Evaluation of CUDA Memory Fence Performance; Berlekamp-Massey Case Study

Hanan Ali
hananahassan@mucsat.sci.eg
Fax: +2034593415

Zeinab Fayez
zeinab.fayez@gmail.com
Fax: +2034593415

Ghada M.Fathy
eng.ghadafathy@gmail.com
Fax: +2034593415

Walaa Sheta
wsheta@mucsat.sci.eg
Fax: +2034247511

Informatics Research Institute
City of Scientific Research and Technological Applications New Borg El-Arab, Alexandria, Egypt

Abstract - Graphics processors Unit (GPU) architectures are becoming increasingly programmable, offering the potential for dramatic speedups for a variety of general purpose applications compared to contemporary general-purpose processors (CPUs). However, GPU architecture depends on multithreading that needs to share data and resources that face memory concurrency issues. Data races and deadlocks are the most challenging Concurrency and consistency issues due to the non-deterministic execution of threads. In this paper, we evaluate the performance of CUDA Memory fence to solve the data race problem via implementing the Berlekamp-Massey Algorithm as a case study. The results showed that CUDA memory fence improves algorithm speed up with small input sequence.

Index Terms-, *Graphics processors, GPU, CUDA, Memory Fence, Memory concurrency*

I. INTRODUCTION

The many cores and high data bandwidth of graphics processing units (GPU) allow a high arithmetic throughput for several computational intensive applications. Although the GPUs are originally designed for graphics acceleration and 3D games, nVIDIA released a series of microprocessors that support the CUDA architecture and allow developers to run a general purpose codes on GPU.

GPUs and highly concurrent devices that allow multiple processors to share data are suffering from memory concurrency issues. These issues are reported in traditional CPU processors as well. Data races and deadlocks are the most challenging Concurrency issues due to the non-deterministic execution of processes. Consequently, an obtained results may vary when execute the same program with the exact same input. GPUs are typically programmed using a Single Thread Multiple Data (STMD) parallel model. This model allows us to execute several threads of the same program even if they take a completely different control flow way. Threads are organized into a computation grid of thread blocks. Each block has an identifier and each thread has an identifier within the thread block. These identifiers are used to map the computation to the data structures. Memory updates performed by a thread might not be perceived by other thread, except for atomic and memory fence (GPU-wide and system-wide) instructions. This is a key feature in order to implement the shared memory model, when the kernel in divided to be executed on multiple GPUs. The same consistency model is provided for the thread blocks running on different partitions

Through our experiences in improving some applications on GPU. Berlekamp-Massey Algorithm (BMA) was the most GPU implementation faced data races and deadlock problems. Therefore, BMA was chosen as a case study to evaluate its performance after using memory fence.

The main purpose of the Berlekamp-Massey Algorithm is to evaluate Binary Bose-Chaudhuri Hocquenghem codes (BCH). Berlekamp published his algorithm in 1968 [1] and it was followed shortly by Massey's publication of a variation on the algorithm in 1969 [2] . The algorithm is most widely used as a fast way to invert matrices with constant diagonals. The algorithm works over any field, but the finite fields that occur most in coding theory are the most often used. it is specifically helpful for decoding various algebraic codes. The algorithm used a key equation to input a known number of coefficients of the generating function and then determine the remaining coefficient of the polynomial. This process is equivalent to finding the linear complexity of the system.

The rest of the paper is organized as follows: Section 2 describes the most relevant related work. Section 3 presents our method and the major optimizations to improve the implementation of BMA. Section 4 describes experimental results. Section 5 contains discussion, conclusion and future work.

II. RELATED WORK

As started above, concurrency Memory issues have become very critical issues in parallel computing. This opportunity should redirect efforts of research in various computer science fields to improve these issues on CPU and GPUs. This section is oriented toward main subject: (1) Exploring

various solutions for concurrency memory issues [3] [4] [5]. (2) Describe the Berlekamp-Massey GPU implementation as a case study [6] [2] [1].

In a software transactional memory (STM) system, conflict detection is the problem of determining when two transactions cannot both safely commit. The authors in [3] presented study about date conflict detection and Validation problem to ensure that a transaction never views inconsistent data, which might potentially cause a doomed transaction to exhibit irreversible, externally visible side effects. Existing mechanisms for conflict detection vary greatly in their degree of speculation and their relative treatment of read-write and write-write conflicts. a lightweight heuristic mechanism "the global commit counter" was represented to reduce the cost of validation and single-threaded execution. The proposed heuristic also allows them to experiment with mixed invalidation, a more opportunistic interleaving of reading and writing transactions.

