CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization [*]

Michael Bauer
Stanford University
mebauer@cs.stanford.edu

Henry Cook
UC Berkeley
hcook@cs.berkeley.edu

Brucek Khailany
NVIDIA Research
bkhailany@nvidia.com

ABSTRACT

As the computational power of GPUs continues to scale with Moore's Law, an increasing number of applications are becoming limited by memory bandwidth. We propose an approach for programming GPUs with tightly-coupled specialized DMA warps for performing memory transfers between on-chip and off-chip memories. Separate DMA warps improve memory bandwidth utilization by better exploiting available memory-level parallelism and by leveraging efficient inter-warp producer-consumer synchronization mechanisms. DMA warps also improve programmer productivity by decoupling the need for thread array shapes to match data layout. To illustrate the benefits of this approach, we present an extensible API, CudaDMA, that encapsulates synchronization and common sequential and strided data transfer patterns. Using CudaDMA, we demonstrate speedup of up to 1.37x on representative synthetic microbenchmarks, and 1.15x-3.2x on several kernels from scientific applications written in CUDA running on NVIDIA Fermi GPUs.

1. INTRODUCTION

The merits of using GPUs for scientific and HPC applications are clear. GPUs are being incorporated into large-scale supercomputing installations throughout the world. On the June 2011 Top500 list, GPUs were included in 3 of the top ten installations [8], and on the Green500 list from the same month, GPUs were included in 5 of the top ten machines [6]. Accelerators designed to exploit fine-grain data parallelism are on track to be the next big advance in supercomputing.

In order for programmers to achieve peak performance on installations with GPUs, their algorithms and code will have to exploit the performance potential of these accelerators. Accomplishing this task mandates new software tools to increase programmer productivity and boost achieved performance. Tools for exploiting common algorithmic patterns such as sorting [1] or canonical HPC algorithms [3, 4] on GPUs have already been introduced. However, for programmers developing new algorithms, such libraries are only small components of much larger applications.

For many programmers, the most difficult part of creating high-performance applications that leverage GPUs is managing the balance between computational intensity and memory bandwidth. Effectively exploiting both GPU computational resources and memory bandwidth is critical to achieving peak per-node performance. This task is complicated because the programmer must use the same parallel hierarchy of threads to carry out both the computation and the transfer of data between memories. This model works well for cases where the size and dimensionality of the data transferred is geometrically similar to the size and dimensionality of the thread hierarchy. However, for many applications there are sufficient differences to create difficulties for the programmer.

In previous accelerators, such as the Cell Broadband Engine [14] and the Imagine Stream Processor [20], the issue of moving data between on-chip and off-chip memories was solved by the use of asynchronous hardware DMA engines. These systems delegated the responsibility of data movement to the hardware, enabling the programmer to focus on optimizing the computation being performed. The driving force behind the CudaDMA project is to provide a similar feature for GPUs at a software level.

In this paper we present CudaDMA, an extensible API for efficiently managing data transfers between the on-chip and off-chip memories of GPUs. CudaDMA enables the programmer to decouple the size and dimensionality of the data from the size and dimensionality of the computation, improving both programmability and performance.

Decoupling is achieved by specializing warps into compute warps and DMA warps. Compute warps are solely responsible for performing computation while DMA warps are solely utilized for moving data between on-chip and off-chip memories. The CudaDMA API provides the abstractions and synchronization primitives necessary for warp specialization.

We present two instances of CudaDMA that support DMA warps for performing common sequential and strided data transfer patterns. These instances encapsulate a variety of expert-level bandwidth optimization techniques, allowing them to be deployed with minimal programmer effort. We also exhibit how custom instances of the CudaDMA API

[*]This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

can be created for application-specific transfer patterns or leveraging advanced programming techniques.

This paper is organized into the following sections. In Section 2 we cover the basics of the CUDA programming model and the challenges it can present. Section 3 introduces the CudaDMA API. We cover the benefits and use cases of CudaDMA in Section 4. Sections 5 and 6 present the performance of CudaDMA on microbenchmarks and real applications. Related work is described in section 7. Sections 8 and 9 discuss future work and offer conclusions.

2. MOTIVATION

2.1 GPU Architecture and CUDA

CUDA is a general purpose programming language for programming GPUs. Each CUDA-enabled GPU consists of a collection of *streaming multiprocessors* (SMs). A SM possesses an on-chip register file, as well as an on-chip scratchpad memory that can be shared between threads executing on the same SM. DRAM memory is off-chip, but is visible to all SMs.

The CUDA programming model targets this GPU architecture using a hierarchy of threads. Threads are grouped together into *threadblocks*, also known as *cooperative thread arrays* (CTAs). CTAs are correspondingly grouped into a subsequent array structure referred to as a *grid*.

When a grid is executed on the GPU, the hardware schedules CTAs onto SMs. All the threads within a CTA execute on the same SM in groups of 32 threads. This collection of 32 threads is referred to as a *warp*. All the threads within a warp share the same instruction stream. Control divergence within a warp can lead to performance degradation, but inter-warp divergence will *not* harm performance. The CUDA programming model encourages the view that all warps will execute the same instruction stream, but there are advantages to breaking through this abstraction.

From the perspective of a thread executing on an SM there are three types of memory. First, each thread is allocated a set of private, on-chip registers in the register file. Second, a thread has access to the on-chip scratchpad memory, called *shared* memory because it is visible to all the threads in the same CTA. Finally, all of the threads on the GPU have access to *global* memory which consists of off-chip DRAM. On-chip memories are two orders of magnitude faster to access than off-chip memory.

Due to the extreme difference in access latencies between on-chip and off-chip memories, CUDA encourages programs to be written in a way that first moves data into on-chip memories. Computation is then performed through this on chip memory. When the computation is completed, results are written back to global memory. Figure 1 illustrates this paradigm. To enable threads in a CTA to coordinate the loading and storing of data, a light-weight barrier mechanism is provided for synchronization.

2.2 GPU Programmability Challenges

The first challenge encountered by programmers when they attempt to program GPUs using the paradigm in Figure 1 revolves around programmability. Programmers routinely choose the size and dimensionality of their CTAs based on the computation being performed as opposed to the size and dimensionality of the data being accessed.

The example paradigm is extremely easy to code and maintain when both the size and dimensionality of a CTA closely matches the properties of the data to be used in shared memory. Each thread loads an element, operates on that element, and then stores the result.

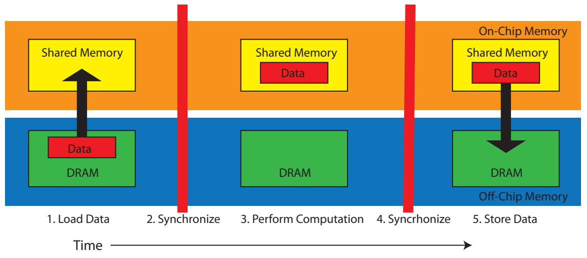


Figure 1: Common CUDA programming paradigm illustrating the movement of data between on-chip and off-chip memories.

