Memory Optimized Dynamic Matrix Chain Multiplication Using Shared Memory in GPU

Girish Biswas(✉) and Nandini Mukherjee

Department of Computer Science and Engineering, Jadavpur University, Kolkata, IN, India
girishbiswas@gmail.com, nmukherjee@cse.jdvu.ac.in

Abstract. Number of multiplications needed for Matrix Chain Multiplication of n matrices depends not only on the dimensions but also on the order to multiply the chain. The problem is to find the optimal order of multiplication. Dynamic programming takes $O\left(n^3\right)$ time, along with $O\left(n^2\right)$ space in memory for solving this problem. Now-a-days, Graphics Processing Unit (GPU) is very useful to the developers for parallel programming using CUDA computing architecture. The main contribution of this paper is to recommend a new memory optimized technique to solve the Matrix Chain Multiplication problem in parallel using GPU, mapping diagonals of calculation tables m[][] and s[][] into a *single combined calculation table* of size $O\left(n^2\right)$ for better memory coalescing in the device. Besides optimizing the memory requirement, a versatile technique of *utilizing Shared Memory* in Blocks of threads is suggested to minimize time for accessing dimensions of matrices in GPU. Our experiment shows best ever Speedup as compared to sequential CPU implementation, run on large problem size.

Keywords: GPU · CUDA · Matrix chain · Memory mapping · Dynamic programming · Memory optimized technique

1 Introduction

Graphics Processing Unit (GPU) is a common architecture in today's machines that can provide high level of performance in graphical platform using many-core processors. Modern GPU offers the developers to use all cores of processors simultaneously to parallelize the general purpose computing. Many studies [2, 4, 6, 8, 9] have been carried out till date to implement parallel algorithms in CUDA for general computational problems. There are some Streaming Multiprocessors (SM) in a GPU device and each SM comprises of many cores (Fig. 1) which may be allocated to threads in parallel. The whole computation in the device is done over a Grid of some Blocks, where each Block is constituted of some number of threads. NVIDIA GPUs provide the parallel programming architecture, called CUDA (Compute Unified Device architecture) [5].

Using GPU architecture for solving the optimization problems with large number of combinations is challenging due to limited memory in the device and minimum dependency between different threads. The problem of Matrix Chain Multiplication arises in

© Springer Nature Switzerland AG 2021
D. Goswami and T. A. Hoang (Eds.): ICDCIT 2021, LNCS 12582, pp. 160–172, 2021.
https://doi.org/10.1007/978-3-030-65621-8_10

many real time applications such as image processing, modern physics and modelling etc. This optimization problem needs to find the optimal order of multiplication of the matrices.

Dynamic programming approach may be applied to find the optimal solution using GPU [2] with diagonal mapped matrices of calculation for assuring coalesced memory access. This approach uses two 2D calculation tables $m[][]$ and $s[][]$, each of n rows and n columns. So, $O(2 \cdot n^2)$ size memory space is required to be allocated in GPU device, which is made of limited space.

The main contribution of our paper is to suggest an efficient way of using only a single combined calculation table of size $O(n^2)$, to be allocated in the device, mapping diagonals of both of $m[][]$ and $s[][]$ into it (Fig. 3). Our memory optimized approach maintains memory coalescing and shows better results choosing proper Block-size in GPU (Sect. 4.B) for the varying number of elements in the diagonals of the tables. Also, a simple trick is taken in our study to use Shared Memory to store only the required dimensions of matrices (Fig. 4) in Blocks of threads executing in parallel in GPU to reduce the access time for accessing dimensions of matrices to enrich the Speedup even more, compared with sequential implementation run in CPU for large datasets. This paper presents the most effective technique with respect to both memory requirements and performance.

The paper is organized as follows, Sect. 2 illustrates the idea of CUDA programming architecture in GPU device. Section 3 provides the Dynamic Programming Techniques to solve the Matrix Chain Multiplication Problem in GPU. Section 4 discusses about our Proposed Approach. Section 5 discusses about the results. Section 6 concludes the paper.

