# Implementing a GPU Programming Model on a Non-GPU Accelerator Architecture

Stephen M. Kofsky, Daniel R. Johnson, John A. Stratton,
Wen-mei W. Hwu, Sanjay J. Patel, and Steven S. Lumetta

University of Illinois at Urbana-Champaign, Urbana IL 61801, USA

**Abstract.** Parallel codes are written primarily for the purpose of performance. It is highly desirable that parallel codes be portable between parallel architectures without significant performance degradation or code rewrites. While *performance portability* and its limits have been studied thoroughly on single processor systems, this goal has been less extensively studied and is more difficult to achieve for parallel systems. Emerging single-chip parallel platforms are no exception; writing code that obtains good performance across GPUs and other many-core CMPs can be challenging. In this paper, we focus on CUDA codes, noting that programs must obey a number of constraints to achieve high performance on an NVIDIA GPU. Under such constraints, we develop optimizations that improve the performance of CUDA code on a MIMD accelerator architecture that we are developing called Rigel. We demonstrate performance improvements with these optimizations over naïve translations, and final performance results comparable to those of codes that were hand-optimized for Rigel.

## 1 Introduction and Background

In this paper, we address the goal of achieving *performance portability* for existing SPMD code, originally tuned for a GPU, when targeting Rigel [4], a throughput-oriented MIMD accelerator architecture. Our approach is based on automatic translation of CUDA's [10] fine-grained SPMD kernels to Rigel. With optimizations that leverage the characteristics of optimized CUDA code, we see a significant speedup across several benchmarks. These optimizations do not change the underlying algorithm in the code, and it is reasonable to assume that they could be automated using compiler analysis.

### 1.1 CUDA

Fine-grained SPMD programming models use multi-level parallel decomposition to target various levels of parallelism for high throughput computation on GPU architectures. We use CUDA in this work, noting that the techniques also apply for languages such as OpenCL [5].

CUDA uses *host* code to set up and launch *grids* of parallel execution. Parallelism within a grid is decomposed hierarchically into two levels. A one- or

A.L. Varbanescu et al. (Eds.): ISCA 2010 Workshops, LNCS 6161, pp. 40–51, 2011.

two-dimensional grid is composed of *thread blocks*, and each one-, two-, or three-dimensional thread block is further decomposed into *threads*. Threads within a block execute cooperatively; they are initiated concurrently and interleaved in execution. Explicit barriers among threads in a thread block can control this interleaving. Blocks are not guaranteed to run concurrently, limiting their interaction.

A kernel function is written in SPMD form and executed by every thread of a grid. Variables within the kernel function are private to each thread by default but can be annotated as *shared* variables. In the case of shared variables, each thread block has a private instance, which all constituent threads may access. The memory available for shared variables is limited in capacity but offers low latency. *Global* variables are visible to all threads, are stored in high capacity DRAM, and are assumed to have long latency access.

The programming model characteristics as well as performance implications of GPU architectures lead programmers to follow specific software design principles. Tasks or operations using the same data should be co-scheduled to leverage locality. Small datasets can be accessed quickly and repeatedly, while streaming or random accesses to large data sets limits performance. A sufficient number of thread blocks are required to fully utilize the on-chip parallelism while block sizes have effects on execution overhead and load imbalance. These trends are common among throughput-oriented architectures; the MIMD accelerator Rigel should benefit from optimizations targeting these points as well.

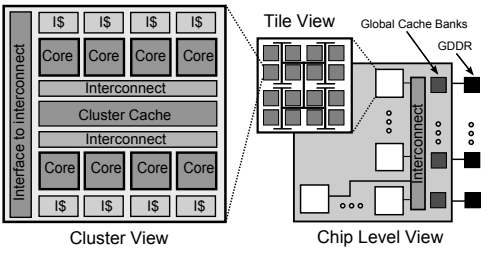
### 1.2 MCUDA

MCUDA [14] is a publicly available source-to-source translation framework for CUDA and has been used in previous work to convert CUDA kernels to parallel C code for CPUs. With MCUDA, CUDA threads within a thread block are combined and serialized within loops, creating code that iterates over the individual CUDA thread indices. MCUDA's translation process increases the work granularity by making thread blocks the smallest parallel task. During execution thread blocks are mapped to separate OS threads to be executed in parallel.

