TECHNIQUES TO MAXIMIZE MEMORY BANDWIDTH ON THE RIGEL COMPUTE ACCELERATOR

BY

WILLIAM J. TUOHY

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, 2011

Urbana, Illinois

Adviser:

Associate Professor Sanjay J. Patel

ABSTRACT

The Rigel compute accelerator has been developed to explore alternative architectures for massively parallel processor chips. Currently GPUs that use wide SIMD are the primary implementations in this space. Many applications targeted to this space are performance limited by the memory all, so comparing the memory system performance of Rigel and GPUs is desirable. Memory controllers in GPUs attempt to coalesce memory requests from separate threads to achieve high off-chip bandwidth. This coalescing can be achieved by the programmer if the address mapping bits are understood, so that neighboring threads create memory requests that do not conflict. MIMD hardware as implemented in Rigel avoids the SIMD costs of serialization of conditional execution paths and load imbalance from varying task lengths. These benefits to the execution hardware come at a cost of reduced memory bandwidth, however, as it is difficult or impossible to orchestrate the memory requests in a way that achieves perfect access patterns as can be done with SIMD hardware. When a program can be decomposed statically and the computation does not vary among threads, then Rigel can achieve bandwidth similar to that of a GPU - but these are precisely the types of programs for which SIMD hardware is well suited. When a dynamic task distribution scheme is used to improve load balance, or the computation runs for different amounts of time on different threads, memory bandwidth can suffer greatly as the access pattern is not likely to be well controlled. Thus, for the types of programs for which MIMD hardware is best suited, the memory bandwidth penalty may be significant and reduce the benefits of MIMD's flexible execution hardware.

To my parents, for their love and support.
To my new bride Sherry, for her unwavering support and encouragement.

ACKNOWLEDGMENTS

I would like to thank the entire Rigel team for being such a great group of people to work with. The many long discussions we had were very stimulating and enlightening. The professors involved in the project, Dr. Sanjay Patel and Dr. Steven Lumetta, were very encouraging and helpful. The National Center for Supercomputing Applications provided access to their GPU cluster for the class project that became Chapter 3. The *rigelsim* simulator used for all of the data collection in Chapter 4 was a true team effort, and I am thankful to have worked with such a strong group.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
BW	Bandwidth
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
FLOPS	Floating Point Operations Per Second
GDDR	Graphics Double Data Rate
GHz	Gigahertz
GPU	Graphics Processing Unit
HW	Hardware
ID	Identification
IO	Input Output
L1	Level 1 Cache
L2	Level 2 Cache
LLC	Last Level Cache
MIMD	Multiple Instruction Multiple Data
RISC	Reduced Instruction Set Computer
RTM	Rigel Task Model
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
SW	Software

CHAPTER 1

INTRODUCTION

The *memory wall* [1] in processor design refers to the fact that over time, the speed of main memory technology comprised of DRAM has not improved as fast as processor logic. Many important applications are now memory bound, meaning the execution time of a program is determined primarily by how fast the data can be moved in and out of memory and is relatively independent of processor speed. This work studies issues related to the memory wall in the realm of compute accelerators such as graphics processing units (GPUs) and the Rigel experimental architecture developed in the Department of Electrical and Computer Engineering at the University of Illinois. Accelerators are used for data intensive applications, and generally utilize more channels of GDDR memory than the main memory system to achieve much higher bandwidth than standard processors such as those from the x86 family.

Modern GPUs are primarily built with multiple wide SIMD processors. SIMD processors contain multiple execution units, or lanes, sharing a single program counter and instruction front end. SIMD is an optimization that allows very efficient computation on perfectly regular data-parallel applications. All execution units of a SIMD processor must perform the same operation on their particular datum. Since many interesting applications have inherent data parallelism, SIMD is very effective at exploiting this parallelism in an efficient manner. Significant portions of the hardware cost of a processor are amortized across multiple data lanes with no loss of performance on such programs. When the computation becomes more irregular, SIMD processors lose some of this efficiency. They use predication for conditional execution, and must serialize execution of different code paths generated by conditional branches. This serialization of conditional execution is one of the main costs of SIMD and can limit performance in some applications.

The lockstep execution of SIMD means that memory load and store requests are issued by each lane simultaneously. If these requests access differ-

ent DRAM banks and channels, they will not conflict with each other and the hardware can service them in parallel. If they do conflict, throughput will be reduced because the memory system is not fully utilized. To avoid such conflicts, the requests from adjacent lanes must have addresses that map to different banks and channels, which requires the programmer to understand the interleaving scheme used in the memory controller hardware. Recent work in GPU programming [2] has shown that by exploiting knowledge of the memory controller address mapping, data structures can be created in a way that maximizes memory bandwidth. Memory access coalescing has been shown to be one of the most important techniques for overcoming the memory wall in GPU applications.

The Rigel accelerator, by contrast, uses a MIMD execution style where each execution unit has independent instruction front ends. There is no need for predication as each lane can follow its own code path. The processor lanes do not operate in lock step, and memory requests are generated and issued independently. Task distribution can be done either statically or dynamically, but is controlled by the application program in either case. Thus MIMD hardware does not suffer from the SIMD cost of serialization of conditional execution. The main memory system is similar to that of GPUs, consisting of multiple channels and multiple banks interleaved based on address bit mapping. With MIMD execution the memory request pattern is likely to be quite different. This work shows that there may in fact be a cost to this MIMD execution model because memory requests are not as well formed and controlled as in SIMD. This means it can be more difficult for the programmer to orchestrate memory movement in a way that achieves the high bandwidth realizable on GPUs. The contributions of this work include:

- Showing that high bandwidth can be achieved on MIMD with careful orchestration using a static partitioning scheme when the computation is regular
- Showing the cost in terms of bandwidth reduction that occurs when either the computation or task distribution scheme introduces variation in the request stream

The rest of this document is organized as follows: Background information including related work and an overview of the Rigel architecture are

given in Chapter 2; GPU memory bandwidth issues are described in Chapter 3; Rigel memory system and bandwidth issues are explored in Chapter 4; Chapter 5 compares the results of the previous two chapters, and compares the techniques applicable to Rigel and modern GPUs; Chapter 6 gives the conclusions of this study.

CHAPTER 2

BACKGROUND

Compute accelerators such as GPUs implement complex main memory controllers to allow high bandwidth to be achieved by the massive number of physical threads running on the chip. Multiple channels, multiple banks per channel, and interleaving techniques allow multiple simultaneous requests from multiple processor threads to be serviced in parallel. To maximize the bandwidth a program achieves, it is necessary to understand the interleaving mechanism so that the data can be divided among threads in a way that reduces the number of requests that create resource conflicts at the memory interface. Prior work on data layout transformations for grid computations on GPUs [2] has shown the benefits of matching the data layout to the memory address interleaving. By properly blocking the grid data to match the address interleaving scheme and thread block size, very high memory request parallelism can be achieved. It is necessary to understand the address interleaving being used in the memory controller, and to design the program to exploit this.

