Accelerating the Dynamic Programming for the Matrix Chain Product on the GPU

Kazufumi Nishida, Yasuaki Ito, and Koji Nakano
Department of Information Engineering,
Hiroshima University,
1-4-1 Kagamiyama, Higashihiroshima, Hiroshima, 739–8527 Japan
$\{$*nishida, yasuaki, nakano*$\}$*@cs.hiroshima-u.ac.jp*

***Abstract*—Modern GPUs (Graphics Processing Units) can be used for general purpose parallel computation. Users can develop parallel programs running on GPUs using programming architecture called CUDA (Compute Unified Device Architecture). The Matrix Chain Product Problem is an optimization problem for finding parentheses of the matrix chain that gives the minimum total number of multiplications necessary to compute the product of the matrix chain. It is well known that this problem can be solved using the dynamic programming technique in $O(n^3)$ time using tables of size $O(n^2)$. The main contribution of this paper is to present an efficient parallel implementation of this $O(n^3)$-time algorithm on the GPU. In our implementation, we have considered the architecture and programming issues of the GPU system. The experimental results show that, for a chain of 16384 matrices generated at random, our implementation in the Nvidia GeForce GTX 480 achieves a speedup factor of 40 over a conventional CPU implementation.**

***Keywords*-Dynamic Programming; Matrix Chain Product; GPGPU; CUDA; Parallel Processing;**

I. Introduction

Recent Graphics Processing Units (GPUs), which have a lot of processing units, can be used for general purpose parallel computation. Since GPUs have very high memory bandwidth, the performance of GPUs greatly depends on memory access. CUDA (Compute Unified Device Architecture) [1] is the architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [2], [3], [4], [5], [6], [7].

Matrix Chain Product Problem is an optimization problem for finding the best parentheses of the matrix chain that minimizes the total number of multiplications [8]. Suppose that a chain of three or more matrices to be multiplied is given. The total number of multiplication may vary depending on the order of multiplication. For example, given 3 matrices A, B, and C, of size 5×1, 1×5, and 5×1, respectively, let us evaluate the total number of multiplications to compute the product $A\cdot B\cdot C$. We have two ways to compute the product: (1) $A\times B$ is computed first (i.e.$(A\cdot B)\cdot C$) and (2) $B\times C$ is computed first (i.e. $A\cdot(B\cdot C)$). Since the multiplication of the matrix is associative, the results of the both ways are the same. Note that, the product of an $l\times m$ matrix and an $m\times n$ matrix needs lmn multiplications and the size of the resulting matrix is $l\times n$. In case of (1), the product $A\cdot B = X$ needs $5\times 1\times 5 = 25$ multiplications and the size of the resulting matrix is 5×5. After that, the remaining product $X\cdot C$ needs $5\times 5\times 1 = 25$ multiplications. Thus, the total number of multiplications is 50. On the other hand, in case of (2), the product $B\cdot C = Y$ needs $1\times 5\times 1 = 5$ multiplications, and the size of the resulting matrix is 1×1. The remaining product $A\cdot Y$ needs $5\times 1\times 1 = 5$ multiplications. The total number of multiplications is only 10 and thus, we should select (2). If we have only three matrices to be multiplied, we can easily find which product should be computed first in the same way. However, if we have more than three matrices, it is not trivial to find the best parentheses that minimizes the total number of multiplications.

Since the number of possible parenthesizations of a chain of n matrices is the Catalan number [8] $\frac{(2n)!}{(n+1)!n!} = \Omega(4^n/n^{3/2})$, it is impossible to evaluate the number of multiplications for every parenthesization. It is known that the dynamic programming technique can be applied to solve the Matrix Chain Product problem in $O(n^3)$ time [9] using tables of size $O(n^2)$.

