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 micro-benchmarks, 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 November 2010 Top500 list, GPUs were included in 3 of the top ten installations [4], and on the Green500 list from the same month, GPUs were included in 5 of the top ten machines [3]. Accelerators designed to exploit fine-grain data parallelism are on track to be the next big advance in supercomputing performance.

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 [5, 7] on GPUs have already been introduced. However, for programmers developing new algorithms, such libraries are only small components to 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 both carry out the computation and to transfer the data. 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 [13] and the Imagine Stream Processor [19] 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 and how it is transferred 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 synchronization primitives necessary for coordinating between compute warps and DMA warps.

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 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 divergence across warps 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.

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 [20]. 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 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-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), through-
Many long-latency memory accesses: If enough long-latency memory accesses are issued so as to fill up the memory system’s buffers, then the 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 sufficiently 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 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 enforced 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 cease 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 penalty for changing subsets of the thread block’s functionality. We call this differentiation technique warp specialization. Warp specialization has been employed previously for efficient parallel implementations of sorting algorithms on GPUs [16]. 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 which code will be executed by DMA warps.

The calls in the API can be divided into two major categories: cudaDMA object instantiations and calls for transfers and synchronization. cudaDMA objects describe the memory access pattern that a particular subset of threads will employ to transfer data. The user can launch and synchronize
compute a number of per-thread variables that will be used are not meant to be fast, but rather are designed to pre-loop at the top of the kernel. The constructors themselves of daDMA simply requires employing two predefined instances sequential, whereas the 2-D matrix data are strided across different layouts in memory. The 1-D vector data are all sequent in Section 6.1.

The subsets of matrix data and subsets of vector data have into shared memory as well as the vector for performance blocks. In this example, we also load subsets of the matrix cient large that they must be loaded iteratively in small into shared memory. However, the vectors are often suffi cient of the vector, making it beneficial to load the vector and the vector. To do so, every thread must access every el-

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 3 presents code implementing a single-precision matrix-vector multiplication (SGEMV) routine from the BLAS dense linear algebra library.

Figure 3: Interface for CudaDMA objects.

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 subsets of vector data have different layouts in memory. The 1-D vector data are all sequen, 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 outside of the iterative loop at the top of the kernel. The constructors themselves are not meant to be fast, but rather are designed to pre-compute a number of per-thread variables that will be used repeatedly in the API calls within inner loops. 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.

