**ABSTRACT**

The dynamic programming paradigm involves various important optimization problems. The set of optimization problems include optimal binary search tree, longest common subsequence, binary knapsack, Matrix chain multiplication (MCM), and many more.

Modern GPUs (Graphics Processing Units) can be used for general purpose parallel computation. One platform for doing so is NVIDIA’s Compute Unified Device Architecture, or CUDA. The example of Matrix Chain Multiplication has been used to introduce the basics of GPU computing in the CUDA environment.

The Matrix Chain Product Problem is an optimization problem for finding parenthesis of the matrix chain that gives the minimum total number of multiplications necessary to compute the product of the matrix chain. Dynamic programming can be used to solve the problem of optimal matrix parenthesization. The results and their analysis reveal that there is considerable amount of time reduction compared with simple left to right multiplication, on applying the matrix parenthesization algorithm and parallel programming. Time reduction varies from 0% to 96%, proportional to the number of matrices and the sequence of dimensions.

.

1. **INTRODUCTION**

Matrix multiplication is a central operation in many numerical algorithms. Its applications are found in many fields including scientific computing, pattern recognition and problems such as counting the paths through a graph. The running time of square matrix multiplication is O(n3) and the running time for multiplying rectangular matrices (one m × p matrix with one p × n matrix) is O(mnp). However, more efficient algorithms exist. Improving the matrix multiplication algorithm can improve the performance of many of those applications. Matrix multiplication is not commutative, but it is associative, so in case of chain matrix multiplication, the chain can be parenthesized in whatever manner deemed best without changing the final product. Using parallel and distributed systems, where the computational work is spread over multiple processors is one way to achieve this. Here, Parallel Computing Model is used because concurrent execution reduces execution time. The parallel computing model is implemented through CUDA language which is a parallel programming language, in order get the full utilization of GPU (Graphics Processing Unit).

* 1. **Introduction to Matrix Chain Multiplication**

In mathematics, matrix multiplication or the matrix product is a binary operation that produces a matrix from two matrices. The definition of matrix multiplication is that if C = AB for an n × m matrix A and an m × p matrix B, then C is an n × p matrix with entries

The basic matrix multiplication algorithm takes time Θ(nmp). If the inputs are all square matrices of size n × n, the running time is Θ(n3), i.e., cubic.

The Matrix Chain Multiplication is the process of multiplying matrices A x B x C x D x . . . ., cumulatively one after the other. Matrix chain multiplication is used in applications such as signal processing.[1]

Each matrix has fixed number of rows and columns and for the multiplication to be feasible, the number of rows of first matrix must be equal to number of columns of second matrix.

Pseudo code to check compatibility of matrices for chain matrix multiplication is as follows:

|  |
| --- |
| *//to check whether the matrices can be multiplied or not*  int compatible(int r[],int c[],int n) *//r[i] and c[i] have no. of rows & columns for matrices i=1,2,…,n*  {  int i;  for(i=0;i<n;i++)  {  if(r[i+1]!=c[i])    return 0;  }  return 1;  } |

Pseudo code for checking feasibility of matrix chain multiplication

Chain Matrix Multiplication gives another problem definition i.e., given a sequence of matrices, finding the most efficient way to multiply these matrices. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

There are many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result will be the same. For example, if there are four matrices say A, B, C, and D, then:

(ABC)D = (AB)(CD) = A(BCD) = ....

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

Suppose, given a list of n matrices. Let us attack the problem with brute-force approach and try all possible parenthesizations. As a result, the number of ways of parenthesizing an expression is very large. For instance, if there is just one item in the list, then there is only one way to parenthesize. Similarly, if there are n items in the list, then there are n − 1 places where the list could be split with the outermost pair of parenthesis, namely just after first item, just after the second item, and so on and so forth, and just after the (n −1)th  item in the list.

On the other hand, when the list is split just after the kth item, it creates two sub-lists to be parenthesized, one with k items, and the other with n − k items. After splitting, consider all the ways of parenthesizing these sub-lists (brute force in action). If there are L ways to parenthesize the left sub-list and R ways to parenthesize the right sub-list and since these are independent choices, then the total is L times R. This suggests the following recurrence for P (n), the number of different ways of parenthesizing n items:

On implementing this algorithm, it is discovered that it is just as slow as the naive way of trying all permutations. Here, a lot of redundant work is being done. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

One simple solution is called memorization: the minimum cost needed to multiply out a specific subsequence is computed and saved each time. If the computed answer is needed again, the saved value is provided without re-computing. Since there are about n2/2 different sub-sequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down to O(n3) from O(2n), which is more than efficient enough for real applications. This is top-down dynamic programming.

There are many different approaches which deal with speedup of Matrix Chain Multiplication. In this document the optimization process and the utilization of GPU have been discussed.

* 1. **GPU Architecture**

Graphics Processing Unit (GPU) is the widely used core technology in multimedia, gaming and graphics based applications, in-order to achieve high performance and fast calculations. The GPU is faster because it is embedded with several cores which can execute concurrently. It also takes the advantage of CPU for task-scheduling and other OS related supporting applications and only concentrates on computing

The CUDA model is worth using for a specific hardware GPU, because the programming language helps to run specific parts of code which are hot-spots in GPU by restricting the compiler.

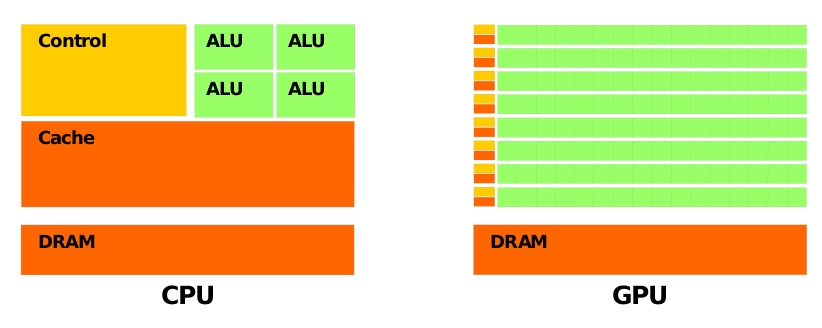


Fig 1.1: Fundamental Difference between CPU and GPU architectures

GPU computing is the use of a GPU (graphics processing unit) together with a CPU to accelerate general-purpose scientific and engineering applications. It offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run significantly faster.