The main contribution of this paper is to present a parallel implementation of this $O(n^3)$-time dynamic programming based algorithm in the GPU. In our implementation, we have considered the architecture and programming issues of the GPU system. CUDA can execute a lot of *Threads* in parallel. An array of Threads constitutes a *Block*, and an array of Blocks constitutes a *Grid*. To allocate the Grid in a GPU system, the host PC calls a special C function called *Kernel*. Each Block in a Grid is assigned to one of the Streaming Multiprocessor in the GPU. If the number of Blocks is larger than that of Streaming Multiprocessors, two or more Blocks may be arranged in one Streaming Multiprocessor. Each Streaming Multiprocessor has usually 32 processor cores which execute Threads in a Block in parallel. To minimize the computing time in the GPU, it is important to balance the numbers of Blocks in a Kernel and that of Threads in a Block. and a single Streaming Multiprocessor can execute multiple Threads with several processing cores in parallel. In our proposed implementation with CUDA, we implemented three Kernels each of which is a kind of function call on the GPU. The granularity of these three Kernels differs for Blocks and Threads. The feature of Matrix Chain Product Problem with dynamic programming is that the amount of the computation varies during the computation. Because of the variation of the amount of the computation, the parallelism of the parallel execution may change. We have evaluated the three Kernels for each step in the algorithm to determine the efficient allocation. According to the result, in our implementation, the fastest Kernel is selected dynamically to achieve efficient parallel execution on the GPU. We have implemented it in a modern GPU system, Nvidia GeForce GTX 480 with 480 processing cores. The experimental results show that our implementation can achieve a speed-up factor of 40 over the sequential implementation on the CPU.

The remaining parts of this paper is organized as follows; Section II introduces concrete algorithm of dynamic programming for Matrix Chain Product Problem. Section III shows GPGPU features in CUDA. Section IV proposes an implementation of Matrix Chain Multiplication Problem on the GPU. The experimental results show in Section V. Finally, Section VI offers concluding remarks.

II. A DYNAMIC PROGRAMMING ALGORITHM FOR MATRIX CHAIN PRODUCT

This section briefly describes the dynamic programming algorithm for Matrix Chain Product Problem. Let $\mathcal{P} = \langle p_0, p_1, \ldots, p_n \rangle$ be a sequence of dimensions of matrix A_i for $i = 1, 2, \ldots, n$, such that the size of A_i is $p_{i-1} \times p_i$. Recall that the product of an $l \times m$ matrix and an $m \times n$ matrix needs lmn multiplications and the size of the resulting matrix is $l \times n$. Let $m_{i,j} (1 \leq i \leq j \leq n)$ denote the minimum number of multiplications to compute the product of $A_i \cdot A_{i+1} \cdot \cdots \cdot A_j$. The goal of the Matrix Chain Product Problem is to compute the $m_{1,n}$ and to find the parentheses that gives the minimum number $m_{1,n}$ of multiplications. The idea of the dynamic programming based algorithm is to compute every $m_{i,j}$ as follows.

First, it should be clear that

$$m_{i,i} \quad = \quad 0.$$

Also, the product of A_i of size $p_{i-1} \times p_i$ and A_{i+1} of size $p_i \times p_{i+1}$ can be computed by $p_{i-1} p_i pi + 1$ multiplications. Thus, we have

$$m_{i,i+1} \quad = \quad p_{i-1} p_i p_{i+1}.$$

We assume that $m_{i,j}$s have been already computed for all i, j $(1 \leq i \leq j \leq n)$ such that $j - i < \alpha$. In other words, we have computed the minimum total number of multiplications of $A_i A_{i+1} \cdots A_j$ of α or less matrices. For $j - i = \alpha$ and $i < k < j$, we need $m_{i,k}$ multiplications to compute the product $A_i A_{i+1} \cdots A_k$. Similarly, we need $m_{k+1,j}$ multiplications to compute the product $A_k A_{k+1} \cdots A_j$. Since the sizes of the resulting matrices are $p_{i-1} \times p_k$ and $p_k \times p_j$, respectively, the product of these two matrices needs $p_{i-1} p_k p_j$ multiplications. Thus, we have,

$$m_{i,j} \quad = \quad \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j).$$

In the dynamic programming based algorithm, tables m and s are used. Table m is used to store the value of $m_{i,j}$ and Table s records the index k that achieves the minimum number of multiplications in computing $m_{i,j}$ for separating $m_{i,j}$ into $m_{i,k}$ and $m_{k+1,j}$. Namely, table s shows the parentheses for the optimal cost. The value of $m_{i,j}$ gives the cost of optimal parentheses to subproblems. The value $m_{1,n}$ shows the optimal parentheses for the input. The details of this algorithm are spelled out as follows:

Matrix Chain Product Algorithm

```
1. $m_{i,i} \leftarrow 0$ for every $i$ $(1 \leq i \leq n)$
2. $m_{i,j} \leftarrow \infty$ for every $i$ and $j$ $(1 \leq i < j \leq n)$
3. for $l \leftarrow 2$ to $n$ do
4.     for $i \leftarrow 1$ to $n - l + 1$ do
5.         $j \leftarrow i + l - 1$
6.         for $k \leftarrow i$ to $j - 1$ do
7.             $q \leftarrow m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j$
8.             if $q < m_{i,j}$ then $m_{i,j} \leftarrow q$, $s_{i,j} \leftarrow k$
```

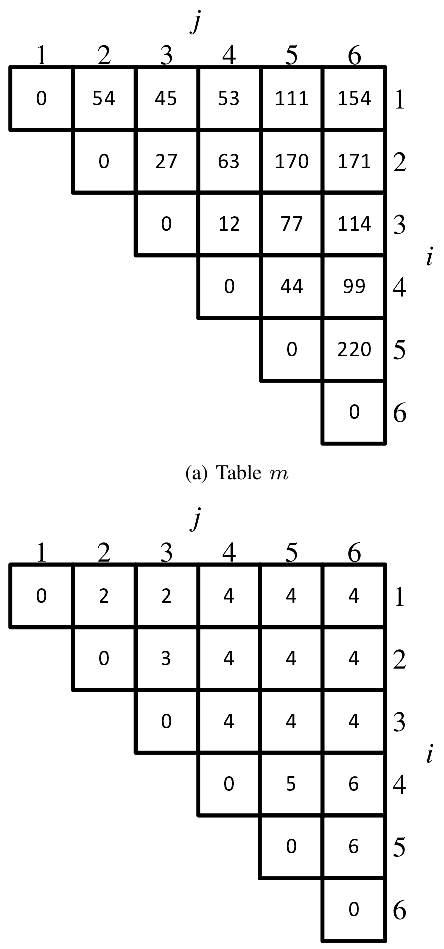
The algorithm first computes $m_{i,i} \leftarrow 0$ for $i = 1, 2, \ldots, n$, that means the minimum costs for chains of length 1, in line 1, Then they are used to compute $m_{i,i+1}$ for $i = 1, 2, \ldots, n-1$, that shows the minimum costs for chains of length $l = 2$, during the first execution of the loop in lines 3–8. The second time through the loop, it computes $m_{i,i+2}$ for $i = 1, 2, \ldots, n-2$, that shows the minimum costs for chains of length $l = 3$, and so forth. At each step, the value $m_{i,j}$, that is computed in lines 7–8, depends only on table entries $m_{i,k}$ and $m_{k+1,j}$ that are already computed.

Figure 1 illustrates an example of tables m and s for a chain of $A_1, A_2, \ldots, A_6$ of size 2×9, 9×3, 3×1, 1×4, 4×11, and 11×5, respectively. From the value of $m_{1,6}$ in Table m, the optimal cost that is the minimum number of multiplications is 154. Also, the optimal parentheses is $(A_1(A_2A_3))((A_4A_5)A_6)$ from table s.

It should be clear that the last 6 lines are dominant in this algorithm. A simple inspection of the nested loop structure yields a running time of $O(n^3)$. Also, the algorithm uses tables m and s of size $O(n^2)$.

III. CUDA ARCHITECTURE

NVIDIA CUDA is a general purpose parallel computing architecture introduced by NVIDIA. It includes the CUDA Instruction Set Architecture (ISA) and the parallel compute engine in the GPU. To program to the CUDA architecture, developers can use C language which is one of the most widely used high-level programming languages.



