D^{2}MA: Accelerating Coarse-Grained Data Transfer for GPUs

D. Anoushe Jamshidi, Mehrzad Samadi, and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{ajamshid, mehrzads, mahlke}@umich.edu

ABSTRACT

To achieve high performance on many-core architectures like GPUs, it is crucial to efficiently utilize the available memory bandwidth. Currently, it is common to use fast, on-chip scratchpad memories, like the shared memory available on GPUs' shader cores, to buffer data for computation. This buffering, however, has some sources of inefficiency that hinder it from most efficiently utilizing the available memory resources. These issues stem from shader resources being used for repeated, regular address calculations, a need to shuffle data multiple times between a physically unified on-chip memory, and forcing all threads to synchronize to ensure RAW consistency based on the speed of the slowest threads. To address these inefficiencies, we propose Data-Parallel DMA, or D^{2}MA. D^{2}MA is a reimagination of traditional DMA that addresses the challenges of extending DMA to thousands of concurrently executing threads. D^{2}MA decouples address generation from the shader's computational resources, provides a more direct and efficient path for data in global memory to travel into the shared memory, and introduces a novel dynamic synchronization scheme that is transparent to the programmer. These advancements allow D^{2}MA to achieve speedups as high as 2.29x, and reduces the average time to buffer data by 81% on average.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—Single-instruction-stream, multiple-data-stream processors (SIMD)

Keywords

GPUs; DMA; Software-managed Caches; Shared Memory; Dynamic Management; Throughput Processing

1. INTRODUCTION

In modern computing systems, *graphics processing units (GPUs)* have become ubiquitous. While GPUs traditionally have been used to efficiently render graphics, over the last decade major GPU vendors have unlocked the potential of their high-throughput, many-core products to be used for more general purpose, albeit massively data-parallel computations. Systems ranging from smartphones [25] to supercomputers [26] can now use GPUs to accelerate applications in domains such as multimedia processing, machine learning, financial modeling, linear algebra, and scientific computing. Using programming models like CUDA [22] or OpenCL [17], software developers can leverage the computational efficiency of many-core GPUs with their data-parallel code across a broad spectrum of devices.

Many-core architectures like GPUs typically achieve high performance when provided thousands of threads to concurrently execute. Problems arise, however, when trying to deliver enough data to these threads for them to compute upon. There is a long latency when loading data from a GPU's off-chip memory, possibly greater than 400 cycles on NVIDIA architectures [22]. Ideally, when threads stall due to such long latency operations, there will be many other threads that are schedulable for execution. While this may be the case for applications that perform a great deal of computation compared to memory transfers, in practice there are many memory-intensive applications that have low compute-to-memory characteristics. In such scenarios, most threads may be requesting data from memory simultaneously, creating high contention for memory resources. These applications will not have enough threads ready for execution, causing their performance to suffer, and simply adding more threads will not necessarily improve this performance.

Simply expanding and improving the existing memory subsystem in GPUs may not necessarily improve memory-intensive application performance either. According to GPU device specifications, there should be ample bandwidth to quickly move data between off-chip memory and the core. After all, most medium- to high-end GPUs are provisioned with 100-300+ GB/s of memory bandwidth to their off-chip global memories [24]. However, when memory-intensive applications attempt to fully utilize the GPU's cores, they can easily saturate the memory resources between the cores and off-chip memory, such as buffers between core load/store units and the memory interconnect or miss status holding registers (MSHRs) in caches. Simply providing more bandwidth or expanding existing memory resources are both too costly and will not necessarily ameliorate this problem [16, 28]. To demonstrate this, a model of the NVIDIA GTX480 was augmented to have infinite memory resources (that is, MSHRs and all buffers between shaders and DRAM), and 28 benchmarks from the NVIDIA SDK were simulated us-

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

http://dx.doi.org/10.1145/2628071.2628072 .

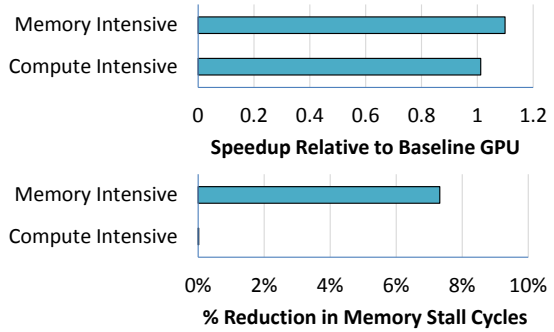


Figure 1: A balanced increase in the amount of memory resources does not guarantee improved performance and bandwidth utilization.

ing this model. Figure 1 shows that the mean speedups and memory stall cycle reductions that this theoretical device achieves for memory intensive applications are limited to roughly 10%, while computationally intensive benchmarks see marginal gains from such costly hardware over-provisioning. Even with infinite memory resources, neither compute- nor memory-bound applications see great improvements in their performance as the number of cycles they stall for memory transfers are only slightly impacted. Thus, even with a vast array of computational and memory hardware provisions, a GPU's performance can still suffer as it struggles to efficiently fetch data from off-chip memory for processing. Other approaches should be considered to better utilize the existing memory resources.

Numerous techniques have been introduced that help improve bandwidth utilization. One commonly used technique is to buffer tiles or blocks of data from global memory in a fast, on-chip shared memory [22]. Once the tile is in shared memory, computation can continue quickly and many delays associated with the GPU's memory subsystem can be greatly reduced. While this paradigm can easily be deployed in software and works with modern GPUs, there are software and microarchitectural inefficiencies that may hamper the performance of applications using this scheme. These include overheads from generating addresses for each load and store instruction, redundancy of data as buffered data will occupy space in both the L1 cache and the shared memory when both memories are in fact unified, coarse-grained synchronization penalties in order to guarantee read-after-write data consistency, and unnecessary register file usage to move data from the global to the shared memory.

Other software techniques like CudaDMA [4] and Sponge [11] attempt to improve this idea by using some GPU computation resources to help manage the buffering. They achieve this by specializing threads to handle transfer of data from global to shared memory, but by doing so they consume computational hardware to generate addresses and perform these transfers and limit the number of threads able to concurrently perform computation. Furthermore, these works do little to address the inefficiencies of the current buffering paradigm on modern GPUs.

Other works have introduced novel GPU prefetching hardware [19, 31] to help applications that do not have enough threads to hide memory latency. Prefetching schemes however are speculative and can potentially degrade bandwidth utilization due to inaccurate data prefetches. An alternative to prefetching is to decouple some memory accesses from a processor's datapath with some additional hardware support. Direct memory access (DMA) controllers are one such example of this. Some single- and multi-core architectures, like the IBM Cell BE [8], use direct memory access (DMA) to move data between peripherals and memories asynchronously with respect to the CPU. By doing so, the CPU avoids consuming execution resources to calculate memory addresses and can reduce stalling when waiting for data by overlapping computation with transfer. However, as DMA was not envisioned for architectures that execute on thousands of threads simultaneously, difficult challenges prevent the direct adoption of DMA for GPUs. Such challenges include selecting which threads should begin a DMA transaction, how to concurrently interleave transactions for many threads, and how to inform threads that their transfer is complete on a non-interruptible architecture. This work addresses these challenges to bring the benefits of decoupled memory accesses provided by traditional DMA-enabled devices to many-core, heavily multi-threaded GPUs.