### 1.3 Rigel

Rigel is a 1024-core MIMD compute accelerator targeting task- and data-parallel visual computing workloads that scale up to thousands of concurrent tasks. The design objective of Rigel is to provide high compute density while enabling an easily targeted, conventional programming model. A block diagram of Rigel is shown in Figure 1(a).

The fundamental processing element of Rigel is an area-optimized, dual-issue, in-order core with a RISC-like ISA, single-precision FPU, and independent fetch unit. Eight cores and a shared cache comprise a single Rigel *cluster*. Clusters are grouped logically into a *tile* using a bi-directional tree-structured interconnect. Eight tiles of 16 clusters each are distributed across the chip, attached to 32 global cache banks via a multistage interconnect. The last-level global cache




(a)

| Parameter | Value | Unit |
|---|---|---|
| Cores | 1024 | – |
| Memory BW | 192 | GB/s |
| DRAM Channels | 8 | – |
| L1I Size | 2 | kB |
| L1D Size | 1 | kB |
| L2 Cluster Cache (Total) | 8 | MB |
| L3 Global Cache (Total) | 4 | MB |

(b)

**Fig. 1.** Block diagram and simulated parameters for the Rigel architecture

provides buffering for 8 high-bandwidth GDDR memory controllers. Figure 1(b) presents additional architectural parameters.

Applications are developed for Rigel using a task-based API, 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.

## 2 RCUDA

RCUDA is a framework that allows for CUDA code to be executed on Rigel. The first component of RCUDA is a CUDA-to-C translation engine, which converts CUDA kernels to C. The second component is a software runtime library that implements CUDA built-in functions and performs load balancing.

### 2.1 Source Code Transformations

CUDA kernel source code is transformed, as shown in Figure 2, so as to be amenable to Rigel's MIMD execution model. Within a cluster, threads can be mapped dynamically to cores and executed serially in a loop between synchronization points in the kernel. Whenever a synchronization point occurs, the thread queue on the cluster is reset so that the cluster can iterate over each thread again. Shared variables are stored as a per-cluster data structure. Each core can read and write to the shared data through the cluster cache. Further, local variables are stored in a cluster level data structure since we allow CUDA threads to migrate between cores within a cluster after a synchronization point. However, local CUDA thread variables that are produced and consumed between synchronization points do not have to be replicated since they are not used when a CUDA thread moves to another core.

Host code, originally targeting an x86 general-purpose CPU, must be hand edited so as not to use programming interfaces that are not supported on Rigel. In addition, host code on Rigel is executed by a single core 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.

## 2.2 Runtime Library

The second major component of the RCUDA framework is a software runtime library that provides work distribution and an implementation of CUDA built-in functions such as `__syncthreads()` and atomics.

