Accelerated Dynamic Programming on GPU: A Study of Speed Up and Programming Approach

Mohsin Altaf Wani
Research Scholar computer science
Mewar University
Chittorgarh
Rajasthan

S.M.K Quadri
Research Supervisor computer sciences
Mewar University
Chittorgarh
Rajasthan

ABSTRACT
GPUs (Graphics processing units) can be used for general purpose parallel computation. Developers can develop parallel programs running on GPUs using different computing architectures like CUDA or OpenCL. The Optimal Matrix Chain Multiplication problem is an optimization problem to find the optimal order for multiplying a chain of matrices. The optimal order of multiplication depends only on the dimensions of the matrices. It is known that this problem can be solved by dynamic programming technique using $O(n^3)$-time complexity algorithm and a work space of size $O(n^2)$. The main contribution of this paper is to present a parallel implementation of this $O(n^3)$-time algorithm on a GPU and to assess the amount of programming effort required to develop this parallel implementation when compared to a similar sequential implementation.

Keywords: Dynamic programming, parallel algorithms, GPU, CUDA

1. INTRODUCTION
The GPU is a specialized processor designed to accelerate computations for drawing and manipulating images[5]-[7]. Modern GPUs have teraflops of processing power and manufacturers are designing them for general purpose computing in addition to their traditional domain of graphics processing. Hence, GPUs have attracted attention of a large number of developers. NVIDIA provides a parallel computing architecture known as CUDA(Compute Unified Device Architecture)[3], the computing engine in the NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in the GPUs. Similarly AMD GPUs can be programmed using OpenCL(Open Computing Language) framework which is specified by the khronos group[8] .

The Dynamic programming is an algorithmic technique used to find an optimal solution of a problem over an exponential number of candidate solutions [2]. In this approach small instances are solved first, store the results, and later, whenever a result is needed, look it up instead of recomputing it. The term "*dynamic programming*" comes from control theory, and in this sense "programming" means the use of an array (table) in which a solution is constructed. The steps in the development of a dynamic programming algorithm are as follows:

i. Establish a recursive property that gives the solution to an instance of the problem.

ii. Compute the value of an optimal solution in a bottom-up manner.

iii. Construct an optimal solution in bottom-up manner.

Principle of optimality must apply in the optimization problem to be solved using dynamic programming. Principle of optimality is said to apply if an optimal solution to an instance of the problem contains optimal solution to all its sub-instances. Several important problems like optimal binary search tree, edit distance problem can be solved using the dynamic programming [1].

The main contribution of this paper is to implement the dynamic programming to solve the chained matrix multiplication problem [2] on the GPU and to assess the amount of programming effort required to develop this parallel implementation when compared to a similar sequential implementation.

2. CHAINED MATRIX MULTIPLICATION AND DYNAMIC PROGRAMMING SOLUTION
The main purpose of this section is to define the chained matrix multiplication problem and to review the dynamic programming algorithm to solve it.

The goal here is to develop an algorithm that determines the optimal order for multiplying 'n' matrices. The optimal order depends only on the dimensions of the matrices. Therefore besides 'n', these dimensions would be the only input to the algorithm. Let $A_1,A_2,A_3,...,A_n$ be the n matrices to be multiplied, the order in which these matrices are to be multiplied is to be found so that minimum number of multiplications are used.

Consider the multiplication of following four matrices:

A_1	X	A_2	X	A_3	X	A_4
40 x 2		2 x 30		30 x 10		10 x 8

Dimension of each matrix appears -under the matrix. Since matrix multiplication is an associative operation, meaning that the order of multiplication of matrices does not matter. There are five different orders in which four matrices can be multiplied, each possibly resulting in a different number of elementary multiplications. in the previous case we have.

$A_1(A_2(A_3A_4)$ 30x10x8+2x30x8+40x2x8 = 3520

$(A_1A_2)(A_3A_4)$ 40x2x30+30x10x8+40x30x8= 14400

$A_1((A_2A_3)A_4)$ 2x30x10+2x10x8+40x2x8 = 1400

$((A_1A_2)A_3)A_4$ 40x2x30+40x30x10+40x10x8 = 17600

$(A_1(A_2A_3))A_4$ 2x30x10+40x2x10+40x10x8 = 4600

Clearly the third order is optimal order for multiplying these four matrices. It is not hard to see that principle of optimality applies.

When multiplying matrix A_1 with dimensions d_0 x d_1 and A_2 with dimensions d_1 x d_2 the number of multiplications needed are d_0 x d_1 x d_2.