This work was extended in a ECE598-HK class project, which attempted to reverse-engineer the address interleaving steering bits for several different GPUs from both NVIDIA and ATI. Micro-benchmarks were used to gather bandwidth data with varying address decomposition among threads. The detailed results of this study are shown in Chapter 3. The Rigel accelerator architecture has some similarities to GPUs, but also many important differences. Until recently GPUs did not employ on-chip cache memory, opting instead for software managed scratchpad local memories. The more recent Fermi architecture from NVIDIA employs cache to front main memory, similar to Rigel's global cache. Some of the techniques that give predictability on the older GPUs are not needed or do not work the same way on Fermi. GPU hardware can coalesce memory requests from different SIMD lanes very effectively. Rigel has a similar number of active threads, but the MIMD hard-

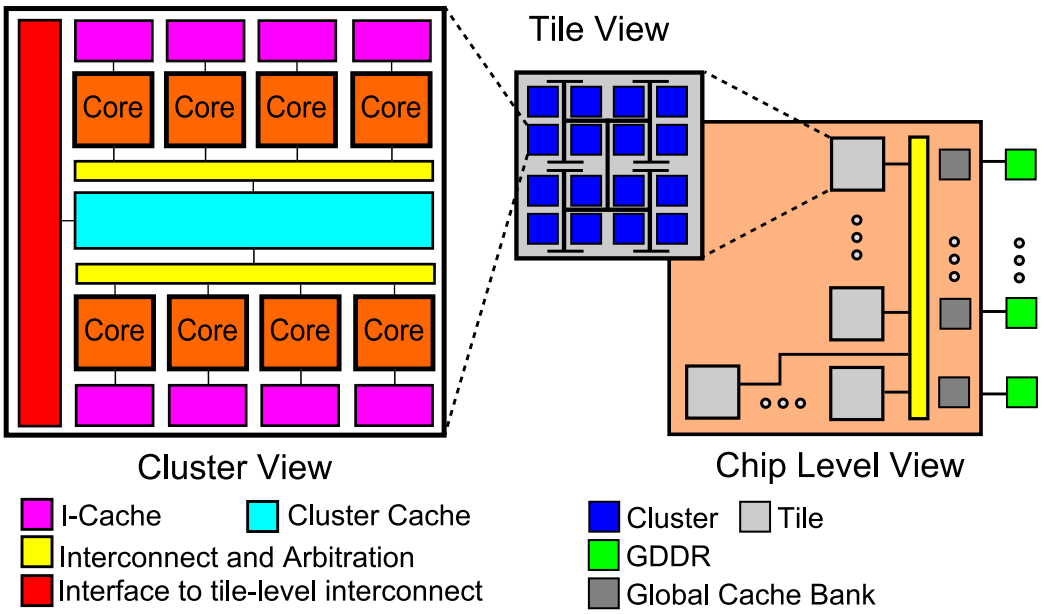

Figure 2.1: Architecture of the Rigel many-core accelerator.

ware may not create the same opportunity to achieve non-interfering accesses during runtime.

The Rigel compute accelerator architecture is a research project used to study massively parallel multiprocessor chip design. Rigel has been designed as a MIMD-based multiprocessor, as opposed to the SIMD hardware that GPUs typically utilize. The Rigel project was intended to study whether MIMD execution could remove the performance costs of SIMD execution on irregular applications, and still be implemented without giving up too much of the SIMD energy efficiency seen on perfectly data-parallel workloads. Rigel publications have shown the design to be physically feasible, and have demonstrated good scalability and fairly low overhead with a software-based dynamic task distribution scheme [3]. Until now however we have not studied the memory bandwidth cost to MIMD execution compared to SIMD, and whether this cost could be overcome.

2.1 Rigel Architecture

A block diagram of Rigel is shown in Figure 2.1. The fundamental processing element of Rigel is an area-optimized dual-issue in-order execution core. Each core has a fully pipelined single-precision floating-point unit, independent

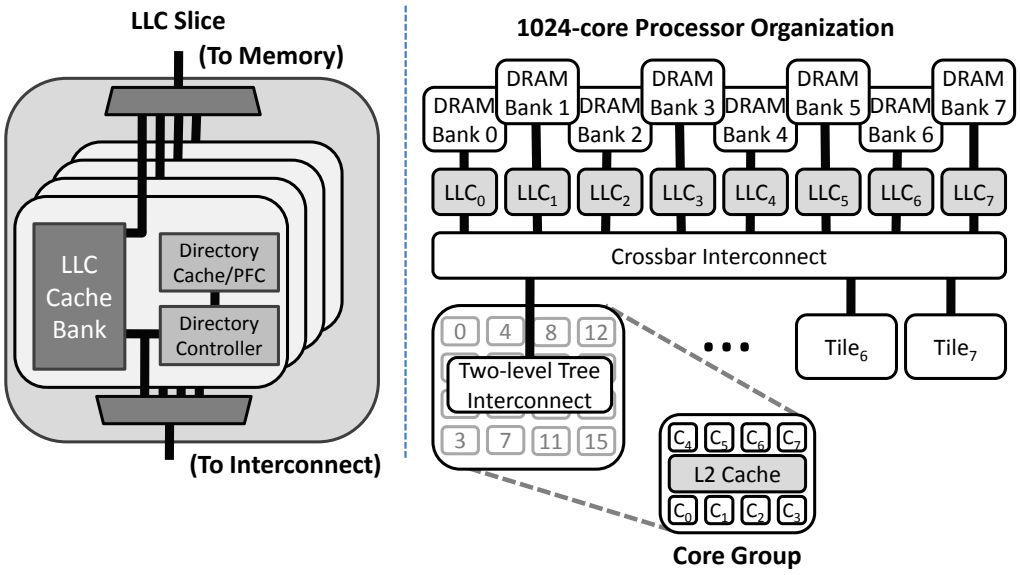

Figure 2.2: Rigel memory system overview.

fetch unit, and executes a 32-bit RISC-like instruction set with 32 general-purpose registers. Cores are organized into *Clusters* of eight cores attached to a shared, unified cache called the *Cluster Cache*.