To improve bandwidth utilization as well as application performance, we propose a decoupled memory access scheme for GPUs called Data-Parallel DMA, or D^{2}MA. D^{2}MA provides an efficient path for tiles of data to travel from off-chip global memory to fast access, on-chip shared memory, while decoupling bulk address generation for these transfers from the rest of the core's pipeline. To accomplish this, a new DMA engine is added to each shader core that is capable of rapidly generating coalesced memory requests given a single base address and the size of a buffer for each resident concurrent thread array (CTA)[1]. The engine is also capable of generating special addressing patterns that are frequently found in data-parallel code, including tiles with a surrounding halo and blank column insertion to avoid bank conflicts in shared memory. When buffer transfers complete, threads waiting for their data tile need to be signaled to continue execution. We introduce a novel waiting scheme that is managed in hardware and maintains read-after-write consistency while being entirely transparent to the programmer. Finally, to minimize the burden of using D^{2}MA in GPU code, a few simple API calls are introduced that allow the programmer to provide the DMA engine with the necessary information to buffer a tile of data in the shared memory.

Our results show that by decoupling coarse-grained memory buffering from the rest of the shader's pipeline, D^{2}MA is able to drastically reduce the number of cycles where all resident CTAs on a shader are stalled waiting for a memory transfer to less than 1%. On average, the duration of a transfer of a tile of data from global to shared memory was reduced by 81%. The geometric mean speedup accomplished by reducing these stalls and transfer times was around 36%.

In summary, the main contributions of this work are:

- A simple new functional unit, the DMA engine, that can rapidly generate coalesced memory requests to global memory in bulk without using the shader's computational execution units.
- Support for specialized address generation for data with halos and for shared memory bank conflict avoidance.

[1]In this paper, we will frequently use NVIDIA/CUDA terminology to describe GPU hardware and programming. A CUDA thread block is a collection of threads that maps to a GPU CTA [27].

- An efficient path for responses from global memory to flow to the shared memory that reduces data redundancy and register usage.
- A novel hardware scheme that enforces read-after-write consistency, is transparent to the programmer, and obviates the need for coarse-grained synchronization.

2. BACKGROUND AND MOTIVATION

This section describes the ways that GPUs hide the memory access latencies and the inefficiencies of buffering data into shared memory, and discusses the challenges of addressing them when using DMA, a tried-and-true solution to such issues for CPUs, on multi-threaded GPUs.

2.1 Hiding GPU Memory Access Latency

One of the most important characteristics of data-parallel programs that target GPUs is that they expose enough thread level parallelism to hide the long latencies of global memory accesses and some arithmetic operations. When the shader core's execution units are waiting for a long latency operation to complete, the shader's warp[2] scheduler will attempt to find another warp that is ready for execution to run in the meantime. When running applications with enough threads and high compute-to-memory ratios, swapping warps may be able to hide most long latency operations, but when an application does not have enough threads or is too memory intensive the scheduler may not have enough ready warps to issue, thus stalling the shader core.

To prevent poor performance for such memory intensive applications, many applications take advantage of low-latency on-chip shared memory by buffering tiles of data from off-chip global memory into it. Each shader core has a limited amount of shared memory available – NVIDIA architectures provide a maximum of 48KB per core, compared to gigabytes of off-chip memory. The scope of data sharing is limited to the threads within a CTA, and the amount of space allocated to a CTA is limited by the number of CTAs that are simultaneously issued to a shader.[3] The programmer must carefully account for this when utilizing shared memory, as the amount of data buffered into shared memory will directly impact how many CTAs can be launched concurrently per shader core. While this may seem restrictive, the shared memory is located so close to a shader's functional units that frequent random accesses to the tile are feasible provided that care is taken to avoid bank conflicts. The same random accesses of the data in global memory would incur coalescing penalties and long round trip latencies to transfer each segment of data [22]. When shared memory buffering is applied, it can greatly improve performance by reducing the cycles during which the shader stalls for global memory accesses.

2.2 Inefficiencies when Buffering into Shared Memory

While buffering data using shared memory can positively impact application performance and off-chip bandwidth utilization, there are inefficiencies in this process for modern GPU hardware [4]. Figure 2 details the many steps required to perform each transfer of data from global into shared memory. Before any transfer can begin, the addresses of the source data and the destination must be calculated. These instructions are fetched, decoded and issued to ALUs for computation (❶), and the results of execution are written back to the register file (❷). While the ALUs are calculating these addresses, the instructions that load from global memory are fetched and decoded and are waiting at issue for the scoreboard to indicate that the addresses are ready as shown by ❸. Once the scoreboard permits their issue, the loads use the memory unit to generate coalesced memory requests. These requests first probe the L1 data cache, where compulsory misses are very likely as each buffer transfer is fetching a new and unique tile of data from global memory. After missing in the L1 cache, the request is sent to the L2 cache and global memory through the interconnect (❹). Meanwhile, the instructions that will store the fetched data into shared memory have progressed to the issue stage, awaiting the results of the loads (❺). After a few hundred shader cycles, the memory responses for these loads will return to the shader. When this happens (❻), the data updates the shader's L1 data cache (as shown by the cross-hatched box) and are written back to the register file. Finally, ❼ shows that the store instructions are issued to the memory unit, and the results of the global loads in the register file are stored into the shared memory. This process repeats until the entire tile is transferred, that is, until all warps involved have completed their stores to shared memory.

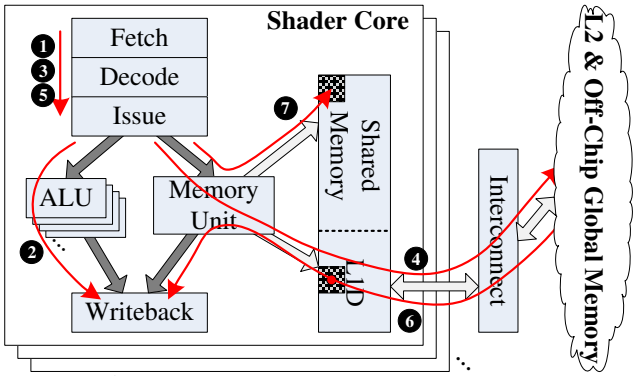


Figure 2: An example illustrating inefficiencies while buffering data into the shared memory.

Close examination of the example in Figure 2 reveals a few key sources of inefficiency during the transfer of a tile from global to shared memory. The first occurs at the ISA level. Each transfer is composed of numerous instructions to generate addresses, launch loads from global memory, and store results to shared memory. The hardware must issue a batch of these instructions for each cache block (128B) of data to be transferred, and since applications buffering into shared memory are likely to be memory-intensive, many instruction batches must be issued. However, the shape of the data being transferred is typically very regular, e.g. data packed in a 2-D matrix. With the existing ISA, there is no way to exploit the regularity of this data without executing all these instructions. The second inefficiency compounds the first with structural hardware limitations – to calculate the base addresses for the loads and stores, the shader's ALUs are consumed to perform additions and multiplications and cannot be used by other warps to perform computations.

[2]In modern NVIDIA GPUs, a warp is a collection of 32 threads that execute the same instructions in lockstep on a shader core [22].

[3]The NVIDIA Fermi architecture allows eight CTAs to be simultaneously resident on a shader core, while Kepler permits sixteen [23].

A third major inefficiency is caused by the microarchitectural layout of the shared memory and L1 data cache. In the NVIDIA Fermi architecture[4], these two memories are actually unified and coexist in the same memory [10, 21]. As shown by ❻, when the response from global memory returns to the shader the L1 is updated, and later when the store to shared memory occurs (❼) the data is again written to a different partition of the same, unified memory. This produces redundant data existing in this unified shader memory, and can pollute the L1 data cache as the recently transferred tile of data is unlikely to be reused by other global memory operations or future tile transfers. Furthermore, because there is no direct path for the load's memory response to go to shared memory, it's journey is lengthened by writing the fetched data to the register file so that a future store instruction can write this data into the shared memory.