	1	2	3	4	5	6
1	0	54	45	53	111	154
2		0	27	63	170	171
3			0	12	77	114
4				0	44	99
5					0	220
6						0

(a) Table m

	1	2	3	4	5	6
1	0	2	2	4	4	4
2		0	3	4	4	4
3			0	4	4	4
4				0	5	6
5					0	6
6						0

(b) Table s

Figure 1. An example of tables m and s computed for $n = 6$

CUDA-enabled GPUs have hundreds of cores that can collectively run thousands of computing Threads. Each core has registers and memories. The on-chip *Shared Memory* allows parallel tasks running on these cores to share data without sending it over the system memory bus.

A. Hardware architecture

CUDA-enabled GPUs consist of several *Streaming Multiprocessors* and two kinds of memory, *Shared Memory* and *Global Memory*. A Streaming Multiprocessor consists of multiple processor cores and a Shared Memory. Figure 2 shows an overview of the CUDA hardware architecture. Global Memory is large but slow, whereas Shared Memory is small but fast.

B. A design of programming environment

In CUDA parallel programming model, a hierarchy of Thread groups consists of a *Grid*, *Block* and *Thread*. The Grid contains multiple Blocks and a Block contains multiple

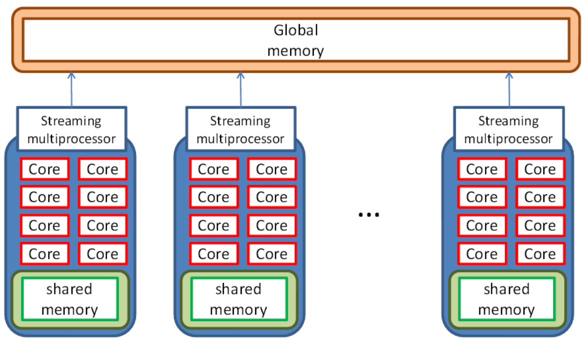


Figure 2. CUDA Hardware Architecture

Threads. A Block is always allocated to a single Streaming Multiprocessor, and a single Streaming Multiprocessor executes Threads in the Block in parallel. Threads in a same Block can be synchronized, however, Threads cannot be synchronized if they are executed in different Blocks. Also, since threads in a block is allocated to a Streaming Multiprocessor, they can access to the same shared memory of the Streaming Multiprocessor. However, threads in different blocks cannot access to the same shared memory. By contrast, all Threads have access to the same Global Memory.

1) Kernel: CUDA C extends C language by allowing the programmer to define C functions, called *Kernels*, that, when called, are executed in parallel by Threads. We configure the number of Blocks and Threads.

In this paper, each diagonal entry is computed in a single Kernel call. Therefore, when the number of matrices is n, Kernel is called $n - 1$ times.

2) Warp and Coalescing: The Streaming Multiprocessor creates, manages, schedules, and executes Threads in groups of 32 parallel Threads called *Warps*. In the Block allocated to each Streaming Processor, Threads in the same Warp execute instructions all together concurrently. When all Threads in the same Warp access 32 sequential and aligned values in the Global Memory, the GPU will automatically combine them into a single transaction.

In this time, if all the Threads access same or continuous memory address, all data are fetched at once. It is called *Coalescing*. This speed-up is very effective, then memory arrangement and access by Threads are very important.

IV. GPU Implementation

In this section, we show an implementation of dynamic programming for Matrix Chain Product using a GPU. Dynamic Programming algorithm for Matrix Chain Product Problem can be parallelized in two ways. One way is the loop in lines 4–8 of Matrix Chain Product Algorithm in Section II. This loop computes the costs in the diagonal entries of tables m and s in Figure 1. Each execution for i in the loop is independent of the others. Another one is