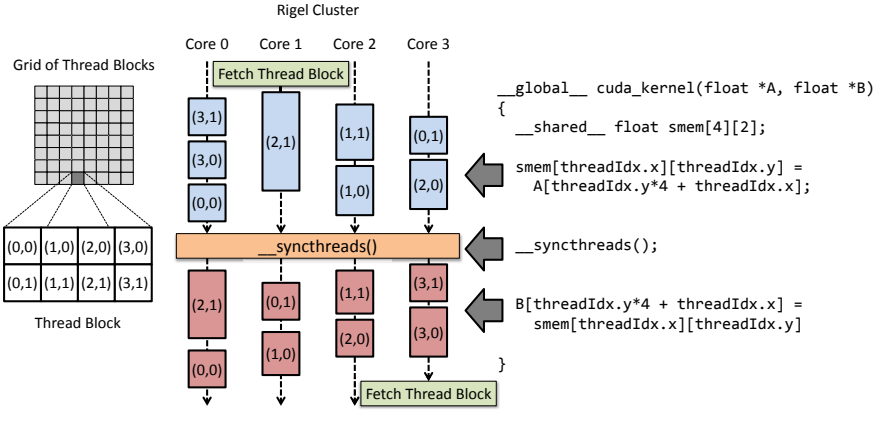



**Fig. 2.** A thread block with one synchronization call mapped to a Rigel cluster

RCUDA handles work distribution hierarchically, at both the global chip level and at the local cluster level. CUDA uses a grid of thread blocks to define work. Threads in a thread block are executed simultaneously, allowing for synchronization across threads in a block. In the RCUDA framework, a thread block is executed on a single Rigel cluster. One core from a cluster attempts to fetch a thread block only when the core is waiting to do work. Fetching on demand allows the thread blocks to be dynamically allocated to clusters. A core 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 2. Each cluster contains eight cores with separate instruction streams and a shared cache. 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.

### 2.3 Kernel Execution

When a kernel is called, one core initializes the runtime and writes the kernel function parameters to global memory. Next, one core from each cluster attempts to fetch a thread block identifier (ID) by atomically decrementing a global counter of remaining thread blocks. If the block ID is nonnegative, the core initializes the cluster task queue, and each core begins executing tasks from the thread block. After completing a thread block, the process is repeated. When all cores in a cluster complete execution, and no more thread blocks are available, the cores enter a barrier where they wait for all other cores on the chip to complete. After all cores enter the barrier, control returns to the host code.

## 3 RCUDA Optimizations

In this section we describe two classes of optimizations that can be applied at the source level to CUDA kernels and their runtime environment. The first class of optimizations are kernel code transformations. The second class of optimizations are runtime optimizations which change how kernel code is executed. We also discuss how these optimizations can be automated.

### 3.1 Kernel Code 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 specialized memories. Additionally, CUDA thread synchronization is a low latency operation on a GPU, but must be done in software on Rigel. We use kernel transformations to remove shared memory and to remove thread synchronization where possible.

**Shared Memory Removal.** Using shared memory is essential for good performance on NVIDIA GPUs since a GPU has limited cache and prefetch capabilities. By using shared memory, programmers can avoid memory bandwidth bottlenecks by locating data in shared scratchpad memories to leverage temporal locality. Many CUDA kernels use shared memory solely to keep a high-bandwidth, read-only copy of global data. Such kernels populate shared memory with the global memory contents before computation, using the shared memory for all computations before writing results back to global memory. For example, the transpose kernel shown in Figure 3 maps the global input data to a shared array on line 6. Indexing into the shared array is based only on the CUDA thread index and `BLOCK_DIM` which is a compile-time constant. Therefore, the assignment establishes a direct parametric relationship between the shared memory indices and the global memory indices from which the shared memory values originated.

Using shared memory in this manner works well on the GPU, but on Rigel it simply creates a redundant copy of memory with the same access time. Instead of using shared memory, a simple mapping function can be used. It is important to note that using a simple mapping function does not allow us to get rid of all shared memory usage. To use a mapping function, shared memory must be

filled in a uniform way based on the thread index [6]. Also, it must be filled in the same basic block across all kernels. The values in shared memory can only be used to store global data and can never be used to store values generated in the thread block itself. While there are several restrictions on when shared memory removal can be used, we find that it works in many common cases. In the transpose kernel example, we can transform the code into that of Figure 4, which instead of assigning values to a shared memory variable, simply defines, given a shared memory index, a mapping expression for the corresponding global memory location holding that value.

```
 1: __global__ void transpose(float odata[][], float idata[][], int width, int height) {
 2:   __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
 3:   // read the matrix tile into shared memory
 4:   unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
 5:   unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
 6:   block[threadIdx.y][threadIdx.x] = idata[yIndex1*width + xIndex1];
 7:   __syncthreads();
 8:   // write the transposed matrix tile to global memory
 9:   unsigned int xIndex2 = blockIdx.y * BLOCK_DIM + threadIdx.x;
10:   unsigned int yIndex2 = blockIdx.x * BLOCK_DIM + threadIdx.y;
11:   odata[yIndex2 * height + xIndex2] = block[threadIdx.x][threadIdx.y];
12: }
```