One thing that is not pictured in Figure 2 is the synchronization required to prevent read-after-write consistency errors. As each warp is responsible for transferring a part of each tile, some warps may finish their transfer before others. If these warps attempt to continue execution and try to access data in shared memory that is still being buffered by other warps, they will not be computing on the desired tile of data, but rather some stale or otherwise incorrect data. To prevent this, the programmer must insert a barrier after this transfer (i.e. *__syncthreads*() in CUDA) which prevents warps from continuing forward until all warps reach this synchronization point. However, this implies that the slowest warp determines the end time of the transfer, and can prevent other warps that may be computing on data independent of the transferred tile from continuing.

All these sources of inefficiency indicates that there is some room for optimization in the GPU's ISA and microarchitecture. To improve upon the modern state-of-the-art of GPUs, we focus on adopting ideas from a well-established feature of CPUs, Direct Memory Access.

2.3 Traditional DMA on CPUs

The problem of gathering data for computation in a timely manner and without excessively consuming processing resources has been a concern for architects since time immemorial. In the early 1980s, Direct Memory Access (DMA) was pioneered by IBM to decouple long duration data transfers between peripheral devices and main memory from the CPU's datapath. Before DMA, if a program requested data from a peripheral, e.g. a floppy drive, it would either (a) poll the device to determine when the transfer completes, forcing the CPU to sit idle until the program can resume, or (b) let the operating system schedule other tasks until an interrupt signals that the data is ready. Both schemes incur high overheads when reading large amounts of data from devices as the CPU was spending a great deal of time calculating new addresses, issuing commands to move the data from a peripheral to memory, and performing bookkeeping to track the transfer's progress. With DMA, the program simply commands a DMA controller, a small hardware addition next to the CPU, to start copying a certain number of bytes between a specified memory address and a selected device. The DMA controller is responsible for generating addresses for data to be transferred to/from, issuing commands to the two endpoints, and arbitrating for the data bus, freeing the CPU's computation resources for other uses. Once a transfer completes, the DMA controller interrupts the CPU to indicate that the program's data is ready in memory, and the program continues execution [12, 27].

Many of the inefficiencies of GPU buffer transfers described in Section 2.2 seem like they could be solved using DMA-style transfers. However, reusing existing DMA designs and ideas for the GPU is not straightforward. As DMA was originally created for systems with very few threads executing simultaneously, DMA requires a reimagining when applied to a GPU capable of executing thousands of threads at the same time. This poses a few major challenges, listed below:

- Which CTA's warps should command a DMA controller to start a transaction?
- How can transactions from many warps be interleaved to occur concurrently?
- How does the DMA engine keep track of such pending transfers with reasonable overheads?

Furthermore, GPUs are not interruptible devices. If a memory transfer is offloaded to a DMA controller, it needs some way to inform the program that its transfer is complete. All of these challenges necessitate a rethinking of DMA before it can be applied to a GPU architecture.

3. D²MA OVERVIEW

As discussed in Section 2, a number of inefficiencies in modern GPUs are exposed when buffering global memory data in a shader's shared memory, which can prevent memory-intensive applications that employ such buffering from achieving their potential performance. The goal of this work is to address these bottlenecks by overlapping computation and buffer transfers, creating a slipstream effect that permits future computations to occur sooner than with the current buffering paradigm. In order to accomplish this, this work proposes a new direct memory access scheme that has been reimagined for GPUs called Data-Parallel DMA, or D²MA. D²MA accomplishes this by: *a)* decoupling address generation and memory accesses for data within the tile being buffered from the ALUs and memory units of the shaders by utilizing a new functional unit: the DMA engine; *b)* providing a direct path for data to travel from the off-chip global memory to the on-chip shared memory; *c)* ensuring that a pipelined stream of successive memory requests are issued to the global memory in a timely manner; and *d)* permitting warps to execute up to the point where an instruction attempts to load data from in-progress buffer transfer before forcing the warp to stop execution and wait for the completion of the transfer. In this section, we will discuss these improvements. Their implementation details will be explored in Section 4.

The current paradigm accomplishes data buffering into the shared memory by performing many address calculations and memory operations that consume the shader's primary functional units, depicted in Figure 2. To free these units for other computation, D²MA decouples almost all buffering operations from the shader pipeline by adding a DMA engine to each shader core as shown in Figure 3. As multiple

[4]The programmer can set the partition size so that 48KB is dedicated to shared memory and 16 to the L1 or vice versa. NVIDIA's Kepler architecture appears to have the same unified shared & L1 data cache with an added 32KB shared/32KB L1 configuration [23].

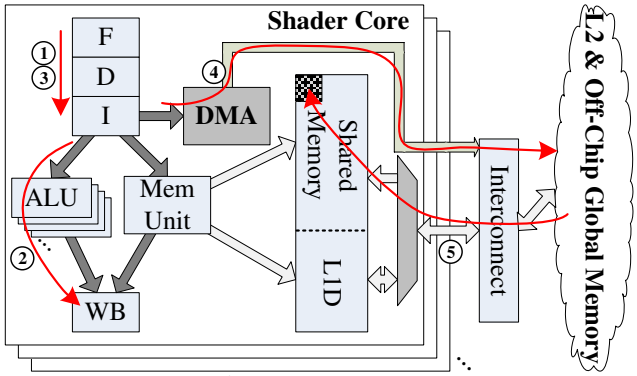


Figure 3: How a DMA engine can efficiently buffer data into the shared memory.

CTAs can be resident in a shader, each shader can be responsible for at least one buffer transfer occurring for each CTA. D^{2}MA's engine is capable of tracking not just one transfer per CTA, but can also permit multiple transfers to occur simultaneously for each CTA. Although DMA accesses are decoupled from the shader's pipeline, the DMA engine still requires the shader's ALUs to calculate two base addresses per tile transfer for each CTA, one for the source and one for the destination (①, ②). Once these two addresses are computed, however, the shader's functional units are never used again to facilitate the buffer transfer for a CTA. Instead, the DMA engine is passed configuration parameters, including the two base addresses, the size of each element and the number of elements in the buffer, and information describing the memory layout of the data being copied (i.e. the access stride for 1D data, or the order and leading dimension of matrix data). These configuration commands are implemented as two new instructions which are fetched, decoded, and wait for their operands (the aforementioned parameters) to be ready at the issue stage (③) like any other warp instruction.

Once a CTA has configured the DMA engine, the engine begins generating global memory addresses to load data from in a pipelined fashion and wholly independent of the shader's ALUs. As the shape of the data being buffered is very regular and is expected to be densely packed, at least until it encounters a stride boundary, the addresses of successive elements will naturally occur in a coalesced manner, obviating extra coalescing logic that exists in the shader's memory unit for general global memory accesses. When addresses are ready, memory requests are directly issued to the global memory through a shared memory port to the interconnect (④). As this memory request goes directly to the L2 and global memories, it bypasses probing the L1 data cache entirely, where it is likely to miss for reasons described in Section 2.2, and avoids modifying state in the L1.

During global address generation, the DMA engine concurrently computes the shared memory addresses associated with global addresses, and requests for data are sent to the global memory. Once responses to DMA requests start returning to the shader core (⑤), D^{2}MA stores each responses' data directly to the shared memory using the generated addresses. By selectively forwarding DMA responses to the shared memory, the data will only exist once in this physically unified L1 and shared memory space. Furthermore, the L1 will not be polluted with buffer data that should only persist until the next tile is transferred.

In order to prevent read-after-write consistency errors during buffering while remaining decoupled from the shader's pipeline, D^{2}MA replaces the coarse-grained synchronization instructions used in the current paradigm with a unique warp waiting scheme that is entirely managed in hardware and is transparent to the programmer. This scheme allows warps to continue execution until an instruction attempts to read data in shared memory that is currently being buffered. When such an event occurs, the load from shared memory gets squashed and its warp's next PC is rolled back to that of the squashed instruction. The warp gets marked as waiting for DMA, allowing the warp scheduler to issue a different warp that is ready for execution to the shader's functional units. As many warps in a CTA may be waiting for a DMA transfer to continue, the DMA engine tracks the waiting warps per CTA, and when the transfer completes it informs the scheduler that these warps may continue execution. When these warps are reissued, their next PC was rewound, allowing the load from shared memory to re-execute, thus preserving program correctness while preventing the shared memory from being read before a buffered tile was ready.