If a problem has a small data size, sophisticated control logic, and/or low-level parallelism, the CPU is a good choice because of its ability to handle complex logic and instruction-level parallelism. If the problem at hand instead processes a huge amount of data and exhibits massive data parallelism, the GPU is the right choice because it has a large number of programmable cores, can support massive multi-threading, and has a larger peak bandwidth compared to the CPU

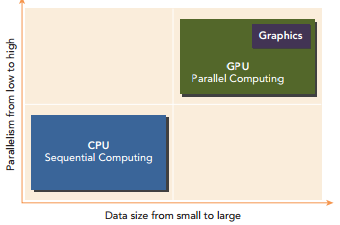


Fig 1.2: Dimensions that differentiate the scope of applications for CPU and GPU

CPU & GPU is a powerful combination because CPUs consist of a few cores optimized for serial processing, while GPUs consist of thousands of smaller, more efficient cores designed for parallel performance. Serial portions of the code run on the CPU while parallel portions run on the GPU.

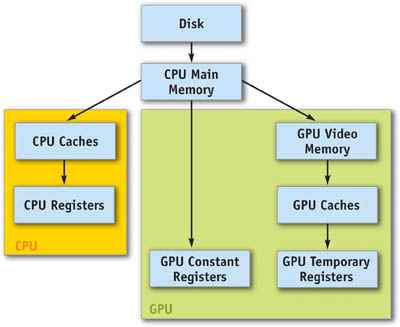


Fig 1.3: GPU Communications with CPU and Memory

* 1. **Parallel Programming Languages**

Parallel programming languages are languages designed to program algorithms and applications on parallel computers. Parallel processing is a great opportunity for developing high performance systems and for solving large problems in many application areas. Many models and languages have been designed and implemented to allow the design and development of applications on parallel computers. Parallel programming languages (called also concurrent languages) allow the design of parallel algorithms as a set of concurrent actions mapped onto different computing elements.

**Process interaction:**

Process interaction relates to the mechanisms by which parallel processes are able to communicate with each other. The most common forms of interaction are shared memory and message passing, but interaction can also be implicit (invisible to the programmer).

**Shared memory (inter-process communication)**

Shared memory is an efficient means of passing data between processes. In a shared-memory model, parallel processes share a global address space where they read and write asynchronously. Asynchronous concurrent access can lead to [race conditions](https://en.wikipedia.org/wiki/Race_condition) and mechanisms such as [locks](https://en.wikipedia.org/wiki/Lock_(computer_science)), [semaphores](https://en.wikipedia.org/wiki/Semaphore_(programming)) and [monitors](https://en.wikipedia.org/wiki/Monitor_(synchronization)) can be used to avoid these. Conventional [multi-core processors](https://en.wikipedia.org/wiki/Multi-core_processor) directly support shared memory, which many parallel programming languages and libraries, such as [Cilk](https://en.wikipedia.org/wiki/Cilk_(programming_language)), [OpenMP](https://en.wikipedia.org/wiki/OpenMP) and [ThreadingBuildingBlocks](https://en.wikipedia.org/wiki/Threading_Building_Blocks), are designed to exploit.

**Message passing**

In a message-passing model, parallel processes exchange data by passing messages to one another. These communications can be asynchronous, where a message can be sent before the receiver is ready, or synchronous, where the receiver must be ready. The [Communicating sequential processes](https://en.wikipedia.org/wiki/Communicating_sequential_processes) (CSP) formalisation of message passing uses synchronous communication channels to connect processes, and led to important languages such as [Occam](https://en.wikipedia.org/wiki/Occam_(programming_language)), [Limbo](https://en.wikipedia.org/wiki/Limbo_(programming_language)) and [Go](https://en.wikipedia.org/wiki/Go_(programming_language)). In contrast, the [actor model](https://en.wikipedia.org/wiki/Actor_model) uses asynchronous message passing and has been employed in the design of languages such as [D](https://en.wikipedia.org/wiki/D_(programming_language)), [Scala](https://en.wikipedia.org/wiki/Scala_(programming_language)) and [SALSA](https://en.wikipedia.org/wiki/SALSA_(programming_language)).

**Implicit interaction**

In an implicit model, no process interaction is visible to the programmer and instead the compiler and/or runtime is responsible for performing it. Two examples of implicit parallelism are with [domain-specific languages](https://en.wikipedia.org/wiki/Domain-specific_language) where the concurrency within high-level operations is prescribed, and with [functional programming languages](https://en.wikipedia.org/wiki/Functional_programming) because the absence of [side-effects](https://en.wikipedia.org/wiki/Side_effect_(computer_science)) allows non-dependent functions to be executed in parallel.[5] However, this kind of parallelism is difficult to manage and functional languages such as [Concurrent Haskell](https://en.wikipedia.org/wiki/Concurrent_Haskell) and [Concurrent ML](https://en.wikipedia.org/wiki/Concurrent_ML) provide features to manage parallelism explicitly.

**Problem decomposition:**

A parallel program is composed of simultaneously executing processes. Problem decomposition relates to the way in which the constituent processes are formulated.

**Task parallelism**

A task-parallel model focuses on processor threads of execution. These processes will often be behaviourally distinct, which emphasises the need for communication. Task parallelism is a natural way to express message-passing communication. In [Flynn's taxonomy](https://en.wikipedia.org/wiki/Flynn%27s_taxonomy), task parallelism is usually classified as [MIMD](https://en.wikipedia.org/wiki/MIMD)/[MPMD](https://en.wikipedia.org/wiki/Flynn%27s_taxonomy) or [MISD](https://en.wikipedia.org/wiki/MISD).

**Data parallelism**

A data-parallel model focuses on performing operations on a data set, typically a regular structured array. A set of tasks will operate on this data, but independently on disjoint partitions. In Flynn's taxonomy, data parallelism is usually classified as MIMD/SPMD or SIMD.

In the actor model there are languages called SALSA, Smalltalk, In the coordination languages Millipede, In data flow languages CAL, SISAL, In distributed Computing model Julia, Limbo languages are there, and In hardware environment Verilog, VHDL. Even in Java we have Multi-thread model in order to achieve parallelism, and other API frameworks such as CUDA, OpenCL, OpenGL, In Object Oriented Programming Languages Java-Aparapiis the extended version of Java, Charm++, C#, C\*, and C++ AMP

This project is designed on the platform CUDA to get the ultimate performance of GPU. CUDA is based on Data parallel model. Details of the CUDA will be discussed in Section 3.1

* 1. **Problem Statement**

The problem statement is Parallel Implementation of Matrix Chain Multiplication algorithm using CUDA on GPU. The execution of Matrix Chain multiplication is much difficult task compared to normal Matrix Multiplication because of the size, so to speed up the process CUDA is being used.

