Cluster Computing (2020) 23:1007–1024
https://doi.org/10.1007/s10586-019-02961-x

Exploring the parallel capabilities of GPU: Berlekamp-Massey algorithm case study

Hanan Ali[1] · Ghada M. Fathy[1] · Zeinab Fayez[1] · Walaa Sheta[1]

Received: 25 December 2018 / Revised: 15 May 2019 / Accepted: 17 July 2019 / Published online: 12 August 2019

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, there are several optimization techniques which are used to maximize the benefit of the GPU resources. This research exploits optimization techniques for CUDA enabled GPU architecture in order to achieve the best possible performance for Berlekamp-Massey Algorithm (BMA) as a case study. Berlekamp-Massey Algorithm (BMA) is one of the best solutions to find the shortest linear feedback shift register which is very important for several applications such as digital processing and cryptography. The experimental results show that the optimized BMA implementation is almost 160 × faster than non-bit CPU serial implementation, 7 × faster than bit serial implementation and 4 × faster than an initial parallel bit implementation.

Keywords Linear complexity · BerlekampMassey algorithm · Parallel computing · GPU · CUDA optimization techniques

1 Introduction

Graphics processing units (GPU) are now widely used to speed up compute intensive applications in different fields. CUDA platform is considered as one of the most important platforms used to program GPU devices. CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing. To improve any algorithm using GPU computational power, four phases are needed (APOD): Assess, Parallelize, Optimize, and Deploy [3]. The first phase is to assess the existing code (serial code) to identify which parts intensively consume the time and computations by analyzing the application with one or more real data sets. The second phase uses GPU programming languages to parallelize the algorithm. The third phase is optimizing the algorithm. CUDA platform contains several optimization techniques, such as: caulased memory access, reduce memory transfer and reduce kernel launch overhead. Finally, deploy phase takes the initial development all the way through to deployment.

This paper focuses on improving the performance of Berlekamp-Massey Algorithm (BMA) as a case study through applying the four APOD phases. Berlekamp-Massey Algorithm (BMA) finds the shortest linear feedback shift register which is used in several applications such as digital processing and cryptography. A linear feedback shift register (LFSR) is widely used to generate data encryption keys and random numbers. It is very useful in various applications, such as communication channels and cryptography. LFSRs and regular structure can thus be easily incorporated into digital circuits. LFSRs can be used to generate exhaustive binary sequences for circuit testing or as pseudo-random number generators. In 1967, Berlekamp designed an algorithm to decode Bose-Chaudhuri-Hocquenghem codes. Massey recognized its relationship to LFSRs and described a simplified version of the algorithm. Given an binary syndrome, the Berlekamp Massey

✉ Ghada M. Fathy
gfathy@srtacity.sci.eg

Hanan Ali
hali@srtacity.sci.eg

Zeinab Fayez
Zeinab.Fayez@gmail.com

Walaa Sheta
w.sheta@louisville.edu

[1] Informatic Research Institute, City for Scientific Research, Alexandria, Egypt

Algorithm (BMA) finds the shortest linear feedback shift register (LFSR) that generates the sequence. For a binary sequence whose length is n, BMA requires a time of $O(n^2)$.

Length of the LFSR can be calculated as the linear complexity (LC) of the syndrome [10] and used to solve Toeplitz systems [9]. Georg Schmidt [18] proposed an approach based on multi-sequence shift register synthesis, which makes it easy to understand and simple to implement. The computational complexity of this shift register based algorithm is of the same order as the complexity of the well known Berlekamp Massey algorithm.

In addition, BMA is one of the best choices in decoding Reed-Solomon codes [17, 20]. Reed-Solomon codes are powerful techniques for correcting errors introduced when a message is transmitted in a noisy environment. These codes are very popular and can be found in compact disc players and NASA satellites used for deep-space exploration [5]. The decoding in Reed-Solomon consists of four steps Syndrome Computation, Key Equation Solver (Berlekamp Massey Algorithm), Chien Search, and Forneys Formula [6].

Authors in [4] investigated the complexity of syndrome less decoding, and compare it to that of syndrome-based decoding. They measured the complexity of each stage on decoding process. The complexity of BMA is one of the most complex processes in decoding.

Moreover, the BMA is commonly used in CD and DVD decoding and error correction. The BMA algorithm decodes Bose-Chaudhuriid-Hocquenghem codes. In 2017, [19] illustrated a decoder implementation for high-rate generalized concatenated (GC) codes. The codes are suited for error correction in flash memories for high reliability data storage. The GC codes are constructed from inner extended binary Bose-Chaudhuri-Hocquenghem (BCH) codes and outer Reed-Solomon (RS) codes.

This complicated process led researchers to apply all their capabilities to improve BMA execution time. BMA can be implemented in either hardware or software. In 1998, Ralf Kotter [11] proposed a hardware-based parallel implementation for the BMA algorithm. An algorithm was proposed that had a regular structure and was suitable for VLSI implementation. Ralf [11] proposed an outline for implementation that was used as the main block γ copies of a modified one-dimensional Berlekamp Massey algorithm, where γ was the order of the first non-gap in the function space associated with the code.

In 1989, Henkel [9] represented BMA in matrix form. Henkels description was based on studying sub-Toeplitz matrices. The main purpose of this description was to clarify and understand the BMA properties more easily than a shift register synthesis. In [7] the authors proposed a software in lieu of a hardware-based RS decoder.

This is accomplished using the Berlekamp-Massey algorithm, implemented on a programmable DSP. This software-based RS decoder using Berlekamp-Massey is implemented on Motorolas MSC8101 Star-Core DSP.

To the best of our knowledge, only two research works have addressed an optimized BMA implementation to improve its execution time [1] [14].

Ali et al. [1] introduced a bitwise CPU implementation as well as a GPU implementation. They studied the optimization of a typical CPU implementation and developed a bitwise CPU implementation. In this implementation, bit operations are used instead of integer operations because the syndrome was stored in 32-bit integer tokens to reduce memory access and, hence, reduce the overall computational time. Additionally, a GPU implementation was developed using the CUDA and NVIDIA M2090 GPU device. The results showed that bitwise CPU code was faster than GPU code for syndrome lengths of up to 2^{22} (4 Mbits). In contrast, the GPU code was faster than the serial implementation for long sequences (those greater than or equal to 4 Mbits), as will be discussed later, in Sect. 5.

Hamidreza [14] used recent advances in SIMD and GPU technologies to achieve a high BMA algorithm performance. The author proposed that the bitwise code is almost 35 times faster than a typical implementation, and the GPU implementation is more than 124 times faster than the CPU implementation for random syndromes with lengths of 2^{24}. The proposed implementation was tested on a K40 NVIDIA GPU device. Moreover, CUDA concurrent kernel execution has been investigated using multiple concurrent streams to process multiple syndromes (with a fixed syndrome length of $(= 2^{24})$.The results showed a speed improvement 2.2 times faster than the GPU implementation when the number of streams was equal to 32. Another approach was studied as a parallel implementation on SIMD Instructions (SSE and AVX). It has been found that the CPU-bit-SSE-1D implementation is 3.1 $\times$ faster than the CPU-bit implementation for short syndrome lengths (2^{11} bits) but its speedup was smaller (1.2 $\times$) for longer syndromes. CPU multi-threading was also investigated [14] using OpenMP, where a speedup of 10 $\times$ was achieved relative to the CPU-bit implementation.

In contrast to [14] this paper attempts to improve the BMA performance by adding several optimization techniques using GPU such as applying Brent's theorem to adjust the best number of threads. Moreover, reduce the number of kernel launched. Finally evaluate different types of inter block synchronization to the algorithm (lock base and lock free synchronization).

The rest of the paper is organized as follows: Sect. 2 gives a brief explanation of BMA. Section 3 Assess the BMA algorithm for a good selection of optimization entry points. Section 4 illustrates the performance optimization

approaches used in GPU implementation. Section 5 shows experimental results. Section 6 compares the results from different optimized implementations as well as compares our results with literature results. Finally, Sect. 7 concludes this research and suggests future work.

2 The Berlekamp Massey algorithm (BMA)

2.1 Overview of the Berlekamp-Massey algorithm

The BMA finds the shortest linear feedback shift register (LFSR) for a given binary output sequence. The algorithm can also be used to find the minimal polynomial of a linearly recurrent sequence in an arbitrary field. The field requirement means that the BMA requires all non-zero elements to have a multiplicative inverse [2]. Reeds and Sloane offer an extension to handle a ring [13]. The algorithm can be mathematically presented as follows Let S_n be a periodic sequence over a field F. The linear complexity L of S_n is the smallest positive integer L such that there are constants coefficients C_0, $C_{L-1} \in F$ with

$$S_{n+L} = C_{L-1}S_{n+L-1} + +C_0S_n \tag{1}$$

In other words, the linear complexity of the sequence of bits is the length of the shortest LFSR that can produce that sequence, which can be found by the BMA. Berlekamp published his algorithm in 1968 [2], and it was followed by Massey publication in 1969 [13]. Massey used the linear feedback shift register (LFSR) to better understand the Berlekamp algorithm and simplify it. The next discrepancy d_n is calculated as follows:

$$d_n = S_n + \sum_{i=1}^{\ln} C_i S_{n-1} \tag{2}$$

The new error-locator polynomial (coefficient) is given by:

$$C_{n+1}(x) = C_n(x) - d_n \div d_m x^{n-m} C_m(x) \tag{3}$$

2.2 The BMA implementation

2.2.1 Serial implementation of BMA

Ali et al. [1] introduced the optimized bitwise serial implementation of BMA shown in algorithm 1 and proved its superiority to a typical serial implementation. Moreover, a GPU implementation was suggested that executed faster than the traditional serial implementation when the syndrome length was greater than 4 Mbits.