To make the on-chip interconnect manageable, up to 16 clusters are physically grouped into a tile. Clusters within a tile share interconnect resources, and there is a single interface from a tile to the rest of the chip. Tiles are distributed across the chip and are attached to *Global Cache* banks via a crossbar interconnect. The global cache provides buffering for high-bandwidth memory controllers and is the point of coherence for memory.

The Rigel memory hierarchy is shown in Figure 2.2. There are 128 cluster caches of 64 kB each, labelled as *L2 Cache* in the figure. Each tile is connected to a crossbar with 8 ports on the processor side (one per tile), and 32 ports on the memory side. The Global Cache is implemented as 32 banks that front 8 GDDR 32-bit memory channels. Four Global Cache banks are fixed to each DRAM channel. The physical memory address space is striped across the memory channels. This high number of memory controllers with per-DRAM-bank queues, and global cache banks, enable significant memory level parallelism for hiding DRAM access latency. Hardware prefetch is available for both the global cache and the cluster cache; both use simple next-line prefetch of a configurable number of lines and can be independently enabled. Both the Cluster and Global caches are 8-way set associative.

2.2 Rigel Simulation and Programming Environment

Rigel is being developed and analyzed with a custom simulator developed from scratch, called `rigelsim`. Rigelsim is a cycle-based and cycle-accurate simulator written in C++ that runs compiled C programs coded for a SPMD model. Rigelsim provides command line options to control various system implementations such as the number of clusters per tile, number of tiles, memory channels, global cache banks, prefetch length, and cache coherence options. It also provides a rich set of statistics at the end of a simulation.

We have developed a runtime library of task-based programming techniques to implement a task queue for bulk-synchronous programming style, or we can program at a lower level that fixes operations to cores based on their hardcoded core (thread) ID number. Each core can be thought of as a single hardware thread, with 8 contiguous threads sharing a cluster cache. Our benchmarks are coded as C programs that use the thread number to determine address decomposition for the data-parallel tests to remove any noise from task-queue overhead. The simulator implements a very accurate model of the GDDR DRAM controllers and the processor cores, and allows us to set timers to extract the runtime of the main portions of the kernels and ignore the setup and IO portions of the tests. Physical design has validated the estimated 1.2 GHz clock cycle.

CHAPTER 3

GPU MAIN MEMORY CHARACTERIZATION

GPU main memory systems are designed to provide high bandwidth when the accesses from adjacent threads in a block perform requests that can be easily coalesced by the hardware. This requires the memory addresses to vary in the bits used to select banks and channels, which requires the programmer to understand the hardware implementation. If the details are not published, a programmer can attempt to reverse engineer the design by testing with micro-benchrmarks. This section describes a set of such tests run on various commercial GPUs using the AC cluster at the National Center for Supercomputing Applications at the University of Illinois.

3.1 GPU Micro-Benchmarks

The goal of micro-benchmarking in this project is to attempt to characterize the memory controller address mapping. When the architecture has a fixed mapping of physical address bits to DRAM bank, channel, and row addresses, we should be able to observe the effects of varying memory channel parallelism through bandwidth measurements. The micro-benchmarks vary the address stride across thread blocks of dimension 1. The idea is to create a single thread per processor to access a large block of memory in parallel without memory coalescing at the controller affecting the measurement. By varying the stride per thread, we can test the bandwidth with different sets of address bits varying per request. When the differing bits address different DRAM channels or banks, memory parallelism will result in higher bandwidth. When the bits address similar channels or banks, the contention will result in lower bandwidth.

The bandwidth tests are designed to create a window of address bits based on thread index, and sweep this window across the address space. There are

two versions of the test. The first does a fast sweep by shifting the window up in powers of two to move the entire window up one bit location per run. The second version starts with the window at the lower address bits and adds a fixed offset of 16 for each run. The second version will toggle small subsets of the address window to show fine-grained differences in the bits.

3.2 GPU Bandwidth Characterization

The GPUs available on the AC cluster at this time include the ATI Radeon 5870, NVIDIA T10 Tesla, and NVIDIA GTX 480 Fermi.

3.2.1 Characterizing ATI Radeon 5870

The ATI architecture shows one region of bandwidth transitions as shown in Figure 3.1. The flat region at the left indicates we are not stimulating steering bits at all, so full bandwidth is achieved. When bandwidth begins to decline rapidly it is clear that one set of steering bits has been reached. The flat low bandwidth region at the right of the diagram indicates we have moved out of the first set of steering bits. From these diagrams we infer that the address bits in the range A[9:13] are the set which should vary to achieve maximum bandwidth.

3.2.2 Characterizing NVIDIA Tesla T10

The NVIDIA Tesla T10 measurement is quite similar to ATI's as shown in Figure 3.2. The entire set of steering bits is contained in A[8:14], but there do appear to be two distinct regions, as the slope flattens briefly between two steeper sections.

Figure 3.3 shows the effect that address stride can have on bandwidth. Varying the stride length by small amounts creates the same effect as varying the total dataset size. When the total input data is larger, the stride length of a simple decompositon grows as well. This figure shows that slight perturbations in the stride length can change the bandwidth drastically. Larger input data can actually improve performance if it creates a better memory

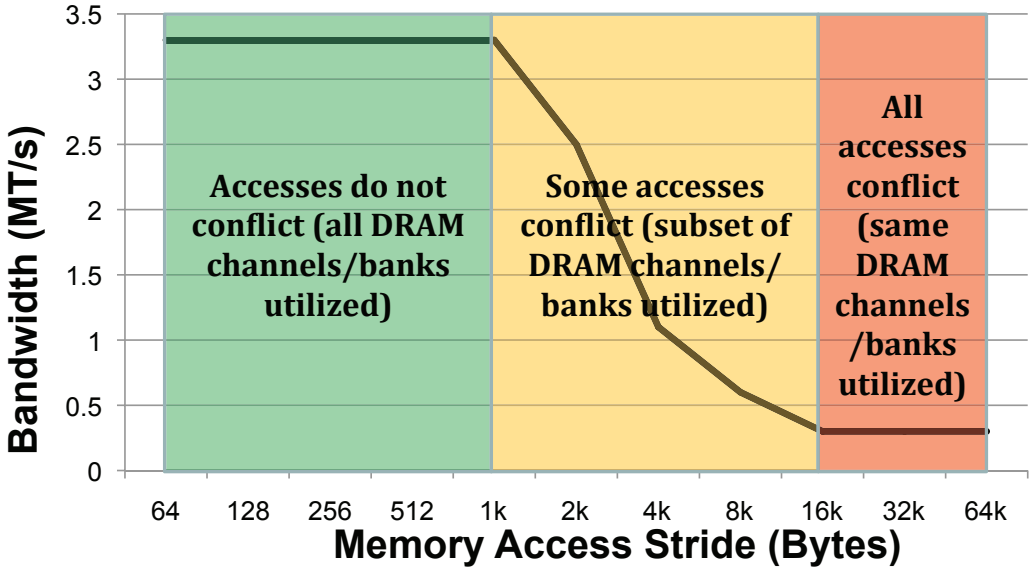

Figure 3.1: Micro-benchmark on ATI Radeon window test.