For the DMA warps, whether or not a particular warp should execute a particular transfer is determined by the boolean method owns_this_thread() of a particular transfer object. 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, 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

```cpp
// CudaDMA class definition

class cudaDMA {
  __device__ cudaDMA {
    const int dma0, const int num_dma_threads, const int num_compute_threads, const int threadIdx_start;
  }
  __device__ void execute_dma(void* src_ptr, void* dst_ptr);
  __device__ bool owns_this_thread();
  __device__ void wait_for_dma_start();
  __device__ void finish_async_dma();
  __device__ void wait_for_dma_finish();

  __device__ void wait_for_dma_start();
  __device__ void finish_async_dma();
  __device__ void wait_for_dma_finish();

  // Interface for CudaDMA objects.
  __global__ void sgemv_cuda_dma(int n, int m, float alpha, float *A, float *x, float *y) {
    __shared__ float buf[VEC_ELMTS];
    __shared__ float mat [VEC_ELMTS][COMPUTE_THREADS];

    cudaDMASequential<sizeof(float)*VEC_ELMTS/ DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ>

    cudaDMASequential(sizeof(float)*VEC_ELMTS, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ, DMA_THREADS_SEQ)

    if (threadIdx.x < COMPUTE_THREADS) {
      dma_ld_0.start_async_dma();
      float res = 0.f;
      for(int i=0; i<n; i++) { dma_ld_0.wait_for_dma_finish();
        for(int j=0; j<VEC_ELMTS; j++) {
          res += mat[i][threadIdx.x]*buf[j];
        }
      }
      dma_ld_0.finish_async_dma();
      if (ind < n) y[ind] = alpha * res;
      else if (dma_ld_0.owns_this_thread()) {
        dma_ld_0.wait_for_dma_start();
        for(int idx=0; idx<n; idx++) { dma_ld_0.wait_for_dma_finish();
          for(int j=0; j<VEC_ELMTS; j++) {
            res += mat[i][threadIdx.x]*buf[j];
          }
        }
      }
    }
    else if (dma_ld_1.owns_this_thread()) {
      dma_ld_1.wait_for_dma_start();
      for(int idx=0; idx<n; idx++) { dma_ld_1.wait_for_dma_finish();
        for(int j=0; j<VEC_ELMTS; j++) {
          res += mat[i][threadIdx.x]*buf[j];
        }
      }
    }

    for (int j =0; j < VEC_ELMTS ; j ++) {
      dma_ld_1.wait_for_dma_start();
    }
    dma_ld_1.wait_for_dma_finish();

    for (int j =0; j < VEC_ELMTS ; j ++) {
      dma_ld_1.wait_for_dma_start();
    }
    dma_ld_1.wait_for_dma_finish();

    dma_ld_1.wait_for_dma_start();
    dma_ld_1.wait_for_dma_finish();

    if (threadIdx.x < COMPUTE_THREADS) {
      dma_ld_0.start_async_dma();
      float res = 0.f;
      for(int i=0; i<n; i++) { dma_ld_0.wait_for_dma_finish();
        for(int j=0; j<VEC_ELMTS; j++) {
          res += mat[i][threadIdx.x]*buf[j];
        }
      }
      dma_ld_0.finish_async_dma();
      if (ind < n) y[ind] = alpha * res;
      else if (dma_ld_0.owns_this_thread()) {
        dma_ld_0.wait_for_dma_start();
        for(int idx=0; idx<n; idx++) { dma_ld_0.wait_for_dma_finish();
          for(int j=0; j<VEC_ELMTS; j++) {
            res += mat[i][threadIdx.x]*buf[j];
          }
        }
      }
    }
    else if (dma_ld_1.owns_this_thread()) {
      dma_ld_1.wait_for_dma_start();
      for(int idx=0; idx<n; idx++) { dma_ld_1.wait_for_dma_finish();
        for(int j=0; j<VEC_ELMTS; j++) {
          res += mat[i][threadIdx.x]*buf[j];
        }
      }
    }
    // SGEMV routine from BLAS.

    Figure 4: A CudaDMA-based implementation of the SGEMV routine from BLAS.
```
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 the cudaDMA API that represent common transfer patterns. In this section we describe the specifics of these instances of cudaDMA as well as how programmers can create their own specialized instances of cudaDMA.

In the example in Figure 4 the sequential access pattern required by the vector data and the strides access pattern required by the matrix data are performed by the cudaDMA subclasses cudaDMASequential and cudaDMAStrided. Figure 5 presents the declaration of the subclass constructors that are used in our SGE MV 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. By making this a template parameter, the compiler is able to perform constant folding and reduce runtime computation. 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.

Note that while these parameterized transfer classes require the programmer to specify how many DMA warps will be used to execute the transfer and how much work each DMA warp will have to do, they do not require the programmer to express which thread issues loads or stores 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 __device__ cudaDMA and are shown here for clarity. The implementation of these four calls is described in Section 4.1. The producer/consumer nature of our synchronization mechanisms allow the programmer to employ a variety of techniques to overlap communication and computation described in section 4.2.

4. CUDADMA METHODOLOGY

As we saw in Section 3, CudaDMA takes advantage of several strategies to implement code with balanced compute-to-memory ratios as efficiently as possible. In this section we investigate the techniques that we use as part of the CudaDMA API to achieve good performance.

4.1 Warp Specialization

In an ideal GPU with an infinite-sized instruction window, an infinite-sized register set for load return data, and support for an infinite number of non-blocking loads, there would be no need for warp specialization. In practice these resources are constrained; warp specialization relieves the pressure on these various resources. Warp specialization allows each subset of threads within a CTA to have their behavior tuned for a particular purpose and to consume specific resources. DMA warps focus on maximizing achieved MLP and using memory system resources, while the compute warps focus on maximizing ILP and usage of compute resources.

To achieve maximum MLP, an instance of a CudaDMA class will ensure that DMA threads use CUDA vector data types for loads and stores (e.g. float4), which consume fewer instruction issue slots to achieve the same memory bandwidth. The memory access patterns for DMA warps will also be engineered to be coalesced and avoid bank conflicts when writing to shared memory. DMA warps generally use a smaller number of registers, since they only need enough for the return data of outstanding loads and for performing pointer arithmetic.

Warp specialization also aids compute warps by reducing the number of registers maintained by their constituent threads for global address calculations. In addition it removes the requirement for the compute threads to contain specific addresses. This abstraction is essential for increasing programmer 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 extensible, we provide a cudaDMAExtension class which contains the synchronization functions but leaves the behavior of the DMA warps up to the programmer.
registers reserved for load return data. Since there are accesses to global memory, there will never be stalls in instruction issue due to pushback from the memory system.