For applications where the size and dimensionality of the CTA and the data are sufficiently different, it is unclear which data elements a thread should be responsible for loading and storing from global memory. An example of such an application would be one that performs a multi-dimensional stencil algorithm. The CTA size is based on the number of output points, but the data that must be transferred to perform the computation is based on the order of the stencil which is unrelated to CTA shape. This mismatch between CTA and data properties will result in many conditional statements that clutter the code and make it more difficult to discern the intention of the programmer when maintaining the code in the future.

2.3 GPU Performance Challenges

In addition to programmability, the other challenge to programming using the paradigm in Figure 1 is performance. A common approach to classifying application performance has been to examine the compute-to-memory ratio of a computation [23]. Algorithms with low compute-to-memory ratios (e.g. BLAS 1 kernels, sparse matrix-vector multiply) typically have little data reuse and can easily saturate the memory system. Algorithms with high compute-to-memory ratios (e.g. BLAS 3 kernels) often have significant spatial and temporal data locality; their performance is dominated by instructions per clock (IPC) limits encountered when consuming on-chip data. The performance of both these classes of applications is dominated by the inherent limits of the underlying computer architecture and cannot be improved beyond these limits at the software level.

With this performance model in mind, we created a synthetic micro-benchmark to illustrate how staging data through shared memory affects application performance. The micro-benchmark runs a generic load/store loop with variable compute intensity over a large dataset by having each CTA process a segment of the data. In order to model the performance effects of staging through shared memory, every CTA executes the first three steps of the programming paradigm shown in Figure 1. CTAs copy 2 KB of data from DRAM to shared memory, synchronize, and perform a computation on the data in shared memory. We ran our benchmark on a Tesla C2050 GPU (ECC off) with 14 SMs and a 1.15 GHz clock. Our kernel launches 2 CTAs per SM (28 CTAs total).

Throughput on the micro-benchmark, plotted as DRAM GB/s, is shown in Figure 2. We vary the number of multiply-

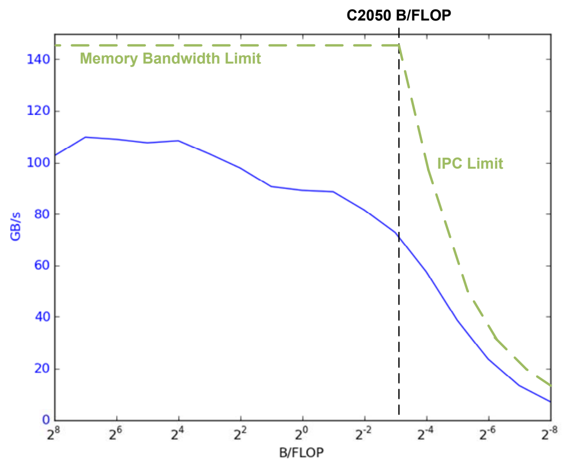


Figure 2: Shared memory staging micro-benchmark performance, with naive transfer implementation.

adds in each loop iteration to perform a sweep from very low compute-to-memory ratios (expressed as bytes/FLOP) to very high. At low compute intensity (BLAS1), throughput is memory-bandwidth limited, sustaining around 75% of the 144 GB/s peak memory bandwidth, compared to a practical limit of 85% of peak when accounting for overheads such as DRAM refresh. At high compute intensity (BLAS3), throughput is limited by instruction-issue bandwidth.

The interesting region is where the application compute intensity approaches the B/FLOP ratio of the underlying hardware. This region is where the most slowdown from staging data through shared memory takes place. At the ratio for our Tesla platform, 0.14 B/FLOP, the benchmark is only sustaining around 50% of peak DRAM bandwidth. For applications with "balanced" computational intensities similar to that of the underlying machine, performance degradation is caused by bottlenecks in both the memory system as well as the computational resources. We now elaborate on the potential bottlenecks in both areas in more detail.

2.3.1 *Computational Bottlenecks*

For an application that has adopted the CUDA programming paradigm introduced in Figure 1, there are three common factors related to data transfers between off-chip and on-chip memories that could limit instruction issue bandwidth and prevent full utilization of all the GPU's computational resources.

Many long-latency memory accesses: If enough long-latency memory accesses fill up the memory system's buffers, then a warp's instruction stream will stall due to the in-order nature of SMs, preventing independent computational instructions from being issued.

Coarse-grained synchronization: Using coarse grained barriers implies that all threads within a CTA must always wait for the slowest warp to finish executing before execution on all warps can continue, even when independent work could be executed.

Data Access Patterns: Accessing data different from the CTA size and dimensionality will require conditional branches which can lead to intra-warp divergence and bank conflicts in accessing shared memory.

2.3.2 *Memory System Bottlenecks*

The other challenge for applications with balanced compute-to-memory ratios is fully exploiting the available memory system resources. The GPU memory system is designed to support many parallel loads and stores in flight simultaneously. Issuing multiple memory accesses simultaneously and allowing the memory system to handle them in parallel is referred to as *Memory Level Parallelism* (MLP).

Achieving high MLP on a GPU can commonly be impeded by two factors.

Instruction Issue: If a warp's instruction stream stalls due to computational resources being over-subscribed then independent memory operations could be prevented from issuing.

Data Access Patterns: Accessing data sufficiently different from the CTA size and dimensionality can prevent memory operations from coalescing, which would result in serialization. This effect severely decreases the MLP achieved by a GPU.

The critical insight into these problems, and the motivation for CudaDMA, is that the bottlenecks in both areas are coupled. The root cause of this entanglement is the requirement encouraged by the CUDA programming model that threads of a CTA perform both memory accesses and computation. By creating specialized warps that perform independent compute and memory operations we can tease apart the issues that affect memory performance from those that affect compute performance. By decoupling the problems, we remove the entanglement and enable the programmer to address performance bottlenecks in balanced compute-to-memory ratio applications in isolation.

3. CUDADMA API

The CudaDMA library provides the basis for moving data between on-chip shared memory and off-chip global memory of CUDA-enabled GPUs. The fundamental assumption underlying the library's API is that, for many algorithms, the most efficient transfer implementation is to divide a thread block into differentiated subsets of threads. As long as this partitioning is performed at warp granularity, there is *no* performance penalty for divergence. We call this differentiation technique *warp specialization*. Warp specialization has been employed previously for efficient parallel implementations of sorting algorithms on GPUs [17]. CudaDMA encapsulates this technique into a library to make it generally available to a wider range of application workloads.

There are two classes of warps into which threads can be assigned using CudaDMA. *DMA warps* are exclusively in charge of transferring data between global and shared memory. *Compute warps* perform the actual computation by processing the data that has been transferred to on-chip memory. The API is designed to aid programmers in specializing their warps by making it clear what code will be executed by compute warps and what code by DMA warps.