4. IMPLEMENTATION AND DESIGN

In this section, we detail the hardware modifications and additions to the shader core that are necessary to implement D^{2}MA. At the heart of this work is the DMA engine, as seen in Figure 3 and detailed in Figure 4. The DMA engine manages the buffering operations for all CTAs issued to a shader. Its key components include: an array of DMA controllers which contain all necessary configuration and bookkeeping metadata for buffer transfers (Section 4.1); a specialized address generation unit capable of creating one memory request for global data per cycle while concurrently generating each datum's associated shared memory address (Section 4.2); and a mechanism to prevent RAW consistency violations caused by attempted accesses to an incomplete transfer's data (Section 4.3). Section 4.4 details D^{2}MA's programming model.

4.1 DMA Controllers

The DMA engine contains one DMA controller per resident CTA, empowering D^{2}MA to manage simultaneous buffering operations for all the CTAs on a shader. Each controller contains all the configuration and bookkeeping metadata necessary to facilitate a buffer transfer in a table called the buffer management table. The tables are provisioned with four entries, allowing each CTA to launch up to four buffering operations at the same time.

When a DMA configuration instruction is issued to the engine, it populates this metadata into entries in the buffer management table, shown in Figure 4, associated with the instruction's CTA and the buffer specified by the instruction. Once configured, the table entry will contain the global and shared base addresses of the tile (64b and 16b, respectively), the number of elements to transfer (16b), the byte size of an element (encoded into 2b[5]), the stride of the data (16b), and flags that indicate the dimensionality of the data (1b[6]) and if a special addressing mode is enabled (2b). There are two supported special addressing modes, detailed in Section 4.2.1. One of the addressing modes creates a halo around the tile of data for stencil applications. This requires extra configuration data as each buffer must know the number of halo rows (16b) and columns (16b) that surround the

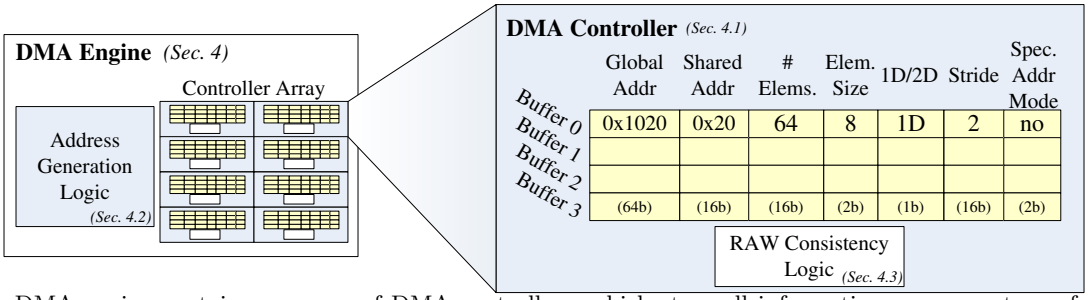


Figure 4: A DMA engine contains an array of DMA controllers, which store all information necessary to perform a buffer transfer, and specialized address generation logic.

tile, an indicator as to the position of the tile relative to the entire data (8b), as well as a constant to be stored in this halo region around the tile's data (16b). The other addressing mode is used to prevent shared memory bank conflicts, and shares these configuration bits with the halo addressing mode to store the number of elements that get stored in a shared memory bank. In total, each DMA controller requires 692 bits (173 bits per buffer) of metadata to configure and track a DMA transfer.

As all warps within a CTA execute the same configuration instructions, once a particular CTA's buffer is fully configured, the engine will not modify the buffer's table entry again and will commit all future configuration instructions.[7] When configured, the controller informs the DMA engine that a buffer transfer is ready for ignition, and the engine can commence this transfer.

4.2 DMA Memory Transaction Handling

When a transfer is ready, the DMA engine uses its address generation logic to efficiently generate memory requests to move this tile of data from global into shared memory. The requests are sent through the shader's existing memory subsystem to the global memory, and when responses are received, data is directly stored to the shared memory without modifying the L1D or requiring any writeback to registers.

Once a CTA's buffer is marked as ready for ignition and the specialized address generation (AGEN) logic is available, the DMA engine routes the buffer's metadata from the associated controller's buffer management table to the address generation logic. The AGEN logic is controlled by a byte counter (Figure 5a) that tracks if all addresses for this transfer have been generated. The total size is calculated by shifting the number of elements by the encoded size of each element and is stored in a register. The counter is incremented depending on whether the buffer's metadata indicates the data stored in global memory is unit-strided, or has a non-unit stride.

When data has a unit stride (Figure 5b), every element is packed contiguously in global memory, so the AGEN logic can easily generate global addresses based on segment boundaries.[8] In this case, the engine's AGEN logic can issue one request for a segment of data per cycle. For the non-unit stride case, data is still contiguous in global memory but is not densely packed. The data that is of interest will exist with distance equal to the stride between each element.

[5]Two bits are needed to represent all data type sizes: 1B, 2B, 4B and 8B.

[6]D^{2}MA currently supports 1D and 2D tile transfers.

[7]Configuration instructions can later change the buffer management table entries once a buffer transfer has completed.

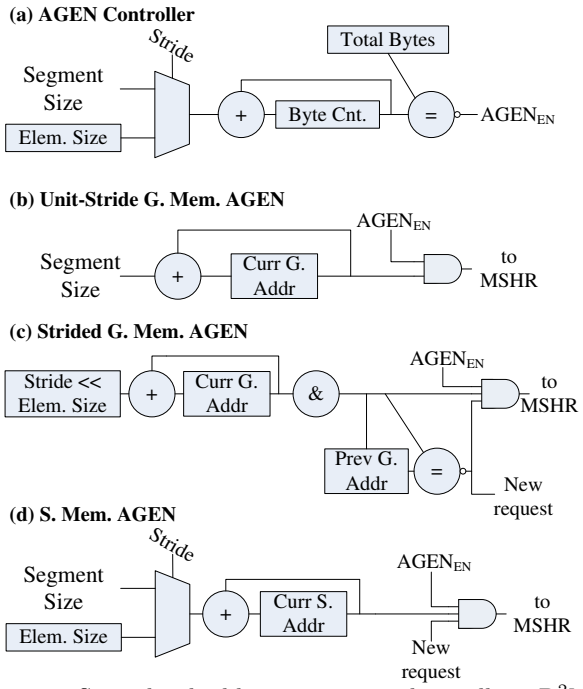


Figure 5: Specialized address generation logic allows D^{2}MA to rapidly issue memory requests and decouples all address generation from the shader's computational resources.

Because of this, some contiguous segments will not contain data of interest, so the AGEN logic only sends requests for those that do. As shown in Figure 5c, the address of the current element is stored in a register. To extract the segment address from this, the lower seven bits are masked off (&). Then, this segment address is compared to the one generated in the previous cycle. If they are not equal, this element is in a new segment and a new global memory request is ready to be issued. Because of this, it is possible that it will take multiple cycles to generate a strided memory request for each segment. As global addresses are generated, the AGEN logic also generates the shared memory addresses associated with each global segment (Figure 5d). If no special addressing mode is specified, data is expected to be contiguous and densely packed in shared memory. If the global addresses are strided, then the shared memory AGEN logic only issues a new shared memory address if the strided AGEN circuit signals that it created a new request.

As discussed in Section 3, D^{2}MA will write a response's data directly to shared memory and not to the L1D. In order

[8]Segments can be 32, 64 or 128B wide for NVIDIA architectures [22].