Whenever small pieces of a larger data structure are being staged through shared memory, there exists an opportunity to overlap computation and communication. Warp specialization is a natural way to encode this overlapping. The compute warps and the DMA warps must be able to coordinate with one another efficiently for this overlapping strategy to be effective.

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. Rather than solely relying on coarse-grain CTA-wide barrier wide synchronization, for CudaDMA, more parallelism can be exposed to the underlying hardware if we can leverage the advantages of fine-grained named producer-consumer synchronization events between threads in a CTA [8].

We accomplish this finer-granularity synchronization by using inlined PTX assembly to express named barriers. There are two such barriers associated with every cudadma object. Two barriers are required to track whether the data buffer in shared memory is full or empty. We also use the PTX instruction bar.arrive, which allows a thread to signal its arrival at a particular barrier without blocking its execution [6]. 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 create blocking barriers. 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 4.2 demonstrates the power of the producer/consumer abstraction by illustrating several different ways to use named barriers for buffering.

4.2 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.

Depending on the compute-to-memory ratios of the application, single buffering may not be enough to saturate the full computational resources or the memory system of a GPU. 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 a valid performance technique especially if the programmer ensures that multiple CTAs are resident on an SM. This 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 loading 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 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 two CudaDMA objects for the same set of DMA warps. 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.

4.3 Access Patterns

As discussed in Section 3.2, there are a few common data transfer patterns that occur frequently enough that we have implemented optimized versions of cudadma objects to provide as a productivity basis for CUDA application programmers. These transfer patterns are characterized by their memory access pattern and a few specific parameters that serve to define that pattern. CudaDMA implements these patterns extensibly via C++ templated classes.

Providing this class-based abstraction encourages the programmer to explicitly state a transfer pattern at compile time. Doing so is advantageous because it confers information that can be used to precompute pointer offsets that are then inserted into the code as immediate operands, rather than values that occupy register space. Our library can also use template meta-programming techniques to determine when to use vector data types to implement a transfer.

The access pattern abstractions are also essential to decoupling the functionality of the transfer from the number of threads being used to implement the transfer. By allowing the programmer to state the nature and parameters of the access pattern explicitly and separately from the number of threads assigned to the transfer, our library abstracts away the messy logic involved in determining how to perform the transfer. Optimized version of cudadma objects for common transfer patterns can be written by expert programmers and then re-used by application programmers providing good performance and high productivity.
Having investigated how compute intensity influences memory bandwidth, we were 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, sustained memory bandwidth is plotted as a function of active warps on each SM (32 warps per SM would correspond to the baseline data plotted in Figure 8). Whereas 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 effectively 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 occupy a GPU. In such cases, CudaDMA could be used to expose more memory-level parallelism 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 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 additional performance effects related to address patterns, which are based on the two dimensions of the strided transfer: the number of elements
being transferred and the size of each of those elements. Figure 10 illustrates how our current implementation of the strided transfer pattern performs across a variety of possible transfer dimensions. While the current implementation favors large elements, future optimizations will improve performance for small element cases.

6. APPLICATION KERNELS
In addition to the micro-benchmarks described above, we ported several kernel 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:

- **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 custom CudaDMA instance was used for loading the matrix elements. The custom instance of CudaDMA was designed to simultaneously issue multiple global loads for matrix data to exploit as much MLP as possible. 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 11, compared to a reference implementation in the open-source Magma BLAS library [18]. 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 the CudaDMA implementations use 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. For smaller sizes, although staging the matrix data through shared increases the instruction count and synchronization events in the CTA, 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.

### 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 in GPU computing 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 are able to decouple the loading of the halo data from the properties of our CTA.

As an example, we implement the 8th order in space, three-dimensional stencil algorithm described by Micikevicius [17]. 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 12 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 its private 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 13 shows the shape of the halo cells. To load these halo cells, our implementation uses an instance of a custom CudaDMA subclass responsible for loading the halo data for a given slice. The custom instance of cudaDMA uses 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 implement 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 implement 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 compare our kernels against the tuned stencil-only computation in [17] 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.

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.