The CudaDMA API is object based. Users create a `cudaDMA` object within a device kernel to manage the transfer of data for a shared memory buffer. Multiple `cudaDMA` objects can be created to manage multiple buffers. Every `cudaDMA` object implements the CudaDMA API seen in Figure 3. The base class of the CudaDMA API provides synchronization methods for coordinating between compute and DMA warps as

```
class cudaDMA {
  __device__ cudaDMA (
                const int dmaID,
                const int num_dma_threads,
                const int num_compute_threads,
                const int dma_threadIdx_start );

  __device__ void execute_dma(
                void* src_ptr,
                void* dst_ptr );

   __device__ bool owns_this_thread();
   __device__ void start_async_dma();
   __device__ void wait_for_dma_start();
   __device__ void finish_async_dma();
   __device__ void wait_for_dma_finish();
};
```

Figure 3: Interface for CudaDMA objects.

well as a call for transferring data between on-chip and off-chip memories. Sub-classes of the CudaDMA API are created for specific transfer patterns by overriding the transfer method. To illustrate specifically how the CudaDMA API works we now present a working example.

3.1 API Example: SGEMV

The most straightforward way of illustrating the use of the CudaDMA API is to examine how it is employed in a piece of application code. Figure 4 presents code implementing a single-precision matrix-vector multiplication (SGEMV) routine from the BLAS dense linear algebra library.

In our implementation, each CTA is responsible for computing the inner product of a subset of rows in the matrix and the vector. To do so, every thread must access every element of the vector, making it beneficial to load the vector into shared memory. However, the vectors are often sufficiently large that they must be loaded iteratively in small blocks. In this example, we also load subsets of the matrix into shared memory as well as the vector for performance reasons that are explained in detail in Section 6.1.

The subsets of matrix data and vector data have different layouts in memory. The 1-D vector data are all sequential, whereas the 2-D matrix data are strided across memory. Handling these different access patterns with CudaDMA simply requires employing two predefined instances of `cudaDMA` objects. These objects are instantiated on lines 7-17 of the code in Figure 4 and are described in further detail in Section 3.2.

The `cudaDMA` objects are declared at the beginning of the kernel, outside of the iterative loop. Once the `cudaDMA` objects are instantiated, we differentiate the warps according to the assignments given to the constructors. This differentiation is implemented by the conditionals on lines 19, 35, and 44 of Figure 4.

By convention, compute warps contain the threads with the lowest thread IDs. These warps enter the main computation loop, lines 20-33, in which they calculate the inner product of the matrix and vector data stored in the shared memory buffers declared on lines 4 and 5.

The DMA warps determine which `cudaDMA` object they are managed by using the API call `owns_this_thread()`. The specialized DMA warps each have their own inner loop (lines 36-42 and 45-51), in which they repeatedly call their transfer object's `execute_dma()` method. Calling `execute_dma()` causes the DMA warps to perform a single execution of that transfer. As SGEMV demonstrates, many transfers can be iteratively launched based on a single `cudaDMA` instance,

```
1  __global__ void
2  sgemv_cuda_dma(int n, int m, float alpha, float *A,
3                 float *x, float *y) {
4    __shared__ float buff[VEC_ELMTS];
5    __shared__ float mat [VEC_ELMTS][COMPUTE_THREADS];
6
7    cudaDMASequential<sizeof(float)*VEC_ELMTS/DMA_THREADS_SEQ>
8    dma_ld_0( 1, DMA_THREADS_SEQ, COMPUTE_THREADS,
9             COMPUTE_THREADS, sizeof(float)*VEC_ELMTS);
10
11   cudaDMAStrided<sizeof(float)*VEC_ELMTS*
12                  COMPUTE_THREADS/DMA_THREADS_STRD>
13   dma_ld_1( 2, DMA_THREADS_STRD, COMPUTE_THREADS,
14            COMPUTE_THREADS+DMA_THREADS_SEQ,
15            sizeof(float)*COMPUTE_THREADS,
16            VEC_ELMTS, sizeof(float)*n,
17            sizeof(float)*COMPUTE_THREADS);
18
19   if (threadIdx.x < COMPUTE_THREADS) {
20     dma_ld_0.start_async_dma();
21     dma_ld_1.start_async_dma();
22     float res = 0.f;
23     for(int i=0; i<n; i += VEC_ELMTS) {
24       dma_ld_0.wait_for_dma_finish();
25       dma_ld_1.wait_for_dma_finish();
26       for(int j=0; j < VEC_ELMTS; j++) {
27         res+=mat[j][threadIdx.x]*buff[j];
28       }
29       dma_ld_0.start_async_dma();
30       dma_ld_1.start_async_dma();
31     }
32     ind = blockIdx.x*COMPUTE_THREADS+threadIdx.x;
33     if (ind < n) y[ind] = alpha * res;
34   }
35   else if (dma_ld_0.owns_this_thread()) {
36     dma_ld_0.wait_for_dma_start();
37     for (int idx=0; idx<n; idx += VEC_ELMTS) {
38       dma_ld_0.execute_dma(x,buff);
39       dma_ld_0.finish_async_dma();
40       dma_ld_0.wait_for_dma_start();
41       x += VEC_ELMTS;
42     }
43   }
44   else if (dma_ld_1.owns_this_thread()) {
45     dma_ld_1.wait_for_dma_start();
46     for (int idx=0; idx<n; idx += VEC_ELMTS) {
47       dma_ld_1.execute_dma(
48         A+idx*m+blockIdx.x*COMPUTE_THREADS, mat);
49       dma_ld_1.finish_async_dma();
50       dma_ld_1.wait_for_dma_start();
51     }
52   }
53 }
```

Figure 4: A CudaDMA-based implementation of the SGEMV routine from BLAS.

where the pointers passed to `execute_dma()` are changed with every iteration (lines 41, 48).

There are two synchronization points defined for every `cudaDMA` object. One synchronization point corresponds to the data having been consumed and the buffer standing empty awaiting the next transfer. The compute threads indicate this status using a non-blocking call to `start_async_dma()` (lines 20,21,29,30). The DMA threads wait to begin this transfer using the blocking call `wait_for_dma_start()` (lines 36,40,45,50). The other synchronization point corresponds to the transfer being complete and the buffer being ready for processing. DMA warps indicate that a transfer is complete using a non-blocking call to `finish_async_dma()` (lines 39,49). The compute warps wait for a transfer to complete using a blocking call to `wait_for_dma_finish()` (lines 24,25). The DMA-side calls are usually abstracted behind `execute_dma()` but are shown here for clarity. The implementation of these four calls is described in Section 4.2. The producer/consumer nature of our synchronization mechanisms allow the programmer to employ a variety of techniques to overlap communication and computation described in more detail in Section 4.3.

3.2 CudaDMA Instances

In section 3.1 we mentioned two specific instances of the CudaDMA interface: `cudaDMASequential` and `cudaDMAStrided`. These two classes are optimized instances of CudaDMA classes. The classes are characterized by their memory access pattern and a few specific parameters that serve to define that pattern. In this section we describe the use of these classes from a programmer's perspective. The techniques used to implement these classes are covered in Section 4.1.

In the example in Figure 4 the sequential access pattern required by the vector data and the strided access pattern required by the matrix data are performed by the `cudaDMA` subclasses `cudaDMASequential` and `cudaDMAStrided` respectively. Figure 5 presents the declaration of the subclass constructors that are used in our SGEMV code.

