increased number of threads, for example, RCUDA-8 Threads executes significantly faster with 2048 total threads than with 1024 threads. The
Figure 5.3: Performance of RCUDA and RTM with an increasing number of clusters and memory bandwidth.

The performance difference is due to the short length of the tasks and how the RCUDA runtime distributes tasks to the clusters. Each cluster fetches a new thread block on demand; however, competition to fetch a block is unfair and clusters can starve. With 1024 tasks, some clusters execute multiple thread blocks while other clusters do not execute any, resulting in load imbalance and performance degradation.

The RCUDA benchmarks show that for tests with homogeneous threads (which execute in similar or identical runtimes), it is advantageous to have more threads in a thread block (and thus fewer thread blocks) as long as the total number of thread blocks is larger than the number of clusters. Having more threads in a thread block means that fewer global atomic operations are executed, and the RCUDA runtime relies primarily on lower-latency, cluster-level atomic operations to distribute work.

5.2.3 Scalability with Execution Units

Both RCUDA and RTM are optimized for parallelism and are designed to utilize more execution units if available. With a well-designed runtime, the execution time should decrease linearly with an increase in the number of execution resources. To test how RCUDA and RTM scale with more cores, a benchmark for each runtime was run multiple times with a different number
of simulated Rigel clusters. With more clusters, the memory bandwidth also increases. The **RCUDA** benchmark executes 1024 thread blocks each with 8 threads, while the **RTM** executes 8192 tasks. Figure 5.3 shows that performance improves for both RCUDA and RTM with an increase in the number of available clusters. In fact, the performance increase is almost linear for both runtimes, which means they are effectively making use of the additional execution resources. These results also show that there is no significant serialization point in either runtime.

5.2.4 Conclusion

The main difference between RCUDA and RTM is their workload distribution characteristics; both runtimes behave similarly in terms of scalability. The data shows that workloads using fine-grained tasks can be efficiently executed on Rigel using RCUDA. It is important to note that these results only consider the load distribution overhead of RCUDA, but in real CUDA code there are other factors to take into account, such as shared memory usage and thread block synchronization. The next section discusses the overhead of mapping CUDA constructs to Rigel.

5.3 RCUDA Performance

In this section, the source translation optimizations discussed in Chapter 4 are applied to a variety of benchmarks with different characteristics. Optimizations are evaluated based on how the performance of the CUDA kernel code changes when the optimization is applied.

5.3.1 Benchmarks

The seven benchmarks shown in Table 5.1 are used to evaluate RCUDA performance. With the exception of **SAXPY**, all benchmark codes were taken from external sources [22], [25] and were originally written to be executed on a GPU. The CUDA kernel source code for **SAXPY** is listed in Appendix B. Table 5.1 lists data sizes and characteristics for all benchmarks. The data sizes were chosen to represent typical GPU workloads, and the input data was taken from the original sources where possible.
Table 5.1: Characteristics of benchmarks used for evaluating performance of RCUDA, including the data set size, number of kernels, the dimensions of the thread blocks and whether shared memory is used in the original code.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Set</th>
<th># Kernels</th>
<th>Thread Block Size</th>
<th>S. Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolve</td>
<td>1024x1024</td>
<td>1</td>
<td>(16,16,1)</td>
<td>Yes</td>
</tr>
<tr>
<td>DMM</td>
<td>1024x1024</td>
<td>1</td>
<td>(16,16,1)</td>
<td>Yes</td>
</tr>
<tr>
<td>Histogram</td>
<td>2M</td>
<td>2</td>
<td>(192,1,1),(256,1,1)</td>
<td>Yes</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>512x512</td>
<td>1</td>
<td>(16,16,1)</td>
<td>Yes</td>
</tr>
<tr>
<td>MRI</td>
<td>8192,8192</td>
<td>2</td>
<td>(512,1,1),(256,1,1)</td>
<td>No</td>
</tr>
<tr>
<td>SAXPY</td>
<td>2M</td>
<td>1</td>
<td>(512,1,1)</td>
<td>No</td>
</tr>
<tr>
<td>Transpose</td>
<td>1024x1024</td>
<td>1</td>
<td>(16,16,1)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Convolve**

The **Convolve** benchmark applies a 2D image filter using a 5x5 kernel. **Convolve** uses a single kernel which executes 256 threads per thread block to compute a 12x12 section of the output matrix. The kernel code itself is divided into two pieces separated by a synchronization point. First, a 16x16 subsection of the input matrix is copied into shared memory. After the synchronization point, only the threads with an X and Y index value of less than 12 compute an output element. The results shown for **Convolve** are from a run using an input matrix with a size of 1024x1024. With a 1024x1024 input matrix, 7396 thread blocks are executed.