1.1 Previous Works

As per our knowledge, very few studies have been made on Matrix Chain Multiplication optimization problem until now for parallelizing the problem especially through GPU. In 2011 Kazufumi Nishida, Yasuaki Ito and Koji Nakano [2] proposed an efficient technique of memory mapping to ensure coalesced memory access for accelerating the dynamic programming for the Matrix Chain Multiplication in GPU. The diagonals of $m[][]$ and $s[][]$ were mapped into rows of 2D arrays in a manner that all elements of a diagonal are consecutive in nature. They have passed all the diagonals of m and s tables to GPU one by one for computation of all elements of each diagonal by Blocks of threads in parallel. But, for a problem size of n matrices, this approach takes memory space of size $(2 \cdot n^2)$, where $O(n^2)$ size of memory is wasted. In addition, much time is wasted in accessing the array of dimensions from Global Memory of GPU by the threads of each Block which can be further accelerated with the help of Shared Memory.

Mohsin Altaf Wani and S.M.K Quadri [4] presented an accelerated dynamic programming on GPU in 2013. They have not used any mapping of m table, but simply used single Block of threads for the computation of a single diagonal of m where each thread independently calculates some elements of that diagonal in parallel. This approach suffers from non-coalesced memory access and does not use multiple Blocks of threads also.

2 GPU and CUDA

NVIDIA introduced CUDA^{TM}, a general purpose computing architecture in GPU in 2006. This massive parallel computing architecture can be applied to solve complex computational problems in highly efficient manner with respect to equivalent sequential solution implemented on CPU. Developers may use the high-level language, C in programming with CUDA [3].

2.1 GPU Architecture

GPU consists of several Streaming Multiprocessors (SM) with many cores and mainly two types of memory: *Global Memory, Shared Memory* (Fig. 1) [3]. Also, each SM has number of *registers* which are fastest and local to SM. Each SM has its own Shared Memory, which can be as fast as registers when bank conflict does not happen. Global Memory of higher memory capacity is potentially 150$\times$ slower than Shared Memory.

2.2 CUDA

In programming architecture of CUDA [3], parallelism is achieved with bunch of *threads* combined into a *Block* and multiple *Blocks* combined into a *Grid*. Each *Block* is always assigned to a single SM while the *threads* in a Block are scheduled to compute as a *warp* of 32 threads at a time. All threads of a Block can be synchronized within that Block and can access the Shared Memory of that assigned SM only. CUDA permits programmers to write C function, called *Kernel* for parallel computations in GPU using Blocks of threads.

2.3 Coalesced Memory Access

If access requests to Global Memory from multiple threads can be assembled into contiguous location accesses or same location access, this request can be performed at once which is known as *coalesced memory access* [3]. As a result of such memory coalescing, Global Memory works nearly as fast as register memory.

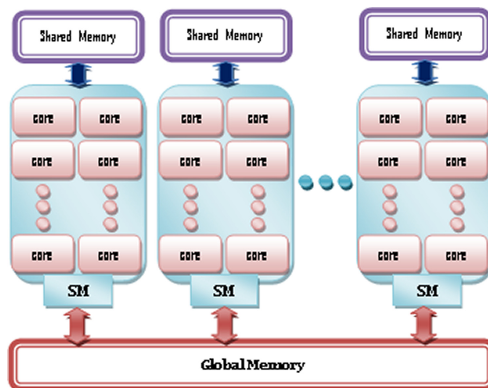


Fig. 1. GPU computing architecture in CUDA

2.4 Shared Memory and Memory Banks

Shared Memory [3, 10] is a collection of multiple banks of equal size which could be accessed simultaneously. Any memory accesses of n addresses from n distinct memory banks can effectively be serviced simultaneously. If multiple threads request to the same bank and to the same address, it is accessed only once and served to all those threads concurrently as multicast at once. However, multiple access requests to different addresses from same bank lead to *Bank Conflict*, which needs much time as requests are served serially [11]. For the GPU devices of compute capability $\geq 2.x$, there are 32 banks with each bank of 32-bits long whereas the warp size is of 32 threads. If multiple threads of a warp try to access data from the same memory address with same bank, there happens no bank conflict also which is termed as multicast. When used with 1Byte/2Byte long data in Shared Memory, each bank contains more than one data. In this case also there is no bank conflict if threads access these data from single bank, as this is taken as multicast [12] in GPU device with compute capability $\geq 2.x$. GPU devices of compute capability = 2.x have the default settings of 48 KB Shared Memory/16 KB L1 cache.

3 Matrix Chain Multiplication Problem

Matrix Chain Multiplication or Matrix chain ordering problem requires to finding the best order for multiplying the given sequence of matrices so that the least number of multiplications are involved. This is merely an optimization problem using the associative property of matrix multiplication. Actually the solution is to provide the fully parenthesized chain of matrices through the optimal order of multiplication.