Another optimization techniques are represented in [4] to overcome the overhead of the global clock in STM. The first technique is Read-Write Lock Allocation (RWLA), which does not exploit any central data structure to maintain consistency of transactions. This method improves performance of STMs only if transactions commit successfully. However, in the event of frequent conflicts, RWLA increases cost of abort and degrades performance. The second optimization technique is an adaptive technique, which dynamically selects either baseline scheme or RWLA. The experimental shown that the adaptive technique is effective and is able to improve performance of transactional applications up to 66%.

Memory barriers or memory fence is used on CPU and GPU to fixed data races and deadlock problem [7] [8]. On CPU barriers are used when a multi-core machine must provide coherent access to a shared resource such as a work or task queue. but GPU architecture is different, The GPU is a many-core machine. Threads are grouped into a set of blocks, and each block is scheduled onto one of the streaming multiprocessors (SM) and run until completion. Many of the CPU implementations for barrier do not directly port to the GPU (they use instructions not available on the GPU), or do not scale well on the GPU because they are meant for different types of processors and different memory systems than that of the GPU. NVIDIA provides intra-block barriers in the form of highly efficient intrinsic (e.g. __syncthreads()). __ syncthreads responsible for waiting until all threads within the same block finished their jobs

On another side, CUDA implement terminology called *__threadfence() to solve* concurrency memory On a larger scale . *__threadfence() function* is responsible for ensuring that all writes to global memory that are made by the calling thread before the call to __threadfence() are observed by all threads in the device as occurring before all writes to global memory made by the calling thread after the call to *__threadfence*()[9]. Memory fence also has overhead that will be discussed in the next section.

GPUs have recently attracted the attention of many application developers as commodity data-parallel coprocessors. To increase performance of Berlekamp-Massey Algorithm as a case study GPU is used. As we mention above, BMA is a good representative of concurrency memory. There is a previous attempt to optimize BMA algorithm. Hanan Ali et al [10] represented bitwise CPU and GPU software implementation for BMA. They implemented code for CPU and GPU. The results showed that bitwise CPU code was faster than GPU code for input up to 2^{20} or 1 MBits. CPU code was used for first 1 MBits, and GPU code was used beyond 1 Mbits. The GPU code becomes faster than serial implementation in long sequences started from 4 Mbits.

III. METHODOLOGY

BMA GPU implementation was started from serial and parallel representation that proposed in [6]**.** As shown in fig.1the initial parallel of BMA depends on three main kernels Multiplication kernel line 4, addition kernel line 9, 14 and copy kernels line 13 , 15.

```
1   N=0;
2   d=S[0];
3   for(N=0; N< LenS; N++)
4       Multiply_Kernel <<<0>>>();
5       reduction_Kernel<<<0>>>();
6       if(d==0)
7          x++;
8       else if(d!=0 && (2L > N))
9           addition_Kernel<<<0>>>();
10          x++;
11      else if (d!=0 && (2L <= N))
12          L=N +1-L;
13          CopyC_Kernel<<<0>>>();
14          addition_Kernel<<<0>>>();
15          CopyB_Kernel<<<0>>>();
16          x=1;
```

Fig.1: initial parallel implementation of BMA

We are focus on reduce the kernel launch by combine the addition kernel line 14 with CopyC kernel line 13 in one kernel. As known in CUDA programming, after terminating the kernel all threads sure to finish writing in the memory before other threads start to read. However, the overhead of kernel launch led to bad performance. To overcome this problem the terminology of memory fence is used.

```
1  __global__ void addition_CopyC(){
2
3      int myId = blockIdx.x*blockDim.x + threadIdx.x;
4      if (myId >= numThread) return;
5
6      temp[myId] = C[myId];
7      __threadfence();
8      myLZ = B[myId];
9      if (shiftLZ) {
10         if(myId == 0) {
11         // shift right >>
12         myLZ >>= shiftLZ;
13         } else {
14             tmp = B[myId-1];
15             myLZ >>= shiftLZ;
16             tmp <<= 32 - shiftLZ;
17             myLZ |= tmp;
18         }
19     }
20     C[myId+offsetC] ^= myLZ;
21 }
```

Fig.2: addition and copy kernel

Fig.2 represents the new kernel that combine CopyC and addition kernel in BMA parallel implementation. CopyC kernel is responsible for reading all values from global array called C and addition kernel responsible for adding and modify on C. the main problem is making sure that all threads read from C before the other threads write or modify on it. Based upon the importance to use thread fence function appear. The role of __threadfence() is making all threads that need to modify on C as shown in line 20 to wait until all threads finish reading from C line 6.