**1.5. Organization of thesis**

The thesis is organized as follows: Chapter 2 discusses the related work done on Matrix Chain Multiplication and existing algorithms and different approaches on them, then Chapter 3 discusses on CUDA the platform which is used for the improvement in the program, optimisation process, the sequential implementation of Chain matrix multiplication and then the further improvement using the advantage of tiled Approach embedded in CUDA, then Chapter 4,5 discuss about Hardware requirements, Software Requirements and the Experimental setup for executing the code then the results and screenshots of executing program, and the graphs which represents the performance improvement and it ends with the conclusion and references

**2. LITERATURE SURVEY**

**Hu &Shing (1981)**

An algorithm published in 1981 by Hu and Shing achieves O(n log n) complexity. They showed how the matrix chain multiplication problem can be transformed (or [reduced](https://en.wikipedia.org/wiki/Reduction_(complexity))) into the problem of [triangulation](https://en.wikipedia.org/wiki/Polygon_triangulation) of a [regular polygon](https://en.wikipedia.org/wiki/Regular_polygon). The polygon is oriented such that there is a horizontal bottom side, called the base, which represents the final result. The other n sides of the polygon, in the clockwise direction, represent the matrices. The vertices on each end of a side are the dimensions of the matrix represented by that side. With n matrices in the multiplication chain there are n−1 [binary operations](https://en.wikipedia.org/wiki/Binary_operation) and Cn-1 ways of placing parenthesizes, where Cn-1 is the (n−1)th [Catalan number](https://en.wikipedia.org/wiki/Catalan_number). The algorithm exploits that there are also Cn-1 possible triangulations of a polygon with n+1 sides.

A regular [hexagon](https://en.wikipedia.org/wiki/Hexagon) has 14 possible triangulations. These correspond to the different ways that parentheses can be placed to order the multiplications for a product of 5 matrices.

For the example, consider a case which requires four sides: A, B, C and the final result ABC. A is a 10×30 matrix, B is a 30×5 matrix, C is a 5×60 matrix, and the final result is a 10×60 matrix. The regular polygon for this example is a 4-gon, i.e. a square.The matrix product AB is a 10x5 matrix and BC is a 30x60 matrix. There are two possible triangulations in this example: triangle having A, B as sides and a triangle having B, C as sides.

The cost of a single triangle in terms of the number of multiplications needed is the product of its vertices. The total cost of a particular triangulation of the polygon is the sum of the costs of all its triangles.

**Strassen’s algorithm**

Strassen introduced an algorithm to compute matrix chain multiplication with minimum number of scalar multiplications and additions compared to conventional matrix multiplication. Huss-Lederman developed an efficient and portable serial implementation of Strassen’s algorithm called Dynamic General Fast Matrix Multiplication (DGEFMM)[7]. DGEFMM is designed to replace Dynamic General Matrix Multiplication(DGEMM) with better performance for all matrix sizes while minimizing the temporary storage. DGEMMS in the IBM ESSL library implements Strassen’s algorithm out-performs the multiplication part of DGEMM,C = op(A) \_ op(B)[8].Various parallel implementation methods are proposed to implement Strassen’s algorithm on distributed memory architectures. Broadcast-Multiply-Roll (BMR) method introduced is performed in two stages using both sequential and parallel approach. In 2004, Ohtaki is focused on a distribution scheme particularly for heterogeneous clusters. Chou proposed method to divide the given matrix into 2\_2 blocks of sub-matrices, then further divide each sub-matrix into four 2\_2 blocks(i.e., 4 \_ 4 blocks) and identified 49 multiplications and with 7 or 49 processors to perform multiplications concurrently. In [9], author proposed the Strassen-BMR method with both the Strassen’s and BMR to achieve concurrency. Grayson developed parallel Strassen’s algorithm for a square mesh of nodes. Desprez[10] simultaneously exploits data and task parallelism employing Strassens algorithm into two disjoint sets.

1. **GPU BASED MATRIX CHAIN MULTIPLICATION USING CUDA**

This section discusses about the solution for the problem of Parallel Implementation of Matrix chain Multiplication, the platform that is used to achieve it i.e. CUDA. The CUDA model is worth using for a specific hardware GPU, and its architecture, and further improvements.

**3.1. CUDA [Compute Unified Device Architecture]: A Platform for Heterogeneous Computing**

CUDA is one of the most popular application programming interfaces for accelerating a range of compute kernels on the GPU. It can enable code written in C or C++ to run efficiently on a GPU with very reasonable programming effort. It strikes a balance between the need to know about the architecture in order to exploit it well, and the need to have a programming interface that is easy to use and results in readable programs.

CUDA is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way. Using CUDA, one can access the GPU for computation, as has been traditionally done on the CPU. The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces, and extensions to industry-standard programming languages, including C, C++, Fortran and Python. A CUDA program consists of a mixture of the following two parts: The host code runs on CPU. The device code runs on GPU. NVIDIA’s CUDA nvcc compiler separates the device code from the host code during the compilation process. The host code is standard C code and is further compiled with C compilers. The device code is written using CUDA C extended with keywords for labelling data-parallel functions, called kernels. The device code is further compiled by nvcc. During the link stage, CUDA runtime libraries are added for kernel procedure calls and explicit GPU device manipulation.

The CUDA platform is also a foundation that supports a diverse parallel computing ecosystem, as shown in figure 3.1.

**CUDA Program Structure**

A typical CUDA program structure consists of six main steps:

1. Allocate GPU memories: The variables which have to be used on the device are allocated using cudaMalloc() function.

2. Copy data from CPU memory to GPU memory: Host to device variable information is copied using functions like cudaMemcpy().

3. CPU instructs the process to GPU: The CPU calls the device function by identifying or searching the code for a function which is declared as global.

4. GPU executes parallel in each core: Parallel execution is performed with the help of blocks and threads in each block.

5. Copy data back from GPU memory to CPU memory: Device to host variable information is copied using functions like cudaMemcpy().