**DMM**

**DMM** performs a dense-matrix multiply of two matrices. In **DMM**’s kernel, each thread block calculates a 64x16 section of the output matrix using 256 threads (each thread generates four output values). The kernel uses blocking so that only a submatrix of each input matrix is accessed at a time. In order to minimize the cost of memory accesses, the submatrices are copied into shared memory, and dot products are calculated using the values stored in shared memory. After all threads have computed the dot products for current submatrices, the kernel copies the next submatrices into shared memory and repeats the process until the edge of the first input matrix is reached. Before exiting, the dot products are written back to the result matrix. Two synchronization points are used in the kernel. The first ensures that the shared
memory is read only when all threads in the block have finished initializing the shared array values. The second synchronization point guarantees that the shared memory is updated only after all threads finish computing the dot products for the current submatrices. The results for DMM shown are from multiplying two 1024x1024 input matrices which uses a total of 1024 thread blocks.

Histogram

Histogram generates a 256 bin histogram for a given input array. The benchmark uses two kernels: the first kernel is used to calculate partial histograms, and the second merges all of the partial histograms together to create the final output histogram. Both kernels fill a shared array with a portion of the input array and then calculate the histogram from the shared array. The results for Histogram are from a run using an input array with 2,097,152 elements, which executes 240 thread blocks for the first kernel and 256 thread blocks for the second kernel.

Mandelbrot

The Mandelbrot benchmark generates a Mandelbrot set. The Mandelbrot kernel is different from the other benchmarks because it is heavily compute bound and does not use a large input data set. Additionally, the load distribution is implemented in the kernel code itself. A single thread in a thread block fetches the next index value through an atomic increment on a global variable. If the index is valid, all threads in the block calculate a pixel value. Threads exit when no more work remains. The host code creates a grid with a size equal to the number of Streaming Multiprocessors. On Rigel a grid size of 128 is used, which is equal to the number of clusters. The results for Mandelbrot are from creating a 512x512 output image by using 128 thread blocks with 256 threads each.

MRI

MRI performs medical image construction as described in [26]. MRI uses two kernels, one to initialize data, and one to do the actual computation. The
first kernel is used to initialize two arrays and uses 512 threads per block where each thread initializes at most two output values. In the second kernel, each thread loops over a portion of the input array and outputs two values. Neither kernel uses synchronization or shared memory. The results for MRI are from generating 8192 pixels with 8192 samples. The first kernel uses a total of 16 thread blocks while the second kernel uses 32 thread blocks.

SAXPY

The SAXPY benchmark implements SAXPY from BLAS. SAXPY uses a single kernel in which each thread block calculates a 4096 element portion of the output array. Every thread block contains 512 threads, and each thread computes eight output elements. The results of SAXPY are from a run using two input arrays, each with 2,097,152 elements which uses 4096 thread blocks.

Transpose

Transpose outputs the transpose of an input matrix in a separate matrix; it does not perform the matrix transformation in-place. Only a single kernel is used, and each thread block performs the matrix transpose for a 16x16 section of the original matrix. Individual threads move a single element from the input matrix to the output matrix. Bounds checking is used so that the matrix size does not have to be a multiple of 16. Transpose was evaluated using a 1024x1024 input matrix, which requires 4096 thread blocks, each with 256 threads.

5.3.2 Baseline Performance

Figure 5.4 shows the normalized speedup of the naïve translation on Rigel over NVIDIA’s Tesla. These results show that the code translation process is sound and does not cause a dramatic performance variation when moving from the Tesla GPU to Rigel, as Rigel has a peak FLOP rate of 1.1 times that of the GPU. These results indicate a starting point towards performance portability, but one should also consider these numbers vs. hand-coded RCUDA as described in Section 5.4.
Figure 5.4: Baseline speedup of naïve translation on Rigel over NVIDIA Tesla T10.

Figure 5.5: Speedup over naïve translation with optimizations applied.
5.3.3 Optimizations

The optimizations discussed in Chapter 4 are applied individually to each benchmark where applicable. The combination of optimizations that result in the fastest code for a particular benchmark are used to create an optimal version. Figure 5.5 shows the speedup of each configuration over the naïve translation.

Kernel Transformations

Shared memory removal is applied to the Convolve, DMM and Transpose benchmarks. Histogram and Mandelbrot use shared memory only to hold temporary values, making our shared memory optimization inapplicable.