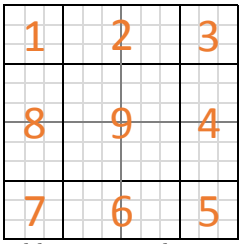


Figure 6: The halo addressing mode must track which region of the entire matrix a tile is coming from to determine what data is stored in the halo. There are nine possible regions.

to accomplish this, the MSHRs in the L1D are shared between the DMA engine and memory unit, and each MSHR has been provisioned with a 16b shared address field. When both the global request and its associated shared memory address(es) are generated by the AGEN logic, they are directly sent to the MSHR table, where the global and shared addresses are written if an entry is available. If the global address of a response from memory matches an MSHR entry that also contains a shared memory address (identifying a DMA response), the data gets stored to that location in the shared memory rather than the in the L1D.

To avoid complications of adding more nodes to the interconnect, the shader's memory port to the on-chip interconnect is shared between the DMA engine and the shader's memory unit. In case of contention between the engine and memory unit, all memory unit requests are given priority to ensure that computation is given priority over DMA operation.

4.2.1 *Special Addressing Modes*

D^{2}MA supports two special addressing modes. The first, called bank conflict resolution, spaces data stored to shared memory out so that when multiple threads access the buffer, they do not all attempt to read from the same bank at the same time. When shared memory addresses are generated, the AGEN logic only permits n elements to be stored per memory bank, as specified by buffer's metadata.

The second addressing mode creates a halo surrounding the buffered tile in shared memory. This is commonly used in image processing and physics simulations to include neighboring data when computing on a tile. This halo is either filled with data from neighboring tiles or with a constant value if that tile is on the border of the entire data. Figure 6 shows that every tile resides in a region that determines what data needs to be stored in its halo in shared memory. Interior tiles (region nine) will have halos that contain data from neighboring tiles. Exterior tiles (regions one through eight) have some rows and/or columns that will contain a neighboring tile's data and others that will contain the halo constant. When a halo is applied to a tile, the AGEN logic needs to know the region from which this tile belongs. The programmer supplies the current tile's region when configuring a new tile transfer. The size of the halo, the halo constant, and the current tile's region are loaded from the buffer's metadata for use by the AGEN logic. Based on the region the tile is in and the number of halo rows and columns for the tile, the AGEN logic will generate global and shared memory addresses for the transfer. For the exterior regions where some of the halo data is constant, the DMA engine will store the halo constant to the exterior halo parts of these tiles in shared memory while the global memory requests are in flight.

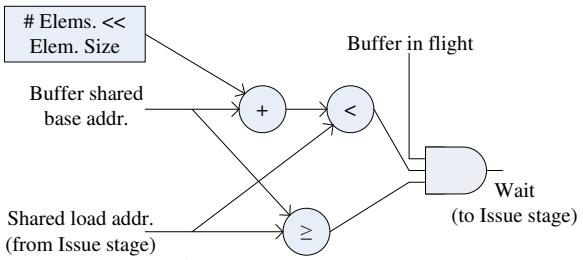


Figure 7: The DMA controller contains logic to check if a read-after-write consistency violation will occur if a shared load is issued that conflicts with an in progress DMA transfer.

4.3 Dynamic Synchronization Support

One of the goals of this work is to obviate the need for all warps to wait for a CTA's transfer to complete. To accomplish this, D^{2}MA dynamically monitors all instructions that load from shared memory while DMA transfers are in progress, and if loads are found to alias with a buffer in transit, their warps are forced to wait until the buffer transfer completes. This allows work independent of the buffered data to continue to execute in these warps past a point where a programmer would normally have forced a coarse-grained synchronization to occur.

Dynamic synchronization is achieved by collaboration between the DMA engine and the warp scheduler. If transfers are in progress and a load from shared memory instruction is encountered at the issue stage, the source address of the load is forwarded to the DMA engine. The DMA engine routes this address to all of its DMA controllers, where Read-After-Write (RAW) Consistency Logic (Figure 4) compares the address to the ranges of possible shared memory addresses for all in-flight buffers. The logic displayed in Figure 7 is used to perform this checking. To determine buffer address ranges, the total number of bytes transferred by a buffer is added to that buffer's shared base address. Two digital comparators then check to see if the shared load's address is within this range. If both comparators return true, and the associated buffer's table entry was fully configured, this indicates that a RAW consistency violation has occurred. In this case, a signal is sent back to the issue stage, informing the scheduler that this load's warp needs to wait. Upon receipt of such a wait signal, the scheduler will mark this instruction's warp as waiting for a DMA transfer to complete, allowing a different warp to be issued, and the scheduler will rewind the next PC of the shared load instruction's warp so that the load can re-execute when the warp is allowed to continue.

When a DMA transfer for a CTA has completed, the DMA engine passes the CTA ID to the warp scheduler so that it may release all warps within the CTA that may have been waiting. Each DMA controller will contain one of the circuits shown in Figure 7 per buffer entry, for a total of four RAW consistency checking circuits per controller. Implementing this functionality in hardware allows a guarantee of consistency while remaining transparent to the programmer.

4.4 D^{2}MA Programming Model

To simplify the process of programming using D^{2}MA, we extend intrinsics that map directly to DMA engine configu-

```
1  __global__ void
2  sgemv_DMA(int n, int m, int n1, float alpha,
3            float *A, int lda, float *x, float *y)
4  {
5    __shared__ float buff[VEC_ELMTS];
6    dma_set_datatype_float();
7    dma_configure_flat_unit_stride(buff,
8                                   x,
9                                   VEC_ELMTS);
10
11   int ind = blockIdx.x * num_threads
12                        + threadIdx.x;
13   A += ind;
14   float res = 0.f;
15   int k = 0;
16   for(int i=0; i<n1; i += VEC_ELMTS)
17   {
18     for(int j=0; j < VEC_ELMTS; j++)
19     {
20       res+=A[0]*buff[j];
21       A+=lda;
22     }
23     k += VEC_ELMTS;
24     if (k < n1)
25       dma_execute(buff, x+k, 0);
26   }
27
28   if (ind<n)
29     y[ind] = alpha * res;
30 }
```

Figure 8: A sgemv kernel written using D^{2}MA.

ration instructions. Figure 8 shows a sgemv kernel written using these intrinsics to enable DMA operation. Lines 6-9 show the initial configuration, where the buffer is configured for source data which is a flat, unit-stride, 1D vector comprised of VEC_ELMTS number of floats. The $dma\_set\_datatype\_float()$ intrinsic sets the data type of this buffer, while the $dma\_configure\_flat\_unit\_stride()$ sets the global and shared base addresses and the size of the buffer. After configuration, the DMA engine will start this transfer, and the kernel's execution will continue. Lines 10-19 are independent of the transfer in progress and can execute without waiting, and if the transfer is still in progress at line 20, the DMA engine will stall the warp attempting to execute this load from the shared buffer. By using D^{2}MA, the programmer does not need to worry about placing $\_\_syncthreads()$ in their code. At the end of the outer loop, the $dma\_execute()$ on line 25 begins the transferring the next tile of data into $buff$. Shown here are only a subset of the available intrinsics, other intrinsics permit configuring 2D matrices and strided accesses, setting the data type to be any possible data type, including packed floats, and enabling the special addressing modes.

