MRAM for Shared Memory in GPGPUs
Prateeksha Satyamoorthy and Sonali Parthasarathy

Abstract—

The application of new emerging Non-Volatile Memory technologies (NVMs) such as MRAMs in other processing contexts poses an interesting challenge. GPGPUs are used to exploit parallelism in non-graphic applications. While shared memory access is much faster than global memory accesses, the size of the available shared memory in current GPGPUs is limited to 16KB which puts a ceiling on the performance benefits that can be reaped. This paper proposes a design methodology to solve this problem by replacing the SRAM shared memory with an MRAM within the fixed area and power constraints. It also describes the design and policy of a write buffer which masks the latencies associated with MRAM writes.

1. INTRODUCTION

Graphic Processors (GPUs) have been exploiting parallelism in graphic processing applications to improve performance for more than three decades. With the advent of General Purpose Graphics Processors (GPGPU), parallel processing has been extended to compute-intensive and other general purpose applications. Since a GPGPU is a secondary processor it operates in conjunction with the CPU to process an application. The CPU copies the data to be processed into the GPGPU’s global memory and once the processing is completed the data is written back to the CPU. An access to global memory takes around 400-600 clock cycles, a rather costly operation. Therefore, in order to speed up the processing, thread accesses to off-chip global memory must be reduced as much as possible and on-chip memories should be made optimal use of. The NVIDIA GPGPUs of the Tesla architecture [1] have on-chip memories such as shared memory and constant caches in the Streaming Processor (SM), texture caches shared amongst 2-3 SMs and a global L2. The shared memory in each SM i.e. the on-chip scratchpad has the same access latency as the register file when there are no bank conflicts. The shared memory is divided into 16 banks providing higher memory bandwidth. This enables quicker access to data precluding the need for global memory accesses. Intuitively one would want to keep all of the data required by the threads of an SM on the shared memory and reap performance benefits. However, scratchpads are only 16KB in size which is one of the main factors limiting the amount of data that can be processed either at any given instance, or by a single CUDA block [1]. A shared memory variable is only visible to threads belonging to a block. The number of threads per block is therefore restricted by the limited memory resources of a processor core. The number of blocks an SM can concurrently process at a time, referred to as the number of active blocks [1] depends on the number of registers per thread and the amount of shared memory per block required for a given kernel [1]. It is desirable to increase the number of active blocks on an SM as it would increase performance by masking latency. For example, consider an application using only doubles values: the 16KB shared memory can accommodate at most 2K double values. If this could be increased, fewer global memory accesses would be required to obtain data for processing. This implies that the same thread block can now do more work. This shows that there exists a need for a higher capacity memory within the fixed area and power constraints.

A new class of memory technology called Non-Volatile Memory (NVM) promise just that. Many NVMs such as Phase-Change Memory (PCM), Magnetoresistive RAM (MRAM), NAND and NOR Flash offer higher density i.e. higher capacity per unit feature size than SRAM. The cell area of PCM, STT-RAM, NAND Flash are 5-8 F² [13], 37-40F² [3] and 6-7 F² (ITRS 2008). In addition, the read latencies/access times are in the order of nanoseconds comparable to SRAMs. One of the attractive features of the NVM technology is its low leakage power. This paper proposes replacing the SRAM shared memory with MRAM (STT-RAM) to tackle the issue of providing higher capacity for a given area. Not only does it provide 4x the density of SRAM but also gives 3.5x total power savings and 5.36x leakage power as we show later on. The increased density provides many advantages. For one, it increases the shared memory double values from 2K to 8K for one time processing. This extra capacity could also be used for double buffering where the next thread block to be executed can stream in its data while the current one is computing. Finally, instead of increasing the scratchpad size, more processing elements could be accommodated in the same area for the same shared memory capacity.