For Convolve, DMM and Transpose, removing the shared memory accesses also allowed for the elimination of the synchronization in each of the benchmarks’ kernels. The performance of Convolve and Transpose benchmarks improved since the redundant memory accesses and synchronization were eliminated after shared memory removal. DMM is the only benchmark for which the optimization did not improve the execution time. The mapping function generated for DMM is complex, requiring costly integer multiplications. Additionally, the mapping function for DMM adds register pressure, forcing the code to use stack memory and thus further reducing performance.

Runtime Optimizations

All benchmarks except Convolve and SAXPY show improved performance when using static scheduling of threads. Convolve is the only benchmark where the amount of work varies greatly between threads. In Convolve, only 144 of the 256 CUDA threads in the thread block actually compute an output value. The irregular code means that some Rigel cores end up with significantly less work with static work partitioning, leading to load imbalance. In the case of SAXPY, the execution time increases by 10% primarily due to how the RCUDA runtime statically partitions work. When using static work partitioning, RCUDA calculates the beginning and end thread index for each Rigel core during the kernel launch stage. Since the thread block execution time is relatively short for SAXPY, the longer kernel launch time resulted in an overall performance reduction.
Table 5.2: Optimizations applied to create optimal configurations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolve</td>
<td>Shared Memory Removal, Thread Fusing</td>
</tr>
<tr>
<td>DMM</td>
<td>Static Work Partitioning, Thread Fusing</td>
</tr>
<tr>
<td>Histogram</td>
<td>Static Work Partitioning, Thread Fusing</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>Static Work Partitioning, Thread Fusing</td>
</tr>
<tr>
<td>MRI</td>
<td>Thread Fusing</td>
</tr>
<tr>
<td>SAXPY</td>
<td>Thread Fusing</td>
</tr>
<tr>
<td>Transpose</td>
<td>Thread Fusing, Shared Memory Removal</td>
</tr>
</tbody>
</table>

Thread fusing improves the performance of all benchmarks. In each case, multiple CUDA threads can be combined, removing redundant calculations. It seems counterintuitive that SAXPY benefits the most from thread fusing but slows down with static work partitioning. After thread fusing, each fused SAXPY thread executes four CUDA threads, which means that the CUDA thread indices are fetched once for every four CUDA threads. Additionally, thread fusing does not require the computation of a start and end thread index since the CUDA thread indices are fetched on demand during runtime.

Optimal Benchmark Configurations

Table 5.2 shows the optimal version of each benchmark, the combination of optimizations that result in the shortest runtime. As shown in the graph in Figure 5.5, there are diminishing returns when combining optimizations. Some optimizations conflict with each other. For example, thread fusion and static work partitioning reduce the granularity of work by combining CUDA threads. Applying both thread fusion and static work partitioning further reduces granularity, which can have a negative impact on load balance if threads do not perform a uniform amount of work. While the results show that thread fusion and static work partitioning sometimes do not perform well together, it is clear that if shared memory removal improves performance by itself, then it also improves performance when combined with other optimizations. Shared memory removal combines effectively with other optimizations because it removes code in the kernel itself and does not duplicate the functionality of other optimization transforms.
Figure 5.6: RCUDA runtime breakdown for na"ive translations.

Figure 5.7: RCUDA runtime breakdown of optimal configurations.
5.3.4 RCUDA Runtime Overhead

This section contains an analysis of the runtime overhead of the RCUDA framework on Rigel. The runtime is broken down into five categories: (1) **Kernel**, the time spent executing the computation portion. (2) **Thread ID**, the overhead of generating the CUDA thread indices when dynamic load balancing is used. (3) **Sync**, the time spent in the `syncthreads()` call. (4) **Barrier**, the amount of time cores spend waiting for kernel execution to complete, representing load imbalance. (5) **Other**, all other overheads including runtime initialization, thread block fetch, and host code.

Naïve Translation

Figure 5.6 shows the RCUDA runtime overhead for naïve benchmark translations. From the chart, it is apparent that thread index generation is quite expensive, particularly for kernels with two-dimensional thread blocks. For one-dimensional thread blocks, the CUDA thread indices are generated from a count of remaining threads. However, the conversion from a one-dimensional count to a two-dimensional index requires a significant amount of computation that can be comparable to the total work of shorter CUDA kernels such as Transpose and Convolve. Additionally, thread indices are generated twice in Transpose and Convolve due to a single synchronization point in each kernel. The time spent in the `syncthreads()` call is low, even though it is implemented in software. For Histogram and SAXPY the barrier constitutes roughly 20% of the runtime. The Histogram code does not generate a large enough grid to utilize the entire chip, so some cores only wait in the barrier without executing any kernel code. SAXPY has a very short kernel, so load imbalance contributes to the high barrier cost. The barrier makes up the majority of MRI’s runtime because of load imbalance. The first kernel utilizes 16 clusters while the second kernel only uses 32 of the 128 available clusters.