Provided the Matrix Chain, containing n matrices $\{A_1, A_2, \ldots\ldots A_n\}$, is to be multiplied where $\{d_1, d_2, d_3, \ldots\ldots d_n, d_{n+1}\}$ is the set of all dimensions of these matrices, described as follows:

$$A_1 : d_1 \times d_2$$
$$A_2 : d_2 \times d_3$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$A_n : d_n \times d_{n+1}$$

Now, the problem is to find the order in which the computation of the product $A_1 \times A_2 \times \ldots\ldots \times A_n$ needs the minimum number of multiplications and hence find the fully parenthesized chain denoting the optimal order of multiplication.

3.1 Solving Technique in Dynamic Programming

Dynamic programming is useful for storing the solutions of small sub-problems and reusing them step by step to combine into greater problems and finally finding the solution of the given problem in time efficient approach. Dynamic programming technique [1] makes it easy to solve the above Matrix Chain Multiplication problem with n matrices using a $m[][]$ table, where $m[i,j]$ denotes the minimum number of multiplications needed

to compute the sub-problem <Ai x Ai + 1 x….x Aj> for 1 < i < j < n. Minimum cost $m[i,j]$ is calculated using the following recursion:

$$\begin{cases} \mathbf{0} & if\ i = j \\ \min_{i<k\leq j}\{\boldsymbol{m}[\boldsymbol{i},\boldsymbol{k}] + \boldsymbol{m}[\boldsymbol{k}+1,\boldsymbol{j}] + \boldsymbol{d_i d_k d_{j+1}} & if\ i < j \end{cases} \quad (1)$$

Here "k" is stored in another table $s[][]$ at $s[i,j]$ when the minimum value for $m[i,j]$ is found. Thus $m[1,n]$ refers to the solution for the full problem and $s[][]$ table is used to determine the parenthesized solution of the chain.

Dynamic programming technique requires time of $O(n^3)$.

3.2 Accelerated Dynamic Programming in GPU

Dynamic Programming approach for solving Matrix Chain Multiplication problem can be easily parallelized by computing for the elements of each diagonal of $m[][]$ and $s[][]$ tables in GPU independently in different threads. But, this is inefficient and time-taking due to lack of coalescing in Global Memory access.

Kazufumi Nishida, Yasuaki Ito and Koji Nakano [2] innovated a technique of memory mapping of m and s to ensure coalesced memory access for accelerating the dynamic programming for the Matrix Chain Multiplication in GPU. m and s are mapped into arrays of $n \times n$ memory spaces in Global Memory of GPU along with the array of dimensions of matrices, $d[]$.

Let us take a problem sample of six matrices ($n = 6$):

$$A1 : 20 \times 25, A2 : 25 \times 50, A3 : 50 \times 35$$
$$A4 : 35 \times 10, A5 : 10 \times 40, A6 : 40 \times 30$$

Dynamic programming starts from the base case $m[i][j] = 0 fori = j$ i.e., the diagonal-1 (Fig. 2) of m. Then, $m[i][j]$ for each diagonal (upper) is to be computed using recursion (1) where the s table is needed to store values only in upper diagonals 2 to n. Required diagonals of m and s tables are mapped in row by row manner (Fig. 2). GPU kernel may be called for computation of diagonals one by one from diagonal-2 to diagonal-n of m and s, which ensures coalescing.