Section 2 of this paper discusses the GPGPU architecture in particular the NVIDIA Tesla Architecture. Section 3 examines the properties and working of an MRAM with special attention to its use as a shared memory. In sections 4 and 4 of this paper we introduce our design methodology and the reasons for our approach. We compare and contrast our work with literature in the field in sections 6. Sections 7, 8 and 9 describe the experiments designed, the benchmarks used and the simulations performed respectively. Performance analysis for our proposed work is discussed in section 10. Section 11 proposes future work which discusses extensions to this paper.
II. GPGPU ARCHITECTURE

Graphic Processing Units (GPUs) are secondary processors which work in conjunction with a CPU to exploit parallelism in graphic applications. Although GPUs have been used in parallelizing graphic applications, the feasibility of executing non-graphic applications on GPGPUs was not recognized until the introduction of the NVIDIA Tesla architecture. At the crux of its architecture is the notion of multiple CUDA cores which can process multiple threads at any given instance of time. Using CUDA, an extension of C, kernel programs are executed on N different multiprocessors by N different CUDA threads. Threads are grouped into blocks and scheduled on a Streaming Multiprocessor (SM) for execution. GEForce 280 GTX, an example of the NVIDIA Tesla architecture, has 240 CUDA cores and can process thousands of threads in-flight (get correct no) at any given time. Embarrassingly parallel applications benefit the most with this many number of cores. The Tesla architecture is a SIMT (Single Instruction Multiple Thread) processor which maps data elements to parallel processing threads. It consists of 16 SMs each consisting of 8 Streaming processors (SPs). Two SMs clubbed together to make up 8 independent processing units called Texture/Processor Clusters (TPCs) (refer fig.1).

The TPU consists of a geometric controller, an SM controller (SMC), two SMs and a texture unit as illustrated in figure 2. Each SM in addition to containing 8 SPs consists of an Instruction cache, a multithreaded instruction fetch and issue unit (MT issue), a read-only constant cache (C cache), two special functional units (SFU) and a shared memory unit. For easy management, the Tesla architecture schedules and executes threads in groups of 32 threads called a warp. Each warp therefore consists of threads of the same type that pertain to a single instruction. Since each SM manages 24 warps, a maximum of 768 threads (which forms a block) gets scheduled on an SM for execution [1]. Each thread has access to its private local memory, shared memory visible to all threads of that block and global memory visible to all threads in the application. In addition, threads can also access texture and constant caches which are visible to all threads in the application. The texture and constant caches are read-only and are optimized for different memory usages. The GPU also contains an L2 cache which caches data from the DRAM of the global main memory to the SPs. All blocks and threads across the SMs can access the global memory. The shared memory has a separate address space which is disjoint from the main memory, as opposed to the single main memory used by global memory, texture and constant caches. To use the shared memory, data needs to be copied from global memory to shared memory via explicit instructions.

III. MAGNETORESISTIVE RAM (MRAM)

An emerging area for memory technologies is non-volatile memory. Non-volatile memory often provides greater density for much less power consumption. Of all the candidates from the non-volatile family which vie for the spot, MRAM is the best fit because of the density and power savings it offers.

Figure 1: Tesla GPU Architecture [1]

Figure 2: Structure of a TPC [1]

Figure 3: Structure of an MRAM cell [12]
MRAM uses magnetic tunnel junction (MTJ) to store information. To understand the working of an MRAM an overview of the MRAM cell structure is necessary. An MRAM cell consists of a transistor and an MTJ stack. The transistor acts as a current rectifier which helps in the read operation. The MTJ consists of two thin ferromagnetic layers separated by a thin dielectric barrier. One layer is pinned in a specific direction while the direction of the other (free) layer can be adjusted. The relative polarization directions of the two layers determine the resistance of the MTJ. If the two layers are parallel the resistance is minimal and a state “0” is recorded while an anti-parallel situation causes increased resistance with a stored state of “1”.

<table>
<thead>
<tr>
<th>Memory Capacity</th>
<th>SRAM</th>
<th>DRAM</th>
<th>MRAM (STT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td>44mm²</td>
<td>49mm²</td>
<td>38mm²</td>
</tr>
<tr>
<td>Latency</td>
<td>4.659ns</td>
<td>5.845ns</td>
<td>Read 4.693ns Write 12.272ns</td>
</tr>
<tr>
<td>Dynamic Energy/Operation</td>
<td>0.103nJ</td>
<td>0.381nJ</td>
<td>Read 0.102nJ Write 2.126nJ</td>
</tr>
<tr>
<td>Leakage Power</td>
<td>5.2W</td>
<td>0.52W</td>
<td>0.97W</td>
</tr>
</tbody>
</table>