5. EXPERIMENTAL EVALUATION

We model D^{2}MA's engine and all necessary microarchitectural modifications in GPGPU-Sim version 3.2.1 [3, 1]. Our baseline GPU is similar to the NVIDIA GTX480 [21] architecture, and the configuration parameters passed to GPGPU-Sim can be seen in Table 1. To evaluate D^{2}MA, we selected all benchmarks from the NVIDIA CUDA SDK [2] and the Rodinia benchmark suite [6, 7] that contained kernels utilizing the current paradigm of buffering highly regular tiles of data into shared memory, and we created D^{2}MA-optimized versions of these kernels instrumented using D^{2}MA's intrinsics discussed in Section 4.4. A list of the benchmarks is shown in Table 2. Overall speedups achieved by using D^{2}MA are surveyed in Section 5.1. In Section 5.2, we analyze the improvement in the duration of a buffer transfer when using D^{2}MA. We explore a breakdown of how D^{2}MA optimizes buffering with a case study in Section 5.3, and compare D^{2}MA to a software DMA scheme in Section 5.4. Finally, we discuss the implementation overhead of D^{2}MA in Section 5.5.

Number of shaders	15
Threads per shader	1536
Threads per warp	32
SIMD lane width	32
CTAs per shader	8
Registers per shader	32768
Shared memory per shader	48KB
DMA engines per shader	1
Warp scheduling policy	Greedy-then-oldest
L1 cache (size/assoc/block size)	16KB/4-way/128B
L2 cache (size/assoc/block size)	768KB/16-way/128B
Number of memory channels	8
Memory bandwidth	179.2 GB/s
Memory controller	Out-of-order (FR-FCFS)

Table 1: GPGPU-Sim configuration

Abbr.	Description	Ref.
DCT	Discrete cosine transform	[2]
MM	Matrix multiplication	[2]
DWT	1D Haar wavelet transform	[2]
TRANS1	Matrix transpose (coalesced)	[2]
TRANS2	Matrix transpose (diagonal)	[2]
FW	Fast Walsh transform	[2]
GAUSS	Gaussian filter	[2]
MEAN	Mean filter	[2]
SOBEL	Sobel filter	[2]
PATH	Shortest path finding	[6, 7]
SGEMV	Matrix-vector mult.	[4]
LUD1	LU decomposition (internal)	[6, 7]
LUD2	LU decomposition (diagonal)	[6, 7]

Table 2: Benchmarks used to evaluate D^{2}MA

5.1 Overall Performance

Figure 9 shows the overall speedup achieved when using D^{2}MA to optimize the transfer of data from off-chip global memory to on-chip shared memory. By efficiently generating memory requests and forwarding responses directly to the shared memory, D^{2}MA is able to achieve up to 2.29x and on average 1.36x speedup over kernels using the current buffering paradigm on modern GPU hardware. All but one D^{2}MA-enabled kernels see some degree of speedup. As D^{2}MA automatically generates addresses and issues memory requests, the shader's ALUs are not used for this task, reducing the number of dynamic instructions executed as shown by Figure 10. On average, D^{2}MA reduces the number of instructions executed by 7%.

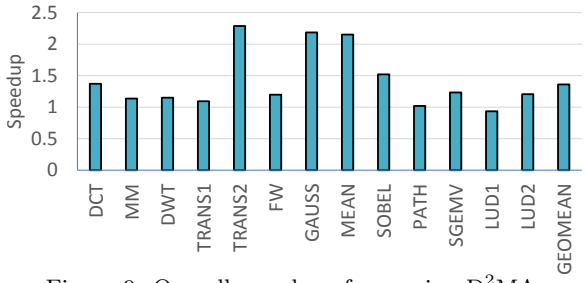


Figure 9: Overall speedups from using D^{2}MA.

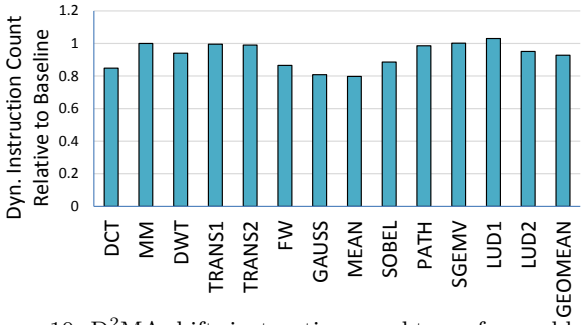


Figure 10: D^{2}MA shifts instructions used to perform address generation and memory transactions to dedicated hardware, reducing the number of dynamic instructions executed when performing a buffering operation.

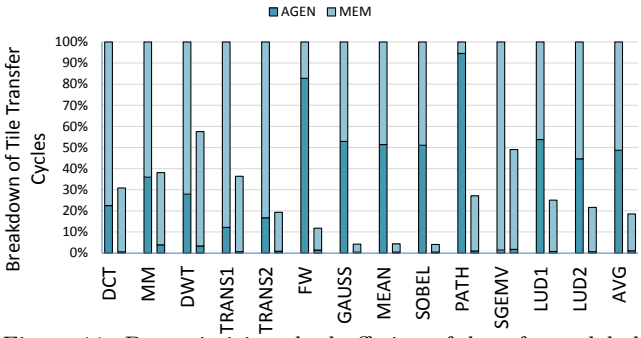


Figure 11: By optimizing the buffering of data from global into shared memory, D^{2}MA on average more than halves the cycles to transfer a tile of data. For each benchmark, the left bar represents baseline results, and the right bar shows D^{2}MA results relative to the baseline. (Lower is better.)

The *GAUSS*, *MEAN*, and *SOBEL* kernels show major benefits as these kernels utilize D^{2}MA's special halo addressing mode. The original code required conditional logic to determine whether the threads needed to store a constant or a neighboring tile's data to the halo surrounding the current tile being buffered. By allowing the DMA engine's specialized address generation logic to handle this computation, the optimized kernels achieve 2.19x, 2.15x, and 1.52x speedups, respectively. These speedups correlate closely with their large reduction in the number of dynamic instructions, which are reduced by up to 20%.

PATH shows a case that gets very little speedup (1.02x). In this case, the buffering operation in *PATH*'s kernel accounts for a very small fraction of its total execution time, so the speedup achieved by optimizing buffering is small. *MM* and *DWT* also exhibit a similar compute-to-memory characteristic as *PATH* and thus get limited speedups of 1.14x and 1.15x, respectively.

*LUD*1 is the only kernel that experiences slight loss in performance. The code footprint of the baseline kernel is very small, and only one transfer is performed per kernel call. Every time the D^{2}MA kernel is called, it must configure the DMA engine, which causes 3% more instructions to be executed compared to the baseline kernel. These factors account for this kernel's 7% performance degradation.

In order to analyze the performance of the remaining benchmarks, we turn to a more detailed study of D^{2}MA's improved buffer transfer performance.

5.2 Analyzing Buffering Performance

When analyzing the buffer transfers within kernels, we found that D^{2}MA is on average able to reduce the transfer time by 81% when compared to the current buffering paradigm. Figure 11 shows the breakdown of tile transfer cycles into address generation (AGEN) and memory transaction (MEM) cycles. It also shows the average improvement in overall transfer time for the D^{2}MA-enabled kernels normalized to that of the baseline kernels.

The significant improvements observed in both address generation and memory transfer cycles are primarily due to the decoupling of a buffering operation from dependencies upon other instructions and the shader's functional units. D^{2}MA's dedicated address generation logic accounts for a 98% reduction in AGEN cycles. As the baseline scheme requires each warp to transfer a part of a tile to shared memory, a transfer's performance becomes dependent on the warp scheduler allowing every warp to fetch their portion of the tile. On the contrary, once a DMA engine is configured, D^{2}MA generates segment addresses and issues them to memory wholly independent of the warp scheduler. This accounts for the average reduction in MEM cycles of 66%.

Generally, all D^{2}MA-optimized kernels see significant reductions in transfer duration due to buffering. For some benchmarks (*MM*, *DWT*, *PATH*), this did not necessarily translate into an overall speedup in execution time. Although *SGEMV* shows an improvement in transfer time comparable to that of *DWT* and *PATH*, *SGEMV* performs much simpler computation compared to these kernels, allowing it to achieve a better overall speedup of 1.23x.