A sequence of arrays will be used to compute a solution. For n matrices A_iA_{i+1} ... A_j where $1 \le i \le j \le n$ let M[i][j] be the minimum number of multiplications needed to multiply matrices A_i through A_j. M[i][j] can be recursively defined as follows.

$M[i][j] = \text{minimum}(M[i][k] + M[k+1][j] + d_{i-1}d_kd_j)$

for all $(i \le k \le j-1)$ ..... recurrence 1

$M[i][i] = 0$.

Using the recursive formula for computing M[i][j], M[i][j] for all $1 \le i \le j \le n$ can be computed in n-1 stages by the dynamic programming as follows:

initialize: M[1][1]=M[2][2] =...=M[n][n] = 0

Step 1: $M[i][i+1] = d_{i-1}d_id_{i+1}$ for all i (1 <=i <=n-1).

Step 2: $M[i][i+2] = \min_{i \le k \le i+1}(M[i][k] + M[k+1][i+2] + d_{i-1}d_kd_{i+2})$ for all $(1 \le i \le n-3)$

.

.

.

Step p-1: $M[i][i+p]=\min_{i \le k \le p+i-1} (M[i][k]+M[k+1][p+i] + d_{i-1}d_kd_{i+p})$ for all $(1 \le i \le n-p-1)$

.

.

.

Step n-1: $M[1][n]=\min_{i \le k \le n-1} (M[1][k]+M[k+1][n] + d_0d_kd_n)$ for all $(1 \le i \le n-1)$.

An example of this procedure is shown below.

Suppose following six matrices with dimensions in braces are given.

A_1 X A_2 X A_3 X A_4 X A_5 X A_6

(5x2) (2x3) (3x4) (4x6) (6x7) (7x8)

Compute $M[i][i] = 0$ for $1 \le i \le 6$

Step 1: $M[1][2] = \min_{1 \le k \le 1}(M[1][k]+M[k+1][2] + d_0d_kd_2)$

$= M[1][1] + M[2][2] + d_0d_1d_2$

$= 0 + 0 + 5x2x3$

$= 30$

The values of M[2][3], M[3][4], M[4][5] and M[5][6] are computed in similar fashion.

Step 2: $M[1][3] = \min_{1 \le k \le 2}(M[1][k]+M[k+1][3] + d_0d_kd_3)$

$= \min(M[1][1]+M[2][3] + d_0d_1d_3, M[1][2]+M[3][3] + d_0d_2d_3)$

$= \min(0 + 24 + 5x2x4, 30 + 0 + 5x3x4)$

$= \min(64, 90)$

$= 64$

The values of M[2][4],M[3][5] and M[4][6] are computed in similar fashion.

Step 3: M[1][4], M[2][5] and M[3][6] are computed in similar manner.

.

.

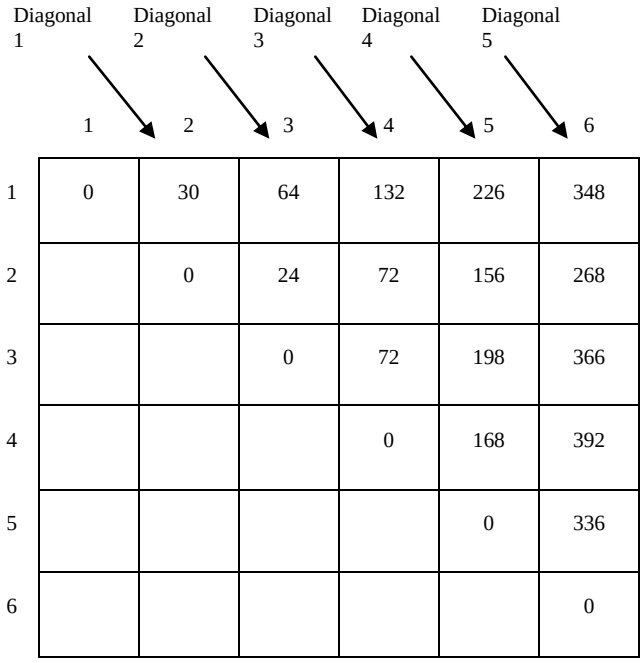
.

Step 5: M[1][6] is calculated using the procedure stated above.

Value of M[1][6] is the solution to our instance and its value is 348.

This procedure can be illustrated by figure given below. Here each M[i][j] is computed from all the entries below and to left of it. the main diagonal is set to 0 initially in accordance to the recurrence 1. the final answer M[1][6] is the solution.

Figure 1.



	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

This dynamic programming algorithm is $O(n^3)$ time algorithm.