6. Destroy GPU memories: The memory allocated to device variables is destroyed using cudaFree() function.

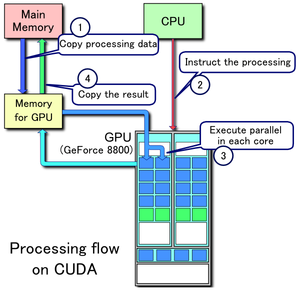
****

Fig 3.1: Processing flow on CUDA

**Managing Memory**

Host and Device Memory Functions:[6]

|  |  |
| --- | --- |
| STANDARD C FUNCTIONS | CUDA C FUNCTIONS |
| Malloc | cudaMalloc |
| Memcpy | cudaMemcpy |
| Memset | cudaMemset |
| free | cudaFree |

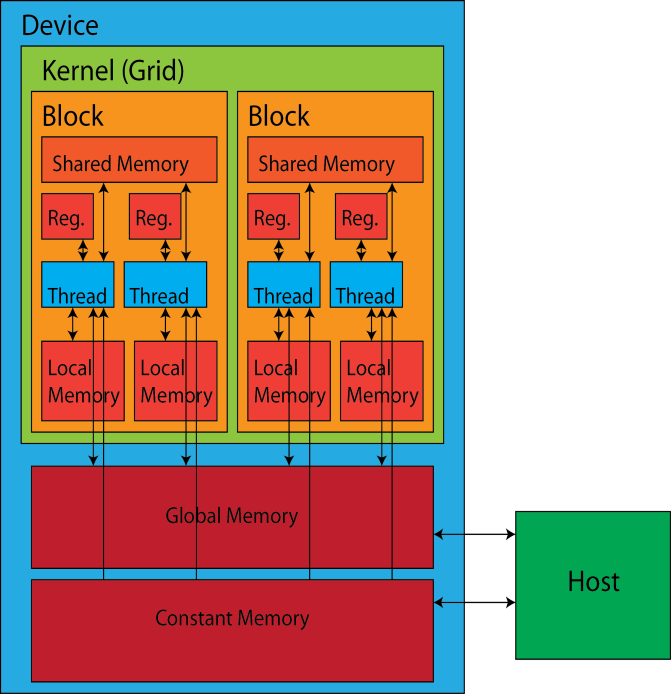


Fig 3.2: Memory Hierarchy

**Organizing Threads**

When a kernel function is launched from the host side, execution is moved to a device where a large number of threads are generated and each thread executes the statements specified by the kernel function. Knowing how to organize threads is a critical part of CUDA programming. CUDA exposes a thread hierarchy abstraction to enable you to organize your threads. This is a two-level thread hierarchy decomposed into blocks of threads and grids of blocks, as shown in figure below.

Threads rely on the following two unique coordinates to distinguish themselves from each other:

1. blockIdx (block index within a grid)

2. threadIdx (thread index within a block)

**Access Grid/Block Variables from the Host and Device Side**

It is important to distinguish between the host and device access of grid and block variables. For example, using a variable declared as block from the host, you define the coordinates and access them as follows: block.x, block.y, and block.z. On the device side, you have pre-initialized, built-in block size variables available as: blockDim.x, blockDim.y, and blockDim.z. In summary, variables for grid and block are defined on the host before launching a kernel, and access them there with the x, y and z fields of the vector structure from the host side. When the kernel is launched, you can use the pre-initialized, built-in variables within the kernel.

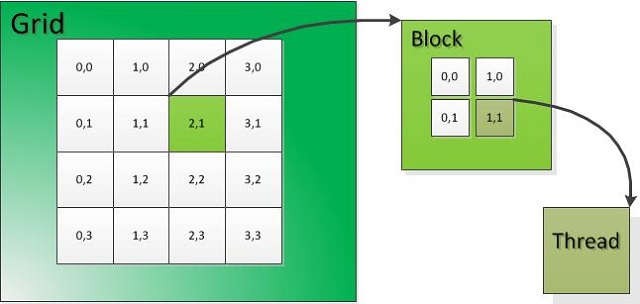
****

Fig 3.3: Two-level thread hierarchy decomposed into blocks of threads and grids of blocks

**Launching a CUDA kernel**

kernel\_name<<grid,block>>(argument list);

**CUDA Kernels are functions with the following restrictions**

1. Access to device memory only
2. Must have void return type
3. No support for a variable number of arguments
4. No support for static variables
5. No support for function pointers
6. Exhibit an asynchronous behaviour

**3.2. Optimization**

Consider the problem of evaluating the product of *n* matrices

M = M1 x M2 x M3 x.... Mn,

where Mi is a matrix of the order m x n and M(i+1) is a matrix of order n x p. The product N = Mi x M(i+1) is a m x p matrix. This N can be computed in time O(mnp). For example, let there be 4 matrices named A, B, C, D of the order (2 x 3), (3 x 4), (4 x 5), (5 x 6) respectively. Now M = A x B x C x D. Since matrix multiplication is associative, the order in which the above chain multiplication is evaluated does not affect the final result. The matrix can be multiplied in the following orders: ((AB)C)D, (AB)(CD), A((BC)D), (A(BC))D, A(B(CD)).

The problem is not actually to perform the multiplication, but to decide the order in which multiplications needs to be performed. Because this order in which the product of matrices is parenthesized affects the number of simple arithmetic operations needed to compute the product, or the efficiency. The chain matrix multiplication problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in complier design for code optimization and in databases for query optimization. The number of arithmetic operations performed for all the above mentioned parenthesizations are:

((AB)C)D = 2x3x4 + 2x4x5 + 2x5x6 = 124

(AB)(CD) = 2x3x4 + 4x5x6 + 2x4x6 = 192

A((BC)D) = 3x4x5 + 3x5x6 + 2x3x6 = 186

(A(BC))D = 3x4x5 + 2x3x5 + 2x5x6 = 150

A(B(CD)) = 4x5x6 + 3x4x6 + 2x3x6 = 228

Clearly the first method is more efficient in all. This gives a picture as the number of operations termed as scalar multiplications is affected by the order in which the product is computed. Thus, the number of scalar operations required depends on optimal parenthesis order. Now in order to determine the optimal parenthesis order, one can proceed in many ways. One is brute force method where the number of operations of all the possible parenthesis order is calculated and the least amongst them is found. Alternative approach is the Dynamic Programming. The first step of the dynamic programming paradigm is to characterize the structure of an optimal solution. For the chain matrix problem, like other dynamic programming problems, involves determining the optimal structure (in this case, a parenthesization). The problem can be divided into sub-problems, whose solutions can be combined to obtain a solution to the global problem.

Tables 1 and 2 show the implementation of Matrix Paranthesization Algorithm. The number of matrices in the table 1 ranges from 1 to 10 and the Sequence of Dimensions ranges from 1 to 10. The number of matrices in the table 2 ranges from 1 to 24 and the Sequence of Dimensions ranges from 1 to 100. The tables show the optimal order for these multiplications, the cost for multiplying the matrices from left to right, the cost for multiplying in optimal order and the reduction in cost due to optimisation.