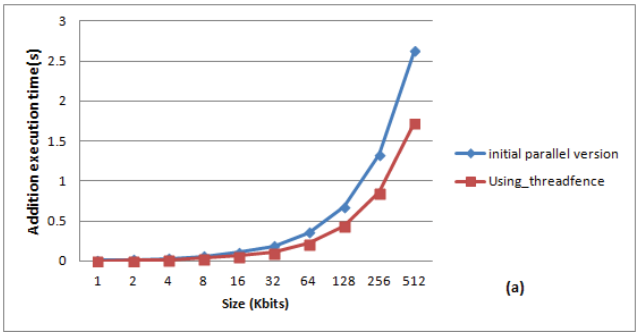
This optimization does not give a significant improvement in all lengths of inputs sequences. It favors in short sequences from 1 KBits up to 4 MBits; the results are presented in experimental results section.

IV. EXPERIMENTAL DESIGN

This section studies the effect of using memory fence on performance of BMA GPU implementation. CUDA and NVIDIA Tesla M2090 are used. Tesla M2090 has 16 streaming multiprocessor (SMP) with 32 cores in each SMP, for a total of 512 cores [11]. There are 6GB of RAM. The server that contains the M2090 device is a Liunx (CentOS) with 2.7 GHz Intel Xeon E5-2680 Processor, 128 GB RAM. NVIDIA has a programming guide for CUDA C [9]. The proposed experiments execute BMA using 512 threads per block. The number of active threads is equal to the number of elements processed at each iteration. The input lengths start from 1 KBits to 64 MBits. The improvement of the proposed algorithm is studied by measuring various metrics such as total execution time, addition time and relative performance improvement.

1. Memory fence optimization

This experiment evaluates the performance of BMA algorithm after combine CopyC kernel and addition Kernel using __threadfence(). The evaluation occurs in terms of additional time, total execution time and Relative performance improvement (speedup).



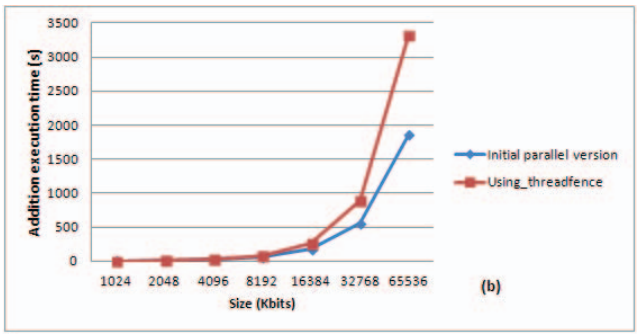
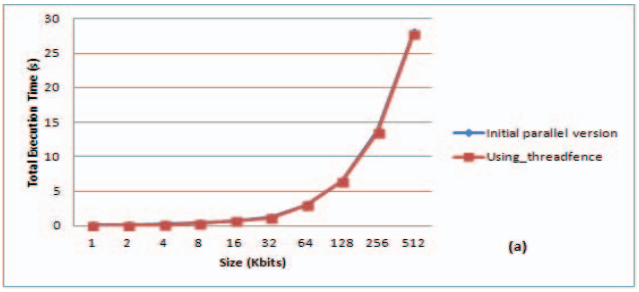


Figure 3: Addition execution time between initial parallel version and threadfence (a) input length from 1 to 512 Kbit (b) input length from 1 to 64 Mbit



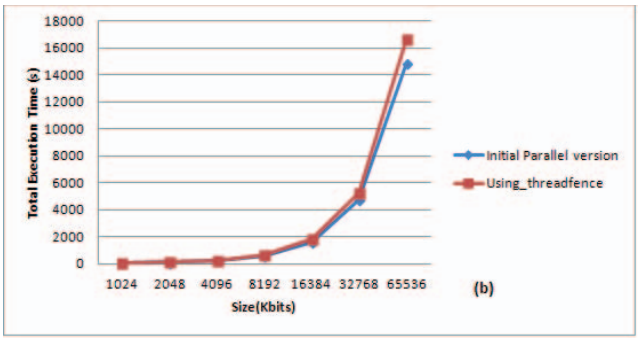


Figure 4: Total execution time between initial parallel version and threadfence optimization (a) input length from 1 to 512 Kbit (b) input length 1 to 64 Mbit

As shown in figure 3,4 the graphs compare the addition time and total execution time between initial parallel version of BMA and after using __threadfence() function.

Graph (a) in figure 3, 4 represents the improvement in small input sequence after using __threadfence() function (from 1 Kbit to 512 Kbit).