**Fig. 3.** Original Transpose kernel using shared memory

```
 1: __global__ void transpose(float odata[][], float idata[][], int width, int height)  {
 2:   unsigned int xIndex, yIndex;
 3:   // Set up the index mapping defined by the shared memory assignment statement
 4:   #define BLOCK(_a,_b) idata[(blockIdx.y * BLOCK_DIM + _a) * width + \
 5:                              blockIdx.x * BLOCK_DIM + _b]
 6:   // write the transposed matrix tile to global memory
 7:   xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
 8:   yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
 9:   odata[yIndex * height + xIndex] = BLOCK(threadIdx.x, threadIdx.y);
10: }
```

**Fig. 4.** Transpose kernel after shared memory and synchronization removal

Thread block synchronization is a relatively low latency operation on a GPU; however, on Rigel `__syncthreads()` is implemented in software. Furthermore, when shared memory removal is effective, much of the synchronization coordinating reads and writes of shared memory becomes functionally meaningless. For instance, after shared memory removal, the `__syncthreads()` call no longer has any semantic effect because threads are no longer sharing data or dependent on each other's operations. In cases like this, the removal of synchronization is both desirable and possible without affecting the code semantics.

In order to automate shared memory removal, shared memory and global memory indexing functions must be the same for each CUDA thread and only depend on compile-time constants and thread or block indices. Also, the shared memory indexing functions must be invertible and substituted into the global indexing function. Once all shared memory is removed, thread block synchronization can also be removed if there are no atomic operations involving global device memory before the synchronization points.

### 3.2 Runtime Optimizations

Unlike GPUs, Rigel uses software to handle the work distribution from CUDA kernels as shown in Figure 5. Using software is advantageous because not every kernel must be executed the same way. We look at static work partitioning and thread fusing to achieve better performance on Rigel.

**Static Work Partitioning.** The RCUDA runtime supports load balancing at the cluster level by allowing individual cores to fetch CUDA threads on demand. Dynamic fetching can be expensive for short threads or threads with many synchronization points. An optimization is to statically assign work to each Rigel core such that each core executes a fixed portion of CUDA threads within a thread block. For static work assignment to perform optimally, the CUDA threads must perform similar amounts of work and the number of CUDA threads should be divisible by eight so that each Rigel core does the same amount of work. Since static work partitioning does not change the kernel code, it can be applied to any kernel without the risk of generating incorrect code.

```
while (TasksLeft) {
  threadIdx = get_next_tid_2d();
  ExecCUDAThread(threadIdx.x, threadIdx.y);
}
```

**Fig. 5.** Original RCUDA worker thread

```
while (TasksLeft) {
  threadIdx.y = get_next_y();
  ExecCUDAThread(threadIdx.x+0, threadIdx.y);
  ExecCUDAThread(threadIdx.x+1, threadIdx.y);
  ExecCUDAThread(threadIdx.x+2, threadIdx.y);
  ExecCUDAThread(threadIdx.x+3, threadIdx.y);
}
```

**Fig. 6.** RCUDA worker thread with fused threads

**Thread Fusing** For some kernels it is advantageous to enforce an execution order as a way to optimize memory accesses. One method to enforce an execution order is to group threads into larger units of work. Thread fusing is a source level transformation that merges threads into a group so they can execute in parallel through software pipelining, as shown in Figure 6.

In CUDA code with a two-dimensional thread block, it is common to see an indexing function based on the thread index. For example:

```
(threadIdx.y * BLOCK_SIZE) + threadIdx.x
```

The `Y` dimension is multiplied by a constant factor, usually the block size or some other constant such as the width of an input matrix. On the other hand, the `X` dimension is used to increment the index. With this information, it is possible to pick an execution ordering of the CUDA threads that is more efficient. It is beneficial to concurrently execute CUDA threads with the same `Y` value so that the cores hit the same cache lines in the shared cluster cache.

In addition to enforcing an execution order, fusing threads is also advantageous since it allows the compiler to optimize across a group of threads. The code in CUDA threads is the same, except for the thread index values, and with thread fusing the compiler is able remove redundant computation, creating faster, more efficient code.