**Table 3.1: Implementation of Matrix Parenthesization Algorithm**

**No. Of Matrices: 1-10, Sequence of Dimensions: 1-10**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No. of**  **Matrices** | **Sequence of**  **Dimensions** | **Optimal**  **Arithmetic Multiplications** | **Left to**  **Right Multiplica- tions** | **Optimal**  **Parenthesizations** | **%**  **Reduction of**  **Cost**  **(d-c)/d\*100** |
| **A** | **b** | **c** | **D** | **e** | **f** |
| 3 | 6,4,6,6 | 288 | 360 | A(BC) | 20 |
| 4 | 8,4,7,3,3 | 216 | 464 | A((BC)D) | 53 |
| 5 | 9,10,6,6,8,3 | 702 | 1512 | A(B(C(DE))) | 54 |
| 6 | 7,4,4,1,7,6,9 | 203 | 861 | (A(BC))((DE)F) | 76 |
| 7 | 7,4,2,8,9,6,7,8 | 616 | 1736 | (AB)((((CD)E)F)G) | 65 |
| 8 | 8,7,6,2,5,4,4,2,2 | 316 | 896 | A(B(C(((DE)(FG))H))) | 65 |
| 9 | 5,2,2,8,3,1,4,5,4,1 | 100 | 475 | A(B((C(DE))(F(G(HI)))) | 79 |

Consider the example of row 5 in Table 1 which consists of multiplication of 5 matrices say A, B, C, D and E. Its dimensions are 9 x 10, 10 x 6, 6 x 6, 6 x 8 and 8 x 3 respectively. They are displayed in the second column as 9,10,6,6,8,3. Its optimal order is A(B(C(DE))) which requires 702 multiplications while the left to right multiplication i.e., ((((AB)C)D)E) requires 1512 multiplications. Thus the reduction of cost is [(1512-702)/1512]\*100=54%

**Table 3.2: Implementation of Matrix Parenthesization Algorithm**

**No. of Matrices: 1-24, Sequence of Dimensions: 1-100**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No. of**  **Matrices** | **Sequence of Dimensions** | **Optimal Arithmetic Multiplications** | **Left to Right Multiplications** | **Optimal**  **Parenthesizations** | **%**  **Reduction of Cost**  **(d-c)/d\*100** |
| **A** | **b** | **c** | **D** | **e** | **f** |
| 3 | 9,95,21,78 | 32697 | 32697 | (AB)C | 0 |
| 6 | 30,10,71,58,9,25,22 | 56982 | 183750 | A((B(CD))(EF)) | 69 |
| 9 | 94,67,56,17,80,68,10,78,7,5 | 98220 | 1273230 | A(B(C(D(E(F((GH)I)))))) | 92 |
| 12 | 42,54,49,22,62,46,93,97,82,59,  24,86,56 | 970214 | 1777734 | ((A(BC))((((((DE)F)G)H)I)J)  )(KL) | 45 |
| 15 | 27,98,89,40,36,82,6,11,3,23,15,  91,87,35,3,43 | 101322 | 816480 | (A(B(C(D(E(F((GH)((IJ)(K(  L(MN)))))))))))O | 88 |
| 18 | 94,30,63,79,52,10,6,13,93,97,3,  8,67,40,38,6,89,61,71 | 139845 | 3518984 | (A(B(C(D(E(F(G(H(IJ))))))))  )(((((((KL)M)N)O)P)Q)R) | 96 |
| 21 | 57,92,76,77,28,13,47,27,3,67,8  9,14,93,16,24,34,14,83,89,92,  33,19 | 166938 | 2827257 | (A(B(C(D(E(F(GH)))))))(((((  (((((((IJ)K)L)M)N)O)P)Q)R) S)T)U) | 94 |
| 24 | 79,68,62,22,98,35,62,99,21,39,  91,79,81,31,11,4,87,90,90,72,5  7,92,36,72,59 | 377216 | 6688377 | (A(B(C(D(E(F(G(H(I(J(K(L(  M(NO))))))))))))))((((((((PQ) R)S)T)U)V)W)X) | 94 |

The following bottom-up approach computes, for each 2 ≤ k ≤ n, the minimum costs of all sub-sequences of length k, using the costs of smaller sub-sequences already computed. It has the same asymptotic runtime and requires no recursion.

|  |
| --- |
| *// Matrix A[i] has dimension dims[i-1] x dims[i] for i = 1..n*  MatrixChainOrder(int dims[])  {  *// length[dims] = n + 1*  n = dims.length - 1;  *// m[i,j] = Minimum number of scalar multiplications (i.e., cost)*  *// needed to compute the matrix A[i]A[i+1]...A[j] = A[i..j]*  *// The cost is zero when multiplying one matrix*  **for** (i = 1; i<= n; i++)  m[i, i] = 0;  **for** (len = 2; len<= n; len++) { *// Subsequence lengths*  **for** (i = 1; i<= n - len + 1; i++) {  j = i + len - 1;  m[i, j] = MAXINT;  **for** (k = i; k <= j - 1; k++) {  cost = m[i, k] + m[k+1, j] + dims[i-1]\*dims[k]\*dims[j];  **if** (cost < m[i, j]) {  m[i, j] = cost;  s[i, j] = k; *// Index of the subsequence split that achieved minimal cost*  }  }  }  }  } |

Pseudo code for Optimal order of Matrix Chain Multiplication

**3.3. Sequential Implementation of Matrix Chain Multiplication**

The Conventional matrix chain multiplication process is shown in Figure 3.4 and proposed approach is shown below

****

Figure 3.4: Sequential Execution Process Flow Diagram

|  |
| --- |
| extern long long int\* mul(long long int \*mat, long long int \*mat2, long long int r1, long long int c1, long long int c2, long long int \*count)  {  long long int i, j, k;  long long int s = 0;  long long int \*res = (long long int \*)malloc(r1 \* c2 \* sizeof(long long int));  for (i = 0; i<r1\*c2; i++)  res[i] = 0;  for (i = 0; i<r1; i++)  {  for (j = 0; j<c1; j++)  {  for (k = 0; k<c2; k++)  {  s= mat[i\*c1 + j] \* mat2[j\*c2 + k];  res[i\*c2 + k] += s;  }  }  }  mat = (long long int \*)realloc(mat, r1 \* c2 \* sizeof(long long int));  for (i = 0; i<r1; i++)  {  for (j = 0; j<c2; j++)  {  mat[i\*c2 + j] = res[i\*c2 + j];  }  }  return mat;  } |

Serial Code for Matrix Chain Multiplication

* 1. **Parallel Implementation of Matrix Chain Multiplication**
     1. **Process Flow**

****

Figure 3.5: Parallel Execution Process Flow Diagram

**I. Pre-Processing**: This function calculates the number of multiplications for each and every order of the given matrices. The order of multiplication of matrices with minimum number of multiplication operations is chosen for GPU computing.