Algorithm 1: C serial implementation of BMA

```
1  B[0]= C[0]=1;
2  for (i = 1; i < lengthS; i + +) do
3      B[i]=C[i]=0;
4  lengthC= 0;
5  m=-1;
6  for (N = 0; N < lengthS; N + +) do
7      d = 0;
8      for (i = 0; i <= lengthC; i + +) do
9          d≙C[i]&S[N − i];
10     if (d) then
11         for (i = 0; i <= lengthC; i + +) do
12             tmp[i] = C[i];
13         for (i = 0; i <= lengthS − N + m; i + +) do
14             C[N + m + i]≙B[i];
15         if (lengthC <= N >> 1) then
16             for (i = 0; i <= lengthC; i + +) do
17                 B[i] = tmp[i];
18                 lengthC = N + 1 − lengthC;
19                 m = N;
```

The syndromes of the algorithm are stored in four integer arrays S, B, C, and tmp, where S contains the binary syndrome sequence of length S, C contains the coefficient values, and B holds the previous values of C. This algorithm can be divided into three main processes: the multiplication process is in lines (8,9), the addition process is in lines (13,14), and the copying process is in lines (11,12) and (16,17). The authors of [1] applied several optimizations to the serial implementation, such as accessing the S array in reverse order using a bitwise operation, and also implemented the BMA algorithm on a GPU. This work paid attention to parallelizing the most computationally intensive parts of the BMA algorithm, as mentioned in the previous section.

2.2.2 The parallel implementation

In this section, we develop the code for an optimized parallel GPU implementation.

The initial parallel implementation of BMA focused on the major time consuming kernels, as shown in Fig. 1: Kernel1_ Multiplication, Kernel2_reduction, Kernel3_Addition, kernel4_ copyC and kernel5_UpdateB.

As we can see in Fig. 1:

a. The CPU is responsible for memory allocation and controls the outer for loop (lines 6 to 19 in algorithm 1). Moreover, it responsible for checking the d value to determine which kernel should be launched.

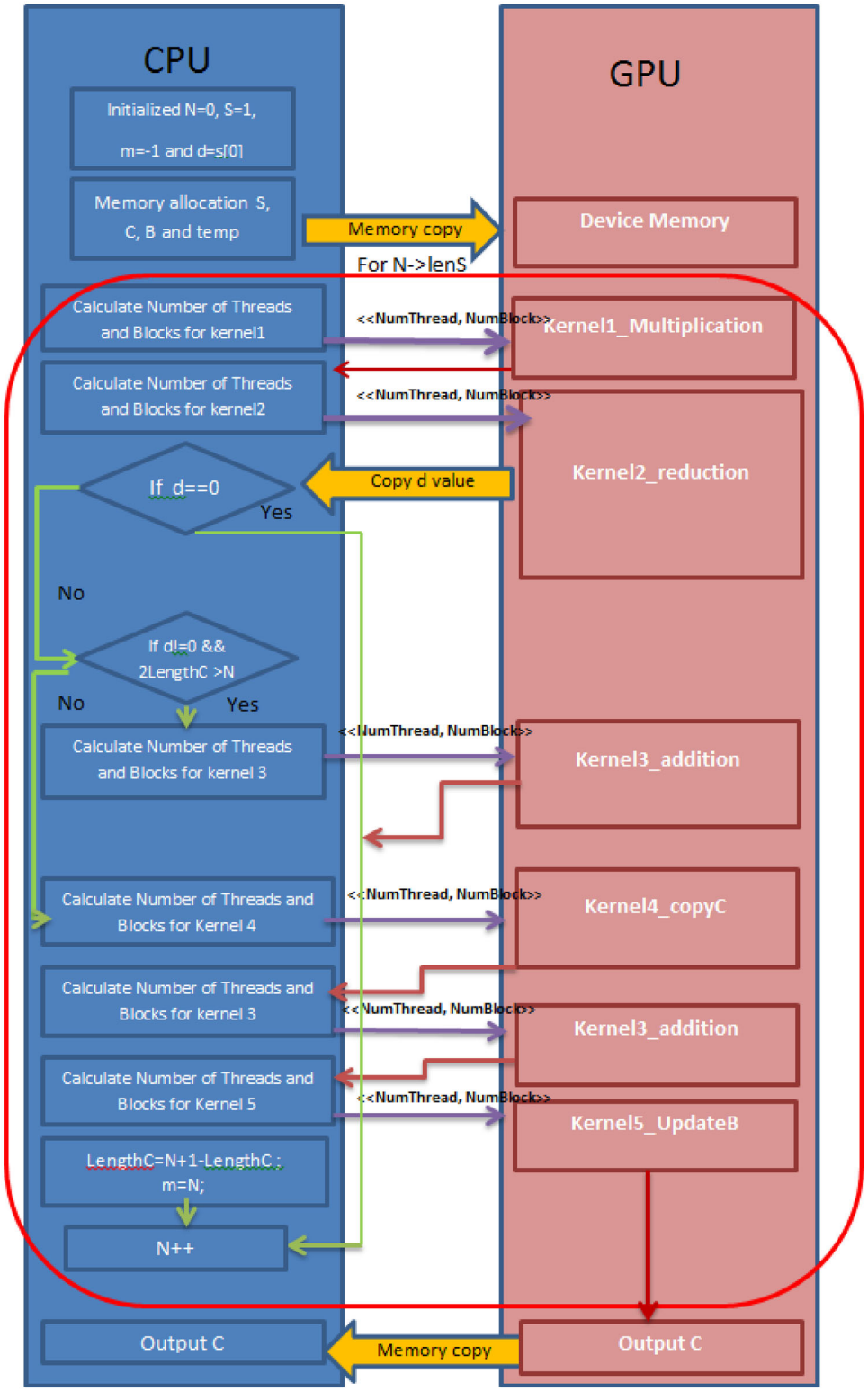


Fig. 1 Flow chart of the initial parallel implementation of BMA

b. The GPU is dedicated to executing the most computationally intensive parts of the algorithm, such as multiplication, as well as the addition operations in lines (8,9) and (13,14).

Figure 2 reveals how the serial code represented in Fig. 3 is parallelized on a GPU device using CUDA 7.5. Sections 1 and 2 perform data allocation and initialization