As discussed in Section 5.1, the three filters (*GAUSS*, *MEAN*, and *SOBEL*) achieve very high speedups by allowing the DMA engine to handle generation of their tile's halo addresses. Figure 10 supports these findings as it shows that the baseline kernels spent a significant number of cycles performing address generation. By allowing dedicated logic to generate these special halo addresses and streamlining the issuing of their memory requests, D^{2}MA greatly reduces each of these kernel's address generation cycles by 99%. Combined with the timely issue of memory requests described in Section 5.1, these benchmarks see an average transfer duration reduction of 96%.

The *TRANS*1 kernel performs a transpose of a large matrix in a sequential, tile-by-tile fashion, while the *TRANS*2 kernel performs the transpose reordered along diagonal strips of the original matrix. While both kernels have similar code, they are buffering tiles in different orders. The baseline sequential kernel (*TRANS*1) suffers from high miss rates (about 50%) in the L2 cache, lengthening its transfer duration. The D^{2}MA version of this kernel achieves limited improvement for this kernel because D^{2}MA's optimizations are not meant to improve L2 caching performance. *TRANS*2's diagonal tile access pattern results in far fewer misses in the L2 with a miss rate of just 4%, and this allows D^{2}MA to improve the reduction in this kernel's transfer time by 17% over that of *TRANS*1. The reduction in transfer time accounts for the major overall speedup of the *TRANS*2 kernel. For similar tile ordering reasons, the access patterns of the two *LUD* kernels similarly account for one kernel performing slightly better than the other.

We also examine how frequently the shader is forced to stall due to memory operations in a buffer transfer compared to the kernel's whole execution time. Shaders are forced to stall during a transfer when all threads amongst the CTAs assigned to the shader are waiting for memory trans-

actions to complete. On average, the baseline benchmarks spend about 7% of their cycles stalled due to buffering inefficiencies. By decoupling the buffering operation from the shader's ALUs and memory unit and more efficiently moving data between the global and shared memories, D^{2}MA nearly eliminates these stall cycles. The D^{2}MA-enabled kernels, on average, stall for less than 1% of their total execution cycles.

5.3 Case Study: Breaking Down Transfer Times

To present a breakdown of transfer and compute times, we examine the execution of $SAXPY$, a scaled vector addition benchmark from [4]. We run a version of the kernel that buffers both vectors to be added into shared memory as our baseline, and optimized this kernel using the D^{2}MA intrinsics. $SAXPY$ contains fairly simple computations, as each tile will contain two vectors that get added together. This makes the buffering operation's transfer time critical when achieving good overall performance. The timelines presented in Figure 12 are from simulated executions of the two versions of the $SAXPY$ kernel, both running two CTAs (b_0 and b_1) on a single shader. Times are broken down into cycles when address generation (*agen*), memory transfer (*mem*), and computation (*comp*) occur.

In Figure 12a, we see the baseline's execution timeline. Each buffering operation has some overhead for address generation before a memory transfer can begin. Because all threads in the CTA will be involved in the transferring of each vector, the two transfers are serialized. Only after all threads involved in the transfer complete can the baseline kernel perform the actual vector addition. If we compare this to the D^{2}MA-optimized kernel's execution, displayed in Figure 12b, address generation is compressed by greater than 2/3, allowing memory transfers to occur sooner. Furthermore, both vectors are buffered simultaneously as each CTA utilizes two of their DMA controller's four available buffers. By improving address generation and overlapping these transfers, two periods of computation occur before cycle 2000, compared to just one for the baseline. Also, the computation in the DMA kernel is able to start slightly earlier than the baseline version, as D^{2}MA's dynamic synchronization does not force all threads to wait when some are ready to continue.

5.4 Case Study: Comparing with CudaDMA

CudaDMA [4] is a software solution that provides DMA-like emulation for CUDA kernels. Using fine-grained synchronization, CudaDMA allows the majority of warps to focus on computation on data while a few warps are tasked with buffering tiles of data from global into shared memory.

To compare D^{2}MA with CudaDMA, we optimized a matrix-vector multiplication ($SGEMV$) kernel for D^{2}MA and ran it along with kernels optimized using CudaDMA and the baseline software buffering approach. All three kernels transferred the subsections of the vector into shared memory using a single buffer. While the CudaDMA-optimized code achieved a 1.10x speedup over the baseline, D^{2}MA surpassed CudaDMA, achieving a 1.23x speedup over the baseline kernel. CudaDMA's warp specialization decouples memory access instructions from those performing computation, allowing for its gains over the baseline. As described in Section 5.1, D^{2}MA's dedicated hardware DMA engine obviates the need for these memory instructions and handles all address generation and issuing of memory requests in a manner that is almost entirely decoupled from the shader's existing functional units and warp scheduling.

Controller Array	45169.4 μm^2	2.40 mW
AGEN Logic	2246.6 μm^2	0.16 mW
RAW Consistency Logic	1493.8 μm^2	0.29 mW
Total per shader	54604.3 μm^2	3.72 mW

Table 3: DMA Engine Hardware Overhead

5.5 Implementation Overhead

Each shader is provisioned with one DMA engine, and all of its controllers require a total of 692 bytes of storage space to track buffering operations. Each of the shader's shared memory and L1D cache's 32 MSHRs are provisioned with 16 bits to track which shared memory address is associated with a memory response's global address, requiring 64 bytes of space. Compared to each shader's 32768 32-bit registers and 64KB shared memory/L1 data cache [21], the storage overhead of D^{2}MA is miniscule.

To estimate the total area overhead of the DMA engine, we created a Verilog model and synthesized it using Synopsis Design Compiler with a commercial 45 nm cell library. Table 3 shows the area and power overheads of a single DMA engine reported by the synthesis of our model. When each of the GTX480's 15 shaders is augmented with a DMA engine, D^{2}MA only requires about 0.016% of the total chip area and increases power consumption by only 0.022%.[9]

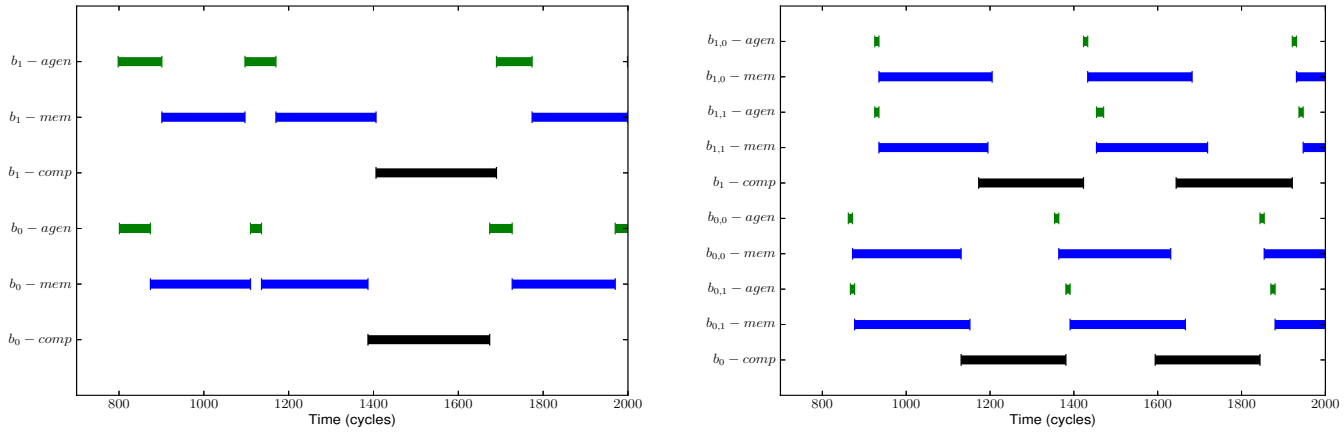
6. RELATED WORK

There are several works that attempt to improve the GPU performance by overlapping computation and memory accesses [4, 11]. Bauer et al [4] propose a software managed approach to overlap compute with memory access. They divide warps into computation and memory warps, where the latter retrieves data for the former. Hormati et al [11] use a high level abstraction to launch helper threads which bring data from global memory to shared memory and overlap computation. While these works improve upon the baseline buffering paradigm in [22], they require some of the shader's computational resources to be used to buffer data, whereas D^{2}MA decouples almost all operations related to buffering using a new functional unit, the DMA engine.