**II. Parallel Function:**  Parallel Execution takes place using Data-level parallelism in GPUs concurrently so as to accelerates the speedup. GPU processing is as shown below.



Figure 3.6: GPU Processing

**III) Output:** Displays the resultant matrix and computation time.

* + 1. **Tiled Approach**

In the proposed Matrix Parallel Multiplication tiled multiplication approach has been used. Each multiplication is implemented simultaneously in separate thread and reducing the time. In CUDA, the time can be reduced even more by using tiled/block approach. In this, partitioning of the computation and matrix data as group is done. From the first matrix the vertical strip and from second matrix the horizontal strips are taken and the partial results are computed by that particular thread and then combined to get the results of one tile of resultant matrix. Likewise, all the tiles are computed to get final result.

Divide each matrix into m × m tiles. For simplicity, let us assume that n is a multiple of m. Each block computes a tile of the product matrix. Computing an m × m tile involves computing n/m products of m × m tiles and summing up the results.

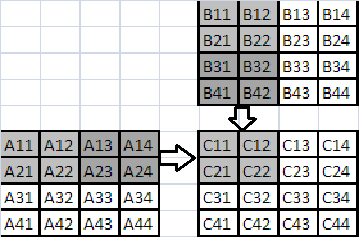


Fig 3.7: Tiled Approach

Tile size also makes the difference in time. Here (above fig) the tile size is 2 x 2 which makes the matrix perfectly divisible. If the size of the matrix is 1024 x 1024, the tile size of 256 x 256 makes the matrix perfectly divisible. There is a trade-off between the tile sizes. If the tile size is near to the size of matrix, then the computation is equal to without tiling and if the tile size is less, then the time complexity is higher. So the tile size has to be chosen carefully.

|  |
| --- |
| extern \_\_global\_\_ void MatrixMul(long long int\* A, long long int \* B, long long int \* C,  long long int numARows, long long int numAColumns, long long int numBColumns)  {  \_\_shared\_\_ long long int sA[32][32];   // Tile size of 32x32  \_\_shared\_\_  long long int sB[32][32];  long long int Row = blockDim.y\*blockIdx.y + threadIdx.y;  long long int Col = blockDim.x\*blockIdx.x + threadIdx.x;  long long intCvalue = 0;  sA[threadIdx.y][threadIdx.x] = 0;  sB[threadIdx.y][threadIdx.x] = 0;  for (int k = 0; k < (((numAColumns - 1) / 32) + 1); k++)  {  if ((Row <numARows) && (threadIdx.x + (k \* 32)) <numAColumns)  {  sA[threadIdx.y][threadIdx.x] = A[(Row\*numAColumns) + threadIdx.x + (k \* 32)];  }  else  {  sA[threadIdx.y][threadIdx.x] = 0.0;  }  if (Col <numBColumns&& (threadIdx.y + k \* 32) <numAColumns)  {  sB[threadIdx.y][threadIdx.x] = B[(threadIdx.y + k \* 32)\*numBColumns + Col];  }  else  {  sB[threadIdx.y][threadIdx.x] = 0.0;  }  \_\_syncthreads();  for (int j = 0; j < 32; ++j)  {  Cvalue += sA[threadIdx.y][j] \* sB[j][threadIdx.x];  }  }  if (Row <numARows&& Col <numBColumns)  {  C[Row\*numBColumns + Col] = Cvalue;  }  } |

Parallel Code for Matrix Chain Multiplication

**4. SYSTEM REQUIREMENTS**

**4.1. Hardware Requirements**

The hardware requirements used in this project is as follows

* Processor : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz 3.40GHz
* RAM : 8 GB
* Hard Disk : 1TB
* GPU : NVIDIA Graphics card

**4.2. Software Requirements**

To execute the CUDA code we need to have some IDE and supporting drivers they are

* IDE : Visual Studio 2015 Enterprise
* Platform & API : CUDA 8.0
* Drivers : NVIDIA GeForce GT730 Graphics Drivers
* Driver Version : 21.21.13.6930
* Number of CUDA : 384

cores

**5. Experimental Results**

**5.1. Environmental Setup**

**Visual Studio**

Microsoft Visual Studio is an [integrated development environment](https://en.wikipedia.org/wiki/Integrated_development_environment) (IDE) from [Microsoft](https://en.wikipedia.org/wiki/Microsoft). It is used to develop [console](https://en.wikipedia.org/wiki/Console_application) and [graphical user interface](https://en.wikipedia.org/wiki/Graphical_user_interface) [applications](https://en.wikipedia.org/wiki/Application_software) along with [Windows Forms](https://en.wikipedia.org/wiki/Windows_Forms) or [WPF](https://en.wikipedia.org/wiki/Windows_Presentation_Foundation) applications, [web sites](https://en.wikipedia.org/wiki/Web_site), [web applications](https://en.wikipedia.org/wiki/Web_application), and [web services](https://en.wikipedia.org/wiki/Web_service) in both [native code](https://en.wikipedia.org/wiki/Native_code) together with [managed code](https://en.wikipedia.org/wiki/Managed_code) for all platforms supported by [Microsoft Windows](https://en.wikipedia.org/wiki/Microsoft_Windows), [Windows Mobile](https://en.wikipedia.org/wiki/Windows_Mobile), [Windows CE](https://en.wikipedia.org/wiki/Windows_CE), [.NET Framework](https://en.wikipedia.org/wiki/.NET_Framework), [.NET Compact Framework](https://en.wikipedia.org/wiki/.NET_Compact_Framework) and Microsoft Silverlight.

NVIDIA® Nsight™ Visual Studio Edition brings GPU computing into Microsoft Visual Studio (including VS2015 Community Edition). This application development environment for GPUs allows you to build, debug, profile and trace heterogeneous compute, graphics, and virtual reality applications built with CUDA, C/C++, OpenCL, DirectCompute, Direct3D, Vulkan API, OpenGL, and the Oculus SDK.