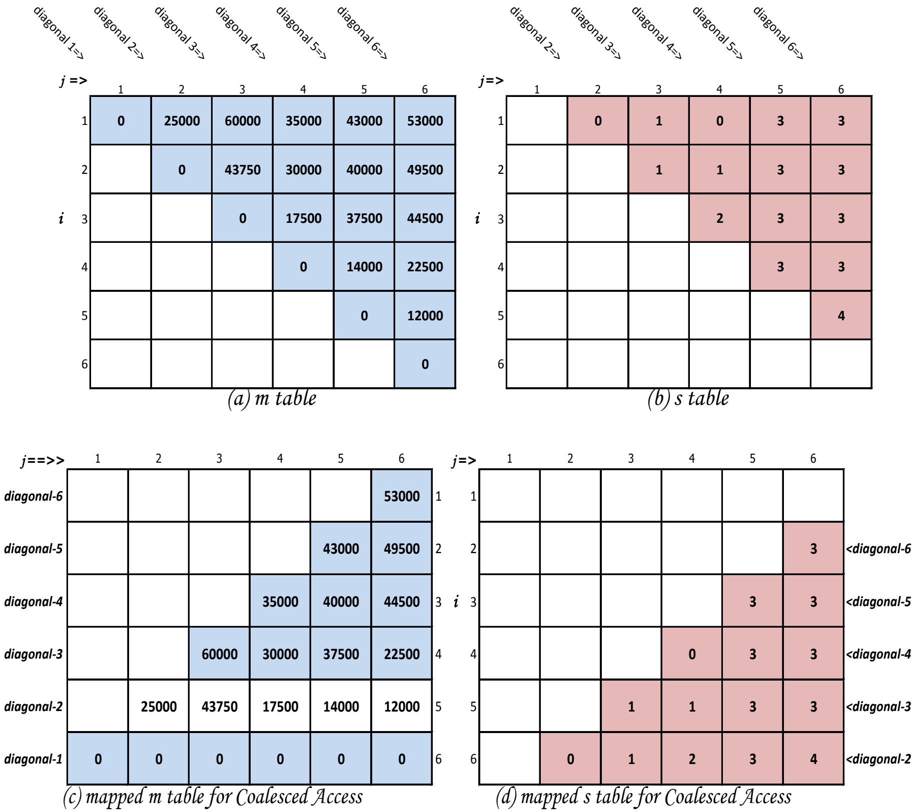
Here, m and s both table are implemented with an array of $n \times n$ elements of memory size $O(n^2)$. But, in computation we do not need the all locations. Say, the number of memory spaces wasted in m and s are W_m and W_s respectively.

Then, $W_m = \{(n-1) + (n-2) + \ldots. + 2 + 1\} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

$$W_s = \{n + (n-1) + \ldots. + 2 + 1\} = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$So, W_m + W_s = \frac{n(n-1)}{2} + \frac{n(n+1)}{2} = n^2$$

This technique suffers from memory wastage of size $\theta(n^2)$ (i.e. half of the spaces allocated) in GPU which is very limited in storage.



(a) m table (b) s table

(c) mapped m table for Coalesced Access (d) mapped s table for Coalesced Access

Fig. 2. Memory Mapping of $\boldsymbol{m}$ & $\boldsymbol{s}$ tables for n = 6

Recursion (1) shows that to compute $m[i][j]$, it needs to access $\boldsymbol{d_i}, \boldsymbol{d_{i+1}}, \boldsymbol{d_{i+2}}, \ldots..\boldsymbol{d_{j+1}}$ from the array of dimensions (d[]) in Global Memory. All threads of a Block need to access the dimensions same way in GPU kernel for computation of a diagonal as shown in Fig. 4. Though this technique ensures the coalesced access of Global Memory from d[] by the threads, this accessing time from Global Memory can rather be reduced much by our new efficient technique of using Shared Memory, which serves much faster with respect to Global Memory.

4 Proposed Approach

4.1 Combined *m* and *s* Table

The most significant thing in our approach is to optimize the memory spaces allocated in Global Memory of GPU device. Without taking two arrays, we have used only an array of $n \times n$ elements for containing m and s tables both combined (Fig. 3), using memory mapping [2, 8] for maintaining the coalesced memory access pattern for better efficiency. Our approach not only assures coalescing but also reduces the space complexity to half

which offers to solve Matrix Chain Multiplication problem of larger datasets even with limited memory in GPU device.

	i \ *j*	1	2	3	4	5	6	
diagonal 1 of m =>	1	0	0	0	0	0	0	
diagonal 2 of m =>	2	25000	43750	17500	14000	12000	3	*<= diagonal 6 of S*
diagonal 3 of m =>	3	60000	30000	37500	22500	3	3	*<= diagonal 5 of S*
diagonal 4 of m =>	4	35000	40000	44500	0	3	3	*<= diagonal 4 of S*
diagonal 5 of m =>	5	43000	49500	1	1	3	3	*<= diagonal 3 of S*
diagonal 6 of m =>	6	53000	0	1	2	3	4	*<= diagonal 2 of S*

Fig. 3. Combined table for Mapped m and s for $n = 6$

In this approach, we have done computations for all diagonal elements of m and s from diagonal-2 to diagonal-n. Each element in a diagonal is calculated by single thread in GPU. After computation, updating values of $m[i][j]$ and $s[i][j]$ for $i < j$ assures the coalescing for each diagonal, resulting in fast and effective memory access.