In order to automate this optimization, the indexing must be analyzed in terms of the CUDA thread index components. On Rigel, having threads that access consecutive elements in parallel is advantageous. If the indexing pattern cannot be determined, and even if the incorrect indexing pattern is chosen, the code still executes correctly, but may not run as efficiently.

## 4 Evaluation

In this section, we describe our simulation and measurement methodology, benchmarks, and results. We discuss the optimizations applied to each benchmark and analyze the change in performance. We then take a detailed look at different implementations of dense-matrix multiplication (`DMM`) and analyze their performance portability.

### 4.1 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 [4]. We use GDDR5 memory timings for the DRAM model. Benchmark and library codes are run in the simulator and are compiled with LLVM 2.5 using a custom backend. Inline assembly was used for global and cluster level atomic operations. Optimizations have yet to be fully implemented in our compiler, and thus were applied by hand editing translated CUDA kernels. Results for CUDA on GPU were gathered on a Tesla [8] T10 4-GPU server using one GPU.

**Table 1.** Benchmarks used for evaluating RCUDA performance

| Name | Data Set | # Kernels | Thread Block Dimensions | Shared Memory |
|---|---|---|---|---|
| Convolve | 1024x1024 | 1 | (16,16,1) | Yes |
| DMM | 1024x1024 | 1 | (16,16,1) | Yes |
| Histogram | 2M | 2 | (192,1,1),(256,1,1) | Yes |
| Mandelbrot | 512x512 | 1 | (16,16,1) | Yes |
| MRI | 8192,8192 | 2 | (512,1,1),(256,1,1) | No |
| SAXPY | 2M | 1 | (512,1,1) | No |
| Transpose | 1024x1024 | 1 | (16,16,1) | Yes |

### 4.2 Benchmarks

We evaluate the seven benchmarks shown in Table 1. With the exception of `MRI` and `SAXPY`, all benchmark codes were taken from external sources and were originally written to be executed on a GPU. Our benchmarks include a 2D image filter with 5x5 kernel (`Convolve`), dense-matrix multiply (`DMM`), 256-bin histogram (`Histogram`), fractal generation (`Mandelbrot`), medical image construction (`MRI`) [13], SAXPY from BLAS (`SAXPY`) and matrix transpose (`Transpose`). `MRI` uses two kernels: the first to initialize data structures, and the second to perform the actual computation. `Histogram` also uses two kernels: the first calculates many partial histograms for a subset of the input array, and the second merges the partial histograms. Table 1 lists data sizes and characteristics for all benchmarks.

### 4.3 Baseline Performance

In Figure 7 we show 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 performance portability.

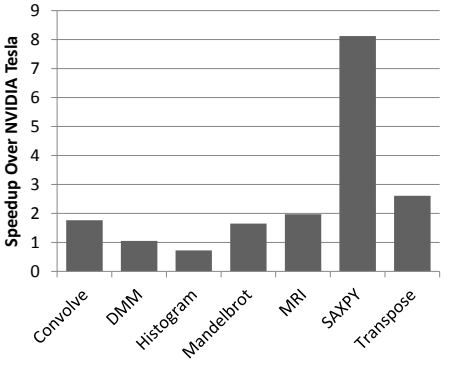



**Fig. 7.** Baseline speedup on Rigel

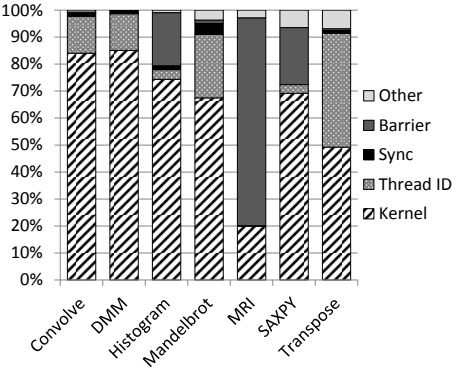



**Fig. 8.** RCUDA runtime overhead

### 4.4 RCUDA Runtime Overhead

We analyze the runtime overhead of our RCUDA framework on Rigel, shown in Figure 8. We break down runtime into five categories: (1) *Kernel*, which is the measurement of the time spent executing the CUDA kernel code. (2) *Thread ID*, is 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*, measuring the amount of time cores spend waiting for kernel execution to complete which represents load imbalance. (5) *Other*, all other overheads including runtime initialization, thread block fetch and host code.