Optimal Configurations

Figure 5.7 shows the runtime overheads for the optimized benchmarks. The optimizations only benefit kernel execution, including the kernel code itself,
thread index generation and possibly synchronization if it can be removed. Other sources of overhead, such as thread block fetch and barrier account for a higher percentage of the execution time in the optimal benchmarks because the overall runtime decreases.

For Convolve, the most significant source of runtime overhead was thread index generation, and it has been reduced from around 10% to less than 2%. Thread index generation was the major source of overhead for DMM as well. The optimal version of DMM has negligible thread index generation overhead since the atomic_get_next_thread() calls were replaced with for loops by static work partitioning. For Histogram, thread fusing and static work partitioning get rid of the thread index generation overhead, but the barrier accounts for a significant amount of runtime, due to workload imbalance. Thread index generation accounts for almost 25% of the runtime with the naïve translation, but only 3% in the optimal configuration. In the naïve translation for Mandelbrot, thread index generation makes up almost 20% of the runtime, but is no longer apparent in the optimal version due to static work partitioning. For MRI and SAXPY, the barrier accounts for a significant amount of the runtime overhead in both the naïve and optimal cases because the source-level optimizations cannot improve workload imbalance. Static work partitioning removed the overhead caused by thread index generation for Transpose.

5.4 DMM Case Study of Performance Portability

This section contains an evaluation of several dense-matrix multiplication implementations running on the GPU and Rigel. As shown in Figure 5.8, the benchmarks include the original CUDA version (CUDA-GPU), a CUDA version optimized for the Rigel architecture (CUDA-Rigel), the original CUDA version with RCUDA optimizations (RCUDA-Opt) and finally a native version for Rigel (RTM-Rigel) which uses RTM. On Rigel, the CUDA-GPU, CUDA-Rigel and RCUDA-Opt benchmarks use RCUDA. Here, each benchmark multiplies two 512x512 matrices.
5.4.1 CUDA-GPU

The CUDA-GPU benchmark is written in CUDA and is tuned for an NVIDIA G80 GPU [27]. The code is translated by RCUDA so that it can be executed on Rigel. Both the GPU and Rigel use identical kernel code, however the host code is different.

Work Distribution

CUDA-GPU uses a thread block size of 16x16 and each CUDA thread computes four elements of the output matrix. Given a 512x512 input matrix, a grid size of 8x32 is used. In this case, there are a total of 256 thread blocks.

CUDA Kernel

The kernel for CUDA-GPU uses a shared memory buffer for both input matrices. The kernel code iterates over the entire matrix as shown in Figure 5.9(a). At the beginning of the loop, a 16x16 block of the first input matrix is placed into shared memory. Next, a 64x16 portion of the second input matrix is copied into another shared memory array. Then, the actual dot product calculations are performed. The code is unrolled by a factor of 16 so that
Figure 5.9: Matrix multiplication techniques used by the three dense-matrix multiplication benchmarks.
the values in the shared arrays can be consumed without an additional loop. The usage of shared memory requires a synchronization point between the initialization of the shared memory and the actual dot product calculation. A second synchronization point after the dot product calculation ensures that the calculations are complete before new sub-matrices are loaded for the next iteration.

5.4.2 RCUDA-Opt

RCUDA-Opt is the CUDA-GPU benchmark with the static work partitioning and thread fusing optimizations applied. Compared to CUDA-GPU, the optimized kernel reduces the runtime overhead by statically assigning threads. The thread index generation accounts for almost 14% of the total execution time for CUDA-GPU but is negligible in RCUDA-Opt because the thread fetch calls are converted into a for loop that iterates over the thread indices.

5.4.3 CUDA-Rigel

The CUDA-Rigel benchmark is written in CUDA and is tuned for Rigel. In order to get the best performance possible, CUDA-Rigel uses a one-dimensional thread block with only eight threads so that one CUDA thread maps to one Rigel core. Additionally, no shared memory or thread block synchronization is used.