4.2 Block-Size Choosing Technique

l $^{\text{th}}$ diagonal (upper) of m and s tables contains $e = n - l + 1$ elements [1]. When computation goes forward from 2nd diagonal to n^{th} diagonal, this number of elements (e) decreases from n to 1. Thus, if B threads/Block are assigned for each diagonal, then the diagonals with $e < B$ do not need the whole Block and some threads remain unutilized in computation. When, e becomes so less with respect to B, most of the threads in the Block are launched in vain.

In previous approach [2], a total Block or multiple Blocks of threads were assigned to the calculation of single element of a diagonal depending on the value of e. We have used a simple approach to assign varying number of threads per Block for e with less number of elements. We need to call the kernel with e/B number of Blocks while Block-size B denotes threads/Block. We have chosen a most suitable value of B provided that some value $\widehat{B}$ is taken as threads/Block satisfying $\widehat{B} \geq e$ and $\widehat{B} mod 32 = 0$ when $e < B$. This technique saves our time, by not launching unnecessary threads for diagonals with few number of elements.

4.3 Using Shared Memory for *d*[]

Similar to m and s tables, $d[]$ array of dimensions of the Matrix Chain is copied to Global Memory in the device. While computation of $m[i][j]$ is done for l $^{\text{th}}$ diagonal of m and s, $j = i + l - 1$ and i ranges from 1 to, where number of elements in the diagonal is $e = n - l + 1$ in dynamic programming technique [1]. For computation of $m[i][j]$, it is

required to calculate all values of $\boldsymbol{d_i d_k d_{j+1}}$ *for* $i < k \leq j$ according to Recursion (1). In kernel, threads in a Block need to access the elements of $d[]$ array in coalesced manner as the threads are accessing contiguous memory locations in parallel (Fig. 4).

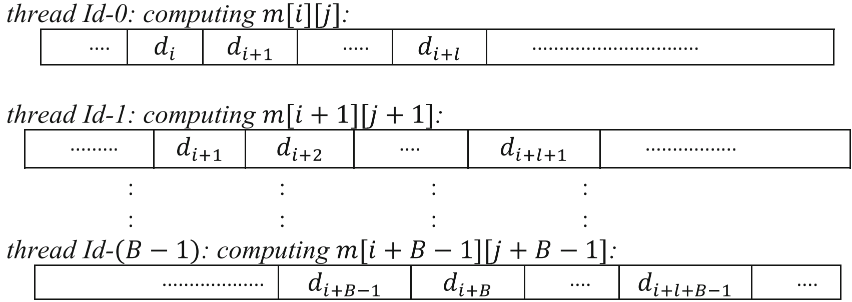


Fig. 4. Memory access patterns by the threads of a Block of Block-size $\boldsymbol{B}$ for $\boldsymbol{l^{th}}$ upper diagonal of m table. Here, $\boldsymbol{j} = \boldsymbol{i} + \boldsymbol{l} - 1$ and $\boldsymbol{i} = \boldsymbol{B} \times \boldsymbol{b_x} + 1$

GPU scheduler schedules Blocks one by one while all threads in that Block compute in parallel in warps of 32 threads at a time. If Block-size is B, then threads have thread-Id from 0 to $B-1$ for any Block with Block-Id b_x, *where* $0 \leq b_x < e/B$ and e/B is total number of Blocks for l th diagonal (Sect. 4.B). When, thread Id 0 of Block Id b_x is used to calculate the value of $m[i][j]$ (i th element of the l th diagonal), it needs to access values of $\boldsymbol{d_i}, \boldsymbol{d_{i+1}} \ldots \boldsymbol{d_{j+1}}$ i.e., $\boldsymbol{d_i}, \boldsymbol{d_{i+1}} \ldots \boldsymbol{d_{i+l}}$ of $d[]$ as shown in Fig. 4 where $i = B \times b_x + 1$ and $j = i + l - 1$. Though this large number of Global Memory accesses can be arranged with coalesced access pattern, this time for memory accesses from Global Memory can be again reduced, as there is a scope to use Shared Memory to serve this purpose much rapidly.