Visual Studio includes a [code editor](https://en.wikipedia.org/wiki/Code_editor) supporting [IntelliSense](https://en.wikipedia.org/wiki/IntelliSense) as well as [code refactoring](https://en.wikipedia.org/wiki/Code_refactoring) The integrated [debugger](https://en.wikipedia.org/wiki/Microsoft_Visual_Studio_Debugger) works both as a source-level debugger and a machine-level debugger. Other built-in tools include a forms designer for building [GUI](https://en.wikipedia.org/wiki/GUI) applications, [web designer](https://en.wikipedia.org/wiki/Web_designer), [class](https://en.wikipedia.org/wiki/Class_(computing)) designer, and [database schema](https://en.wikipedia.org/wiki/Database_schema) designer. It accepts plug-ins that enhance the functionality at almost every level—including adding support for [source-control](https://en.wikipedia.org/wiki/Source_control) systems (like [Subversion](https://en.wikipedia.org/wiki/Subversion_(software)) and [Visual SourceSafe](https://en.wikipedia.org/wiki/Visual_SourceSafe)) and adding new toolsets like editors and visual designers for [domain-specific languages](https://en.wikipedia.org/wiki/Domain-specific_language) or toolsets for other aspects of the [software development lifecycle](https://en.wikipedia.org/wiki/Software_development_lifecycle) (like the [Team Foundation Server](https://en.wikipedia.org/wiki/Team_Foundation_Server) client: Team Explorer).

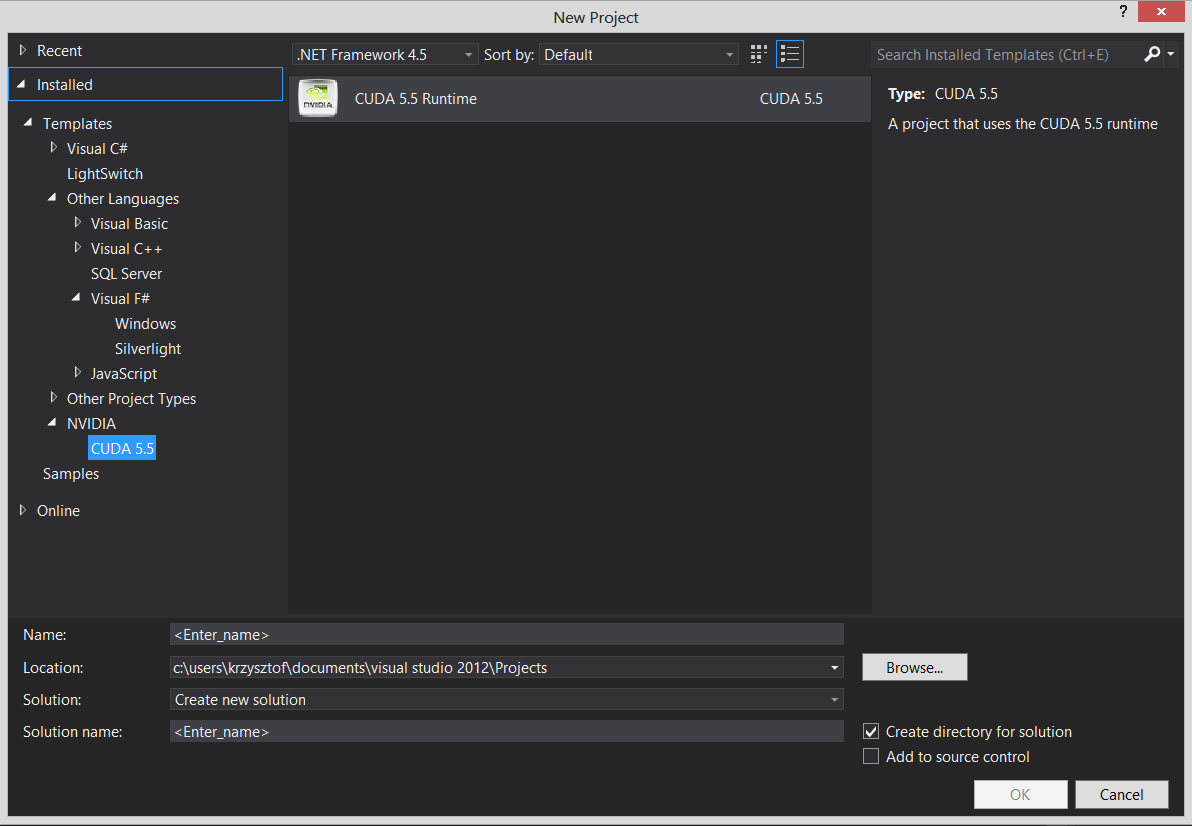


Fig 5.1: CUDA new project in Visual Studio

To multiply several matrices repeatedly and cumulatively we need to have matrices and the matrices should be taken in an order that they can be multiplied. As a rule of thumb two matrices can be multiplied if and only if the no. of columns of 1st matrix is equal to no. of rows in 2nd matrix. So we have to take the matrices in particular order.

Implementing the matrices in programming language requires 2 dimensional arrays. This program takes 5 matrices i.e. five 2 dimensional arrays but to avoid the problem of sending two dimensional matrices to the kernel multiplication function, it takes one dimensional arrays in which the matrix elements are stored using row major so that the array is logically a 2D array. The elements of the arrays are assigned randomly using rand() function available in C standard library. The following is a code snippet to show this:

|  |
| --- |
| int i, j, p=2; //p is equal to matrix no. plus 1  for (i = 0; i<r[0]; i++)  {  for (j = 0; j<c[0]; j++)  {  a1[i\*c[0] + j] = rand() % p;  }  } |

Since, the multiplication function can multiply two matrices at once but there are five matrices based on the order generated, so the following structure is used for easier manipulation.

|  |
| --- |
| typedef structll  {  char c; //represents matrix name; Ex: A or B or….  long long int \*mat;  long long intrv;  long longi nt cv;  structll \*next;  }node; |

To monitor the performance improvement, the following pattern has been used for generating no. of rows and columns. The matrices are generated randomly at the time of execution. Here, the initial value is 2000 and it is incremented by 3000 after every execution of the loop.