Work Distribution

Blocking is used to calculate the matrix multiply. Each thread block computes an 8x8 block of the resultant matrix, as demonstrated in Figure 5.9(b). There are 8 threads in each thread block and each CUDA thread computes an 8x1 section. Since there are only 8 CUDA threads, only one thread is mapped to a Rigel core during a thread block’s execution, minimizing the overhead of the RCUDA runtime. An additional optimization is that only a portion of the input matrices are used at the same time. Implicit grouping occurs because a thread block uses its index (X and Y value) to compute which block of the matrix it computes and the RCUDA runtime executes
blocks in a defined order. Figure 5.10 shows the ordering of a 32x32 matrix. The number in each box is the order in which blocks are executed.

CUDA Kernel

Each kernel independently calculates its portion; there is no synchronization with other CUDA threads or use of shared memory. The kernel computes eight dot products simultaneously by fetching the data horizontally from the first input matrix and vertically from the second. The inner loop is manually unrolled by a factor of 8 so that there are eight fetches from the second input matrix but only one from the first input matrix in every iteration of the loop.

5.4.4 RTM-Rigel

The RTM-Rigel benchmark is written in C and uses the Rigel Task Model (RTM). The code is designed for Rigel and is heavily tuned so that the compiler generates code that rivals hand-written assembly.

Workload Distribution

RTM-Rigel uses blocking to perform the matrix multiplication as shown in Figure 5.9(c). The benchmark only works on a subset of the matrix by blocking the work up into two different levels. First, the input matrices are divided into 256x256 sections. For each 256x256 section, tasks are enqueued.
in parallel with each task to calculating dot products for an 8x8 block in the 256x256 section, making for 1024 tasks for each parallel enqueue. Each task performs a maximum of 64 iterations. Once all 1024 tasks complete, additional tasks are enqueued to perform the next 64 iterations, if required. The partial results of the tasks are added to the partial results of the task that performed the previous 64 iterations for the same 8x8 block. The process is repeated until all dot products are calculated. For example, given input matrices with a length and width of 256, the work is divided into 256x256 sections. In this case there is only one section. Next, the work is divided further since tasks only iterate over 64 elements. In this case four separate parallel enqueues are required because 256 divided by 64 is four. Since tasks are enqueued in groups of 1024, and four parallel enqueues are required, 4096 tasks are executed.

RTM Task

As described above, a task computes an 8x8 block of the larger 256x256 section, but only iterates a maximum of 64 times as opposed to the full matrix as in the other benchmarks. Elements from the matrices are fetched eight at a time. The inter loop of the task computes the sum of one element in the 8x8 section.

5.4.5 Results

On Rigel, both RCUDA-Opt and CUDA-Rigel perform better than the native implementation (RTM-Rigel). The performance difference is due to the uniform nature of the DMM computation. The less restrictive programming model provided by RTM adds extra overhead for features that are not used, such as task scheduling and the ability to enqueue work at any time. The CUDA-Rigel implementation performs the best, due to its memory access pattern, in which all accesses to the input and output matrices are cache aligned. For each memory access of a matrix, the entire cache line is used. However, CUDA-Rigel achieves quite poor performance on the GPU because the block size is not large enough to fully utilize the Streaming Multiprocessors, and the kernel accesses global memory directly for computations. The RTM-Rigel benchmark follows a similar approach as CUDA-Rigel, but uses
a finer-grained blocking resulting in more tasks which in turn increases the overhead of the software runtime.

These results show that it is possible to take high-performance CUDA code originally written for a GPU and still get good performance on Rigel after translating to code and applying optimizations. Given that the GPU-tuned version, RCUDA–Opt, obtains 94% of the performance of the Rigel-tuned version CUDA–Rigel, the right choice for performance portability is to use RCUDA–Opt on Rigel and CUDA–GPU on the GPU.
CHAPTER 6

CONCLUSION AND FUTURE WORK

Achieving performance portability across a MIMD architecture and a GPU is possible, but only for SPMD codes originally tuned for a GPU. When optimizing CUDA code for a GPU, a programmer must adhere to constraints of a more restrictive execution model, creating code with characteristics that can be leveraged to obtain good performance on other architectures. Such information may also be available in CUDA code written for Rigel, a MIMD architecture, but if so, we have not been able to identify it. Given that a platform like Rigel allows MIMD execution, we expect some algorithms will be hard to map efficiently onto a GPU. Applications requiring these algorithms are likely to be written using a programming model other than CUDA, making it more difficult to attain performance portability with Rigel code.