We have used Shared Memory to reduce the access time and the number of Global Memory accesses. Though all threads run independently, they need to access only the elements of $d[]$ in a range and those values are to be used by multiple threads in that Block. This range is $\boldsymbol{d_i}, \boldsymbol{d_{i+1}} \ldots \boldsymbol{d_{i+l+B-1}}$ (Fig. 4). Therefore, the total Block of threads uses only these $\boldsymbol{l} + \boldsymbol{B}$ elements of $d[]$ from Global Memory. Only these $l + B$ number elements of $d[]$ are copied to Shared Memory for each Block. Each thread of a Block can access Shared Memory of that Block in very fast and effective manner with no bank conflict. If we use 1Byte/2Byte long data for each element of $d[]$, the warp of 32 threads access distinct elements of $d[]$ and results in no bank conflict as discussed in Sect. 2.4 with the help of multicasting. Due to space limitation of Shared Memory, dimensions can be stored as 1Byte long data. We have used this approach along with the memory optimization technique (Sect. 4.1) and hence got very effective results.

5 Performance Evaluation

Tests have been carried out on the Matrix Chains containing different number of matrices whose dimensions are randomly generated in the range [1, 100]. Our experiment is made

over a long range of Matrix Chain length (n) i.e. 1000 to 14000 number of matrices in the chain.

We have used NVIDIA GeForce GT 525 M graphics card of 1 GB for parallel computation using GPU and Intel Core i5 @2.5 GHz with 4 GB RAM for sequential processing in CPU. Our GPU device is of Fermi architecture [7] which has 16 SMs of 32 cores each, i.e. total 512 cores and allows maximum 1024 threads/Block.

While $e = n - l + 1$ specifies the number of elements in l^{th} diagonal of m table, we obtained the best speed up when we have passed varying number of threads (multiple of 32) for $e < 768$ and 768 threads/Block for $e >= 768$ for l^{th} diagonal for better occupancy in the GPU kernel.

In Table 1, we can compare the execution time (in sec.) of our Memory Optimized Approach in GPU with the Sequential Approach in CPU and other two approaches: Memory Unmapped Approach (diagonals of m[] and s[] tables are not mapped) and previous Memory Mapped Approach (diagonals of m[] and s[] tables are mapped to different arrays). Our Memory Optimized Approach using GPU shows increasingly better performance as the problem size increases in comparison to CPU according to Fig. 5, showing in logarithmic scale.

Table 1. Execution time (in sec.) of the Sequential Approach in CPU and different Approaches in GPU: Memory Unmapped Approach and Previous Technique of Memory Mapped Approach and our Memory Optimized Approach.

Matrix Chain length ($\boldsymbol{n}$)	Sequential Approach in CPU ($\boldsymbol{t_{seq}}$)	Unmapped Approach in GPU ($\boldsymbol{t_{pll}}$)	Previous Memory Mapped Approach in GPU ($\boldsymbol{t_{pll}}$)	Our Memory Optimized Approach in GPU ($\boldsymbol{t_{pll}}$)
1000	1.2695	2.45833	0.312	0.3058
2000	13.1489	19.81633	1.33893	1.3018
3000	49.5611	66.682	3.96608	3.63
4000	152.9227	155.0192	7.79234	7.0868
5000	253.3594	297.1772	16.69396	13.9917
6000	449.1240	513.3569	26.90393	22.509
7000	793.9893	834.8231	44.34579	36.4512
8000	1441.5231	1243.0041	55.12547	47.5437
9000	1688.754	1761.305	91.98259	75.0826
10000	2295.8261	2411.3491	118.25785	96.9248
11000	3302.2165			137.9615
12000	4844.4406			149.7597
13000	5947.9613			221.5745
14000	9044.3004			254.0719

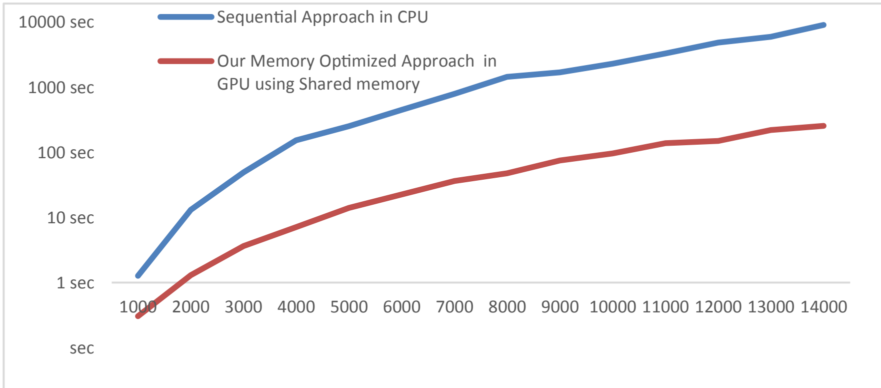


Fig. 5. Comparison of Execution time (in sec.) of Our Memory Optimized Approach in GPU vs. Sequential CPU implementation for $n = 1000 \ to \ 14000$.