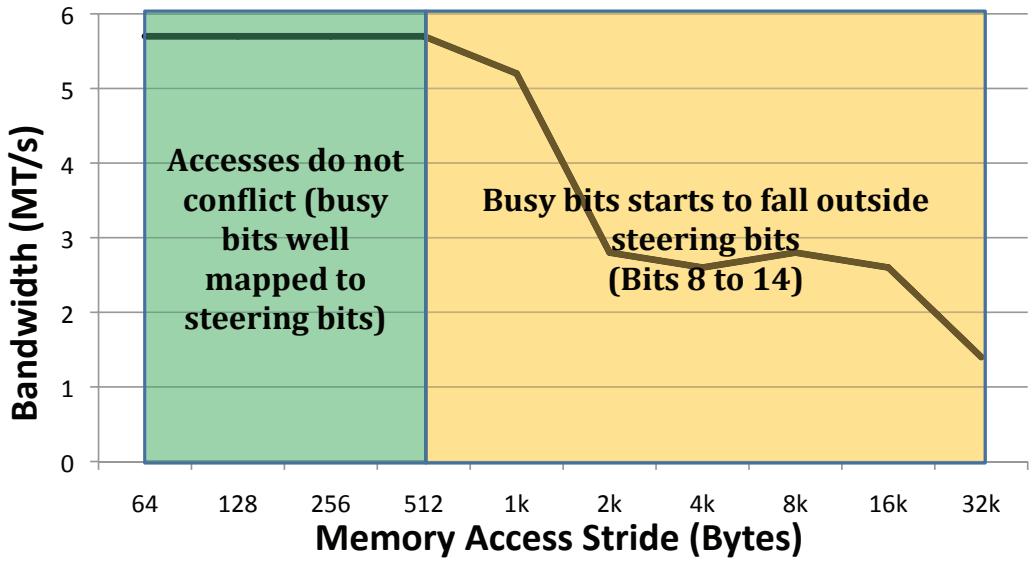

Figure 3.2: Micro-benchmark on Tesla T10.

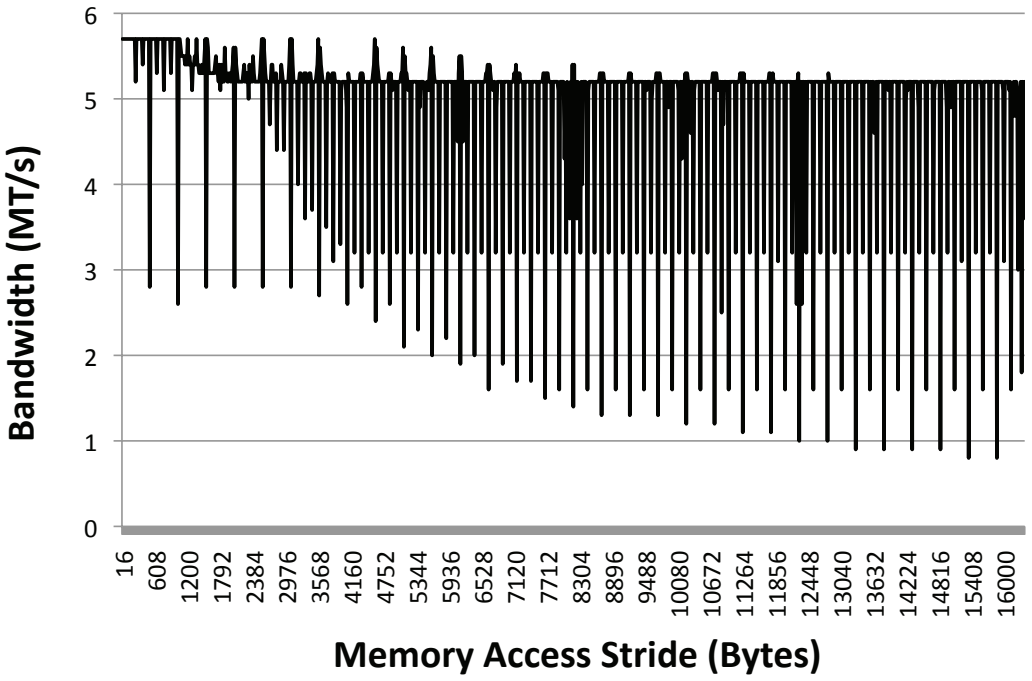

Figure 3.3: Throughput on NVIDIA Tesla GPU when varying the address stride by 16 B per sample. This figure shows how varying the address stride can lead to unpredictable bandwidth, where slightly larger strides can achieve higher throughput if the address decomposition becomes more favorable.

address request pattern that reduces contention. Similar effects are observed on Rigel, which will be discussed in Chapter 4.

3.2.3 Characterizing NVIDIA GTX480 (Fermi)

The Fermi architecture differs from the earlier NVIDIA architectures in that it contains L1 caches at the processors. It also contains a shared L2 cache that fronts main DRAM. Caching is controlled by different instruction set attributes. There are four modes of cache operation that are chosen by passing options to the assembler. The four modes are listed below.

1. *Cache All,* indicated with the *ca* option and extension, is the default and caches accesses at both levels

2. *Cache Global,* indicated with the *cg* option and extension, caches only at the L2 cache and bypasses the L1

3. *Cache Streaming,* indicated with the *cs* option and extension, is meant for streaming applications where data is likely to be accessed only once

4. *Cache Volatile,* indicated with the *cv* option and extension, indicates that L1 cached data is to be considered volatile and should be treated as invalid for future accesses

The micro-benchmarks were run with all four settings on Fermi. When L1 caching is used, the measured bandwidth is much higher than Tesla, and when L1 caching is disabled the bandwidth is a little lower. The main difference observed on Fermi compared with the other architectures was that the bandwidth never varied as the window of striding changed. Fermi is believed to use a very different DRAM addressing scheme that involves address hashing and is not necessarily repeatable, depending on several conditions. For these reasons, Fermi does not seem amenable to static analysis of DRAM mapping that would predict a benefit from data layout transformations.