Both subclasses are defined by the following parameters, which are used in all classes derived from `cudaDMA`:

- `int dmaID`: A unique identifier for synchronization
- `int num_dma_threads`: The number of DMA threads that will be used to carry out this transfer.
- `int num_compute_threads`: The number of compute threads that will synchronize with this transfer.
- `int dma_threadIdx_start`: The starting location of this transfer's assigned threads within the thread block.

In addition to the default parameters, each subclass introduces additional parameters that define specific aspects of their behavior. `cudaDMASequential` transfers simply move a contiguous block of memory; the only parameter they need is the size of the transfer in bytes (line 10). `cudaDMAStrided` transfers fetch multiple chunks of data, each offset from the other by a user-defined stride; they have additional parameters to define element size, element count, and source and destination strides (lines 16-17).

The template parameter, `MAX_BYTES_PER_THREAD`, is used by both classes to calculate constant offset values used within the transfer functionality. This value is the maximum number of bytes that each thread might have to transfer at runtime; smaller transfer sizes will work correctly as well. Requiring a maximum value to be defined at compile time increases code efficiency but is not as restrictive as requiring the actual transfer size to be defined at compile time.

The critical insight concerning both of these CudaDMA instances is that they do not require the programmer to specify how the transfer should actually be performed. By allowing the programmer to state the nature and parameters of the access pattern explicitly and separately from the implementation of that transfer, our library abstracts away the logic involved in determining how best to perform the transfer. Optimized versions of CudaDMA instances for common transfer patterns can be written by expert programmers and then re-used by application programmers, providing good performance and high productivity.

In some cases, programmers would prefer to define their own data transfer pattern while still leveraging the power of the CudaDMA API. To make the CudaDMA library more easily extensible, we provide a `cudaDMACustom` class which contains implementations of the synchronization functions but leaves the behavior of the transfer unspecified. The

```
template <int MAX_BYTES_PER_THREAD>
class cudaDMASequential : public cudaDMA {
  __device__ cudaDMASequential (
               const int dmaID,
               const int num_dma_threads,
               const int num_compute_threads,
               const int dma_threadIdx_start,
               const int sz )

template <int MAX_BYTES_PER_THREAD>
class cudaDMAStrided : public cudaDMA {
  __device__ cudaDMAStrided (
               const int dmaID,
               const int num_dma_threads,
               const int num_compute_threads,
               const int dma_threadIdx_start,
               const int el_sz,
               const int el_cnt,
               const int src_stride,
               const int dst_stride )
```

Figure 5: Constructors for the two access patterns used in SGEMV.

techniques used for optimizing custom implementations of the CudaDMA API are covered in section 4.1. Further information on CudaDMA instances can be found in the CudaDMA user manual [5].

3.3 CudaDMA Generality

Use of the CudaDMA API is predicated on the computation being a *streaming computation* [20, 21]. Our definition of streaming is a computation which loops over a dataset too large to fit in on-chip memory by processing a sub-block of that dataset during each loop iteration. For example, the SGEMV application from Section 3.1 loads the input vector in multiple loop iterations or stages. For each stage, data is first loaded into shared memory and then is consumed. This process is repeated multiple times to compute each output value. Requiring that a computation be streaming allows us to amortize the overhead of the coordination between the compute and DMA warps over the many inputs that will be processed by a single CTA.

While many CUDA applications are explicitly streaming computations, many more are not. However, any application written in CUDA can easily be transformed into a streaming computation. This transformation is accomplished by making a single CTA in a CudaDMA application responsible for the work of multiple CTAs from the original application. The CudaDMA approach to GPU programming is therefore general enough to be applied to any CUDA program.

4. CUDADMA METHODOLOGY

In Section 3 we demonstrated how the CudaDMA API operates from a programmer's perspective. In this section we investigate the performance-enhancing techniques that are enabled by the CudaDMA API.

4.1 Optimizing Performance with Warp Specialization

Optimizing the performance of a GPU kernel is primarily about managing constrained resources such as registers, shared memory, instruction issue slots, and memory bandwidth. Warp specialization allows subsets of threads within a CTA to have their behavior tuned for a particular purpose which enables more efficient consumption of constrained resources. There are several techniques that we use in conjunction with warp specialization to conserve resources.

Hoisting Pointer Arithmetic: The CudaDMA API allows hoisting of pointer arithmetic out of inner loops and into constructors which permits us to pre-compute offsets and save integer arithmetic issue slots. By reducing the number of intermediate temporaries required for pointer arithmetic we can also save registers.

Templating Parameters: Although we avoid using template parameters for exact values for programmability reasons, we do employ them to place bounds on parameters to CudaDMA classes. Template parameters enable the compiler to perform constant folding, saving both integer arithmetic and registers.

Exploiting MLP: DMA warps can issue loads and stores of vector data types (i.e. `float4`) to fully saturate memory bandwidth [22]. This optimization also saves memory instruction issue slots enabling DMA warps to exploit even greater application MLP.

Exploiting ILP: Exploiting application ILP at lower occupancy has been shown to be an effective performance technique [22]. By decoupling the compute and DMA warps, CudaDMA enables this approach without sacrificing memory system performance. Fewer compute threads can also save registers.

Fine-Grained Synchronization: Using fine-grained producer-consumer synchronization primitives reduces barrier overheads and ensures that there will always be active warps available for the SM to schedule.

Prefetching Data: The CudaDMA API enables DMA warps to prefetch data into registers before waiting for the compute warps to instruct them to write the values into shared memory. Prefetching enables better overlapping of computation and memory accesses.

Avoiding Memory Conflicts: DMA warp behavior is not constrained by the logical structure of the data transferred, enabling loads and stores from DMA warps to be engineered for maximal global memory coalescing and minimal shared memory bank conflicts. This optimization saves instruction issue slots by avoiding instruction replays and optimizes both global and shared memory bandwidth usage.

In addition to these explicit techniques, warp specialization improves performance by allowing the compiler to do a better job of instruction scheduling and resource allocation. Warp specialization separates memory and compute operations into two different instruction streams. The compiler's job is greatly simplified by only having to optimize a few metrics in independent instruction streams, rather than multiple performance metrics across a single, mixed stream leading to better machine code.

4.2 Fine-Grained Synchronization

The efficacy of the CudaDMA API and the technique of warp specialization is contingent on the ability to perform efficient synchronization events at a finer granularity than the width of an entire CTA. The canonical `__syncthreads()` intrinsic generates a CTA-wide barrier instruction. Every thread in the CTA must issue the instruction, and every thread must wait until all others arrive at the barrier, possibly leading to unused warp-issue slots when waiting for the last warps to arrive at a barrier. More parallelism and higher efficiency can be achieved by leveraging the advantages of fine-grained, named producer-consumer synchronization events between threads in a CTA [9].