We see that thread index generation is quite expensive, particularly for kernels with two-dimensional thread blocks. For two-dimensional thread blocks, the CUDA thread indices are generated from a count of remaining threads. 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 smaller 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. We find that the time spent in the `__syncthreads()` call is low, even though it is implemented in software. We see that in `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 and `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.

### 4.5 Optimizations

We apply optimizations individually to each benchmark and then combine the beneficial optimizations to create an optimal version of each benchmark as shown in Figure 9.

Shared memory removal was applied to the Convolve, DMM and Transpose benchmarks. Removing the shared memory accesses also allowed for all the synchronization points to be removed. DMM was the only benchmark where the optimization did not improve the runtime because the mapping function generated for DMM is complex, requiring costly multiplication instructions. Histogram also uses shared memory, but uses shared memory as a scratch pad, not to store global values, so the shared memory cannot be removed.

All benchmarks except Convolve and SAXPY showed an improvement when using static scheduling of threads. Convolve is the only benchmark where the amount of work varies greatly between threads because not all CUDA threads compute an output value. SAXPY has very short kernels, so the overhead of statically dividing the workload is significant, and the runtime increases by 10%. Thread fusing improves the performance of all benchmarks; in every case, multiple CUDA threads can be combined, removing redundant calculations.

The optimal version of each benchmark uses the combination of optimizations that results in the fastest runtime. Convolve uses shared memory accesses removal along with thread fusing. DMM and Histogram use static work partitioning and thread fusing. Mandelbrot, MRI, Transpose and SAXPY only use thread fusing.

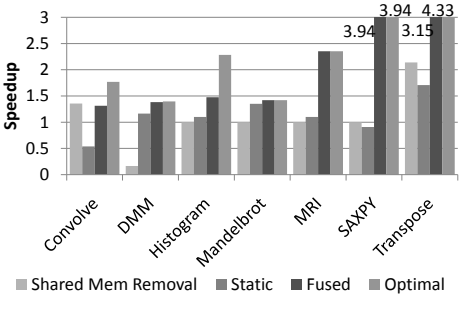



**Fig. 9.** Speedup over naïve translation

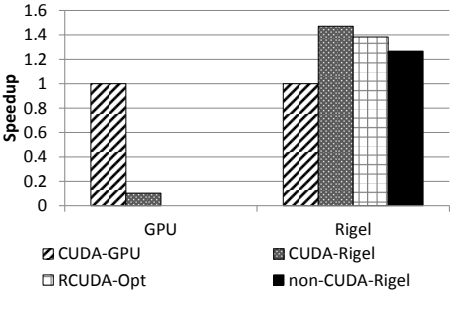



**Fig. 10.** Speedup of DMM benchmarks

### 4.6 DMM Case Study of Performance Portability

We evaluate several implementations of DMM running on the GPU and Rigel as shown in Figure 10. The benchmarks include the translated CUDA version

(`CUDA-GPU`), the CUDA version with our optimizations (`RCUDA-Opt`), a CUDA benchmark written for our architecture (`CUDA-Rigel`) and finally a native version for Rigel (`non-CUDA-Rigel`) which uses a custom task-based programming model targeted at the architecture. The custom task programming model is less restrictive than CUDA because individual tasks may have different instructions and tasks can be enqueued at any time during execution. Here, each benchmark is computing a matrix multiply between two 512x512 matrices.