CHAPTER 4

RIGEL MAIN MEMORY CHARACTERIZATION

The previous chapter explored whether the memory addressing logic can be interpreted from micro-benchmarks, and how one would use this knowledge to improve bandwidth utilization. Such methods are needed when the memory controller logic is not known or the information is not published. For the Rigel project this information is already known, so it is not necessary to apply the same set of tests. This chapter moves on to use micro-benchmarks that represent pieces of real applications that move data in and out of memory. The goal is to study how these tests perform on MIMD hardware using different work distribution schemes.

4.1 Rigel Memory System

The Rigel main memory system contains multiple, interleaved channels of GDDR memory. The address interleaving must be exploited to achieve high bandwidth. If the stream of memory requests creates contention at the memory channels, bandwidth will suffer greatly. This chapter explores the ways that MIMD hardware as implemented in Rigel can create or avoid contention at the memory controllers. A set of micro-benchmarks are used here to study memory system behavior under different conditions. Since the Rigel memory hierarchy contains multiple levels of cache, it is important to consider their effect when designing the benchmarks. The cache controllers can be optionally configured to generate next-line prefetch requests from 1 to 4 lines, with no hardware prefetch by default. The graphs in this section will show bandwidth attained by varying the number of clusters running. Each cluster contains 8 HW threads. The number of tiles activated is increased first, keeping just one cluster per tile active until 8 tiles is reached. Then the number of clusters per tile is increased until the maximum of 128 clusters are active.

This is done because the on-chip network aggregates traffic within a tile to a single port of the global crossbar, as seen in Figure 2.1 and Figure 2.2.

The processors and hardware threads on Rigel have a fixed ID number that can be used to partition work and address ranges. This style of programming is common in OpenMP [4] where the programmer declares the number of threads to create and uses the thread ID number to partition work. As an alternative, the Rigel team has developed a software runtime library that implements a dynamic task-based queueing system. This scheme, known as the Rigel Task Model (RTM), creates a single logical work queue that can be physically distributed across the clusters. The model and semantics of RTM are similar to the Carbon [5] system except that it is implemented entirely in software. Any thread can add work to the queue in the form of a task descriptor, and any thread can remove tasks from the queue and perform work. The scheme implements a form of bulk synchronous programming, a common parallel programming model where the computation is decomposed into intervals that contain a set of independent work. Global barriers are used to separate intervals, and data communication among tasks occurs across intervals by forcing all data produced during one interval to be globally visible. Tasks can be of varying length and new tasks are dequeued by threads as soon as the current task is completed. This dynamic scheme has the benefit of improving load balance when the tasks are of varying length, since a new task can begin immediately independent of what other threads are doing.

It will be shown that bandwidth close to the physical limit can be achieved on Rigel if the requests are orchestrated perfectly. This can be done with a static workload partitioning using thread ID number running a regular computation, that is, one that does not vary the amount of computation from one thread to another. When this is done, the hardware threads will be running the same code at the same time, and will stay effectively synchronized even without any hardware or software enforcement. This has the effect of generating all of the memory requests from each thread at the same time, and if these requests are to non-conflicting memory regions the chip will achieve very high bandwidth. There can be a significant amount of computation performed on the data, as long as the computation is the same on all threads. When the dynamic RTM programming model is used, or when the computation varies among threads in either the static or dynamic model, there can be a significant bandwidth penalty of as much as 50%.

4.2 Memory Bandwidth with Static Partitioning

This section studies the achievable memory bandwidth on Rigel, as well as the balance of computation and bandwidth resources in the design. In this context *balance* refers to the amount of computation that can be performed per data element while sustaining maximum bandwidth. The memory system on Rigel, as with most high performance processors, is quite complex. As shown in Figure 2.2, the on-chip interface to main memory consists of a full crossbar with 16 ports on one side for each tile, and 8 ports on the other side with each connected to a subset of banks of the global cache. The global cache is a single logical cache that fronts main memory. From the processors there is no way to tell whether a request is serviced by the global cache or by main memory.

The address interleaving of the memory channels and banks is fixed on Rigel, allowing a predictable mapping of physical address to memory location. Exploiting the interleaving to achieve multiple parallel accesses at the memory controllers is required to maximize off-chip bandwidth. This is similar to the memory access coalescing strategies needed in GPUs to maximize their bandwidth. The goal is to activate as many non-conflicting requests as possible in parallel to allow each channel to run at a high rate.

Table 4.1 shows how much the access time can vary for the same memory structure. A simple memory block read test was used, with two different address decompositions: a simple `BlockSize / threadID` style where each thread loaded a contiguous block, and a strided version, where each thread loaded a small 128 element chunk and then loaded blocks `128 x threadID` elements distant. These two schemes are illustrated in Figure 4.1. This static strided partitioning method was able to achieve close to the peak bandwidth available to the hardware, while the simple scheme achieved less than half. The 128 element size perfectly matches the address bits used by the memory controllers to determine bank and block access, so perfect interleaving is achieved.

All of the schemes tested in this section benefit some from manually unrolling the inner loop that walks the array. The difference is shown in Figure 4.2, using linear scale this time so that the difference in magnitude is visible. The rest of the tests in this chapter will use the unrolled versions unless otherwise specified.

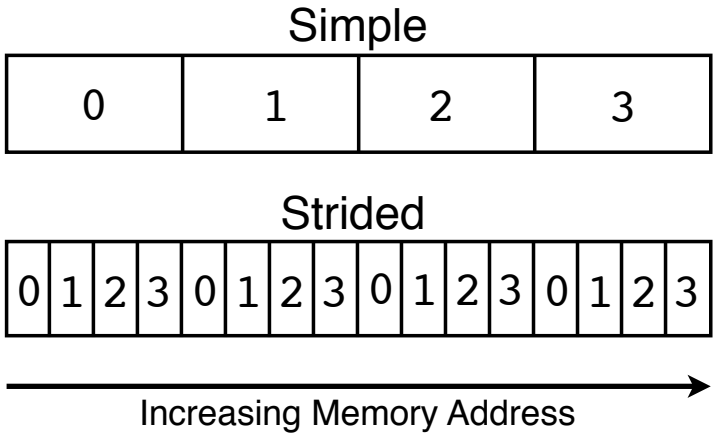

Figure 4.1: Address decomposition by thread ID for four threads, with a simple decomposition above and strided below.

4.2.1 Statically Partitioned Benchmarks

For the bandwidth tests the goal is to measure how close to peak bandwidth a program gets and what had to be done to the program to achieve the best performance. The goal is to show how well the programmer has to understand the memory hierarchy to achieve high bandwidth. Modern DRAM based systems transfer data from off-chip in fixed length bursts, regardless of how much of that data is actually requested or used. The goal here is to measure the actual amount of data transferred across the chip pins, so the micro-benchmarks used will consume all of the data transferred with a single use per element. There is no temporal reuse of data, but the effects of cache line transfers are inherent since the DRAM burst is matched to the cache line size. All of the data in a cache line is used once in sequential order, so the data is transferred once across the chip pins. The tests here are simulated with different configurations of active threads and clusters, and different

Table 4.1: Peak throughput comparison of Simple and Strided address decomposition.