In addition to the stencil-only code, we also implement the finite difference wave equation code from [17]. This code employs the same read pattern as the stencil-only code, but also requires two additional loads from global memory. Due to the extra memory operations, this algorithm has a
Table 1: 3D Stencil: 512x512x512

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (ms)</th>
<th>Throughput (Mpoints/s)</th>
<th>Bandwidth (GB/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>27.83</td>
<td>3746.6</td>
<td>76.88</td>
<td>1.00</td>
</tr>
<tr>
<td>halo-only-single</td>
<td>26.38</td>
<td>5007.6</td>
<td>81.08</td>
<td>1.055</td>
</tr>
<tr>
<td>halo-only-double</td>
<td>31.66</td>
<td>4173.8</td>
<td>67.58</td>
<td>0.879</td>
</tr>
<tr>
<td>halo-only-manual</td>
<td>26.12</td>
<td>5004.3</td>
<td>81.85</td>
<td>1.080</td>
</tr>
<tr>
<td>block-halo-single</td>
<td>24.16</td>
<td>5466.7</td>
<td>88.53</td>
<td>1.152</td>
</tr>
</tbody>
</table>

Table 2: 3D Stencil: 640x640x400

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (ms)</th>
<th>Throughput (Mpoints/s)</th>
<th>Bandwidth (GB/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>25.22</td>
<td>4872.2</td>
<td>79.18</td>
<td>1.000</td>
</tr>
<tr>
<td>halo-only-single</td>
<td>23.74</td>
<td>5116.5</td>
<td>84.12</td>
<td>1.062</td>
</tr>
<tr>
<td>halo-only-double</td>
<td>26.71</td>
<td>4296.2</td>
<td>69.53</td>
<td>0.587</td>
</tr>
<tr>
<td>halo-only-manual</td>
<td>24.20</td>
<td>5098.7</td>
<td>82.53</td>
<td>1.042</td>
</tr>
<tr>
<td>block-halo-single</td>
<td>22.30</td>
<td>5599.4</td>
<td>89.53</td>
<td>1.131</td>
</tr>
</tbody>
</table>

Table 3: 3D Stencil: 800x800x200

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (ms)</th>
<th>Throughput (Mpoints/s)</th>
<th>Bandwidth (GB/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>25.22</td>
<td>4872.2</td>
<td>79.18</td>
<td>1.000</td>
</tr>
<tr>
<td>halo-only-single</td>
<td>23.74</td>
<td>5116.5</td>
<td>84.12</td>
<td>1.062</td>
</tr>
<tr>
<td>halo-only-double</td>
<td>26.71</td>
<td>4296.2</td>
<td>69.53</td>
<td>0.587</td>
</tr>
<tr>
<td>halo-only-manual</td>
<td>24.20</td>
<td>5098.7</td>
<td>82.53</td>
<td>1.042</td>
</tr>
<tr>
<td>block-halo-single</td>
<td>22.30</td>
<td>5599.4</td>
<td>89.53</td>
<td>1.131</td>
</tr>
</tbody>
</table>

much lower compute-to-memory ratio and CudaDMA performs on par with the reference implementation, but is unable to achieve any speedups.

7. RELATED WORK

Our work on CudaDMA was inspired by previous parallel accelerators, such as the Cell Broadband Engine [13] and the Imagine Stream Processor [19]. 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 [15]. CudaDMA encapsulates this technique in order to make it more generally available to a range of application workloads.

Virtualized warps were proposed by [14] 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 whose compute-to-memory ratio is close to that of the underlying hardware is likely to benefit from being ported to CudaDMA. We plan to investigate promising application areas such as FFT algorithms and image filtering algorithms. The performance benefits of CudaDMA on applications with non-uniform memory access patterns such as sparse matrix operations is of particular interest. 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. Finally, we will explore using advanced GPU programming techniques such as leveraging texture caches in conjunction with DMA warps to improve performance.

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 [11] abstractions in languages such as Python [10] and Scala [12] 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 [9], 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 compiler technology, one area of future research is in the area of programming models and compilers that are warp-specialization-aware. These programming models and compilers would explicitly enable programmers to create specialized warps and ensure that specialized warps were co-scheduled on the hardware 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 memory transfers. CudaDMA performs best on applications with balanced compute-to-memory ratios by allowing the programmer to delegate the responsibility of data transfers on a GPU. Programming frameworks that enable the programmer to program directly to the memory hierarchy, such as Sequoia [9], will likewise be able to exploit CudaDMA by using it as part of its generic runtime target for performing transfers on GPUs.

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 [11] abstractions in languages such as Python [10] and Scala [12] 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 [9], 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 compiler technology, one area of future research is in the area of programming models and compilers that are warp-specialization-aware. These programming models and compilers would explicitly enable programmers to create specialized warps and ensure that specialized warps were co-scheduled on the hardware 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.
10. REFERENCES