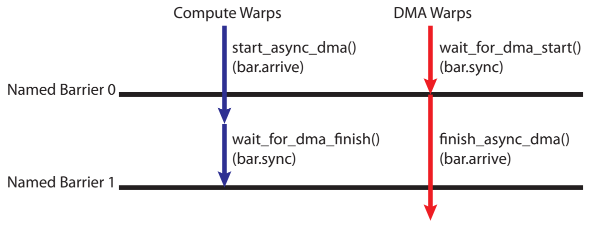


Figure 6: Use of named barriers in CudaDMA.

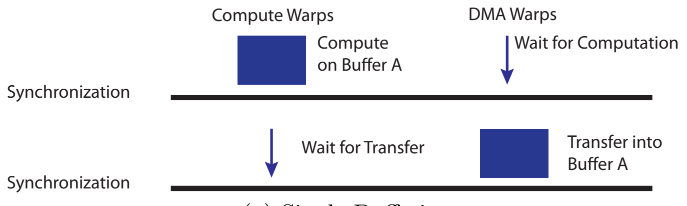
We accomplish fine-grained synchronization by using inlined PTX assembly to express *named barriers*. Named barriers are hardware resources that support a barrier operation for a subset of warps in a CTA and can be identified by a unique name (e.g. immediate value in PTX). There are two named barriers associated with every `cudaDMA` object. Two barriers are required to track whether the data buffer in shared memory is full or empty. We use the PTX instruction `bar.arrive`, which allows a thread to signal its arrival at a named barrier without blocking the thread's execution [7]. This functionality is useful for producer-consumer synchronization by allowing a producer to indicate that a data transfer has finished filling a buffer while permitting the producer thread to continue to perform work. Similarly, a consuming thread can use the same instruction to indicate that a buffer has been read and is now empty. For blocking operations we use the PTX instruction `bar.sync` to block on a named barrier. Figure 6 presents a graphical depiction of the way named barriers operate as well as their relation to the CudaDMA API calls described in Section 3.1. Section 4.3 demonstrates the power of the producer/consumer abstraction by illustrating several different ways to use these named barriers for buffering data transfers.

4.3 Buffering Techniques

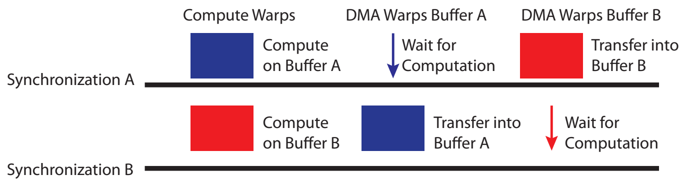
The simplest approach to writing code using CudaDMA is to allocate a separate buffer for each transfer to be performed and to associate a `cudaDMA` object with each buffer. We refer to this approach as *single buffering* since there is a single buffer for each transfer being performed by a set of DMA warps. Figure 7(a) illustrates how single buffering works in CudaDMA. The important aspect of single buffering is that at any point in time only the DMA threads or the compute threads are active, indicating that the memory system or computational resources may not be fully utilized. Despite this possibility, single buffering is still an effective technique, especially if the programmer ensures that multiple CTAs are concurrently resident on an SM. This concurrency will allow the hardware to overlap the compute threads of one CTA with the DMA warps of another.

If single buffering is not exploiting enough MLP to keep the memory system busy, an alternative is to create two buffers with two sets of DMA warps for transferring data. We call this two-buffer technique *double buffering*. Figure 7(b) shows how double buffering works with CudaDMA. The compute threads will always be busy computing on one of the two buffers. Additionally at least one set of DMA warps will be issuing memory operations at all times ensuring better MLP. Double buffering does incur a cost by having many DMA warps. If an application is limited by registers, then double buffering will waste these resources as some DMA warps will be inactive at all points in time.

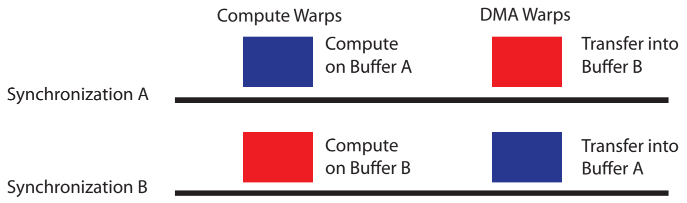
In order to deal with the resource constraints of double buffering and to ensure all DMA warps are active at all



(a) Single-Buffering



(b) Double-Buffering



(c) Manual Double-Buffering

Figure 7: Different buffering techniques using CudaDMA. Large blocks indicate warps are performing useful work, while thin arrows indicate warps move quickly to the next synchronization point. Different colors correspond to different buffers.

times, we introduce a third buffering technique where one set of DMA warps are shared across two buffers and two CudaDMA objects. We call this buffering approach *manual double buffering* since it requires managing the same set of DMA warps with two CudaDMA objects. Manual double buffering is demonstrated in Figure 7(c). In this technique all warps are active at all times ensuring that resources are efficiently utilized. Manual double buffering also gives the programmer the most control over how warps are scheduled on an SM. Manual double buffering is not strictly better than double buffering; Section 6.1 provides an example where double buffering exploits MLP better than manual double buffering.

5. MICRO-BENCHMARKS

To predict the potential benefits of the CudaDMA library on real applications, we evaluated a number of common usage patterns using standalone micro-benchmarks. All experiments were run on an NVIDIA Tesla C2050 GPU with 14 streaming multiprocessors and ECC disabled. All applications were compiled using CUDA 4.0 RC versions of the NVIDIA software toolchain and executed using 270.* versions of the CUDA driver.

The first micro-benchmark evaluates the benefits of staging data from global memory through shared memory using CudaDMA with compute and DMA warp specialization, using the same synthetic micro-benchmark described in Section 2.3 across a range of compute intensities. Results comparing the CudaDMA approach to the baseline approach without warp specialization are shown in Figure 8. The baseline approach uses 16 total warps (512 threads) and transfers 4 bytes per thread on each loop iteration (2 KB total).

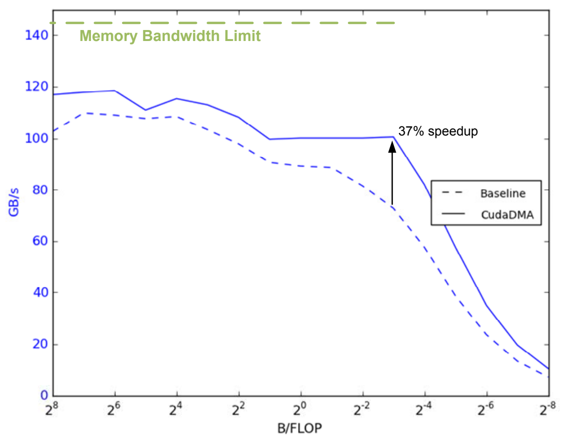


Figure 8: CudaDMA staging through shared with different compute intensities

The CudaDMA approach uses 16 compute warps and 4 DMA warps, staging the same 2 KB of data through shared memory every loop iteration. At very low compute intensity (>1 B/FLOP), execution time is memory-bandwidth limited and CudaDMA shows moderate speedup from using low-overhead producer-consumer synchronization instead of the heavier-weight barriers. At moderate compute intensity similar to the machine ratio, greater speedups are achieved by exposing more MLP via warp specialization. At high compute intensity, the synchronization overhead and requirement for good MLP become lower and the speedups using CudaDMA decrease.