Speedup factors of the approaches in GPU over the CPU implementation (shown in Table 2) are computed as follows:

Table 2. Speedup achieved by different Approaches in GPU: Memory Unmapped Approach and Previous Technique of Memory Mapped Approach and our Memory Optimized Approach.

Matrix Chain length (n)	Unmapped Approach in GPU (t_{seq}/t_{pll})	Previous Memory Mapped Approach in GPU (t_{seq}/t_{pll})	Our Memory Optimized Approach in GPU (t_{seq}/t_{pll})
1000	0.52	4.07	4.15
2000	0.66	9.82	10.10
3000	0.74	12.50	13.65
4000	0.99	19.62	21.58
5000	0.85	15.18	18.11
6000	0.87	16.69	19.95
7000	0.95	17.90	21.78
8000	1.16	26.15	30.32
9000	0.96	18.36	22.49
10000	0.95	19.41	23.69
11000			23.94
12000			32.35
13000			26.84
14000			35.60

$$Speedup = \frac{Execution\ time\ in\ CPU\ Approach}{Execution\ time\ in\ GPU\ Approach}$$

Our approach acquired as much Speedup as 35.6 (Table 2) for the problem size (n) of 14000 matrices in the chain. Due to lack of available space in our GPU device (1 GB), other Approaches (Table 1) can be run over the datasets of Matrix Chain of length (n) only upto 10000 in our GPU device because of higher memory requirements, whereas our Memory Optimized Approach runs successfully upto Matrix Chain of length (n) upto 14000. Speedup factor our approach is compared with other GPU approaches in Fig. 6. It's quite clear that our technique not only requires much less memory in GPU device but also performs all time better.

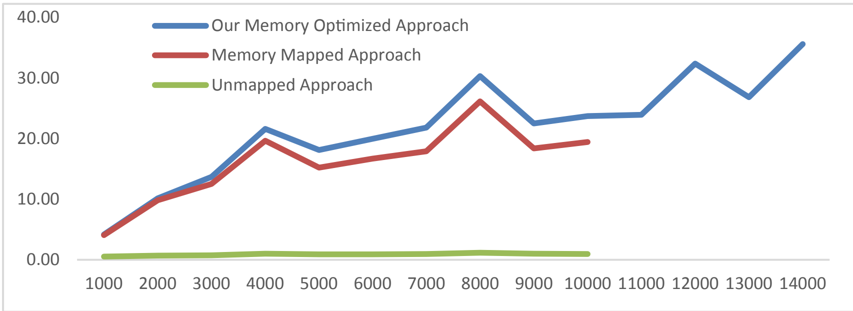


Fig. 6. Comparison of Speedup of Our Approach with other Approaches in GPU for $n = 1000\ to\ 14000$.

We have used the Shared Memory to reduce Global Memory Access (Sect. 4.3), but the previous best Memory Mapped Approach used only Global Memory to access the array of Dimensions $d[]$. We have listed the total number of Memory Accesses from the array of Dimensions $d[]$ stored in Global Memory of GPU for Our Approach and the Memory Mapped Approach in Table 3. Our Approach needs much less number of Memory Accesses from $d[]$ as compared to the previous best Approach (Fig. 7). Our Approach is not only better in this access count, but also copies the required portion Global Memory of $d[]$ to Shared Memory in coalesced manner. For there is only 48 KB Shared Memory in our device, here, we took 1Byte space for each element of $d[]$ as it is in the range of [1, 100]. Thus, our Approach reduces the access time to a certain remarkable factor with the help of Shared Memory with no bank conflict.