Table 1: Comparison of Memory Technologies for a 65nm technology

Table 1 shows that MRAM provides 4 times more density than SRAM for slightly lesser area (SRAM has a feature size of 50F2 to 120F2). The read latencies and dynamic energy per read operation are nearly the same for both. However, leakage power is 5.36 times less for MRAM as compared to SRAM [2]. This is the upper bound estimate as this is for STT-MRAM (Spin Transfer Torque MRAM). The previous generation MRAM uses a current induced magnetic field for the write operation i.e. the write operation is magnetic. When the current is passed through the bit line and the word line, an external magnetic field is induced which polarizes the free layer. As the MTJ size decreases the demagnetization field increases leading to higher write power consumption and lower scalability. In comparison, the STT-RAM’s read/write operation is electrical. The spin-polarized electrons exert a “torque” on the free layer which changes the polarity of the layer [3]. Since the current is directly applied to the MTJ the need for an external magnetic field is precluded. This greatly reduces the power consumption. Further, the threshold current needed for state reversal decreases with decrease in MTJ size [4]. This accounts for the scalability aspect. In spite of these characteristics, STT-MRAM does exhibit 3 times the latency of SRAM and 20 times the dynamic energy for write operations. However the justification for using STT-RAM is its low leakage power benefits which more than makes up for the high dynamic energy in write operations. For the rest of the paper we use MRAM to refer to STT-RAM.

Another recent MRAM technology is the Spin-Injection MRAM. For conventional MRAMs write currents needed for magnetic reversal is inversely proportional to the line width. In order to support this high current, transistors with high capacity are required. However, the increased capacity transistor and word line leads to an increase in the cell area (12F2) almost twice as much as a DRAM. Spin-Injection MRAM tackles this issue by applying current directly to the TMR device. The threshold current needed for this reversal decreases with line width reducing the area requirement. Further, the absence of the word line causes an additional reduction in area bringing the cell area down to 6F2 [5]. In fact, the read/write time access times can be as low as 2ns. However, dynamic and leakage energy values for Spin-Injection MRAM aren’t currently available which necessitates the use of STT-RAM for our experiments.

To justify the usage of MRAM, values from Table 1 were used to calculate the total energy consumed by continuous operation of both SRAM and MRAM. Total energy was computed as the product of Total Power and Time (1s), where Total Power is the sum of Read Dynamic Power, Write Dynamic Power and Leakage Power. These calculations yielded Total Power values of 5.24J for SRAM and 1.16J for MRAM. Since the write latency of MRAM is 3 times that of SRAM, the number of writes for 1s of operation would be 3 times less. Hence, we calculated the total energy consumed by MRAM for the same number of reads and writes as SRAM, which turned out to be 1.51J. Therefore, we concluded that the minimum energy consumed by an MRAM memory cell would be 3.472 times less than SRAM. In addition, MRAM is 4 times denser than SRAM. Hence, for same memory capacity MRAM would give up to 13.88 times less power consumption.

IV. DESIGN METHODOLOGY

This paper discusses a new shared memory design which gives higher performance benefits for GPGPUs. The original SRAM shared memory is replaced with MRAM. The primary contributor to latency is the write latency associated with the MRAM. For the existing technology, we intend to solve this latency problem by adding a write-buffer (refer Fig. 4). This write buffer is implemented in SRAM to give the same write latency as an SRAM scratchpad. Read requests are serviced by both the write buffer and MRAM shared memory (scratchpad). Recent values are read from the write buffer, whereas older values are read from the scratchpad. Writes from the processing elements (SPs) are done to the write buffer. And upon arriving at commit time, are committed to the MRAM shared memory.
Density and power benefits apart, MRAM can be integrated into the processor die in layers above those of conventional wiring. This creates exciting possibilities for GPUs by providing stacked layers of scratchpads, caches and main memory on top of the processing elements. Additionally, MRAMs are being explored to use MLCs (Multi-level cells) [? ] where one memory cell can store multiple values [rephrase] thereby increasing the density. In fact for a 4-level MLC the density increases as much as 16 times. While the use of MRAM has been explored within the memory hierarchy as the last-level cache either as a complete replacement for SRAM [2] or in the form of a hybrid-technology solution [6] and for main memory [7], there is little work that has examined on how it could be used at higher levels of the hierarchy. Maashri et al. discusses MRAM in the context of the GPU but their paper concentrates on the effects of 3D stacking caches specifically the texture and Z caches on the GPU performance [8]. Restricting MRAM use to these caches limits the variety of applications that can make use of the MRAM technology i.e. only graphic applications use texture and Z caches. Our paper in an effort to generalize the use of MRAM for non-graphic applications replaces the SRAM shared memory with MRAM. Further, shared memory capacity is bigger than the read-only caches which would benefit more from the increased memory capacity.