Memory prefetching is another way of improving memory accesses for GPUs [19, 31]. MT [19] provides hardware and software prefetching mechanisms designed for GPUs which exploit the existence of common memory access behavior among fine-grained threads. APOGEE [31] is another hardware prefetching mechanism which adapts to the memory access patterns found in graphics and scientific applications. Jog et al [13] propose a prefetch-aware scheduling scheme that enables a simple prefetcher to be effective in tolerating memory latencies. D^{2}MA differs from these works in that it is not predictive and relies on explicit instructions to fetch data. Due to this, D^{2}MA does not suffer from inaccuracy and only fetches data when instructed.

Different warp schedulers have also been introduced to improve the utilization of GPU memory systems [18, 9, 15, 20, 30, 14]. Lakshminarayana and Kim et al [18] evaluated various scheduling techniques for DRAM optimization. For systems with no hardware-managed caches, they proposed a scheduler which is fair to all warps. Gebhart et al [9] proposed a two-level scheduling technique to improve

[9]The NVIDIA GTX480 architecture has a 529 mm^2 die size and a TDP of 250 W [5].



(a) Baseline

(b) D^{2}MA

Figure 12: When two CTAs of $SAXPY$ are run with DMA support (12b), its transfer times are significantly reduced, allowing more computation to occur in the same time frame when compared to the baseline (12a). $b_{X,Y}$ indicates the CTA ID (X) and buffer number (Y).

register file energy efficiency. Kayiran et al [15] reduced the saturation of the memory subsystem by throttling the number of CTAs that are active on a SM. Narasiman et al [20] proposed a two-level warp scheduler which divided warps into fetch groups, and first scheduled within the fetch groups and then switched groups when all warps within a fetch group were stalled. Rogers et al [30] have shown similar results for a two-level scheduler. Jog et al [14] proposed a CTA-aware Locality BLP scheduler which reduced cache contention and used memory-bank level parallelism to improve performance. D^{2}MA currently improves energy performance independent of the warp scheduler, however new scheduling schemes to further improve memory transaction overlapping will be considered in future work.

Rogers et al [30] proposed a cache-conscious scheduling mechanism which used locality detection hardware in the cache to moderate the number of warps that accessed the cache and controlled cache thrashing. Gebhart et al [10] proposed a unified scratchpad, register file and data cache to better distribute the on-chip resources depending on the application. LAMAR [29] introduced a locality-aware memory hierarchy which provides the ability to tune the memory access granularity. LAMAR used a hardware predictor to predict when fine-grained accesses occurred, and optimized bandwidth utilization for such accesses. Unlike LAMAR, D^{2}MA's goal is to improve the memory utilization for coarse-grained accesses.

7. CONCLUSION

In order to achieve high performance on GPUs, effectively utilizing memory bandwidth is critical. One common technique to improve the memory utilization is have the programmer explicitly buffer tiles of data from off-chip memory into a fast on-chip memory for computation. Although this technique improves the performance of memory-bound applications, there is more room for improvement by reducing the overhead of generating addresses for each load and store instruction, and eliminating the redundancy of data as buffered data will occupy space in both the L1 cache and the shared memory when both memories are in fact unified.

In this work, we propose D^{2}MA to provide an efficient way to buffer tiles of data from global memory into fast access shared memory. This engine decouples the bulk address generation required to transfer tiles from the rest of the shader pipeline. This way, shaders can continue execution while DMA is transferring data from the global memory. It also supports different special addressing modes that are common in various GPU applications. D^{2}MA is also equipped with a novel waiting scheme that is managed in hardware and maintains read-after-write consistency while being entirely transparent to the programmer. Across 13 different kernels, D^{2}MA yields an average of 36% performance improvement while reducing the duration of a transfer of a tile of data from global to shared memory by 81%.

Acknowledgements

We would like to thank the anonymous reviewers whose excellent feedback helped shaped the final draft of this work, as well as Janghaeng Lee, Ankit Sethia, and Prateek Tandon for their insights and comments. This research was supported by the National Science Foundation under grant CNS-0964478.

8. REFERENCES

[1] "GPGPU-Sim," http://gpgpu-sim.org.

[2] "NVIDIA GPU Computing SDK," http://developer.nvidia.com/gpu-computing-sdk.

[3] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 163–174.

[4] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing gpu memory bandwidth via warp specialization," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 12:1–12:11.

[5] J. S. Brunhaver, K. Fatahalian, and P. Hanrahan, "Hardware implementation of micropolygon

rasterization with motion and defocus blur," in *Proceedings of the 2010 Conference on High Performance Graphics*, 2010, pp. 1–9.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of the IEEE Symposium on Workload Characterization*, 2009, pp. 44–54.

[7] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proc. of the IEEE Symposium on Workload Characterization*, 2010, pp. 1–11.

[8] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: A performance view," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 559 – 572, Sept 2007.

[9] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proc. of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 235–246.

[10] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," ser. Proc. of the 45th Annual International Symposium on Microarchitecture, 2012, pp. 96–106.

[11] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 381–392.

[12] P. Horowitz and W. Hill, *The Art of Electronics, Second Edition*. Cambridge University Press, 1991.

[13] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 332–343.

[14] A. Jog, O. Kayiran, C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," ser. 18th International Conference on Architectural Support for Programming Languages and Operating Systems, 2013, pp. 395–406.

[15] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 157–166.

[16] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[17] KHRONOS Group, "OpenCL - the open standard for parallel programming of heterogeneous systems," 2013. [Online]. Available: http://www.khronos.org

[18] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin, "Dram scheduling policy for GPGPU architectures based on a potential function," *Computer Architecture Letters*, vol. 11, no. 2, pp. 33–36, 2012.

[19] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *Proc. of the 43rd Annual International Symposium on Microarchitecture*, 2010, pp. 213–224.

[20] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 308–317.

[21] NVIDIA, "Fermi: Nvidia's next generation CUDA compute architecture," 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architect - ure_Whitepaper.pdf.

[22] *CUDA C Programming Guide*, NVIDIA, Oct. 2012.

[23] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Kepler GK110," 2012, www.nvidia.com/content/PDF/NVIDIA_Kepler_GK110 _Architecture_Whitepaper.pdf.

[24] ——, "Whitepaper: NVIDIA geforce GTX 680," 2012, http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[25] ——, "NVIDIA Tegra K1: A new era in mobile computing," 2014, http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-K1-whitepaper.pdf.

[26] Oak Ridge Leadership Computing Facility, "Titan: Built for science," 2012, http://www.olcf.ornl.gov/wp-content/themes/olcf/titan/Titan_BuiltForScience.pdf.

[27] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface, Fourth Edition*. Elsevier, 2012.

[28] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated CPU-GPU systems," in *Proc. of the 46th Annual International Symposium on Microarchitecture*, 2013, pp. 457–467.

[29] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *Proc. of the 46th Annual International Symposium on Microarchitecture*, 2013, pp. 86–98.

[30] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," ser. Proc. of the 45th Annual International Symposium on Microarchitecture, 2012, pp. 72–83.

[31] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," in *Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 73–82.