Table 3. Number of Global Memory Accesses (in million) from the Dimension array $\boldsymbol{d}[]$ in Our Approach and Previous Memory Mapped Approach in GPU

Matrix Chain length ($\boldsymbol{n}$)	Previous Memory Mapped Approach in GPU ($\boldsymbol{C}_1$)	Our Memory Optimized Approach ($\boldsymbol{C}_2$)
1000	16767	103
2000	133733	487
3000	450900	1281
4000	1068266	2614
5000	2085833	4620
6000	3603599	7427
7000	5721566	11164
8000	8539732	15964
9000	12158099	21956
10000	16676666	29268

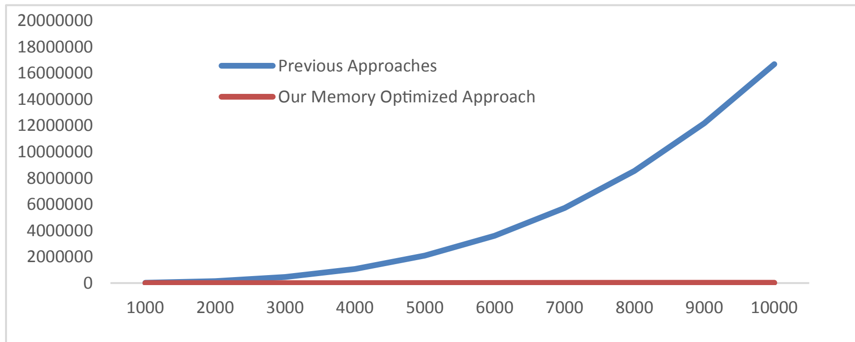


Fig. 7. Comparison of Number of Global Memory Accesses (in million) from the Dimension array $\boldsymbol{d}[]$ between Our Approach and Previous Memory Mapped Approach in GPU for $n = 1000\ to\ 10000$.

6 Conclusion

In this paper, we have presented a new Dynamic Programming technique for parallel processing in CUDA enabled GPU to solve the problem of Matrix Chain Multiplication. All of the previous Approaches, known to us, needed two $n \times n$ size arrays ($O(2.n^2)$) to keep m and s tables which are required in Dynamic Programming to solve this problem. Here, we have suggested a technique to use only one $n \times n$ size array ($O(n^2)$) to which m and s both tables are to be mapped for minimizing the memory requirements in GPU. This allows us to solve problems of Matrix Chain with larger number of matrices in small sized memory in GPU device. Only with the GPU device of 1 GB memory, we

have successfully run our Memory Optimized Technique upto matrix chain of length 14000, where the other approaches stuck at only 10000.

Another technique, we used, is to copy only the required elements of array of dimensions to Shared Memory. It reduces the memory access time and accelerates the execution of our Memory Optimized Approach. Our approach shows so vigorous results with nearly monotonously increasing speedup with respect to CPU on increasing the problem size and further proficient compared to other approaches. We have achieved the speedup factor of 35.6 over CPU-based approach for a randomly generated chain of 14000 matrices, which is unparalleled to other techniques. As a future scope, our approach could be run on the GPU device with much storage space and predictably, larger speedup could be achieved for larger Matrix Chain compared to other techniques. Hence, our paper proposes a new Memory Optimized and more efficient technique to solve Matrix Chain Multiplication problem using dynamic programming assisted with shared memory.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press and PHI, New Delhi (2012)
2. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the matrix chain product on the GPU. In: Second International Conference on Networking and Computing, pp. 320–326 (2011)
3. NVIDIA, CUDA C Programming Guide Version 4.2 (2012). https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accessed 22 Jun 2020
4. Wani, M.A., Quadri, S.M.K.: Accelerated dynamic programming on gpu: a study of speed up and programming approach. In: Int. J. Comput. Appl., 0975–8887 (2013)
5. NVIDIA, CUDA ZONE. https://developer.nvidia.com/cuda-zone. Accessed 12 Jul 2020
6. Fauzia, N., Pouchet, L.N., Sadayappan, P.: Characterizing and enhancing global memory data coalescing on GPUs. In: IEEE/ACM International Symposium on Code Generation and Optimization (2015)
7. Whitepaper NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed 15 Jul 2020
8. Ito, Y., Nakano, K.: A GPU implementation of dynamic programming for the optimal polygon triangulation. IEICE Trans. Inf. Syst., **D**(12), 2596–2603 (2013)
9. Pimple, M.R., Sathe, S.R.: Analysis of resource utilization on GPU. Int. J. Adv. Comput. Sci. Appl., **10**(2) (2019)
10. Bergeron, J.P.: Programming of shared memory GPUs shared memory systems, University of Ottawa (2011)
11. NVIDIA, DEVELOPER ZONE. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory. Accessed 26 Jul 2020
12. NVIDIA, Shared memory bank conflicts with byte arrays. https://forums.developer.nvidia.com/t/shared-memory-bank-conflicts-with-byte-arrays/20553/4. Accessed 26 Jul 2020