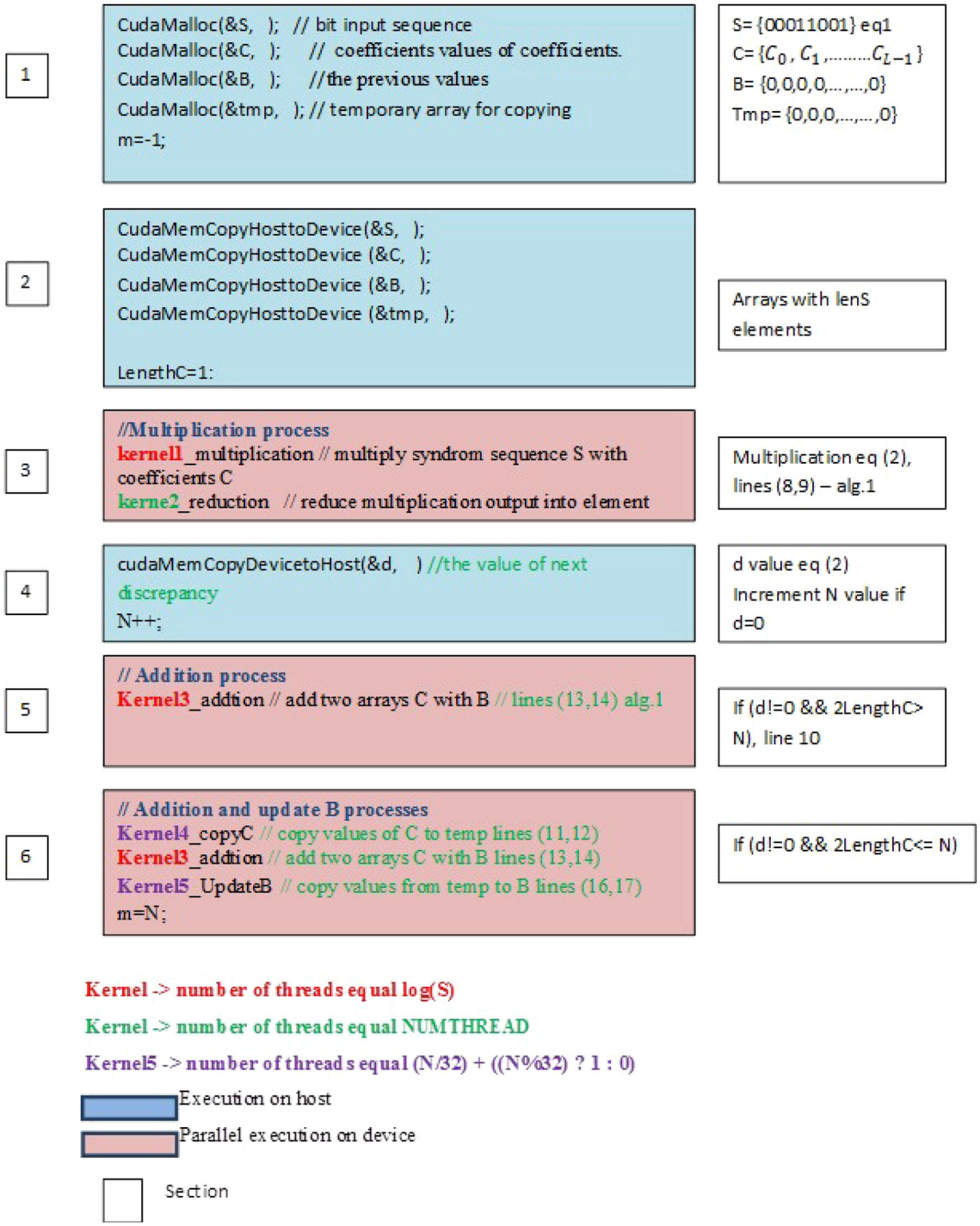


Fig. 2 Initial parallel CUDA implementation for BMA

on the device. Section 3 multiplies S and C using two kernels: kernel1_multiplication and kerne2 reduction.

The GPU kernel function kernel1_multiplication (Num_{thread}, Num_{block}) performs the multiplication of lines (8,9) algorithm 1 The number of thread Num_{thread} is equal to $log(lengthS)$ and the number of thread blocks is calculated as follows:

$$Numblock = (Numthread/NUMTHREAD) + (Numthread\%NUMTHREAD)\,?1 : 0)$$

where NUMTHREAD is the maximum number of active threads. Section 5 adds B to C when the condition

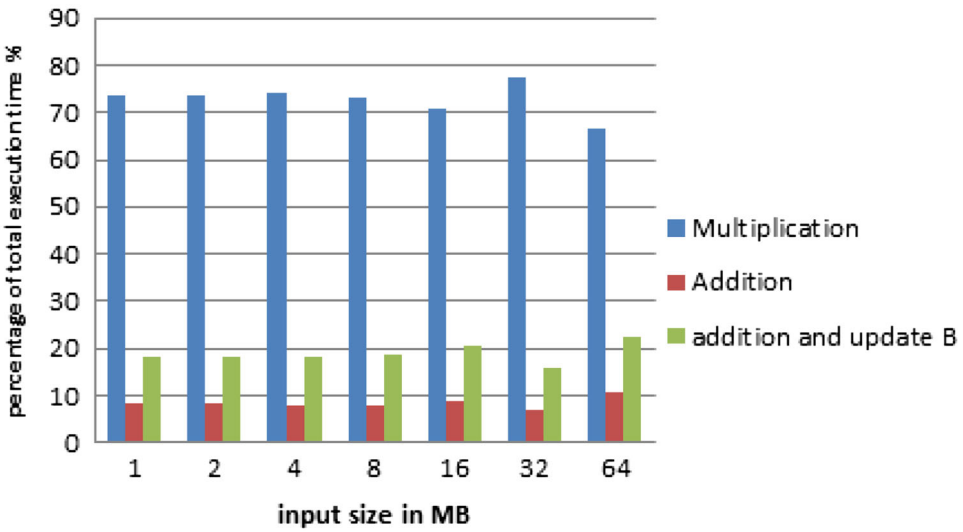


Fig. 3 Profile analysis of the main computation (Color figure online)

($d! = 0 \&\& L > N/2$) is true; this operation is implemented in kernel3_additon (Num_{thread}, Num_{block}). This kernel is responsible for adding the two arrays, C and B, and each thread adds one pair of elements. The number of threads Num_{thread} is equal to the number of elements processed in the iteration:

$$Num_{thread} = (N/32) + ((N\%32)?1 : 0).$$

$$Num_{block} = (Num_{thread} + NUMTHREAD1)/NUMTHREAD.$$

Section 6 is executed conditionally when the condition ($d = !0 and L <= N/2$) is true, and it consists of three main kernels: kernel4_ copyC (Num_{thread}, Num_{block}), kernel3_addition (as in Sect. 5, Fig. 2) and kernel5_updateB (Num_{thread}, Num_{block}). The process is executed as follows: first, a copy of C is stored in tmp, then B is added to C, and then B is updated to the old value of C that was stored in tmp.

3 Assessment

When studying/revisiting the BMA algorithm, we found that the multiplication process that calculates d in algorithm 1 lines (8 and 9), (Fig. 2 Kernel1_multiplication and kernel2_reduction) is repeatedly executed N times, where N is equal to syndrome length. On the other hand, addition and copy processes; Fig. 2. Kernel3_ addition, kernel4_-copc, kernel5_copyB are conditionally executed depending on the value of d. The complexity of multiplication is greater than that of addition; therefore, kernel1_multiplication as well as kernel2_reduction strongly recommended for optimization because they are the kernels most frequently launched, and have the highest complexity. The percentage of time consumed by each portion in the initial parallel version is presented in Fig. 3.

Profile analysis revealed that almost 70% of the total computation time was consumed in the multiplication process, while only 30% was spent in the addition process and in the addition with update B process. Our initial GPU implementation was executed several times for different syndrome lengths; then, the multiplication, addition, and copy kernel GPU times were plotted against the syndrome lengths. Based on this analysis, potential efforts should be directed towards optimizing the multiplication process; kernel1_multiplication and kernel2_reduction. The addition and copy kernels will also be optimized in a second optimization stage.

It is expected that speeding up the multiplication kernels will significantly improve the overall performance of BMA, in comparison to any speed improvements in the addition and copy kernels. In the next section, we present several approaches for tuning the performance of our initial GPU implementation.

4 Optimization/performance tuning of GPU initial implementation

4.1 Multiplication process optimization

4.1.1 Applying Brent's theorem

As mentioned above, multiplication is the most computationally intensive process, and it is also the most frequently executed.

Thus, the multiplication process is divided into two kernels: kernel 1_multiplication and kernel2_reduction to

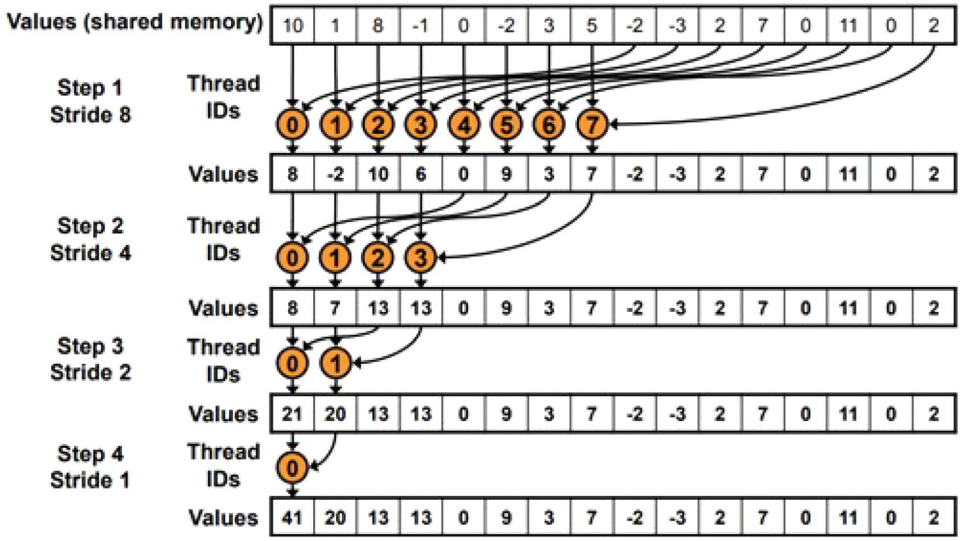


Fig. 4 Parallel reduction [8]

reduce number of global memory accesses. Based on Brent's theorem, which suggests that each thread can perform $O(logN)$ sequential work [12], each thread is allowed to process more than one element according to the following equation:

$$\begin{aligned} &\textit{Number of elements per thread} \\ &\quad = log(\textit{total number of elements}). \end{aligned}$$

This approach results in a large performance improvement, as presented in the experimental section.

4.1.2 Minimizing number of kernel launches

As discussed in Sect. 3, both kernel1_multiplication and kernel2_reduction are executed repeatedly N times, where N is the syndrome length. Therefore, merging both kernels into one kernel would limit the overhead of launching and help to minimize the overall time of the multiplication process. The multiplication process is responsible for multiplying C and S (Fig. 4, Table 1).

Each thread in kernel1_multiplication, multiplies one element of C by the corresponding element of S, i.e., resulting in a temporary data structure saved in shared memory. Subsequently, kernel2_reduction, is launched to sum all the elements of tmp into one value, d, such that $d = \sum_{i=1}^{lenghtC} tmp$. In Fig. 5 it is well known that launching a new kernel will implicitly force synchronization among the existing working thread blocks. This is the case when kernel2_reduction is launched after kernel1_multiplication.