CUDA code tuned for a GPU can be efficiently mapped onto Rigel using a source-to-source translation process with optimizations that leverage the characteristics of high-performance CUDA code. When source-level optimizations are applied, there is a significant speedup across several benchmarks. The source-level transformations do not change the underlying algorithm of the original SPMD code, and it is reasonable to assume these transformations can be automated using standard compiler analysis.

This work is not limited to the SPMD codes written in CUDA. OpenCL [28] is a parallel programming model that is platform independent and designed to work on a variety of architectures including CPUs, GPUs and heterogeneous architectures. Kernels written in OpenCL are comparable to CUDA kernels in the sense that they both target SPMD programming models with a similar memory model. Due to the commonality with CUDA code, the optimizations presented in this thesis would likely work for OpenCL codes as well.

The software transformations used by RCUDA can be extended to other platforms. For example, some heterogeneous architectures combine general
purpose processing cores with specialized accelerator units. If a software application is limited by the accelerator unit resources, it is advantageous to use the general purpose processor to execute the accelerator code. The software transforms used in RCUDA can be used to speed up the accelerator code on the generic cores lacking hardware support for CUDA constructs. Two examples of applicable architectures are the Cell [29] and the AMD Fusion [30].

RCUDA provides a lightweight software task distribution system that efficiently distributes fine-grained tasks across the system. These features could be beneficial to architectures that rely on software to distribute tasks such as Intel’s Larabee [31] and Single-chip Cloud Computer [32].
CHAPTER 7

REFERENCES


This appendix contains the RCUDA runtime code which executes CUDA kernels on Rigel after the source translation process.

```c
// Cluster Variables
typedef struct cluster_state_s {
    uint32_t MaxThreads; // Number of CUDA threads in current thread block
    uint32_t ThreadCount; // Number of CUDA threads left to execute
    volatile uint32_t Sense; // Syncthreads barrier sense
    volatile uint32_t WaitCount; // Count of cores waiting in Syncthreads barrier
    volatile uint32_t DequeueWaitCount; // Count of cores waiting to fetch a thread block
    volatile uint32_t DequeueSense; // Dequeue barrier sense
    volatile uint32_t CurrentBlock; // Index of current thread block
    uint32_t KernelLaunchCount; // Number of kernels launched
} cluster_state_t;

// Global Variables
uint32_t GlobalBlockCount = 0; // Number of thread blocks left to execute
uint32_t GlobalThreadCount = 0; // Number of CUDA threads in each thread block
uint32_t GlobalKernelLaunchCount = 0; // Number of kernels launched by core 0
dim3 GlobalBlockDim; // Dimensions of thread blocks for current kernel
dim3 GlobalGridDim; // Dimensions of the grid for current kernel

#define MAX_NUM_CLUSTERS 128

cluster_state_t cluster_state[MAX_NUM_CLUSTERS] __attribute__ ((aligned (32)));

// _rcuda_init: initializes the RCUDA library
void __rcuda_init() {
    uint32_t ThreadNum;
    uint32_t NumThreadsPerCluster;

    ThreadNum = RigelGetThreadNum();
    NumThreadsPerCluster = RigelGetNumThreadsPerCluster();

    if ((ThreadNum % NumThreadsPerCluster) == 0) {
        uint32_t index;
        RIGEL_GET_CLUSTER_NUM(index);
        cluster_state[index].MaxThreads = 0;
        cluster_state[index].ThreadCount = 0;
        cluster_state[index].Sense = 0;
        cluster_state[index].WaitCount = 0;
    }
}
```

72
void __rcuda_kernel_launch(__cuda_kernel_function kernel_function, dim3 grid, dim3 block)
{
    dim3 blockIdx;
    uint32_t ThreadNum;
    uint32_t ClusterNum;
    uint32_t NumThreadsPerCluster;

    ThreadNum = RigelGetThreadNum();
    ClusterNum = RigelGetClusterNum();
    NumThreadsPerCluster = RigelGetNumThreadsPerCluster();
    cluster_state_t *cs = &cluster_state[ClusterNum];

    // Thread 0 does the setup for all the cores
    if (ThreadNum == 0) {
        uint32_t TotalThreads = block.x * block.y * block.z;
        uint32_t TotalBlocks = grid.x * grid.y * grid.z;
        // Do some error checking
        if (grid.x <= 0 || grid.y <= 0 || grid.z <= 0)
            assert(0 && "grid values must be greater than 0");
        if (block.x <= 0 || block.y <= 0 || block.z <= 0)
            assert(0 && "block values must be greater than 0");
        if (TotalThreads > 512)
            assert(0 && "Error, too many threads in kernel launch
");
        cs->MaxThreads = TotalThreads;
        cs->KernelLaunchCount++;