Having investigated how compute intensity influences memory bandwidth, we are also interested in determining the number of DMA warps required to saturate memory bandwidth. Figure 9 shows the impact that the number of DMA warps has on the ability of CudaDMA to saturate memory bandwidth. In this graph, the micro-benchmark is executing a saxpy kernel with a fixed B/FLOP ratio of 6. Rather than vary the compute intensity, we vary the number of active warps on each SM and plot sustained memory bandwidth as a function of active warps per SM (32 warps per SM would correspond to the baseline data plotted in Figure 8). While the baseline approach requires a total of 40 warps to expose enough MLP and reach the achievable peak bandwidth of 120 GB/s, the CudaDMA approach is able to saturate the memory system with just 4 DMA warps per SM, independent of the number of compute warps.

Saturating memory bandwidth at low warp counts is relevant to application performance for several reasons. First, since many applications have smaller dataset sizes, there may not be sufficient parallel threads to fully saturate the memory system. In such cases, CudaDMA could be used to expose more MLP by launching additional DMA threads within each CTA to load data from DRAM and stage it through shared memory, even if the underlying computation had no reuse. Second, for some workloads, CudaDMA has the potential for being more register-efficient than the baseline approach for medium compute-intensity kernels. If all global memory accesses are moved from compute warps to DMA warps, that frees up registers within the compute warps that would have been needed for address calculations

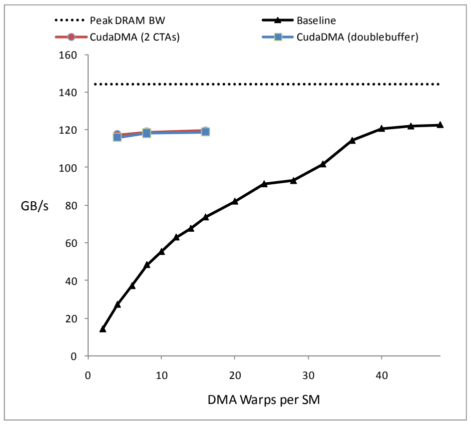


Figure 9: CudaDMA staging through shared with different DMA warp counts

or for load return data. Once register space for these values is no longer needed by the compute warps, the space can instead be used to store intermediate results related to the actual computation.

Sequential DMA patterns are convenient for measuring issues related to MLP and thread coordination, as all their memory accesses are fully coalesced and exhibit ample spatial locality for good memory system performance. Strided DMA patterns introduce further performance effects related to the strided transfer's parameters: the number of elements being transferred, the size of each of those elements, and the stride between successive elements. CudaDMA employs three different implementations of the strided transfer pattern in order to reduce performance variation for different element sizes and counts. We investigate the performance of `CudaDMAStrided` in greater detail in [5], but the performance is comparable to that of `CudaDMASequential` for most elements sizes and counts.

6. APPLICATION KERNELS

In addition to the micro-benchmarks described above, we ported several benchmarks, indicative of common supercomputing applications with moderate FLOP/byte ratios, to CudaDMA in order to better understand the advantages of warp specialization and the features of CudaDMA.

6.1 BLAS2: SGEMV

BLAS is a collection of library calls used by many scientific applications for performing math operations on dense vectors and matrices. BLAS is decomposed into three groups of calls: vector-vector operations in BLAS1, matrix-vector operations in BLAS2, and matrix-matrix operations in BLAS3. BLAS2 calls have moderate compute-to-memory ratios and therefore represent the ideal case for achieving good performance with CudaDMA. We selected single-precision matrix-vector multiplication (SGEMV) as an example from the set of BLAS2 calls.

We implemented SGEMV using several different implementations, each using CudaDMA. In every implementation, we launch CTAs responsible for handling an inner product between a group of rows in the matrix and the vector. We implemented six versions of SGEMV using CudaDMA:

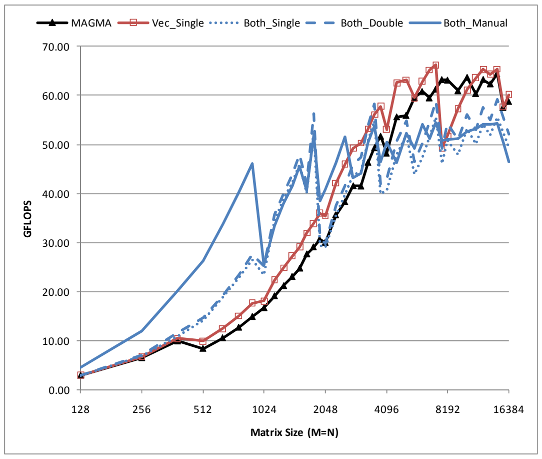


Figure 10: Performance on SGEMV

- `vec-single`: Uses `cudaDMASequential` to move the vector data into shared memory.
- `vec-double`: Double-buffered variant of `vec-single`.
- `vec-manual`: Manually double-buffered variant of `vec-single`.
- `both-single`: Both matrix and vector data are loaded into shared memory. Vector data is still loaded by `cudaDMASequential`, but a `cudaDMAStrided` instance was used for loading the matrix elements. Note that unlike the vector data, staging the matrix data through shared memory is only used for communication between DMA and compute warps, not because there is any reuse of the matrix data by the compute warps.
- `both-double`: Double-buffered variant of `both-single`.
- `both-manual`: Manually double-buffered variant of `both-single`.

Performance results on a sweep of square matrix sizes for four of these SGEMV implementations are shown in Figure 10, compared to a reference implementation in the open-source Magma BLAS library [19]. Similar to the `vec-single` implementation, the Magma implementation of SGEMV also loads vector data into shared memory and directly loads matrix data from global memory into thread registers. However, all threads in a CTA in the Magma implementation are responsible for both loading data and performing math operations whereas CudaDMA uses warp specialization.

For smaller matrices, the `both-*` implementations show speedups of up to 3.2x compared to the reference implementation whereas the `vec-single` implementation shows no speedup. Although staging the matrix data through shared increases the instruction count and synchronization events in the CTA, for smaller sizes the additional MLP exposed by launching CudaDMA threads to load the matrix data significantly improves the sustained memory bandwidth. As matrix size increases, the number of rows processed in parallel on each SM also increases, and performance improves due to more available MLP. For these sizes, the additional MLP exposed by the `both-*` implementations is not beneficial and the launching of additional threads can in fact be counterproductive and lead to slowdowns. However, the `vec-single` case shows moderate improvements of 2%-10% for most matrix sizes due to lower synchronization overheads compared to the Magma reference implementation.