Therefore, integrating those two kernels (kernel1_multiplication and kernel2_reduction) into only one kernel requires an inter-block synchronization mechanism to guarantee correct results.

Table 1 symbols are used in BMA implementations

Symbols	Explanation
S	Integer array of binary syndrome sequence
C	Integer array of binary coefficient
B	Integer array of the previous binary coefficient
tmp	temporary array to copy coefficient values
N	Is equal to syndrome length
L	Linear complexity of syndrome
d	The next discrepancy
Kernel1_Multiplication	GPU function to compute Multiplication process
Kernel2_reduction	GPU function reduce a collection of values to a single value
Kernel3_Addition	GPU function to add coefficient values from C to the previous coefficient B
Kernel4_copyC	GPU function used to copy coefficient Values from C to tmp array
Kernel5_updateB	GPU function used to copy coefficient values to tmp array and add C to B then update B with previous coefficient values from tmp

This optimization imposes a small modification to the end of kernel1_multiplication to guarantee that the reduction will not begin until all the blocks have finished their multiplication operations, and it is where synchronization is required before staring the reduction process. To start the reduction process, we need to recalculate the number of active blocks in the new stage, as follows:

$$\begin{aligned} activeBlocks = (&Num_{block} + NUMTHREAD \\ &- 1)/NUMTHREAD. \end{aligned} \quad (4)$$

merging kernel3_addition with Kernel4_copyC

```
1. → enter;
2. bid ←Block ID//The value of block ID in the grid
3. tid ←thread ID //The value of thread ID in the grid
4. Temp[tid]=C[tid] // copy values from C to temp array
5. __threadfence();
6. // adding computation between Temp and B[tid],
   Temp[tid]=Temp[tid]+B[tid]
7. // update C array with new value from Temp
8. C[tid+offset]=Temp ;
9. → exit
```

Fig. 5 Kernal3_addition with Kernel4_copyC

4.1.3 Inter-blocks synchronization

Many CPU implementations of barrier synchronization do not work on a GPU because of the synchronization cost. There are two levels of GPU synchronization: intra-block and inter-block synchronization. NVIDIA provides intra-block synchronization in the form of highly efficient intrinsic operations (e.g., syncthreads()) but does not provide a direct mechanism for inter-block synchronization [21]. Merging the multiplication and reduction kernels into one step reduces the need to launch of many kernels. However, the inter-block synchronization overhead must be considered. There are two approaches of inter block synchronization:

Lock-base and lock-free. Lock-base synchronization uses GPU atomic functions. An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. For example, atomicAdd() reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address.

The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads.

In other words, no other thread can access the atomicAdd() address until the operation is complete [15]. To obtain the best performance when using atomic functions and avoid deadlock, the maximum number of blocks is calculated. The Tesla M2090 has 16 SMP blocks [16], and 512 threads per block are used. The maximum number of resident blocks per SMP is three. Therefore, the block limit is Based on Eq. (5), the block limit is 48.

$$Max\ number\ of\ Blocks = (Number\ of\ SMPs\ number\ of\ resident\ blocks\ per\ SMP) \quad (5)$$

This arrangement avoids deadlocks resulting from waiting for other blocks to finish while no more resources are available to start new blocks, and results in a significant improvement in algorithm performance. We use another inter-block synchronization approach, the lock-free approach, to evaluate the performance [21].

The lock-free approach uses two arrays of synchronization variables and does not rely on costly atomic operations. The basic idea of the lock-free approach is to assign a synchronization variable to each thread block so that each block can record its synchronization status independently without competing for a single global mutex. However, the GPU lock-free performance does not yield a significant improvement compared to the lock-based approach. The justifications and results will be presented in experiment 1 in the Experiments and Results section.

4.2 Optimizing the addition process

Section 6, Fig. 2 shows a certain condition ($d! = 0$ && $2lengthC <= N$) implemented on the GPU and presented in algorithm 1, lines (9,10,12). In which the value of B is added to the current value of C, and then B is updated to the previous value of C (C's value before adding B). Figure 5 explains the integration of the two kernels: kernel3_addition and Kernel4_copyC. In GPU programming, after terminating the kernel, all the threads need to ensure that writing to memory is complete before other threads start reading from memory. However, the overhead of the kernel launch led to reduced performance. To overcome this issue, we applied the memory fence method [15] as presented in Fig. 5.

Kernel4_copyC is responsible for reading all the values from the global array, and kernel3_addition responsible for adding B to C.

The main problem is, we need to ensure that all the threads have completed reading from C before the other threads write or modify it.

Thus, the thread fence method (__threadfence ())prevents any modification to C, as shown in line 8, until all threads have finished reading from C in line 4. This optimization improves the algorithm performance on short sequences (from 1 to 4 Kbits) as presented in the experimental section.

Fig. 6 Experiment metrics

Metrics	Definition
Total Execution Time	The total time from starting BMA program to get C & L values
Addition Time	The time consumed by Kernel3_addition
Multiplication Time	Time consumed by both OF kerenel1_multiplication & kernel2_reduction
Relative performance improvement	The speedup that occur between initial parallel BMA implementation to the optimized version
Speedup	The improvement that occurs between a typical serial BMA implementation against optimized parallel BMA

5 Experimental results

5.1 Experimental configuration

The system used for the performance evaluations is equipped with 2 × Intel Xeon E5-2660 CPUs (2.2 GHz, 20 MB cache, 8.0 GT/s QPI, Turbo, 8c, 95 W) and 128 GB of DDR3memory (with ECC, DDR3-1600 + , with 100 + GB/s peak bandwidth). The operating system is a Linux (CentOS). The GPU is an NVIDIA Tesla M2090 with 16 streaming multiprocessors (SMPs) and 32 cores in each SMP, for a total of 512 cores, with 6 GB of RAM.

BMA was implemented in CUDA 7. The proposed experiments execute BMA using 512 threads per block.

The number of active blocks and active threads are calculated as discussed in the parallel BMA implementation section. The syndrome lengths were varied from 1 Kbit to 64 Mbits.

5.2 Experiments

The experiments evaluate the performance of the proposed parallel BMA using total execution time, speedup, and relative performance improvement, as shown in the table in Fig. 6.

5.2.1 Experiment 1

This experiment evaluates the first optimization on multiplication process that integrates the multiplication kernel with the reduction kernel using lock- based inter-block synchronization. Multiplication execution time, total execution time and relative performance improvement are measured. The program is executed using different binary sequence lengths, from 1 Kbit to 64 Mbits. Figure 7 shows the multiplication execution time after adopting the lock-based synchronization.

Using lock-based inter-block synchronization improved the performance by avoiding kernel launch overhead. The kernel launch overhead is the time needed to set up the kernel code for execution in the cores [15].

As shown in Figs. 7a and 8a, syndrome lengths from 1 Kbit to 32 Kbit do not result in a significant improvement because those lengths used only one thread block; thus, no synchronization is required. However, syndrome sequences from 64 Kbit to 64 Mbit use multiple thread blocks, so improvement was evident in the multiplication execution time and the total execution time after the two kernels (kernel1_multiplication and kernel2_reduction) were combined into one using lock-base inter -block synchronization as shown in Figs. 7b and 8b. Figure 9 shows the relative performance improvements in multiplication and total execution after the second optimization (compared to the initial GPU implementation).

5.2.2 Experiment 2

Experiment 2 evaluates the effect of using different inter-block synchronization approaches, including lock-free