V. THE WRITE BUFFER

The write buffer is placed between the processing elements (SPs) and the MRAM shared memory. A write buffer is a temporary place for storing recently written values which will eventually be written through. In conventional write-buffers, if the data requested is present in the buffer, read requests on a miss in the memory are stalled until the data is written through to the memory from where it’s later read. On a write request, data if found in the write buffer is modified before being written through. As shown in Figure 4, the design of our write buffer is such that if the data requested for read is found in the write buffer it is read from the write buffer itself without any stalls. For a write request, new values are written into the write buffer unless the data is already present in which case it is modified. Write-through to the MRAM shared memory can happen either every fixed interval (for example every 4th clock cycle) or when the write buffer is full. We intend to evaluate various possibilities.

Another design issue is that of area. The area of the scratchpad could be kept the same in which case an optimal area tradeoff between the write buffer and scratchpad could be converged upon. Alternately, the capacity of scratchpad could be doubled to 32KB while keeping the write buffer relatively small. This would shrink the size of scratchpad and hence the SM. A direct implication of this would be the potential to allow more SMs to be placed on the GPU thereby increasing the throughput.

VI. RELATED WORK

A. For caches in GPGPUs

Implementing texture and Z caches with MRAM by 3D stacking has been explored by Maashri et al [8]. They make the case that texture and Z caches affect the performance of applications. But the caches they explore are mainly used by graphics applications and do not really address GPUs in the context of general purpose computing. Also, this paper addresses the latency problem by 3D stacking MRAM over the GPU. The reasoning behind this being that 3D stacking allows shorter wires and hence reduced latencies. In contrast, this paper uses a write buffer to mask the write latencies.

B. For L2 and L2/ L3 caches in CPUs

Li et al [7] and Dong et al [8] explore MRAM for L2 and L2/L3 respectively. These studies are for the CPU which presents a very different execution environment from a GPU. Besides, L2 and L3 latencies have lesser impact on the performance of an application and it is difficult to get performance improvements after a certain point no matter what size the L2 or L3 is increased to.

C. For Main Memory in CPUs

Desikan et al explore MRAM as an on chip main memory by 3D stacking it on the processor. This study proposes having MRAM on chip as a fast accessible alternative to having an off-chip main memory whose access latency is hundreds of cycles. They propose MRAM as a high-bandwidth low latency technology and do not touch on the latency issue any further. This limits their findings to 3D on-die integration of MRAM.

VII. EXPERIMENTS

In order to prove the benefits of bigger shared memory, we wrote a micro-benchmark which uses variable amounts of shared memory. This benchmark also determines the number of blocks that could be scheduled on an SM at any given time. To demonstrate the effects of having varying amounts of shared memory on an actual GPU and to limit the number of blocks that can be concurrently executed on an SM, different amounts of shared memory were blocked out using an optional variable while launching a kernel for dynamic shared memory allocation per SM. For varying block sizes i.e. the number of threads in a block the effect of block size on performance is observed. Inside the kernel intensive arithmetic operations
such as multiply and divide are performed in a loop to maximize access to shared memory. This represents the behavior of converging algorithms.

A. Effect of shared memory capacity

As we can see in Figure 5, the performance of a block with 16 threads (in red) using 384B of shared memory for computation decreases as the amount of available shared memory decreases. The effect of shared memory on performance seems to be most pronounced in blocks with smaller sizes. This behavior depends on the number of blocks that can be scheduled on an SM with a given shared memory usage. As the amount of shared memory used by 2 blocks cannot be accommodated within an SM only single blocks can execute on an SM at a time. Figure 5 shows how after the computational shared memory reaches and crosses its limit the performance remains fairly constant. Performance of blocks whose sizes are much less than available shared memory is most affected. The average performances of blocks whose sizes are bigger are better than the best performance of smaller blocks. This shows how bigger shared memories could enable bigger blocks which perform better.