Scheme	**Bandwidth**
Simple	80GB/s
Strided	170GB/s

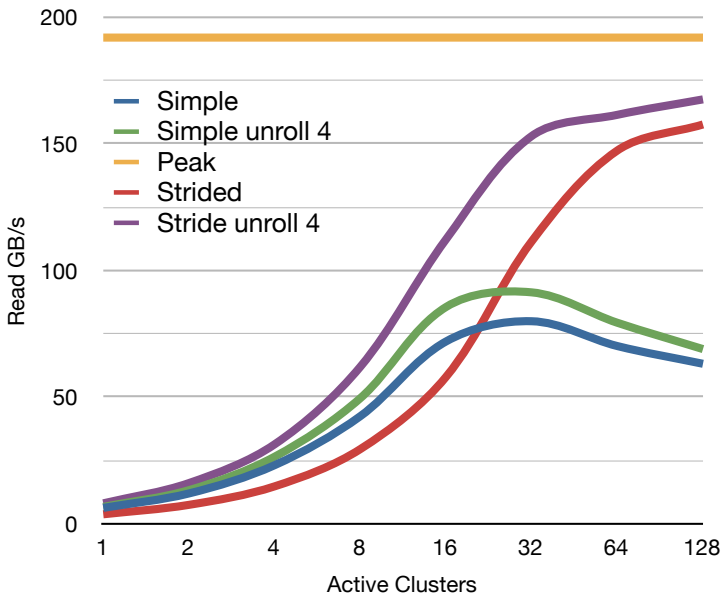

Figure 4.2: The difference between simple loops and manually unrolled by four iterations is shown. The difference can be significant in the middle of the curve.

amounts of HW prefetch enabled. The details are described with the results. Next-line prefetch is a latency avoidance technique, so it is not expected to improve raw bandwidth measurements. Measurements confirm this is the case. The benchmarks start a simulator-based timer right before the parallel access portion begins, and stop the timer after all threads have completed the main loop. This timer is used to compute the achieved bandwidth since the program knows the size of the data structure accessed. Multiple data sizes were tested and found to have no effect on the final bandwidth. A structure of 4 MB is used in the tests unless otherwise stated, a size chosen to match the on-chip cache hierarchy. A program running through a large data set would likely be blocked to move data of near this amount to take advantage of the cache.

For the Rigel system, we have started out with the Block Memory Read test that simply reads a large block of memory in parallel. The idea is to get the number of memory requests up to a large amount by dividing up the address range of a large array among clusters. Rigel's relatively high number of DDR channels and memory controller queues require a large number of requests to approach peak bandwidth. The architecture is designed with the

assumption of massive data parallelism and the MIMD architecture enables a large number of independent requests. However, the interleaving effects of the interconnect and program divergence can reduce DRAM row locality. Knowing the addressing logic across channels and banks, and controlling the program data address breakdown across threads, gives the potential to still achieve a high bandwidth for regular patterns that will not have any control flow divergence. Execution of these programs on regular data structures would maintain synchronized requests across the threads.

4.2.2 Bandwidth Results with Static Scheme

Figures 4.3 and 4.4 show the measured BW for a pure memory read pattern using different address decompositions. The same data is plotted in both figures. Figure 4.3 uses a log scale to show the perfect scaling of bandwidth at the lower left part of the curve, when the system is not generating enough requests to fully utilize the available bandwidth. Since Rigel uses in-order processors with a single outstanding read request, doubling the number of active clusters doubles the number of active read requests. Figure 4.4 uses a linear scale on the Y-axis to more clearly show the difference in magnitude. Next-line prefetch for the strided pattern is also shown in Figure 4.4 in red; the effect is negligible as expected. The *Simple* address decomposition is the easiest to code; simply divide the range into contiguous chunks with one per thread. With a simple coarse-grained decomposition of addresses, requests from adjacent cores will be to addresses far apart in memory. In the worst case scenario, simultaneous requests from adjacent cores will map to the same DRAM bank, causing maximum contention and minimum throughput. As seen in Figure 3.3 in Chapter 3, small changes in the data size can create the same effect, but the result is not predictable if the exact data size is unknown. The difference in throughput is more than 2x just by changing the memory access pattern to better match the DRAM controller's address mapping. The *Strided* style matches the memory mapping at the DRAM controller to the per-thread assignment so that simultaneous requests achieve maximum DRAM row locality and bank-level parallelism.

These figures indicate that near-maximum bandwidth is achieved with only about 1/4 of the hardware threads active. Enough memory requests

Figure 4.3: Memory read bandwidth (log scale) vs. number of active clusters, showing the bandwidth achieved with a simple and a strided address decomposition. The strided version reaches near the chip's peak of 192 GB/s, and more than doubles the simple version.

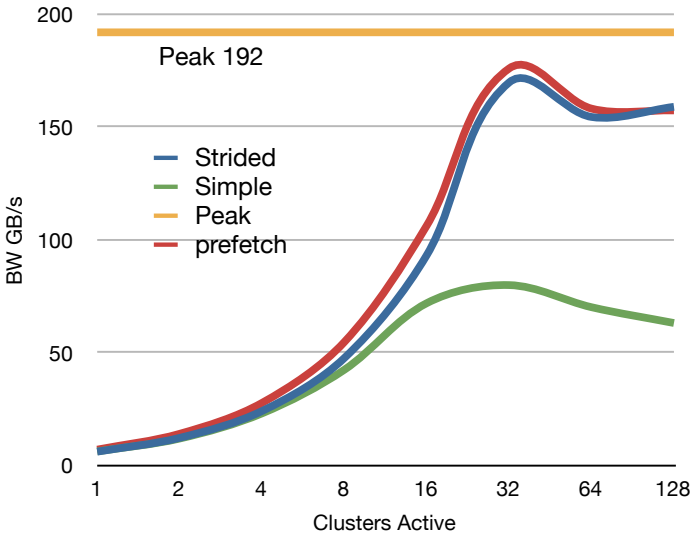

Figure 4.4: Measured read bandwidth vs. cluster count. Linear scale used to highlight the magnitude of differences. Rigel has 8 HW threads per cluster. Prefetch on strided case had negligible effect, shown in red.

are generated to keep the memory system fully utilized. As more threads are added the memory requests will just queue up, so average latency would increase without improving bandwidth much. This bandwidth test has no computation performed on the data, so the hardware threads are generating memory requests at near their maximum rate. Since the processors in Rigel operate strictly in order, with computation on the data elements the rate of requests from each thread would be reduced. The previous graphs indicate that Rigel should be able to sustain a reasonable amount of computation and still maintain maximum memory throughput. This will be explored in the next section.