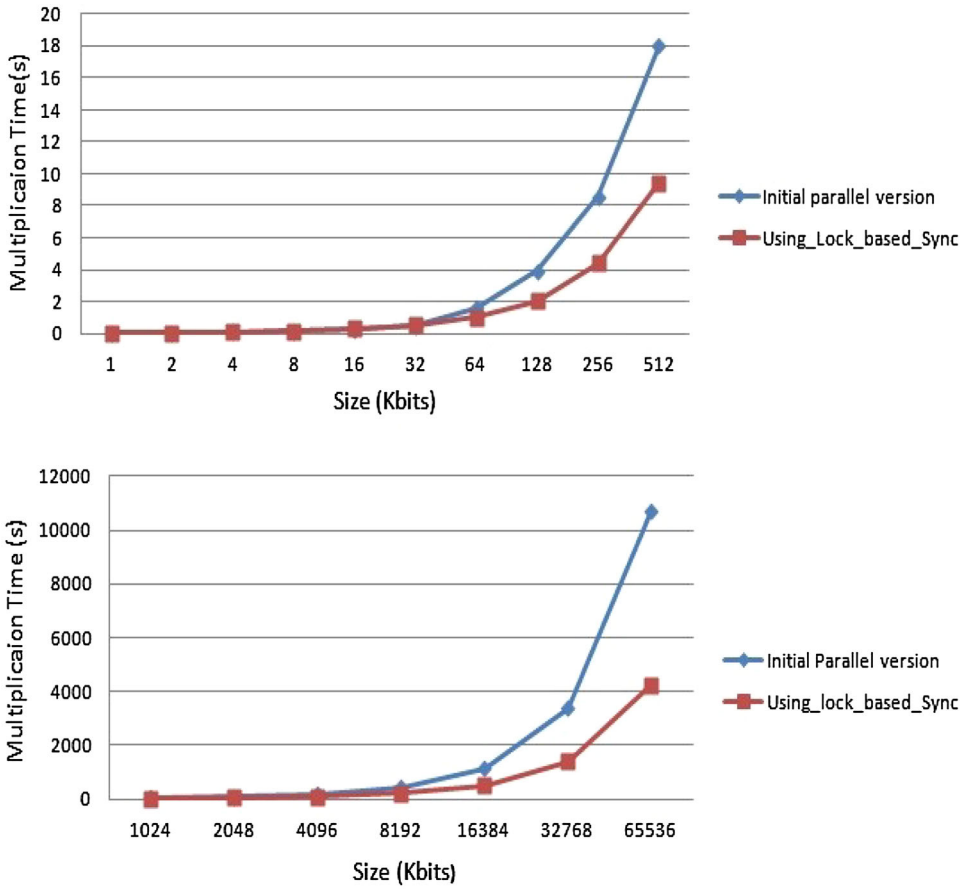


Fig. 7 Multiplication execution time using lock-based synchronization

synchronization. The synchronization overhead is measured by the synchronization ratio, which is the ratio of the time spent in inter-block synchronization to the total time consumed by the multiplication kernels, such that synchronization ratio + computation ratio = 1. Figure 10 depicts the synchronization ratio using two synchronization approaches: lock-based and lock-free. As expected, the synchronization overhead with short sequences in both approaches is 0% because for short sequences, execution is performed using only one thread block. Longer syndrome sequences use multiple blocks; hence, the synchronization ratio increases.

The synchronization ratio decreases as the computation ratio increases. The performance of atomic operations is up to 20 faster in Fermi compared to the GT200 generation [16]. Thus, using the new generation of atomic functions with the improved CUDA capabilities leads to less synchronization overhead in lock-based than in lock- free inter-block synchronization, as opposed to the results of [21]. Figure 11 shows the total execution time using the two types of inter-block synchronization. Lock-based and lock-free inter-block synchronizations achieve the same performance after using Fermi atomic functions and obtain better performance for longer syndrome lengths.

5.2.3 Experiment 4

The goal of this experiment is to evaluate the performance after adopting the threadfence () function. One kernel is used to integrate both kernel3_addition and kernel4_copyC instead of launching two separate kernels. The evaluation is based on the time required for _3 addition, with regard to total execution time and relative performance improvement (speedup). The results in Figs. 12a and 13a show that using the __threadfence () function achieves good performance on short syndrome lengths (from 1 Kbit to 4 Mbits) because it reduces the kernel launch time by blocking the calling thread until all threads writing to global or shared memory have completed before allowing the calling thread to read, which avoids delaying the kernel launch. The __threadfence() overhead appears with long syndrome lengths, as shown in Figs. 12b and 13b.

Based on the results in Figs. 12, 13 and 14 shows the a speedup comparison between the initial parallel version and the version that uses the threadfence() function.