3. GPU AND CUDA

CUDA (compute unified device architecture) is a parallel computing architecture with a parallel programming model and instruction set architecture that is used to harness the parallel compute engine in Nvidia GPUs to solve many complex computational problems in a more efficient way than on a CPU.

CUDA parallel programming model has a hierarchy of thread groups called grid, block and threads[3]. A single grid is composed of multiple blocks, each of which has equal

number of threads. Blocks are allocated to streaming multi processors such that all threads in a block are executed by the same streaming multi processor in parallel. All threads can access the global memory. However, threads in a block can only access shared memory of streaming processor to which block is allocated; shared memory is on chip unlike global memory which is DRAM on board having higher latency. Threads in different blocks cannot share data in shared memory.

The multiprocessor executes threads in groups of 32 parallel threads called warps. Threads composing a warp start together at the same program address, however they are free to branch and execute independently. But a divergent branch may lead to poor efficiency[4].

Threads can access data from multiple memory spaces. Each thread has its own register and private local memory. Each block has a shared memory with high bandwidth only visible to all threads of the block.

CUDA C extends C language and allows programmers to define C functions known as *kernels* in CUDA. By invoking a kernel, all blocks in a grid are allocated to streaming multiprocessors. The kernel call terminates when all threads in a block finish the computation. All threads in a block are executed by a single streaming multiprocessor, they are barrier synchronized by calling CUDA C __syncthreads() function[3]. However there is no direct way to synchronize threads executing in different blocks.

This paper is about study of parallel implementation of a dynamic programming algorithm on a GTX 570 GPU having 480 CUDA cores in 15 Streaming multiprocessors.

4. PARALLEL IMPLEMENTATION OF CHAINED MARRIX MULTIPLICATION USING DYNAMIC PROGRAMMING

Purpose of this section is to show an implementation of chained matrix multiplication using dynamic programming on GPU. A parallel implementation of this algorithm on GPU will be presented keeping the implementation as close to a CPU based implementation as possible to assess the amount of programming effort required on part of a programmer.

4.1 Parallel Algorithm

The algorithm for chained matrix multiplication is designed as follows: Stages (diagonals) shown in figure1 are to be computed bottom up, first main diagonal is to be initialized to 0's then diagonal 1 is to be computed, after computing diagonal 1 computation moves onto diagonal 2 and soon till last diagonal is computed. Diagonal 2 cannot be computed before diagonal 1 is computed as is demanded by the algorithm.

This parallel implementation also follows the same principle, after initializing the main diagonal to zero a kernel can be launched to compute each term in diagonal one using recurrence 1. Each term of diagonal 1 can be calculated by a single thread of execution or multiple threads of execution. The approach here is to use a single thread for calculating each term of the diagonal. Once diagonal 1 is computed one can proceed to compute diagonal 2, however CUDA doesn't guarantee to maintain order of execution of blocks of threads. This can lead to issues related to synchronization. To eliminate this problem multiple kernel calls are issued, one after another completes. Each kernel call is issued for computing each diagonal in order. This addresses issues related to synchronization.

Here is the algorithm

Algorithm 1.

```
__global__ void minMultGPU(int diag, int *d, int *M, int lim)
{
    int t = (blickDim.x * blickIdx.x) + threadIdx.x;
    int tid = -1;
    if(t < lim)
        {
          for(;tid < lim; tid += blockDim.x)
           {
              int  j = tid + diag;
              int k = tid;
              int dt = d[tid-1]*d[j];
              int min = M[tid*N + k] + M[(k+1)*N + j] + dt* d[k];
              k++;
              while(k <= j-1)
               {
                  int temp= M[tid*N + k] + M[(k+1)*N+j]+dt*d[k];
                  if(temp < min)
                       min = temp;
                  k++;
               }
              M[tid*N + j] = min;
           }
        }
}
```

This kernel is invoked from host code where *diag* is the diagonal being computed, *d* is the array containing dimensions of matrices and *lim* is used to set the number of elements to be computed in the diagonal.

4.2 Programming Effort For Developing Parallel Algorithm.

This algorithm is as close to as close to a CPU based implementation as possible. This will help in assessing the effort needed for developing this parallel algorithm.

If algorithm 1 is observed, it closely resembles a CPU based implementation but once a kernel is invoked, thousands of instances of the kernel run simultaneously on the GPU. This brings into picture the inherent difficulty in parallel algorithms. A close look at algorithm 1 reveals that, there is a separate check *(if(t < lim))* to limit the number of threads that compute the elements in the diagonal, this type of check is common in almost all the algorithms running on a GPU as it prevents unnecessary computations. All the threads are executing in parallel, this can be visualized as a separate processor computing each element in the diagonal being computed so each instance will have its separate local variables making debugging difficult when compared to a CPU based implementation.