        // Update the global block and grid
        GlobalBlockDim = block;
        GlobalGridDim = grid;

        RigelGlobalStore(TotalThreads, GlobalThreadCount);
        RigelGlobalStore(TotalBlocks, GlobalBlockCount);
        RigelGlobalStore(cs->KernelLaunchCount, GlobalKernelLaunchCount);
    } else if (ThreadNum % NumThreadsPerCluster == 0) {
        int32_t KernelLaunchCount;
        // Increment the launch count
        cs->KernelLaunchCount++;
        // Wait for thread 0 to update the global block count
        do {
            RigelGlobalLoad(KernelLaunchCount, GlobalKernelLaunchCount);
        } while(KernelLaunchCount < cs->KernelLaunchCount);

        RigelGlobalLoad(cs->MaxThreads, GlobalThreadCount);
    }
    // Cluster Barrier
```c
{
  uint32_t StartSense = cs->Sense;
  uint32_t NewCount;
  // do an atomic increment on the wait count
  ATOMIC_CLUSTER_INC(NewCount, cs->WaitCount);

  if (NewCount == NumThreadsPerCluster) {
    // Update the wait count
    cs->WaitCount = 0;
    // Update the sense
    cs->Sense = !StartSense;
  } else {
    // Wait for the sense to be updated
    while (cs->Sense == StartSense);
  }
}

// Execution: Each cluster fetches and executes a thread block while work remains
while (1) {
  // Fetch next thread block
  int32_t CurrentBlock = dequeue_block();
  if (CurrentBlock < 0)
    break;

  // Calculate the block index
  blockIdx.y = (int)((float)(CurrentBlock))/((float)(GlobalGridDim.x));
  blockIdx.x = CurrentBlock - (int)(((float)(blockIdx.y)*((float)GlobalGridDim.x)));
  blockIdx.z = 0;

  // Call into the kernel code
  (*kernel_function)(blockIdx, GlobalBlockDim, GlobalGridDim);
}

// Global Barrier: Wait for all clusters
{
  int32_t BlockCount;
  int32_t NumClusters;
  NumClusters = -NumClusters;
  do {
    RigelGlobalLoad(BlockCount, GlobalBlockCount);
  } while(BlockCount != NumClusters);
}
```

```c
#define ATOMIC_CLUSTER_INC(Result, Counter) \
  asm volatile ( \
    "L%=_inc_count: 
    ldl %0, %1 
    addi %0, %0, 1 
    stc $1, %0, %1 
    beq $1, $0, L%=_inc_count 
    :="&r"(Result) \ 
    : "r"(&Counter)\ 
    : "1" \ 
  );
```

```c
#define ATOMIC_CLUSTER_DEC(Result, Counter) \
  asm volatile ( \
```
L%=_inc_count:
  ldl %0, %1
  subi %0, %0, 1
  stc $1, %0, %1
  beq $1, $0, L%=_inc_count

#define RIGEL_ATOMIC_GET_NEXT_TID(Result) \
(uint32_t ClusterNum; \
 RIGEL_GET_CLUSTER_NUM(ClusterNum); \ 
 cluster_state_t *cs = &cluster_state[ClusterNum]; \
 ATOMIC_CLUSTER_DEC(Result, cs->ThreadCount);

// atomic_get_next_tid_1d: gets the next thread index for 1d thread blocks
int32_t atomic_get_next_tid_1d(dim3 *threadIdx) {
  RIGEL_ATOMIC_GET_NEXT_TID(threadIdx->x);
  threadIdx->y = threadIdx->z = 0;
  return threadIdx->x;
}

// atomic_get_next_tid_2d: gets the next thread index for 2d thread blocks
int32_t atomic_get_next_tid_2d(dim3 *threadIdx, int xDim) {
  int32_t index;
  RIGEL_ATOMIC_GET_NEXT_TID(index);
  threadIdx->y = (int)(((float)index)/((float)xDim));
  threadIdx->x = index - (int)(((float)xDim)*((float)(threadIdx->y)));
  threadIdx->z = 0;
  return index;
}