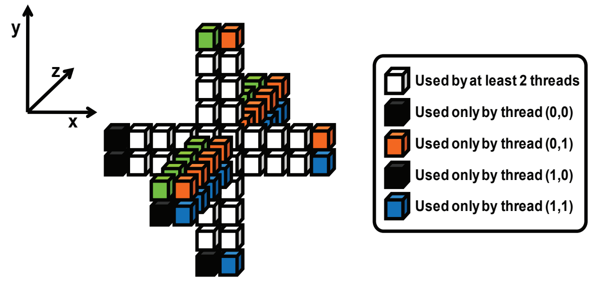

y
z
x
Used by at least 2 threads
Used only by thread (0,0)
Used only by thread (0,1)
Used only by thread (1,0)
Used only by thread (1,1)

Figure 11: Data usage of an 8th order stencil [18].

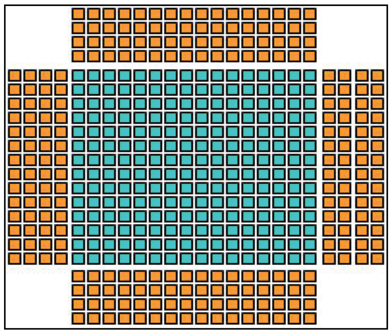


Figure 12: Halo cells for an 8th order stencil [18].

6.2 3D Finite Difference Stencil

Stencil computations are common in many scientific applications that leverage numerical solvers to implicitly compute solutions to differential equations. Stencil computations require reads of many different data values to compute an output value. There is good locality in stencil computations due to the significant overlap between the input data used by adjacent output points. The difficulty in performing stencil computations originates from dealing with the boundary conditions of the stencil. The boundary output points at the edge of the region assigned to a particular CTA require reading additional blocks of data from outside the set of data shared between the CTA's internal output points. We refer to this extra boundary data as *halo* data. Loading the halo data is challenging on a GPU because CTA dimensionality and size is often tied to the dimensionality and size of the output data instead of the input data. Using CudaDMA we were able to decouple the loading of the halo data from the shape of the CTA.

As an example, we implemented the 8th order in space, three-dimensional stencil algorithm described by Micikevicius [18]. The algorithm works by slicing the 3D space in the X and Y dimensions. Each CTA is assigned an X-Y slice and walks through space in the Z direction. Figure 11 shows the data read by each of four threads in a CTA for computing a single output point. The key feature expressed in the read pattern is the large number of cells that are accessed by multiple threads in the same X-Y slice. To facilitate this sharing, the data for the current X-Y slice is placed in shared memory. Each thread then keeps the remaining forward and backwards elements in the Z dimension in on-chip registers.

Since this is an 8th order stencil, a CTA must load the halo data for any thread within 4 elements of the boundary. Figure 12 shows the shape of the halo cells. To load these halo cells for a given slice, our implementation used a custom `cudaDMA` object. The custom instance of `cudaDMA` used two DMA warps for loading the vertical halo regions (one for the top and one for the bottom), as well as a DMA warp for every sixteen rows to load the horizontal halo regions.

Using our custom CudaDMA halo-cell loader we implemented single-buffer, double-buffer, and manual double-buffer versions of the stencil computation, referred to as `halo-only-single`, `halo-only-double`, and `halo-only-manual` respectively. We also implemented a version, called `block-halo-single`, that, in addition to loading the halo cells with CudaDMA using single-buffering, also loaded the leading elements of the primary block cells with a separate `cudaDMA` object using single-buffering.

We compared our kernels against the tuned stencil-only computation in [18] over three different problem sizes. Tables 1, 2, and 3 show the results. Execution time is averaged over 100 runs of the experiment.

Performance gains of 13-15% over the optimized Micikevicius code are achieved by loading both the halo cells as well as the main block data using DMA warps in a single-buffered approach. As shown by the performance of `halo-only-single`, part of this performance gain is due to using DMA warps to get as many halo loads in flight as possible. The rest of the performance is achieved by loading the main block using data transfers of `float4` elements to achieve more MLP and reduce instruction count.

`halo-only-double` performs worse than `halo-only-single` because the additional DMA warps oversubscribed memory resources, causing loads to take longer than in the single-buffer case. Manual double-buffering is able to recoup this performance. In some cases, manual double-buffering exceeds the performance of single-buffering by exploiting MLP in a more controlled manner, instead of relying on the hardware to intelligently schedule the DMA warps.

6.3 1D Fast Fourier Transform

One dimensional fast Fourier transforms are a critical component of many signal processing applications and form the basis for higher dimensional Fourier transforms. Fast Fourier transforms have balanced compute-to-memory ratios that can be adjusted depending on the choice of radix sizes. We implemented CudaDMA versions of the single-precision 128 point and 256 point radix kernels from the CUFFT library [3] that have been highly-optimized for large powers of two.

After analyzing CUFFT's performance we discovered that it was already saturating memory bandwidth and there was little potential for speedup. However, CUFFT required 32 warps per SM to saturate memory bandwidth and was spilling registers to local memory as a result. Rather than attempt to improve memory bandwidth, we used CudaDMA to achieve the same memory bandwidth at lower occupancy which saved registers and prevented spilling. Instead of the 32 warps per SM required by CUFFT, our CudaDMA implementation used only 16 compute warps and 8 DMA warps.

The CudaDMA API enabled us to explore a large slice of the possible performance space. We implemented 48 different CudaDMA object variations that included performing loads by `float2` or `float4`, using main memory or texture cache for loads, loading 4 or 8 or 16 points per DMA thread, and performing different transposes through shared memory all in conjunction with the same compute kernel.

Kernel	Time (ms)	Throughput (Mpoints/s)	Bandwidth (GB/s)	Speedup
Reference	27.83	4746.6	76.85	1.00
`halo-only-single`	26.38	5007.6	81.08	1.055
`halo-only-double`	31.66	4173.8	67.58	0.879
`halo-only-manual`	26.12	5055.4	81.85	1.065
`block-halo-single`	24.16	5467.6	88.53	1.152

Table 1: 3D Stencil: 512x512x512

Kernel	Time (ms)	Throughput (Mpoints/s)	Bandwidth (GB/s)	Speedup
Reference	33.14	4845.0	78.41	1.00
`halo-only-single`	30.97	5185.1	83.92	1.058
`halo-only-double`	37.37	4296.2	69.53	0.887
`halo-only-manual`	31.33	5125.1	82.95	1.058
`block-halo-single`	29.10	5517.7	89.30	1.139

Table 2: 3D Stencil: 640x640x400

Kernel	Time (ms)	Throughput (Mpoints/s)	Bandwidth (GB/s)	Speedup
Reference	25.22	4872.2	79.18	1.000
`halo-only-single`	23.74	5176.5	84.12	1.062
`halo-only-double`	28.71	4280.0	69.55	0.878
`halo-only-manual`	24.20	5078.7	82.53	1.042
`block-halo-single`	22.30	5509.4	89.53	1.131

Table 3: 3D Stencil: 800x800x200

Problem Size	CUFFT (ms)	CudaDMA (ms)	Speedup
524288	0.231	0.227	1.017
1048576	0.457	0.489	0.935
2097152	0.916	0.982	0.933
4194304	1.909	1.939	0.985
8388608	3.894	3.827	1.017
16777216	8.040	7.995	1.006
33554432	18.309	18.154	1.008
67108864	37.191	36.978	1.006

Table 4: 1D FFT (averaged over 1000 runs)

Performance results for inputs that use the two kernels can be seen in table 4. In most cases saving registers was enough to create small speedups, while in a few cases the savings were not enough to overcome the overhead of the additional CudaDMA transfer through shared memory. In all cases CudaDMA achieved the same memory bandwidth at lower occupancy than CUFFT. While this lower-occupancy technique has a small benefit now, we believe it will have greater impact on future architectures as the disparity between computational power and memory bandwidth continues to grow and on-chip memory becomes a scarcer resource.