Besides all that is stated above there are difficulties posed by the architecture of the GPU. The cache memory in GPU doesn't work transparently in compute based algorithms as is the case in CPUs. If caches are to be used in GPU different techniques are needed to use them explicitly, shared memory, registers, constant memory, textures are used in GPU to gain benefits of cache memory.

considering what has been stated in the above paragraph, the algorithm 1 is not using any of these techniques like shared memory, textures, constant memory to speed up the given algorithm further. The use of these techniques no doubt increases the speed of given algorithms manifold in certain instances but using them in any algorithm increases the amount of programming effort required to write that algorithm. Since all these memories are limited, the algorithm to use these memories requires techniques like tiling and pre caching data which introduces further complexity in any algorithm. This involves reading from off chip DRAM into shared memory and then writing back into the off chip DRAM multiple times or use of certain access patterns like coalescing to reduce memory traffic and making the GPU use its cache memory.

Once algorithm is optimized for GPU, carrying out each optimization adds to the amount of time needed to develop the algorithm, increases the debugging effort required and introduces additional complexity into the algorithm.

Synchronization becomes a major issue in parallel algorithms. In the given dynamic programming algorithm this issue can be prominent if you try to synchronize blocks of threads executing in parallel. To avoid these issues one simple solution is to resort to issuing separate kernel calls to calculate diagonals one after another is computed.

5. EXPERMENTAL RESULTS

Nvidia GeForce GTX 570 with 480 processing cores(15 Streaming multiprocessors with 32 cores each) and 1.25GB GDDR5 DRAM has been used for implementing the dynamic programming algorithm for chained matrix multiplication presented in this paper. For the purpose of estimating speedup of a GPU based implementation, a CPU based implementation has also been developed. An Intel core i5 760 running at ~3.3 GHz and 8GB RAM is used to implement sequential version of chained matrix multiplication algorithm using dynamic programming.

Table 1 shows computing time in seconds for matrix chains of size 2500,3000,3500,4000,4500, 5000

Table 1. The computing time on GPU and CPU in milliseconds for different input instances where input size is the number of matrices to be multiplied.

Table 1.

Input Size	2500	3000	3500	4000	4500	5000
Time on GPU	7354.3	11232	16867.2	22577.7	31098.2	41045.9
Time on CPU	25860	43066	76067	102872	173508	226978

Table 1 shows us, that for small instances of chained matrix multiplication GPU implementation is three times faster on average than CPU based implementation. Once larger instances are considered the GPU based implementation is 5.5 to 6 times faster than CPU based implementation. So a best possible speed up factor of 6 is possible for large instances of chained matrix multiplication using the given implementation.

6. CONCLUSION

In this paper an implementation of dynamic programming algorithm for chained matrix multiplication on GPU has been proposed. This implementation closely resembles a CPU based serial implementation, so that minimum amount of additional programming effort is required. Though this algorithm requires minimum additional effort, still it manages to be six times faster than a CPU based implementation. This proves that even a small amount of work on our part can achieve significant amount of speed up on current generation of GPUs. In this case up to six times as compared to CPU based implementation However the speed of these algorithms can significantly increase if techniques like tiling, pre caching and memo1ry coalescing are used but the use of these techniques can be involving and require more effort on part of the programmer.

7. REFERENCES

[1] Cormen, T. H., Lieserson, C.E., Rivest, R.L. 1990 *Introduction to Algorithms*. MIT Press

[2] Neapolitan, R and Naimipour, K. 2003 Foundations of Algorithms using C++ pseudocode.

[3] Nvidia Corp. 2011 Nvidia CUDA programming guide version 4.1.

[4] Nvidia Corp. 2011 CUDA C Best Practices Guide version 4.1.

[5] W. W. Hwu. 2011 *GPU* Computing Gems Emerald *Edition*. Morgan Kaufmann,.

[6] D. Man, K. Uda, Y. Ito, and K. Nakano. Dec. 2011 "A GPU implementation of computing euclidean distance map with efficient memory access," in Proc. of International Conference on Networking and Computing, , pp. 68–76.

[7] A. Uchida, Y. Ito, and K. Nakano. Dec. 2011 "Fast and accurate template matching using pixel rearrangement on the GPU," in Proc. of International Conference on Networking and Computing , pp. 153–159.

[8] AMD. 2011 Introduction to OpenCL programming.