B. Effect of block size

Figure 6 shows how for a fixed amount of available shared memory increase in block size increases performance. Also, the performances of the blocks are best when shared memory available is in excess of that required.

VIII. BENCHMARKS

The selection of benchmarks were based on those that showed different access patterns of shared memory and used varied amounts of it to demonstrate how bigger shared memory availability increases overall performance. For our experiments we chose two benchmarks from the Rodinia Benchmark Suite: Speckle Reducing Anisotropic Diffusion (SRAD) and Needleman-Wunsch (NW).

Structural grid applications involve the decomposition of the computation into highly spatial sub-blocks such that any change to one element depends on its neighbors. SRAD (Speckle Reducing Anisotropic Diffusion) is an example of a structural grid application. Figure 7 shows the computation pattern for a grid where a modification to the selected cell depends on its the four neighboring cells. It is essentially a diffusion algorithm for ultrasonic and radar imaging applications that uses partial differential equations (PDEs) [10]. It processes the images by identifying and removing locally correlated noise known as speckles without destroying important image features. Since SRAD is a structural grid application with computations over sets of neighboring pixels, it takes advantage of the on-chip shared memory. However, the working size of each thread block is limited to only 16 KB of shared memory and hence multiple thread blocks are needed. This limits the size of data blocks to 16x16 which gives poor performance for large data sets. Increasing the shared memory capacity using MRAM would allow bigger blocks sizes thereby improving performance.
The second benchmark used in our experiments is a dynamic programming application which solves an optimization problem by storing and reusing the results of its sub problem solutions. Needleman-Wunsch is a global optimization method for the alignment of sequences typically used for protein and DNA sequences. The pairs of sequences are organized in a 2D matrix and the algorithm is carried out in two steps. In the first step the sequence values are populated in the matrix from top-left to bottom right. To get the optimal alignment, the pathway with the maximum score is obtained where a score is the value of the maximum path ending at that cell [11]. This benchmark exploits the advantages of shared memory as each data element is used four times to calculate the values of four different elements. Figure 8 shows a graphical representation of the two kinds of parallelism. The small blocks represent the data elements while the larger blocks depict all the data elements that belong to a single thread block. Data elements on the same diagonal in a thread block are concurrently executed, while thread blocks on the same diagonal within the entire matrix are executed in parallel. The baseline SRAM shared memory per SM offers a maximum capacity of 10KB and so these benchmarks were modeled to use this capacity to its fullest i.e. thread blocks of size 16 i.e. 16x16 data blocks. Increasing this memory capacity enables the scaling of the benchmark to 32, 64 and 128 blocks which would result in more parallelism and hence high performance.

In order to model the MRAM shared memory and write buffer GPGPU-Sim was used [14]. GPGPU-Sim is a simulator that models various aspects of massively parallel GPU architectures. It parallelizes the existing CUDA SDK with highly programmable pipelines similar to contemporary applications to support the CUDA parallel thread execution (PTX) instruction set. The simulator transforms the CUDA application course code into host CPU C code and device C code to run on the GPU. However instead of using NVIDIA’s libcuda.a the simulator employs a customized libcuda.a for linking. The customized version contains stub functions which are used to set up and start simulation sessions of the kernels on the simulator. GPGPU-Sim provides many configurable parameters such as number of shader cores, warp size, number of threads, number of registers, shared memory size, constant cache size, texture cache size, DRAM bandwidth, number of access ports and so on. Changing the shared memory capacity as well as the number of access ports available for it models behavior of shared memory with sizes larger than 16KB which is the current limit for the GPU. The simulations were performed assuming a write buffer, allowing us to assume the read and write latencies of the SRAM write-buffer and MRAM shared memory system, to be the same as that of the SRAM shared memory system which exists. We realize that there would be additional stalls and latencies when the simulator is modified to accurately route the instructions bound towards the shared memory through the write buffer, which would mostly involve tag lookup to ascertain whether the value to be processed is present, hence read, from the write buffer or MRAM shared memory. We intend to incorporate this in our future work. The simulator was also modified to obtain results to reflect the latencies of a pure MRAM shared memory system for comparison.