7. RELATED WORK

Our work on CudaDMA was inspired by previous parallel accelerators, such as the Cell Broadband Engine [14] and the Imagine Stream Processor [20]. These systems addressed the issue of moving data between on-chip and off-chip memories by including asynchronous hardware DMA engines. By allowing the programmer to delegate the responsibility of data movement to the hardware, their programming models enabled the programmer to focus solely on optimizing the computation being performed. In creating CudaDMA, our aim is to provide a similar abstraction for GPUs, albeit implemented in software rather than hardware.

The OpenCL specification defines functions that provide asynchronous copies between memory spaces [2]. These functions are primarily intended to support the hardware DMA engines of the Cell processor, and lack many of the features of CudaDMA. The copies must be performed by all threads in a threadblock, and so provide no opportunity for warp specialization. The function assumes a sequential copy pattern, and lacks the ability to offload pointer arithmetic operations via a constructor. The barrier functionality is coarse-grained, but does allow waiting on multiple named barriers.

Warp specialization has previously been proposed for efficient implementations of sorting algorithms on GPUs [16]. CudaDMA encapsulates this technique in order to make it more generally available to a range of application workloads.

Virtualized warps were proposed by [15] as a way to deal with different tasks at a warp-granularity in CUDA. Unlike CudaDMA, virtual warps still map onto physical warps that execute the same instruction stream.

8. DISCUSSION AND FUTURE WORK

Any GPU application with a compute-to-memory ratio that is close to the underlying hardware is likely to benefit from being ported to CudaDMA. We plan to investigate the performance benefits of CudaDMA on applications with non-uniform memory access patterns such as sparse matrix operations. Since CudaDMA consists primarily of a header file and can run on current compiler technology it should be easy to incorporate it into many existing applications.

To aid in the adoption of CudaDMA we plan to expand the base set of CudaDMA instances to include additional transfer patterns such as sparse patterns and transposes. We also plan to support the use of CudaDMA without warp specialization. Although this may not lead to performance improvements, it could help with programmability and code maintainability by abstracting the complexity of data access patterns behind an API.

Another domain in which we expect CudaDMA to make a valuable contribution is for frameworks that automatically compile to GPU hardware by generating CUDA or PTX code. Such frameworks leverage high-level [12] abstractions in languages such as Python [11] and Scala [13] to express parallelism and remove the burden of targeting specific machines from the programmer. If these frameworks choose to incorporate CudaDMA into their backends, they can reap the benefits of its performance gains for managing data transfers on a GPU. Programming frameworks that enable the programmer to program directly to the memory hierarchy, such as Sequoia [10], will likewise be able to exploit CudaDMA by using it as part of its generic runtime target for performing transfers on GPUs.

While the CudaDMA interface is currently supported by existing compilers, one area of future research is in the area of programming models and compilers that are warp-specialized-aware. These programming models and compilers would explicitly enable programmers to create specialized warps and ensure that specialized warps were co-scheduled and could share resources. Warp-specialization-aware compilers could then compile code for different warps independently, optimizing the usage of limited resources such as registers more effectively than current compiler technology.

CudaDMA has shown the benefits of emulating an asynchronous DMA engine in software on a GPU. Another area of future research is to examine possible performance gains that could be achieved by incorporating an actual hardware DMA engine onto a GPU.

9. CONCLUSION

In this paper we presented CudaDMA, an extensible API for efficiently managing data transfers between the on-chip and off-chip memories of GPUs. CudaDMA enables the programmer to decouple the shape of data from how the data is transferred by creating specialized DMA warps for memory transfers. CudaDMA performs best on applications with balanced compute-to-memory ratios by allowing the programmer to optimize the specialized DMA warps and compute warps independently. We show speedups between 1.15x-3.2x on several GPU kernels from common scientific applications.

Acknowledgments

This research was funded in part by DARPA contract HR0011-10-9-0008, Microsoft (Award #024263), Intel (Award #024894), and U.C. Discovery (Award #DIG07-10227). Additional support comes from UC Berkeley ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. Michael Bauer is supported under the DARPA UHPC Program and by an NVIDIA Research Fellowship. Henry Cook is supported by an NVIDIA Research Fellowship. The authors also wish to thank Sean Treichler and Paulius Micikevicius for their insights.

10. REFERENCES

[1] Cuda toolkit 4.0 thrust quick start guide. `http://developer.download.nvidia.com/compute/cuda/4\_0\_rc2/toolkit/docs/Thrust\_Quick\_Start\_Guide.pdf`, January.

[2] The opencl specification, version 1.0. `http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf`, April 2009.

[3] Cuda cufft library. `http://developer.download.nvidia.com/compute/cuda/4\_0\_rc2/toolkit/docs/CUFFT\_Library.pdf`, February 2011.

[4] Cuda toolkit 4.0 cublas library. `http://developer.download.nvidia.com/compute/cuda/4\_0\_rc2/toolkit/docs/CUBLAS\_Library.pdf`, February 2011.

[5] Cudadma repository. `http://code.google.com/p/cudadma/`, April 2011.

[6] Green 500 supercomputers. `http://www.green500.org/`, June 2011.

[7] Ptx isa. `http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/ptx_isa_2.3.pdf`, February 2011.

[8] Top 500 supercomputers. `http://www.top500.org/`, June 2011.

[9] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 2–13, May 1995.

[10] M. Bauer, J. Clark, E. Schkufza, and A. Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia. In *Principles and Practices of Parallel Programming*, PPoPP'11, February 2011.

[11] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Principles and Practices of Parallel Programming*, PPoPP'11, pages 47–56, February 2011.

[12] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective embedded jit specialization. In *Programming Models for Emerging Architectures*, 2009.

[13] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Principles and Practices of Parallel Programming*, PPoPP'11, February 2011.

[14] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation; a performance view. *IBM Journal of Research and Development*, 51(5):559 –572, sept. 2007.

[15] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[16] D. Merrill and A. Grimshaw. Revisiting sorting for gpgpu stream architectures. Technical Report Technical Report CS2010-03, University of Virginia, February 2010.

[17] D. G. Merrill and A. S. Grimshaw. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 545–546, New York, NY, USA, 2010. ACM.

[18] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[19] R. Nath, S. Tomov, and J. Dongarra. Accelerating gpu kernels for dense linear algebra. In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, pages 83–92, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. LÃşpez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *In 31st International Symposium on Microarchitecture*, pages 3–13, 1998.

[21] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 49–84. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45937-5_14.

[22] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference*, GTC '10, 2010.

[23] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, April 2009.