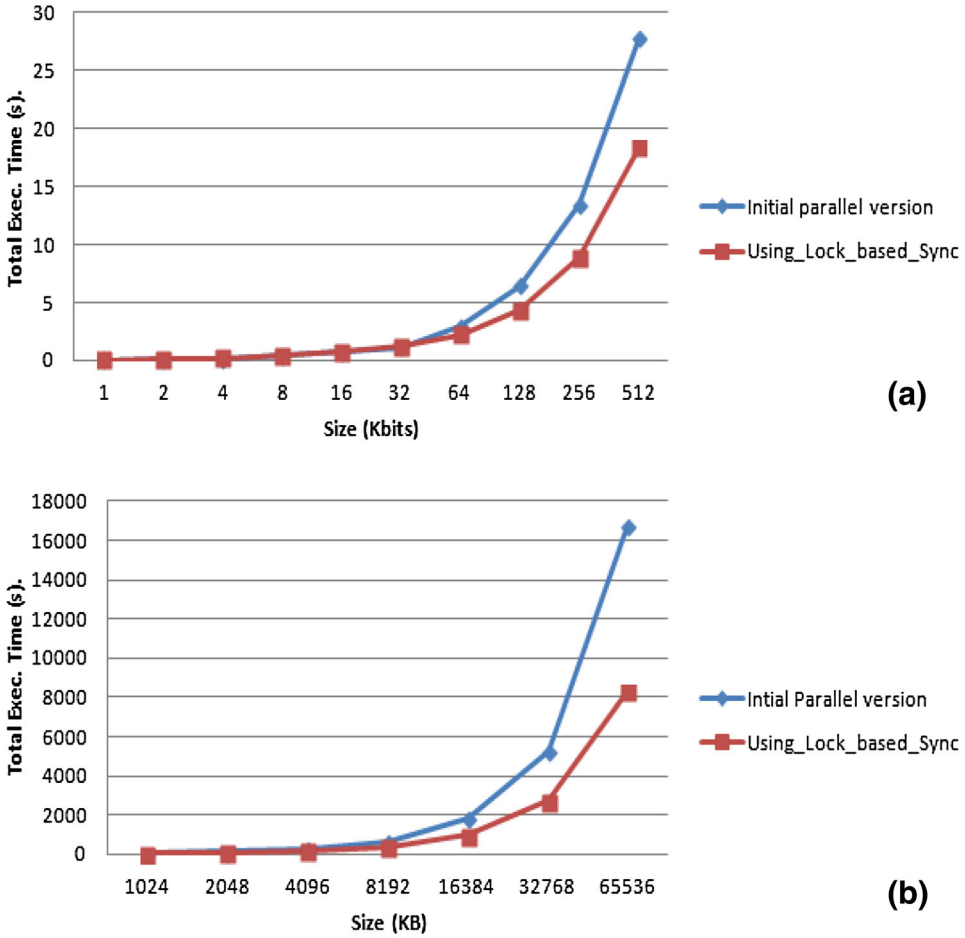


Fig. 8 Total execution time using lock-based synchronization

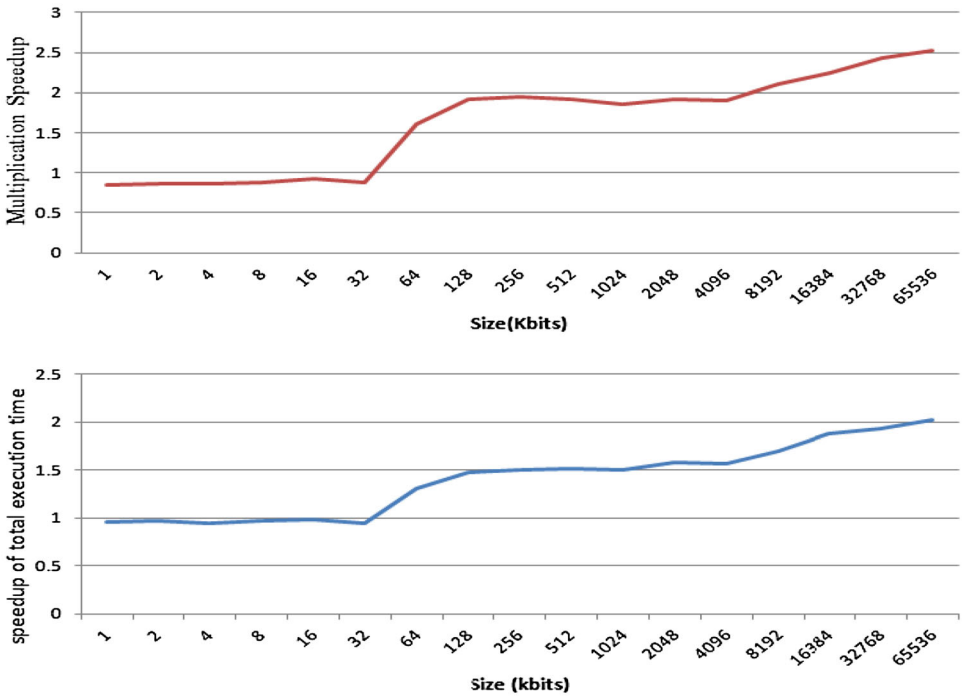


Fig. 9 Performance improvement after optimizing multiplication kernel

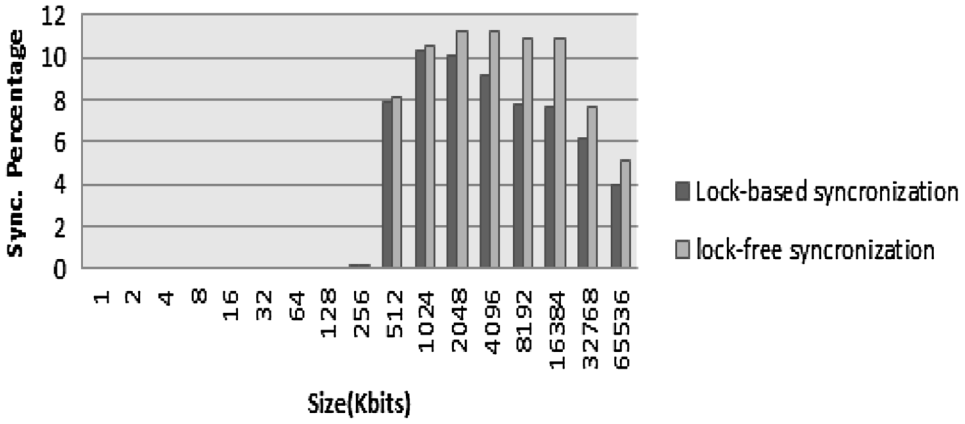


Fig. 10 lock base and lock free synchronization ratios

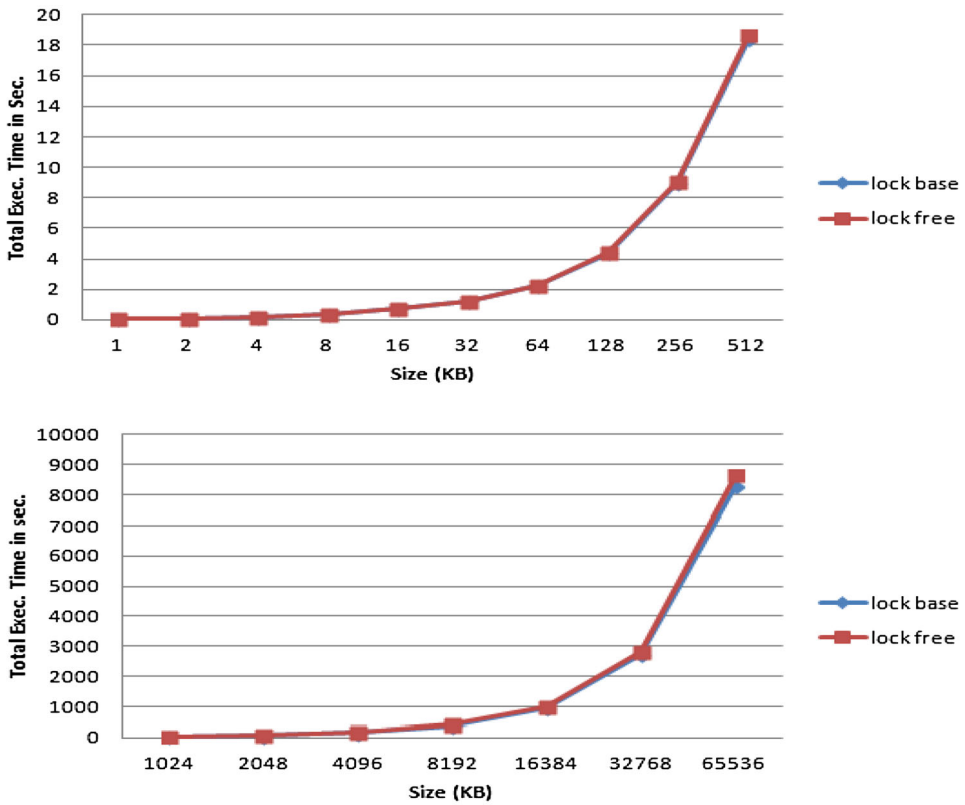


Fig. 11 Total execution time using different inter block synchronization

6 Discussion

This paper presents an optimized implementation of BMA developed to improve the total execution time by capitalizing on GPU resources and CUDA features. Various optimization approaches were attempted and evaluated through several experiments.

The graph in Fig. 15, illustrates the total execution time of different approaches for optimizing the BMA algorithm. The testing syndrome sequences used in the evaluation varied from 1 Kbit to 16 Mbits. Clearly, the algorithm performance improvement started only with sequences of 16 Kbits: all versions result in almost the same total execution times with smaller syndrome lengths. This is because a small sequence uses only one block, while longer sequences use larger numbers of blocks. However, larger block numbers lead to increased inter-block synchronization overhead; thus, the synchronization time ratio should be minimized when performing much computation. using the threadfence() function is one solution for solving

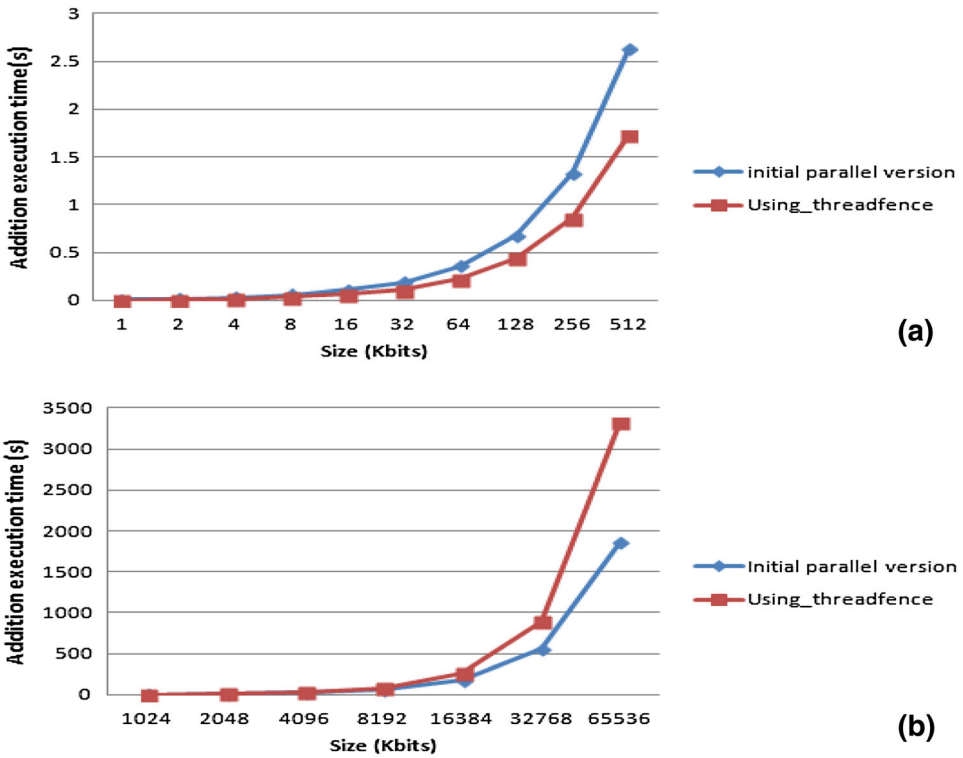


Fig. 12 kernel3 addition execution time for optimization using threadfence a) short sequences b) long sequences

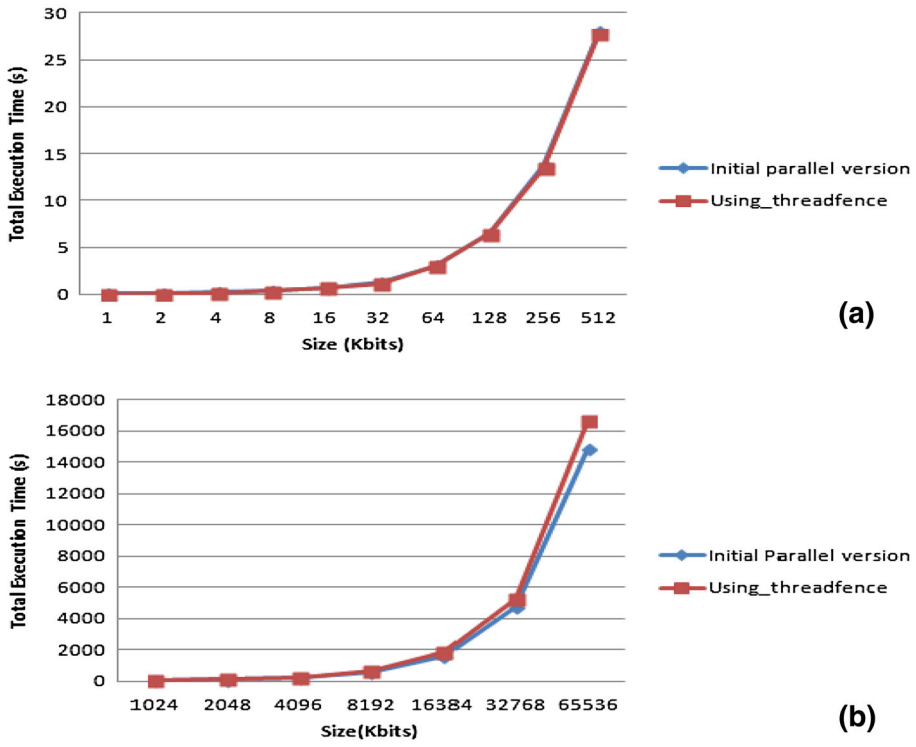


Fig. 13 Total execution time of BMA after using threadfence **a** short sequences, **b** long sequences