4.2.3 Throughput with Computation

In this section, a more realistic benchmark will be used that takes an input data structure, performs some computation on it and writes the results to a different output data structure. The amount of computation will be varied, and the memory bandwidth achieved will be measured. The computation used in these tests is a string of dependent arithmetic operations. With no computation, this reduces to a simple data copy operation. Since there are now two memory accesses per element, the magnitude of the througput is a lower number. The addition of memory write commands has an effect on the DRAM performance since it changes the request pattern seen at the controllers, potentially reducing DRAM row locality and adding contention.

Figure 4.5 shows the measured throughput vs. number of active threads for varying amounts of computation on Rigel. All threads are performing the same computation, and the static partitioning with strided access pattern is used. The family of curves show the performance for varying amounts of computation, from zero for the *Copy* curve, and from one to 40 multiplication operations per word read from memory.

The curves in Figure 4.5 show how effective the large number of threads are at increasing throughput under varying computational loads. As expected based on the graphs in the previous section, Rigel has capacity to do more work without loss of throughput since high BW is achieved with relatively few threads. At 32 clusters (256 threads), throughput drops with any amount of computation beyond a single operation, but increasing active threads scales

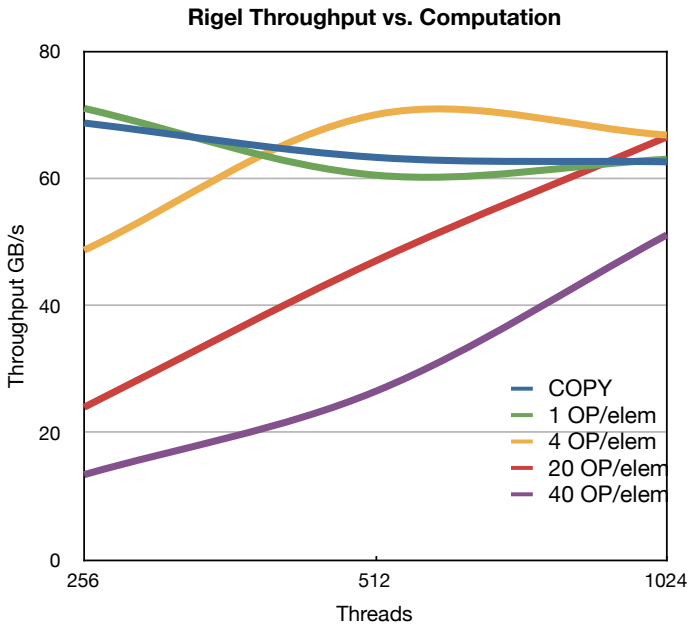

Figure 4.5: Measured bandwidth vs. thread count for Varying computation. With 8 threads per cluster, the Rigel graph depicts thread counts of 256, 512 and 1024.

it back up to the maximum. As the number of operations performed per data element increases, each active thread will have a proportionally lower memory BW demand since they are in-order blocking cores. Activating more threads increases the memory BW requested at the controller back to a level that achieves full throughput. Rigel can support up to 20 dependent operations at 128 clusters (1024 HW threads), a fairly significant amount of computation. In this test there is no ILP in the computation since it uses a chain of dependent multiplies inside of a called function. If there were more ILP inherent in the computation it could be sustained at maximum throughput.

From this perspective Rigel appears to be an effective CMP design since it achieves high throughput with no extra programmer effort. The large number of threads available to run computations do not require any special programming techniques. If the computation is longer than this, there is a drop in throughput, as one would expect.

4.3 Memory Bandwidth with Dynamic Partitioning

This section uses the Rigel Task Model to identify and distribute sections of memory to be accessed by the HW threads. The goal is to compare memory system performance to the statically scheduled strided address pattern. This pattern achieved high bandwidth with the static schedule. The dynamic version can see a significant reduction in achieved bandwidth, which would offset the gains of MIMD execution hardware relative to SIMD hardware.

4.3.1 Dynamically Partitioned Benchmarks

The RTM runtime system places task descriptors in a software-managed queue structure, and allows any processor to dequeue and execute tasks in any order. The benchmarks in this section are similar to the statically partitioned set, except that the task descriptor number is used instead of the HW thread ID number to select a section of the address space. Task descriptors are created with an incrementing value that is used as the task number, and each HW thread that receives a descriptor uses that number to calculate the section of memory to access. The same data structures and timer based measurement scheme are used.

The bandwidth numbers shown in this section measure the time to move the data structure by the program. They do not account for the memory access overhead incurred by the runtime system. Since the RTM queues are software structures, memory accesses occur there as well. The cache system helps reduce the time overhead, but the accesses themselves may interfere with the DRAM locality of the program accesses. The Rigel simulator calculates the time spent in the runtime routines, and this overhead is reported in this section. The data shows that the time spent in the runtime is a small fraction of the overall time and does not account for very much of the bandwidth reduction. Multiple decompositions were tested, where threads could access multiple memory regions per task. The RTM overhead was reduced very slightly, while the overall bandwidth recorded varied more significantly. This indicates that the dynamic execution itself, not the runtime system overhead, is the main cause of reduced bandwidth.

4.3.2 Bandwidth Results with Dynamic Scheme

Figure 4.6 shows a comparison of the static and dynamic bandwidth achieved for the strided access pattern. The RTM tasks access a single 128 B segment, which is the same size used in the static strided benchmark. RTM has the effect of placing all of the segments into a queue in some non-monotonic order, and pulling them out in some other order. The effect is to mix up the segments significantly. The software queue system also has a serializing effect on the hardware threads, which acts to stagger their execution in time.