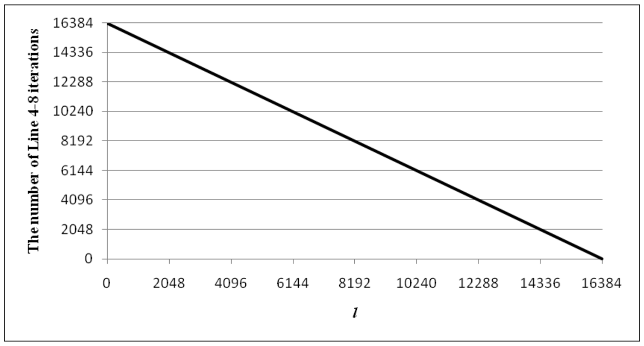


Figure 3. The number of iterations of the loop in lines 4–8 for each l

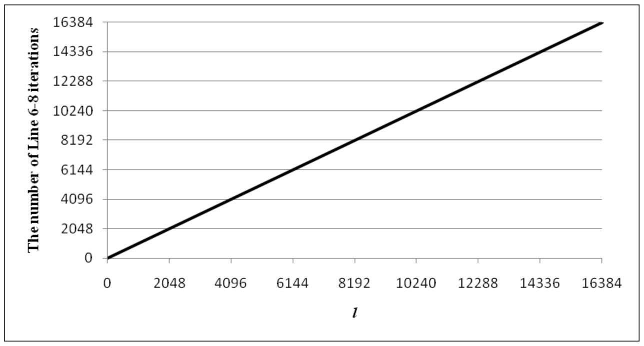


Figure 4. The number of iterations of the loop in lines 6–8 for each l

the loop in lines 6–8. This loop computes the cost for one entry of tables. It is a section to search minimum number of multiplication for a single subproblem. This also does not have dependency, therefore we can compute in parallel, too.

Let us consider the amount of the computation of Matrix Chain Product Algorithm with dynamic programming for each step. We focus on the amount for each execution of the loop in lines 4–6 that computes each diagonal entry of tables s and m. There are two loops that correspond to lines 4–8 and 6–8. Figures 3 and 4 show the numbers of iterations for each l, respectively. According to the figures, the number of computed entries is decreased if l is increased. On the other hand, the amount of the computation to compute each entry is increased if l is increased. The number of total iterations for l is a product of the numbers of iterations of the two loops. Figure 5 shows the amount of the computation for each l. the amount of the computation is varied with l. Therefore, the parallelism in parallel execution depends on l.

In CUDA, it is not easy to obtain good parallelization performance since the performance depends on various factors, for example, Occupancy, a number of Streaming Multiprocessors, amount of computation, and so on [10]. In particular, Matrix Chain Product algorithm has 2 parallelization points and these computation amount changes for l. If a specific allocation is best at the beginning of

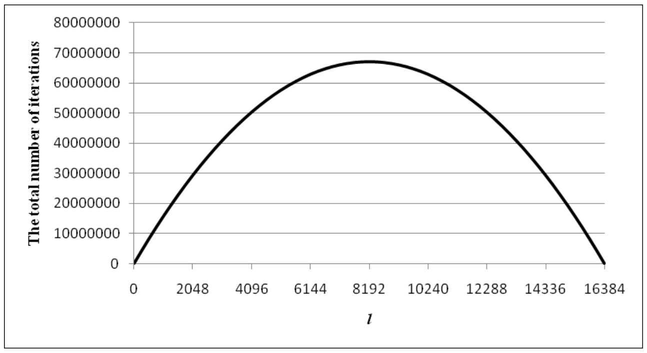


Figure 5. The amount of the computation for each l

process, it is maybe not best allocation at the end of process, or it could be the worst allocation. According to the discussion above, the best allocation cannot be determined uniquely. In our implementation, there are three types of parallelism *OneThreadPerOneEntry*, *OneBlockPerOneEntry*, and *BlocksPerOneEntry*. They are Kernels in the CUDA architecture that performs execution for one diagonal entry in tables m and s. The execution corresponds to computation in lines 4–8. The difference of them is the number of Blocks and Threads for computing one entry that is the execution in the lines 6–8. The details of them are shown as follows.

OneThreadPerOneEntry: This is a Kernel that allocates one Thread to compute one entry in the tables m and s. The execution in the lines 6–8 is done by one Thread. Each Thread computes the cost for one entry and stores the result to the Global Memory in parallel. Since each Block has multiple Treads, in this Kernel, several entries are computed by one Block.