The reason for this, is reduce kernel launch time by blocking the calling thread until all threads write to the global memory or shared memory then allow calling thread to read which led to, avoid delay of kernel launch.

However, The *__threadfence()* overhead appears for long input sequence as shown in Figure 3,4 (b). This led to the inverse relationship between Relevant Performance improvement and length of input sequence; figure 5.

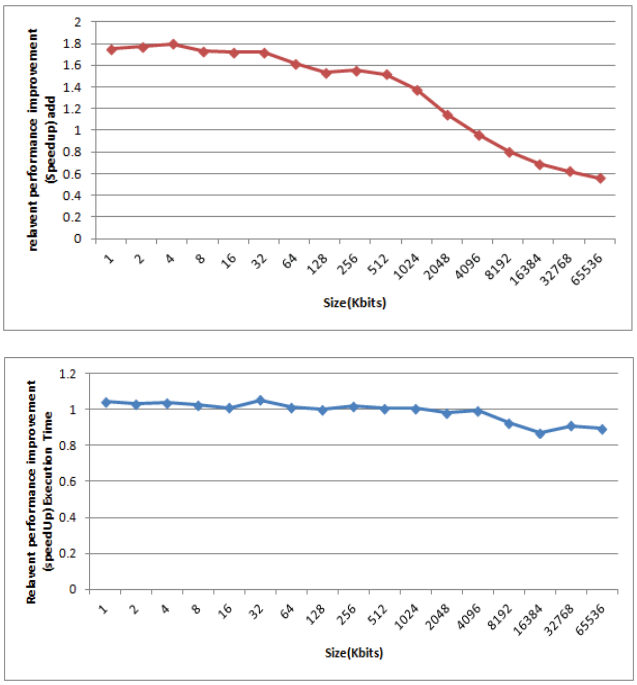


Figure 5: Relevant Performance improvement in addition and execution time (a) input length from 1 to 512 Kbit (b) input length 1 to 64 Mbit

We can conclude that using Memory fence achieves a good performance in applications that used a small amount of data to avoid its overhead.

V. CONCLUSION AND FUTURE WORK

GPUs have recently attracted the attention of many application developers as commodity data-parallel coprocessors. The newest generations of GPU architecture provide easier programmability and increased generality. However, GPU implementation still lacks some difficulties such as memory management and inters block synchronization.

In this paper, we have presented read-write memory conflict problems and used Berlekamp-Massey Algorithm as a case study. our study evaluated the performance of using Memory fence terminology in Nivida and CUDA programming. Memory fence used to solve read write memory conflict problem.

BMA was the best choice for large importance in the field of coding theory and it contains massive parallelism. The results shows that memory fence improve algorithm speed up in small input sequence. Because, it is reduce kernel launch time by blocking the calling thread until all threads read from the global memory or shared memory then allow calling thread to write which led to, avoid delay of kernel launch.

On another hand, memory fence has a big overhead that led to get bad performance in large input sequence.

In future work, we will discuss several optimizations to solve memory management problems with best performance by using inter- block synchronization and compare more than one technique.

REFERENCE

[1] Berlekamp, E.R, " Algebraic coding theory," in *: McGraw-Hill*, New York, 1968.

[2] J. Massey, "Shift-register synthesis and BCH decoding," vol. 15, no. 1, pp. 122 - 127, 1969.

[3] V. J. M. W. N. S. M. L. S. Michael F. Spear, "Conflict Detection and Validation Strategies for Software Transactional Memory," in *Distributed Computing*, Springer link, pp. 179-193.

[4] A.Ghanbari Bavarsad,E.Atoofian, "Read-Write Lock Allocation in Software Transactional Memory," in *International Conference on Parallel Processing,IEEE*, 2013.

[5] R. Farber, CUDA Application Design, Elsevier, 2011.

[6] H.Ali, M. Ouyang, A. Soliman,W. Sheta, "Parallelizing the Berlekamp-Massey Algorithm," in *the Second International Conference on Computing, Measurement, Control and Sensor Network (CMCSN)*, 2014.

[7] J. D. O. Jeff A. Stuart, "Efficient Synchronization Primitives for GPUs," in *ACM*, 2011.

[8] A. O. W. paper, "Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12 Update 2: Part 2, Memory Barriers and Memory Fences," Oracle , 2010.

[9] N. Corporation, "NVIDIA CUDA C programming guide," July, 2013.

[11] “Whitepaper NVIDIA’s Next Generation CUDATM Compute Architecture:Fermi,”[Online].Available: http://www.nvidia.com/content/pdf/fermi_white_papers/ nvidiafermicomputearchitecturewhitepaper.pdf.