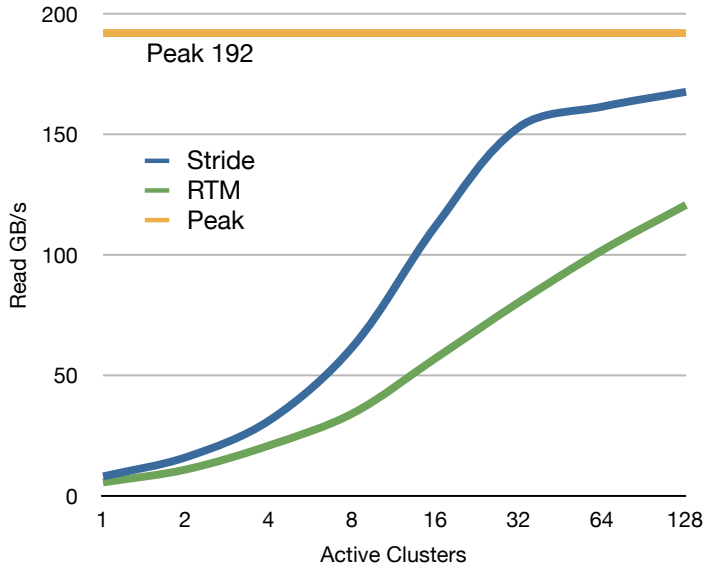

Figure 4.6: Measured bandwidth vs. cluster count. The static strided pattern is shown for reference; the RTM curve is significantly lower for the entire range.

4.4 Memory Behavior Comparison

The Rigel simulator produces a rich set of statistics for various component subsystems of the chip. The main memory system provides statistics such as the number of read and write accesses, the number of row activate commands, the average number of bursts per activate command, and many others. The number of activate commands and the number of bursts per activate are indications of the amount of row locality that occurred during the execution of a program.

The software runtime system incurs a small overhead as shown in the first column of Table 4.2. The overhead is much smaller than the difference in throughput seen in the dynamic RTM benchmarks compared to the static versions. Table 4.2 also shows some of the relevant DRAM controller statistics regarding memory throughput. DRAM accesses that require new row activate commands incur longer latency than accesses that hit an open row. The last two columns in the table show the number of row activate commands, and the number of DRAM bursts per activate command. The latter is a measure of how many requests hit an already-open row. Since the static and dynamic versions of these benchmarks move the same amount of user data, these two columns indicate that the static version achieves significantly higher DRAM row locality. This data indicates that actual DRAM performance, not task overhead, is the primary factor in the reduced throughput of the dynamic versions.

Table 4.2: Rigel Task Model overhead and DRAM locality statistics for different active thread configurations. RTM overhead is only relevant to the dynamic case. DRAM statistics are shown for both dynamic and static versions.

Threads	RTM Overhead	Row Activate Cmds		Bursts Per Activate	
		static	dynamic	static	dynamic
256	6.5%	7.6K	44.7K	18.81	4.21
512	5.7%	20.6	50.1K	5.26	3.31
1024	7.2%	48K	62.3	2.83	2.48

CHAPTER 5

DISCUSSION

Table 4.2 shows the DRAM statistics converging as the number of active threads increases. For the static pattern the 25% active point clearly has the best DRAM statistics. This is also the point where the static pattern reached peak throughput. At 512 threads the DRAM statistics are still significantly better for the static version compared to the dynamic. When the requests from the program do not oversubscribe the memory system by a large amount, the DRAM locality can be maintained.

GPUs programmed using the CUDA [6] language have a library function called `syncthreads()`. This function forces a local barrier among the threads of a particular thread block, but not across the entire program. Although extra synchronization is typically avoided so as not to incur the overhead, in the case of SIMD hardware synchronizing the threads before a set of memory access can actually improve performance by allowing the memory system to coalesce the requests. By designing the program to properly divide the memory pattern across the thread block and synchronizing the threads before a set of requests, the highest possible memory throughput can be achieved. Rigel has no such function available, either in hardware of software. The data presented has shown that significant memory performance is lost by allowing the threads of Rigel to run independently for long periods. It is likely that Rigel would benefit from some form of periodic synchronization such as that available in CUDA.

The design of the Rigel cluster and tile blocks would have an effect on such a synchronization scheme. Each cluster on Rigel has only one physical port to the tile, and each tile has only one port to the crossbar feeding the memory system. There is no notion of thread blocks in Rigel. A cluster would seem the natural unit to provide a synchronization mechanism, but the single physical port has the effect of serializing the accesses from a single cluster. As long as these requests are to different channels or banks there would still

be memory parallelism, and requests from other tiles would occur at the same time. The balance of number of tiles, clusters per tile, DRAM channels and crossbar ports are all factors that would affect the ideal decomposition. A synchronization scheme like this would need the lowest possible latency, so it would need hardware support. A software scheme would not be able to create the same-cycle execution that a hardware scheme could achieve. It is not obvious at first glance what the organization of such a hardware synchronization scheme should be. There is a large design space to consider in implementing something like `syncthreads()` for Rigel. The data gathered for this study indicates there could be a benefit to such a scheme.

CHAPTER 6

CONCLUSION

MIMD execution hardware is intended to alleviate the penalties associated with SIMD hardware, such as serialization of control flow divergence or variation of workloads among threads. The lock-step execution utilized by SIMD hardware is beneficial to the complex memory controller logic used in modern compute accelerators, however, enabling them to realize the high memory bandwidth available on such systems. MIMD hardware such as Rigel is capable of achieving similarly high bandwidth when the hardware threads stay effectively synchronized by use of a static work partitioning of a regular computation. However, this type of computation is almost certainly SIMD-friendly, so it is not likely that MIMD hardware can achieve the same level of efficiency as SIMD in this case. Programs where SIMD hardware is not seen as effective, such as highly varying computation across data elements or threads, are the ones that MIMD hardware is intended to improve. The data gathered for this study has shown that MIMD hardware can see a significant bandwidth reduction from the chip's potential on such irregular programs.

On memory bound applications, the memory bandwidth that can be effectively used may be a more important component than the execution unit efficiency. While SIMD hardware may lose out on execution unit performance, if high memory bandwidth can still be realized then SIMD could still achieve high performance overall. If MIMD hardware cannot effectively achieve high bandwidth on such computations, the improvement it gains in execution efficiency of the computation portion of a program would be offset by the reduced memory system performance. Direct comparison between GPUs and Rigel for such cases is difficult, since programs that are not perceived to be SIMD-friendly are not commonly implemented for GPUs. More are being developed for various scientific fields, with the common theme that they are organized to maximize memory bandwidth.

As with most problems in engineering, you cannot get something for free.

Improving one aspect of a system usually comes at a cost to another area, often in an unexpected manner. This work motivates more study to determine if there are ways to introduce enough thread synchronization in hardware to maintain a regular, high performance memory request stream without giving up all of the benefits MIMD achieves in execution. A lightweight hardware scheme within a cluster, for example, that forced synchronization or coalescing of the requests from a cluster could improve memory bandwidth and is worth investigating.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[2] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 513–522.

[3] J. H. Kelm et al., "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," in *Proceedings of the International Symposium on Computer Architecture*, June 2009, pp. 140–151.

[4] OpenMP Architecture Review Board, "OpenMP application program interface," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[5] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 162–173.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.