OneBlockPerOneEntry: This Kernel allocates one Block to compute one entry in the tables m and s. The execution in lines 6–8 is done by several Threads in one Block. Each Block divides the computation for the entry and assigns them to each Thread. Each Thread computes the cost for the allocated range and stores the result to the Shared Memory in parallel. After that the minimum one is computed and stored to the Global Memory by one Thread. Since each Block has multiple Threads, in this Kernel, multiple Threads in each Block compute one entry. The parallelism is middle between OneThreadPerOneEntry and BlocksPerOneEntry.

BlocksPerOneEntry: This Kernel allocates multiple Blocks to compute for one entry. This Kernel consists of two Kernel calls to synchronize between Blocks. One is to compute the costs by Blocks in parallel. The other is to obtain the minimum one. In first Kernel call, the computation for one entry is divided and they assigned to each Block. Each Block computes the cost for the allocated parts and stores the result to the Shared Memory with several Threads in parallel. After that the minimum one for each Block is computed and stored to the Global Memory by one Thread. To obtain the minimum one, one Block computes the minimum and stores the result to the Global Memory.

In OneThreadPerOneEntry, the number of processed data is the most. On the other hand, in BlocksPerOneEntry, the number of processed data is the least. However, OneThreadPerOneEntry requires the least Threads, and BlocksPerOneEntry requires the most Threads. The number of processed data and Threads in Kernel OneBlockPerOneEntry is between OneThreadPerOneEntry and BlocksPerOneEntry.

In OneThreadPerOneEntry, to perform Coalescing access for Global Memory, memory mapping of tables m and s is changed as follows. Tables m and s are two-dimensional arrays. In ordinarily, these arrays are mapped to one-dimensional array as shown in Figure 6(a). However, this implementation will cause diagonal access by Threads (Figure 6(b)). Threads should access the memory in sequential address for Coalescing effect. Then, if memory sequence is as Figure 6(c), we change their memory mapping to the 2-D array. Because of this mapping, Threads could access serial memory address. Note that since Coalescing access can be done in the identical Block, Coalescing access cannot be used in OneBlockPerOneEntry and BlocksPerOneEntry.

V. Performance evaluation

We have evaluated our Matrix Chain Product algorithm implemented on the GPU. For the purpose of comparison, we have also implemented a conventional software approach running on a single CPU on a PC. We have used Nvidia GeForce GTX 480 with 480 processing cores (15 Streaming Multiprocessors which has 32 processing cores) running in 1.4GHz and 3GB memory. To evaluate software approach that is the sequential program in C language, we have used Intel Core i7 870 running in 2.93GHz and 8GB memory.

First, we have evaluated three Kernels OneThreadPerOneEntry, OneBlockPerOneEntry, and BlocksPerOneEntry for the whole computation. We have measured execution time to compute Matrix Chain Product for a chain of 16384 matrices, that are generated randomly, for all kernels. In this evaluation, we used each Kernel for the whole computation. Tables I shows the execution time of each Kernels for various settings of the number of Threads and Blocks. According to the table, the execution time depends of the settings of the number of Threads and Blocks and OneThreadPerOneEntry with 32 Threads per Block is the best performance.

Next, we have evaluated three Kernels OneThreadPerOneEntry, OneBlockPerOneEntry, and BlocksPerOneEntry for computing diagonal entries that corresponds to each execution of the loop in lines 3–8. Figure 7 (a)–(c) shows the execution time of the three Kernels for l. In this evaluation, we used the fastest setting for each Kernel and l. We can see that the shapes of these three graphs are similar to that of the graph in Figure 5. From the results, Figure 7 (a)–(c) shows the execution time such that the fastest Kernel for each l is selected. Table II shows which Kernel is selected for l. According the results, the execution time of the combination of the Kernels using Table II is shown in Table III. Compared with the execution time of CPU implementation, GPU implementation achieved approximate 40 times speed up. Note that due to the page limitation, we have shown only the case where the number of matrices is 16384. The best combination for other cases depends on the number of matrices. However, it can be also obtained by the same manner.