On Rigel, we find that both `RCUDA-Opt` and `CUDA-Rigel` perform better than the native implementation (`non-CUDA-Rigel`). The performance difference is due to the very uniform nature of the DMM computation, and having a less restrictive programming model adds extra overhead for features that are not used. The `Rigel-CUDA` 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. The `non-CUDA-Rigel` benchmark follows a similar approach, but uses a finer-grained blocking resulting in more tasks which in turn increases the overhead of the software runtime. These results show that performance portability is achieved between the GPU and Rigel, the optimized CUDA code (`RCUDA-Opt`) is less than 10% slower than the best performing code on Rigel (`CUDA-Rigel`).

## 5 Related Work

GPUocelot [1] is a binary translation framework that allows for execution of CUDA code on a multi-core CPU without recompilation. ZPL [12] is an implicitly parallel programming language designed to achieve performance portability [7] across various MIMD parallel computing platforms. OpenCL is a parallel programming model that is platform independent and designed to work on a variety of architectures including CPUs and GPUs. Kernels written in OpenCL are similar to CUDA kernels and optimizations similar to those presented would likely work for OpenCL applications. Autotuning [2,3,9] is a means to improve performance via automatic optimization using runtime measurement and can help to provide performance portability by optimizing performance intensive kernels for particular platforms. Autotuning may be used in combination with the techniques presented in this work, for example by [11].

## 6 Conclusion

We find that achieving performance portability on a MIMD architecture is possible for existing SPMD code originally tuned for a GPU. With optimizations that leverage the characteristics of optimized CUDA code, we see a significant speedup across several benchmarks. These optimizations do not change the underlying algorithm in the code and it is reasonable to assume that they could be automated using standard compiler analysis.

**Acknowledgments.** The authors gratefully acknowledge generous donations by Advanced Micro Devices, Intel Corporation and Microsoft Corporation as well as support from the Information Trust Institute of the University of Illinois at Urbana-Champaign and the Hewlett-Packard Company through its Adaptive Enterprise Grid Program. Lumetta was supported in part by a Faculty Fellowship from the National Center for Supercomputing Applications. The content of this paper does not necessarily reflect the position nor the policies of any of these organizations.

## References

1. Diamos, G.: The design and implementation ocelot's dynamic binary translator from ptx to multi-core x86. Technical Report GIT-CERCS-09-18, Georgia Institute of Technology (2009)
2. Frigo, M., Johnson, S.: FFTW: an adaptive software architecture for the fft, vol. 3, pp. 1381–1384 (May 1998)
3. Dongarra, R.C.J.: Automatically tuned linear algebra software. Technical report, Knoxville, TN, USA (1997)
4. Kelm, J.H., Johnson, D.R., Johnson, M.R., Crago, N.C., Tuohy, W., Mahesri, A., Lumetta, S.S., Frank, M.I., Patel, S.J.: Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In: Proceedings of the International Symposium on Computer Architecture, pp. 140–151 (June 2009)
5. Khronos OpenCL Working Group. OpenCL Specification, 1.0 edn. (December 2008)
6. Kofsky, S.M.: Achieving performance portability across parallel accelerator architectures. Technical Report (to Appear), Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL (2010)
7. Lin, C.: The portability of parallel programs across MIMD computers. PhD thesis, Seattle, WA, USA (1992)
8. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro 28(2), 39–55 (2008)
9. Moura, J.M.F., Johnson, J., Johnson, R.W., Padua, D., Prasanna, V.K., Püschel, M., Veloso, M.: SPIRAL: Automatic implementation of signal processing algorithms. In: High Performance Embedded Computing, HPEC (2000)
10. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue 6(2) (2008)
11. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S.-Z., Stratton, J.A., mei, W., Hwu, W.: Program optimization space pruning for a multithreaded GPU. In: CGO 2008: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 195–204. ACM, New York (2008)
12. Snyder, L.: The design and development of ZPL. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, pp. 8–1–8–37. ACM, New York (2007)
13. Stone, S.S., Haldar, J.P., Tsao, S.C., Hwu, W.W., Sutton, B.P., Liang, Z.P.: Accelerating advanced mri reconstructions on GPUs. J. Parallel Distrib. Comput. 68(10), 1307–1318 (2008)
14. Stratton, J.A., Stone, S.S., Hwu, W.-M.W.: MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs, pp. 16–30 (2008)