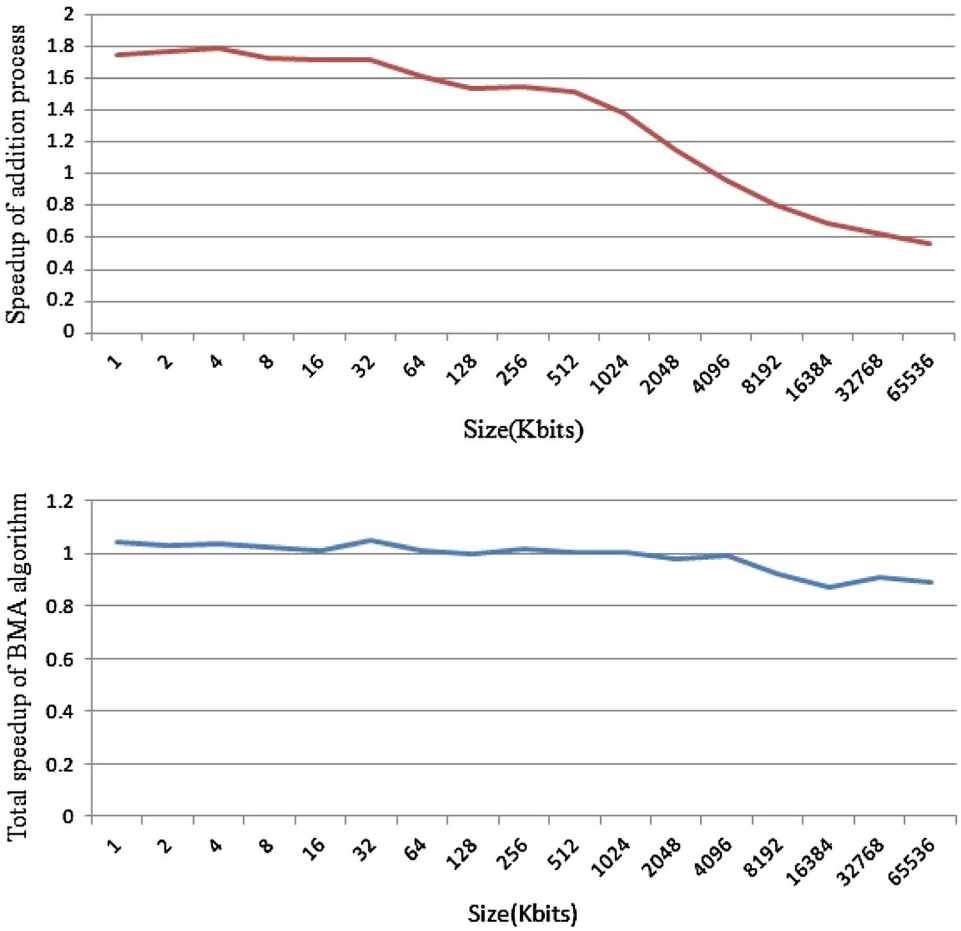


Fig. 14 Performance improvement after optimizing the addition kernel

problems involving memory race conditions. According to the graph in Fig. 15, when using the threadfence(), good performance was achieved on small sequences, (see Fig. 15a). Nevertheless, threadfence() function overhead became evident with large sequences.

Optimizations using lock-based and lock-free synchronization using inter- block synchronization avoid the kernel launch overhead. As evident in the graph, under both techniques, the performance on small sequences does not improve. Lock-based synchronization achieves the best performance because of the use of a new generation of atomic functions. To the best of our knowledge, little prior research exists that has attempted to speed up the BMA algorithm via GPU parallelization [1, 14]. The approaches in those two existing studies are discussed in detail in Sect. 1; however, we provide a quick comparison between their results and ours in 16, which shows a table of execution times for CPU integer, CPU bitwise, and the initial GPU version implemented in [1] compared with our final optimized version using lock-based synchronization. The results from [14] are not implemented on the same hardware platform as ours and are not explicitly mentioned as in [1]. Thus, only the final speed improvements gained by using a GPU could be compared.

Figure 16 shows the speedups of the two parallel implementations of BMA. The speedup of both parallel implementations (Initial GPU version and the version using lock-based synchronization) were acquired by dividing the computational time of serial cpuInt implementations of BMA [1] by the computational time of the parallel versions. The cpuInt version is a C-language implementation of the algorithm in Fig. 17.

Finally, Fig. 18 shows the speedup among several parallel implementations and the serial cpubit implementation proposed in [1, 14]. Speedup is obtained only from syndrome sequences of 4 Mbits or longer. Using lock-based and lock- free synchronization results in the largest improvements. For example, for a syndrome length of 64 Mbits, this version achieves a speedup of up to six times that of the sequential version.

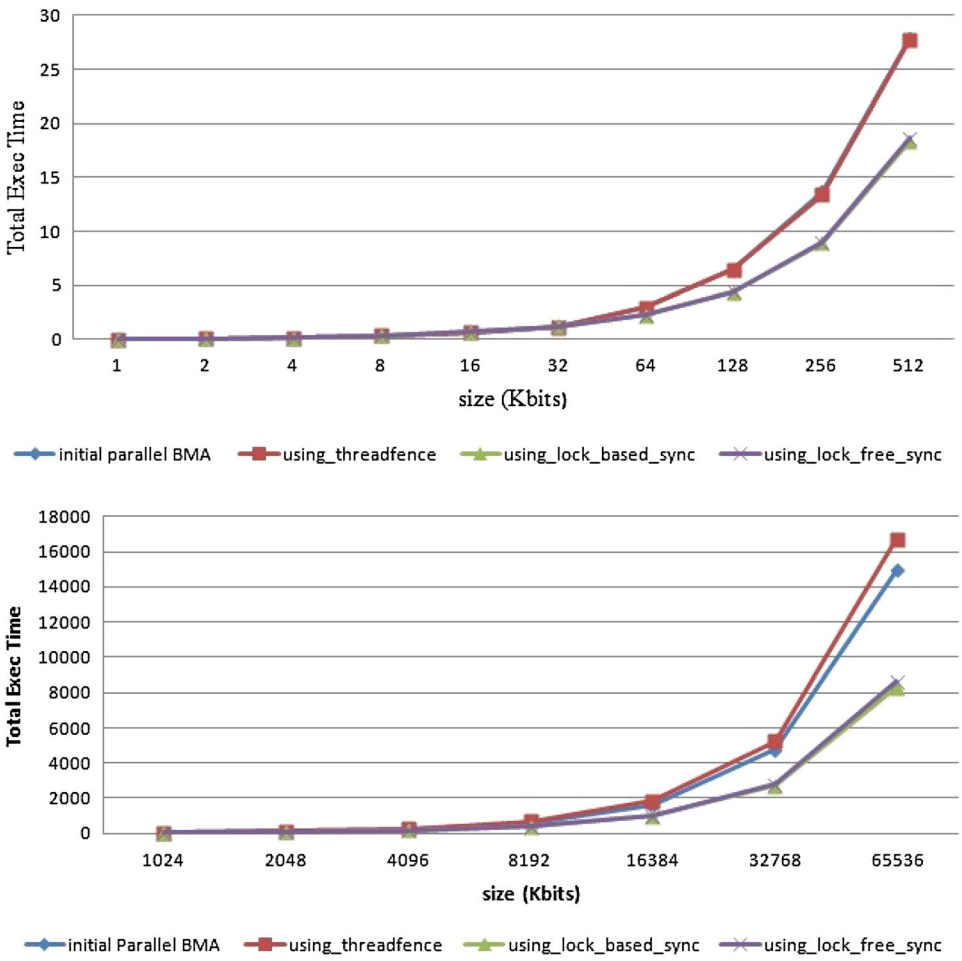


Fig. 15 Total execution Time **a** for syndrome size 1–512 Kbits **b** for syndrome size 1–64 Mbits

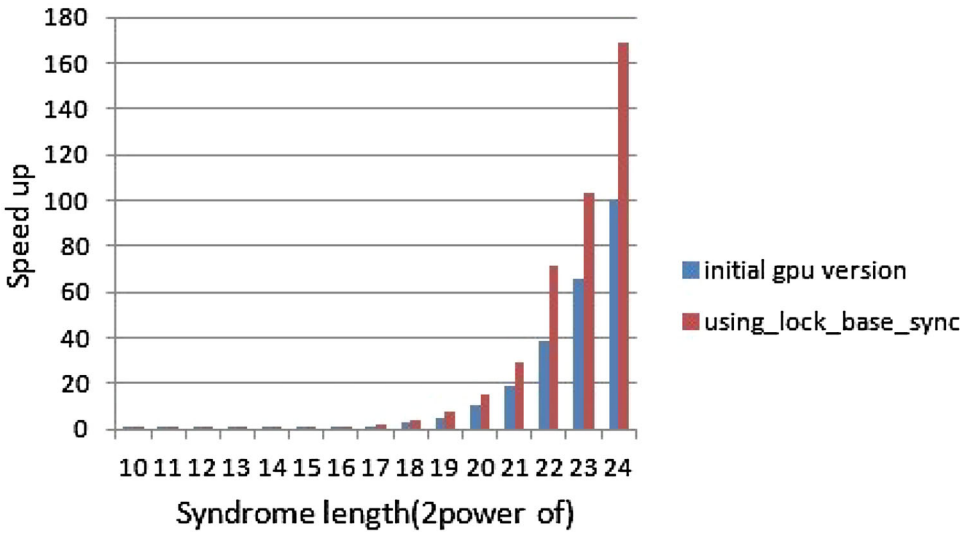
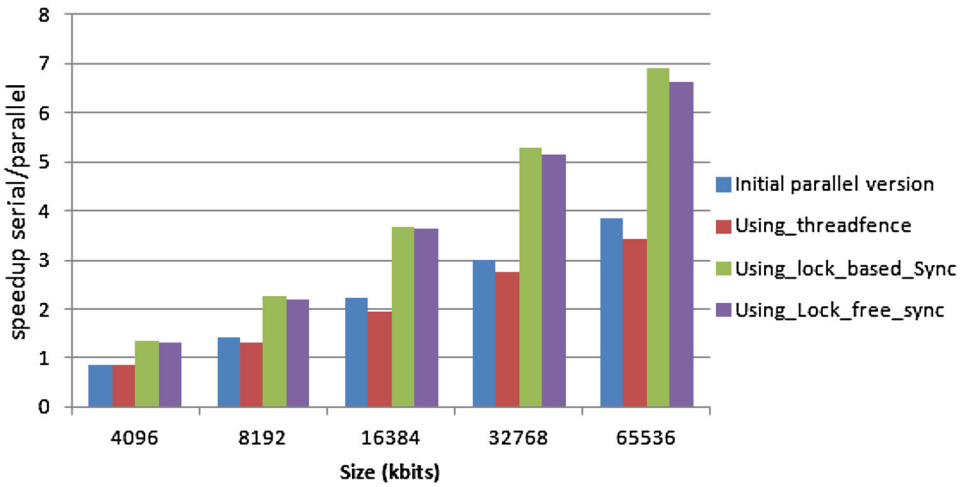