Table II
OPTIMAL COMBINATION OF KERNELS

Range for l	Fastest Kernel
$2 \leq l \leq 3668$	OneBlockPerOneEntry
$3669 \leq l \leq 15406$	OneThreadPerOneEntry
$15407 \leq l \leq 15662$	OneBlockPerOneEntry
$15663 \leq l \leq 16384$	BlocksPerOneEntry

Table III
COMPUTING TIME FOR $n = 16384$

GPU[s]	CPU[s]	Speed-up
701.6	29,282	41.7

VI. Concluding remarks

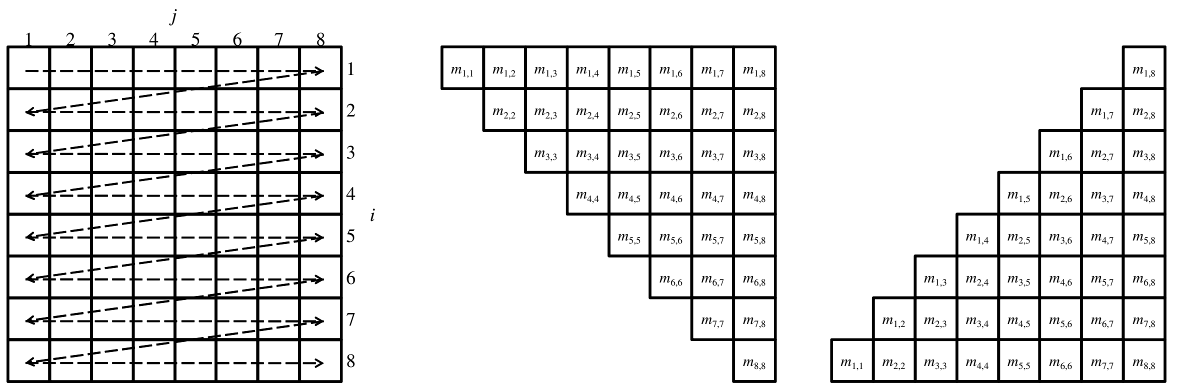
In this paper, we have proposed an implementation of Matrix Chain Product Problem on the GPU. In the proposed implementation with CUDA, we implemented three types of Kernels that is a kind of function call on the GPU. The granularity of these three Kernels is different. Based on the variation of the amount of the computation during the computation. In our implementation, the fastest Kernel is selected dynamically to achieve efficient parallel execution on the GPU. In practice, we have implemented it in a modern GPU system, Nvidia GeForce GTX 480. The experimental results have shown that, for a chain of 16384 matrices that are generated randomly, our implementation achieves a speedup factor of 40 over the sequential CPU implementation. Although an $O(n \log n)$-time algorithm for the Matrix Chain Product Problem is known [11], [12], our implementation gives a new technique to implement a table-based dynamic programming algorithm.

References

[1] NVIDIA, "CUDA ZONE," http://www.nvidia.com/page/home.html.

[2] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," in *Proceedings of Asilomar Conference on Signals, Systems, and Computers*, October 2008, pp. 171–175.

[3] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell, "A parallel implementation of k-means clustering on GPUs," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, July 2008, pp. 340–345.



(a) Memory Mapping Direction (b) Normal Memory Mapping (c) Memory Mapping for Coalesced Access

Figure 6. Memory Mapping of the tables m and s

Table I
PERFORMANCE OF EACH KERNELS ($n = 16384$)

(a) Execution time of OneThreadPerOneEntry [ms]

Threads per Block	32	64	128	256
	729,428	977,575	1,017,610	1,112,098

(b) Execution time of OneBlockPerOneEntry [ms]

Threads per Block	32	64	128	256
	912,642	1,111,794	1,137,166	1,056,523

(c) Execution time of BlocksPerOneEntry [ms]

	Blocks per Entry			
Threads per Block	2	4	8	16
32	1,571,230	1,561,411	1,493,823	1,338,796
64	1,251,733	1,221,975	1,115,941	1,001,370
128	1,215,875	1,059,025	970,862	1,038,715
256	1,128,318	989,715	1,095,406	1,447,205

[4] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” in *Proceedings of the 14th International Conference on High Performance Computing*, 2007, pp. 197–208.

[5] Z. Wei and J. JaJa, “Optimization of linked list prefix computations on multithreaded GPUs using CUDA,” in *Proceedings of International Parallel and Distributed Processing Symposium*, 2010.