// __rigel_sync_threads: this implements a cluster barrier
void __rigel_sync_threads() {
  uint32_t NewCount;
  uint32_t NumThreadsPerCluster;
  uint32_t ClusterNum;
  ClusterNum = RigelGetClusterNum();
  cluster_state_t *cs = &cluster_state[ClusterNum];
  uint32_t StartSense = cs->Sense;
  ATOMIC_CLUSTER_INC(NewCount, cs->WaitCount);
  NumThreadsPerCluster = RigelGetNumThreadsPerCluster();
  // see if we need to update the sense
  if (NewCount == NumThreadsPerCluster) {
    cs->ThreadCount = cs->MaxThreads; // Reset the cluster thread count
  }
}
cs->WaitCount = 0; // update the wait count

cs->Sense = !StartSense; // update the sense
} else {
    while (cs->Sense == StartSense); // wait for the sense to change
}
}

// dequeue_block: Fetches a new thread block for cluster
int32_t dequeue_block() {
    uint32_t NewCount;
    uint32_t NumThreadsPerCluster;
    uint32_t ClusterNum;

    ClusterNum = RigelGetClusterNum();

    // Get the cluster state
cluster_state_t *cs = &cluster_state[ClusterNum];

    // Get the current sense
    uint32_t StartSense = cs->DequeueSense;

    // Do an atomic increment on the wait count
    ATOMIC_CLUSTER_INC(NewCount, cs->DequeueWaitCount);

    // See if we need to do the dequeue and flip the sense
    if (NewCount == 1) {
        // Do the dequeue
        RigelAtomicDEC(cs->CurrentBlock, GlobalBlockCount);

        NumThreadsPerCluster = RigelGetNumThreadsPerCluster();

        // Wait for all threads before we update shared data
        while (cs->DequeueWaitCount != NumThreadsPerCluster);

        cs->ThreadCount = cs->MaxThreads; // Set the threads count

        cs->DequeueWaitCount = 0; // Update the wait count

        cs->DequeueSense = !StartSense; // Update the dequeue sense
    } else {
        // Wait for the sense to be updated
        while (StartSense == cs->DequeueSense);
    }

    // Return the result (same for each thread in cluster)
    return cs->CurrentBlock;
}
APPENDIX B

CUDA KERNEL CODE LISTING

This appendix contains all of the custom CUDA kernel code for the benchmarks discussed in Section 5.3. The code listed was used as the input to the RCUDA source-to-source translator.

B.1 DMM (CUDA-Rigel)

typedef struct {
    float f0, f1, f2, f3, f4, f5, f6, f7, f8;
} float8;

#define FLOAT8_DOT(a,b) \
    sum0 += a->f0 * b->f0; \
    sum1 += a->f0 * b->f1; \
    sum2 += a->f0 * b->f2; \
    sum3 += a->f0 * b->f3; \
    sum4 += a->f0 * b->f4; \
    sum5 += a->f0 * b->f5; \
    sum6 += a->f0 * b->f6; \
    sum7 += a->f0 * b->f7;

__global__ void
matrixMul( float* C, float* A, float* B) {
    int i, indexA, indexB;

    indexA = blockIdx.y*8 + threadIdx.x) * WidthA;
    indexB = blockIdx.x*8;

    float sum0 = 0.0f;
    float sum1 = 0.0f;
    float sum2 = 0.0f;
    float sum3 = 0.0f;
    float sum4 = 0.0f;
    float sum5 = 0.0f;
    float sum6 = 0.0f;
    float sum7 = 0.0f;

    float8 *a = (float8*)(A + indexA);
    float8 *b = (float8*)(B + indexB);
for (i = 0; i < WidthA; i += 8) {
    FLOAT8_DOT(a,b); b += WidthA/8;
    FLOAT8_DOT(a,b); b += WidthA/8;
    FLOAT8_DOT(a,b); b += WidthA/8;
    FLOAT8_DOT(a,b); b += WidthA/8;
    FLOAT8_DOT(a,b); b += WidthA/8;
    FLOAT8_DOT(a,b); b += WidthA/8;
    FLOAT8_DOT(a,b); b += WidthA/8;
    a++;
}

float8 *c = (float8*)(C+indexA+indexB);

c->f0 = sum0;
c->f1 = sum1;
c->f2 = sum2;
c->f3 = sum3;
c->f4 = sum4;
c->f5 = sum5;
c->f6 = sum6;
c->f7 = sum7;
}

B.2 SAXPY

#define SCALE_A 2.1f
#define SCALE_B 1.0f

__global__ void Saxpy(float* C, float* A, float* B) {
    int index = (blockDim.x*blockIdx.x + threadIdx.x)*8;

    C[index+0] = SCALE_A*A[index+0] + SCALE_B*B[index+0];
    C[index+1] = SCALE_A*A[index+1] + SCALE_B*B[index+1];
    C[index+2] = SCALE_A*A[index+2] + SCALE_B*B[index+2];
}