X. Evaluation

A. Speckle Reducing Anisotropic Diffusion (SRAD)

SRAD was modeled to make use of a maximum of 16KB per SM shared memory. Replacing the shared memory with MRAM gives four times the capacity of SRAM. We get results for blocks with 64, 256, 1024 and 65536 threads by varying the shared memory size from 8KB to 512KB. Figure 9 shows the simulation time obtained for the above values for 8 concurrent Cooperative Thread Arrays (CTAs) [14], in other words block size. Comparing the patterns for the four block sizes we find that with smaller block sizes(8x8, 16x16) the simulation time is greater than larger block sizes (32x32, 64x64). This is fairly intuitive because for the same shared memory capacity a larger block will execute more threads in parallel. Also, for small block sizes increasing the shared memory does not show a significant increase in performance i.e. reduction in simulation time. This is because the block contains fewer threads which spend most of their time scheduling blocks across SMs. One anomalous data point was an increase in simulation time when the
shared memory size was increased to 512KB. Here, in spite of the large capacity, the performance is limited by the larger number of CTAs that need to be scheduled.

Figure 9: SRAD – 8 concurrent CTAs or blocks per shader core (SM)

Comparing figures 9 and 10 we see that the time taken for an 8x8 block to run for 8 concurrent CTAs is 14% more than that of the same block running for 32 concurrent CTAs. An increase in the number of concurrent CTAs decreases time spent in scheduling thereby giving better performance. For larger block sizes i.e. 32X32 and 64X64, increasing shared memory gives a decrease in simulation time (fig 9 and 10). Notice the absence of data points for block sizes of 16X16, 32X32 and 64X64. These are attributed to those cases when the amount of shared memory used by the threads of a block exceeds the available shared memory capacity.

Figure 10: SRAD – 32 concurrent CTAs or blocks per shader core (SM)

B. Needleman-Wunsch (NW)

Similar experiments were performed on the NW benchmark. Increasing the block size from 16X16 to 32X32 to 64X64 shows a drastic reduction in simulation time. However, further increase in block size does not yield better performance. This is due to the limited number of threads that are scheduled on an SM which are imposed by the CUDA libraries used by GPGPU-Sim.

Needleman-Wunsch differs from SRAD on in the amount shared memory required and the frequency of the reads and writes to the shared memory. NW uses around 4 times less shared memory than SRAD but performs more computations using shared memory than SRAD. This is the reason for the resulting bigger performance improvement in NW (average 70%) than SRAD (average performance improvement of 16%). This also proves the case that applications which perform intensive computations can obtain a boost in performance if shared memories larger than 16KB are available.

Figure 11: Needleman-Wunsch - 32 concurrent CTAs or blocks per shader core (SM)

XI. DISCUSSION AND FUTURE WORK

In this paper we have evaluated an MRAM based shared memory against SRAM to show the benefits of higher capacity shared memory for the same area as an SRAM based shared memory. However, we have not modeled the exact routing of shared memory bound operations through the write buffer as we intended to do. This we propose to do as future work. Also, the results we obtained could be the upper bound on the performance that can be obtained from the MRAM shared memory and SRAM write buffer system.

We envision this work being extended to other memories in the memory hierarchy i.e. register file, the caches, and device main memory to study the effects on performance by implementing them in MRAM.

Another direction for future work would comprise evaluating area tradeoffs for including the write buffer apart from converging on an optimal write buffer size.
XII. CONCLUSION

The high density, low leakage power benefits of MRAM technology make it an attractive candidate for use in the memory hierarchy. GPGPUs used in parallelizing non-graphic computations contain SRAM shared memory per SM. However, this shared memory is limited to 16KB in the current Tesla architecture. This paper provides a high capacity MRAM replacement for shared memory.

XIII. REFERENCES


6. X. Wu et al., “Hybrid Cache Architecture with Disparate Memory Technologies”, ISCA 2009


15. Include values for real hardware (copy, process and copy back)

Synthetic benchmark

write buffer