[6] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, “Implementations of parallel computation of Euclidean distance map in multicore processors and GPUs,” in *Proceedings of International Conference on Networking and Computing*, 2010, pp. 120–127.

[7] K. Ogawa, Y. Ito, and K. Nakano, “Efficient Canny edge detection using a GPU,” in *International Workshop on Advances in Networking and Computing*, 2010, pp. 279–280.

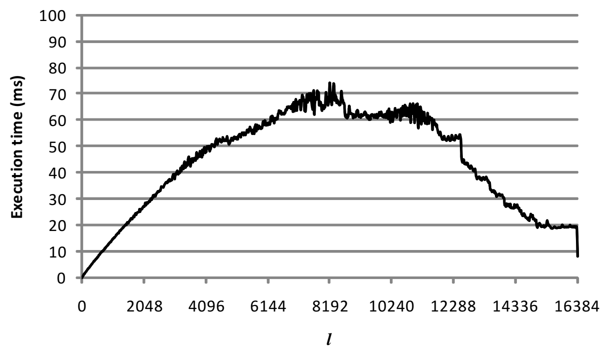
[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001.

[9] S. S. Godbole, “On efficient computation of matrix chain products,” *IEEE Transactions on Computers*, vol. 22, no. 9, pp. 864–866, 1973.

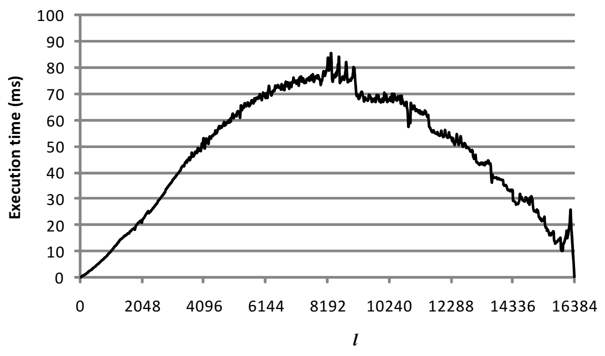
[10] NVIDIA, *NVIDIA CUDA Programming Guide*, July 2009.

[11] T. Hu and M. Shing, “Computation of matrix chain products. part I,” *SIAM Journal on Computing*, vol. 11, pp. 362–373, 1982.

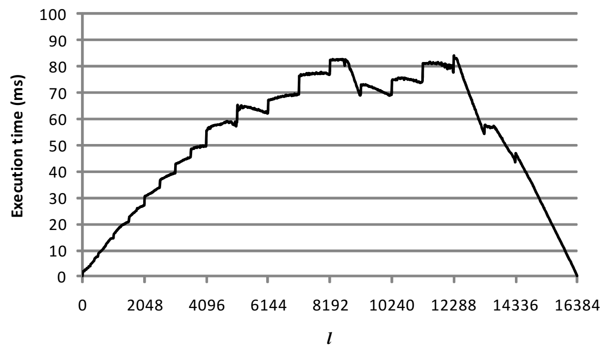
[12] ——, “Computation of matrix chain products. part II,” *SIAM Journal on Computing*, vol. 11, pp. 228–251, 1984.



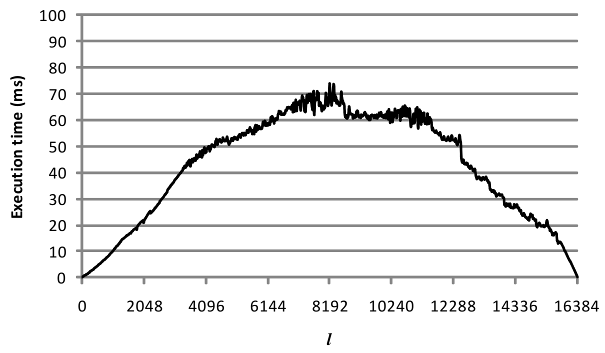
(a) OneThreadPerOneEntry



(b) OneBlockPerOneEntry



(c) BlocksPerOneEntry



(d) Optimal Combination of Kernels

Figure 7. Computing time of three Kernels (a)–(c) and their combination (d)