Fig. 16 Speedup comparison between initial gpu versions with the optimized version (Color figure online)

Fig. 17 Comparison of our result with previous results from literature

Input lengh(n)	Execution time	(seconds)			
	CpuInte [4]	CPU bitwise [4]	Initial_gpu_ Implementation	Optmized GPU implemntation (lock_based_sync)
2^{10}	0.002265	0.000151	0.046333	0.033333
2^{11}	0.005126	0.000466	0.092	0.039765
2^{12}	0.02926	0.00154	0.184	0.196667
2^{13}	0.07254	0.00558	0.370667	0.391667
2^{14}	0.22568	0.00806	0.687333	0.604
2^{15}	0.5265	0.0195	1.137667	1.284333
2^{16}	2.1721	0.0749	2.94	2.306333
2^{17}	9.021	0.291	6.448667	4.631
2^{18}	37.44	1.17	13.43833	9.395667
2^{19}	150.4	4.7	27.75233	20.39267
2^{20}	579.7	18.7	56.57967	38.78867
2^{21}	2409.6	75.3	125.5207	81.80433
2^{22}	10240	320	262.814	143.3151
2^{23}	43989	1333	666.7553	427.278
2^{24}	184756	5434	1832.519	1091.205

Fig. 18 Speedup of different parallel implementations (Color figure online)



7 Conclusion

In this study, several optimization approaches are applied to the BMA algorithm to achieve high performance using recent advances in GPU technology. First, we optimized the multiplication process by using two different types of inter-block synchronizations (lock-based and lock- free) to integrate two device kernels (multiplication and reduction).

This optimization is almost 160 times faster than the serial CPU implementation and 7 times faster than the serial bit implementation. The second optimization manages the total number of active threads by using log(n) elements for each thread. Finally, adopting the threadfence() function helped in integrating the two kernels that have memory race conditions and achieves good performance on short syndrome sequences, as shown in the experimental results. The proposed implementation achieves the best performance compared with the previous work on BMA using the same architecture.

References

1. Ali, H., Ouyang, M., Soliman, A., Sheta, W.: Parallelizing the Berlekamp-Massey algorithm. Int. J. Comput. Sci. Inf. Secur. **13**(11), 42 (2015)
2. Berlekamp, E.: Algebraic Coding Theory. World Scientific Publishing, Singapore (2015)
3. Bradley, T.: Assess, parallelize, optimize, deploy. https://devblogs.nvidia.com/assess-parallelize-optimize-deploy/ (2012)
4. Chen, N., Yan, Z.: Complexity analysis of Reed-Solomon decoding over GF (2 m) without using syndromes. EURASIP J. Wirel. Commun. Netw. **2008**(1), 843634 (2008)

5. Didier, F.: Efficient erasure decoding of Reed-Solomon codes. http://arXiv.org/arXiv:0901.1886 (2009)
6. Elsaid, H.A.E.A.: Design and Implementation of Reed-Solomon Decoder Using Decomposed Inversion less Berlekamp-Massey Algorithm. Faculty of Engineering, Cairo University, Giza (2010)
7. Greenberg, S., Feldblum, N., Melamed, G.: Implementation of the Berlekamp-Massey algorithm using a DSP. In: Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems. ICECS, pp. 358–361. IEEE (2004)
8. Harris, M.: Optimizing CUDA. SC07: High performance computing with CUDA (2007)
9. Henkel, W.: Another description of the Berlekamp-Massey algorithm. IEE Proc. I-Commun. Speech Vision **136**(3), 197–200 (1989)
10. Katz, J., Shacham, H.: Advances in cryptology–CRYPTO 2017. In: Proceedings of the 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, vol. 10401, 20–24 Aug 2017, Springer (2017)
11. Kotter, R.: A fast parallel implementation of a Berlekamp-Massey algorithm for algebraic-geometric codes. IEEE Trans. Inf. Theory **44**(4), 1353–1368 (1998)
12. Mark, H.: Optimizing parallel reduction in CUDA. NVIDIA CUDA SDK **2**, 15 (2008)
13. Massey, J.: Shift-register synthesis and bch decoding. IEEE Trans. Inf. Theory **15**(1), 122–127 (1969)
14. Mohebbi, H.: Parallel SIMD CPU and GPU implementations of Berlekamp–Massey algorithm and its error correction application. Int. J. Parallel Program. **47**(1), 137–160 (2018)
15. Nvidia, C.: Programming guide (2010)
16. Nvidia, W.: Whitepaper NVIDIAS next generation CUDA compute architecture. ReVision, pp. 1–22 (2009)
17. Sarwate, D.V., Shanbhag, N.R.: High-speed architectures for Reed-Solomon decoders. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **9**(5), 641–655 (2001)
18. Schmidt, G., Sidorenko, V.R., Bossert, M.: Syndrome decoding of Reed–Solomon codes beyond half the minimum distance based on shift-register synthesis. IEEE Trans. Inf. Theory **56**(10), 5245–5252 (2010)
19. Spinner, J., Freudenberger, J.: A decoder with soft decoding capability for high-rate generalized concatenated codes with applications in non-volatile flash memories. In: Proceedings of the 30th Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 185–190. IEEE (2017)
20. Tilavat, V., Shukla, Y.: Simplification of procedure for decoding reed-solomon codes using various algorithms: an introductory survey. Int. J. Eng. Dev. Res. **2**(1) (2014)
21. Xiao, S., Feng, W.C.: Inter-block GPU communication via fast barrier synchronization. In: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–12. IEEE (2010)

Hanan Ali is an Associate Professor of Computer and Systems, IRI Research Institute of the City of Scientific Research and Technology Applications (SRTA-City). She finished her Ph.D. in Distributed Virtual Environments at the University of Alexandria, Faculty of Engineering in 2008 where she also received her M.Sc. degree in Neural Networks in 1998 and her Bsc degree in 1994. She published in ISI journals and supervised some M.Sc. students from the Faculty of Engineering. She is interested in mobile cloud computing, GPU programming, HPC computing, virtual reality and modeling and simulation. She is currently the executive manager of the Cloud Center of Excellence in Alexandria."

Ghada M. Fathy is an Assistant Researcher at IRI Research institute, of the City of Scientific Research and Technology Applications (SRTA-City). She graduated from Faculty of Computer and Information Cairo University in 2010. She received M.S. degrees in Virtual Reality and distributed system from Faculty of Computer and information, in 2015. Her research interests include High performance computing, Computer graphics, GPU programming and Distributed system. She is currently a member in the Cloud Center of Excellence in Alexandria."

Zeinab Fayez is Researcher Assistant at IRI Research institute, of the City of Scientific Research and Technology Applications (SRTA-City). She graduated from Faculty of Computer and Information Cairo University in 2010. She is M.S. Student. Her research interests include High performance computing, Computer graphics, GPU programming and Distributed system."

Walaa Sheta is a Professor of Computer and Informatics, Informatics Research Institute of the City of Scientific Research and Technology Applications (SRTA-City) where he served as a faculty since 2001. He completed his Ph.D. at the University of Alexandria in a frame of channel system with University of Manchester, UK in 2001. He received his M.Sc. degree (1993) in the Information Technology and B.Sc. degree (1989) in Science from the University of Alexandria. He received his M.Sc. degree (1993) in the Information Technology and B.Sc. degree (1989) in Science from the University of Alexandria. He held a Visiting Professor position at the University of Louisville, USA and University of Salford, UK. He advised approximately 25 masters and doctoral graduates. His research contribution and consulting spans the areas of computer graphics, 3D image processing, artificial intelligence applications and simulation and modeling. Professor Sheta co-chaired several international conferences sponsored by IEEE and, with his research group, published in peer-reviewed journals. He participated and led several national and multinational projects funded by NSF, European Commission (FP-7) and STDF.