**Table 5.1: Matrices size input**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A(rows) | A(cols) | B(rows) | B(cols) | C(rows) | C(cols) | D(rows) | D(cols) | E(rows) | E(cols) |
| 1 | 2000 | 100 | 100 | 2000 | 2000 | 100 | 100 | 100 | 100 | 100 |
| 2 | 100 | 5000 | 5000 | 100 | 100 | 5000 | 5000 | 100 | 100 | 100 |
| 3 | 100 | 100 | 100 | 8000 | 8000 | 100 | 100 | 8000 | 8000 | 100 |
| 4 | 100 | 100 | 100 | 100 | 100 | 11000 | 11000 | 100 | 100 | 11000 |
| 5 | 14000 | 100 | 100 | 14000 | 14000 | 100 | 100 | 100 | 100 | 100 |
| 6 | 100 | 17000 | 17000 | 100 | 100 | 17000 | 17000 | 100 | 100 | 100 |
| 7 | 100 | 100 | 100 | 20000 | 20000 | 100 | 100 | 20000 | 20000 | 100 |
| 8 | 100 | 100 | 100 | 100 | 100 | 2300 | 2300 | 100 | 100 | 2300 |
| 9 | 26000 | 100 | 100 | 26000 | 26000 | 100 | 100 | 100 | 100 | 100 |
| 10 | 100 | 29000 | 29000 | 100 | 100 | 29000 | 29000 | 100 | 100 | 100 |
| 11 | 100 | 100 | 100 | 32000 | 32000 | 100 | 100 | 32000 | 32000 | 100 |
| 12 | 100 | 100 | 100 | 100 | 100 | 35000 | 35000 | 100 | 100 | 35000 |

**5.2. Results**

In this when the code is executed, the result is displayed on the command prompt which prompts us for the number of matrices to be multiplied. It takes upto 5 Matrices.

It computes the optimal order for the chain matrix multiplication and displays the string. It then stores the parallel and serial multiplication time taken for all the optimal orders. The command prompt looks as shown in the screenshot below.

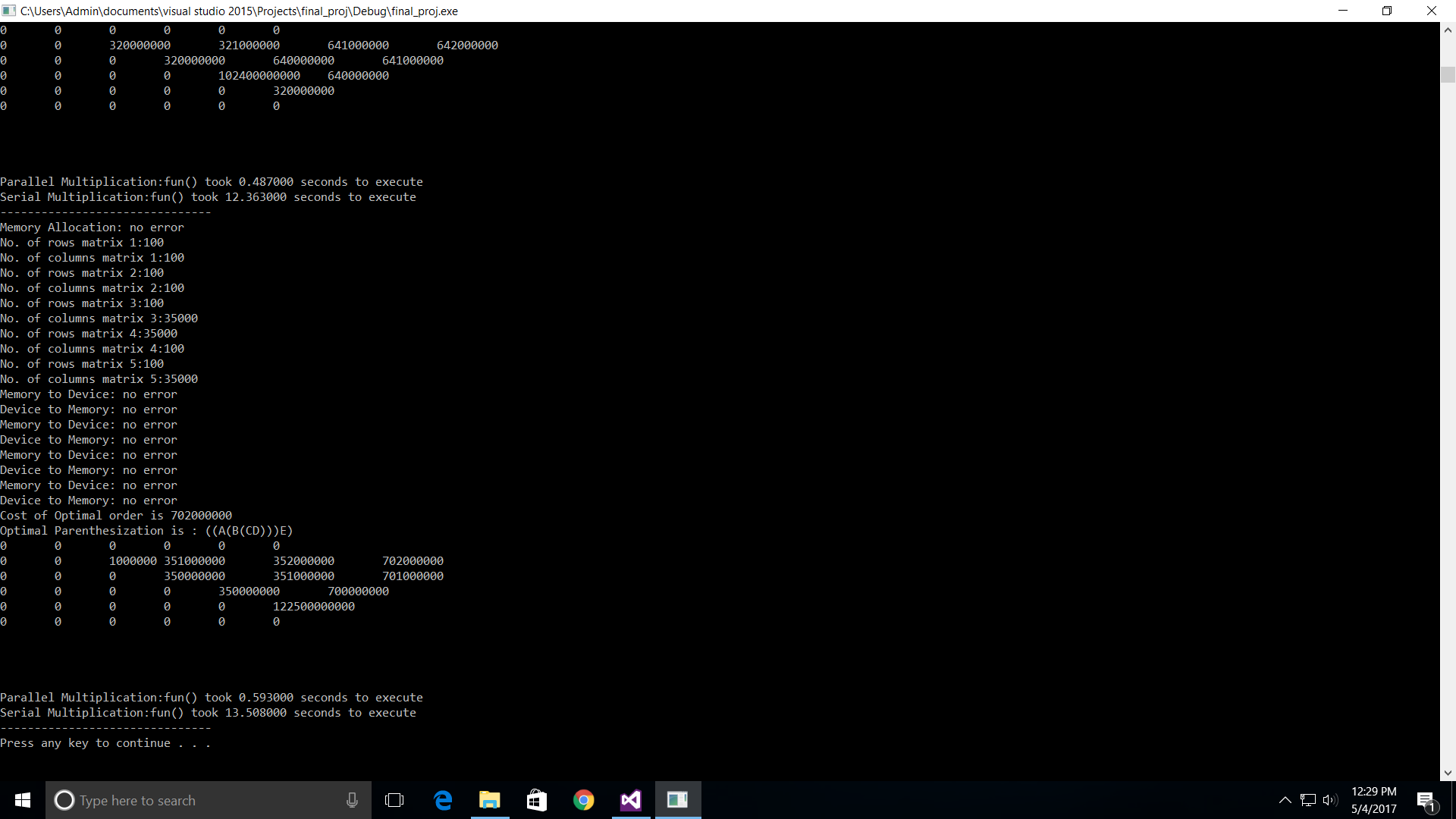


Fig 5.2: Results of Multiplication Operation

Matrix size varies from 100 to 35000

Screenshot shown in fig 5.2 displays the parallel and serial execution time for matrices of size:

A            100 X 100

B            100 X 100

C            100 X 35000

D            35000 X 100

E             100 X 35000

Serial Execution time (s): 13.508 sec

Parallel Execution time (p):0.593 sec

Speedup percentage = [(s-p)/s]\*100 =95.61%

After displaying the sizes of arrays, the command prompt shows whether there is an error while copying memory contents from host to device and device to host.

As mentioned in the table 5.1, the program calculates parallel and serial time for the optimal order, for each case in a loop. When profiler is used and CPU, GPU usage is selected, the following report is produced at the end of the execution. Figure 5.3 shows CPU and GPU usages.

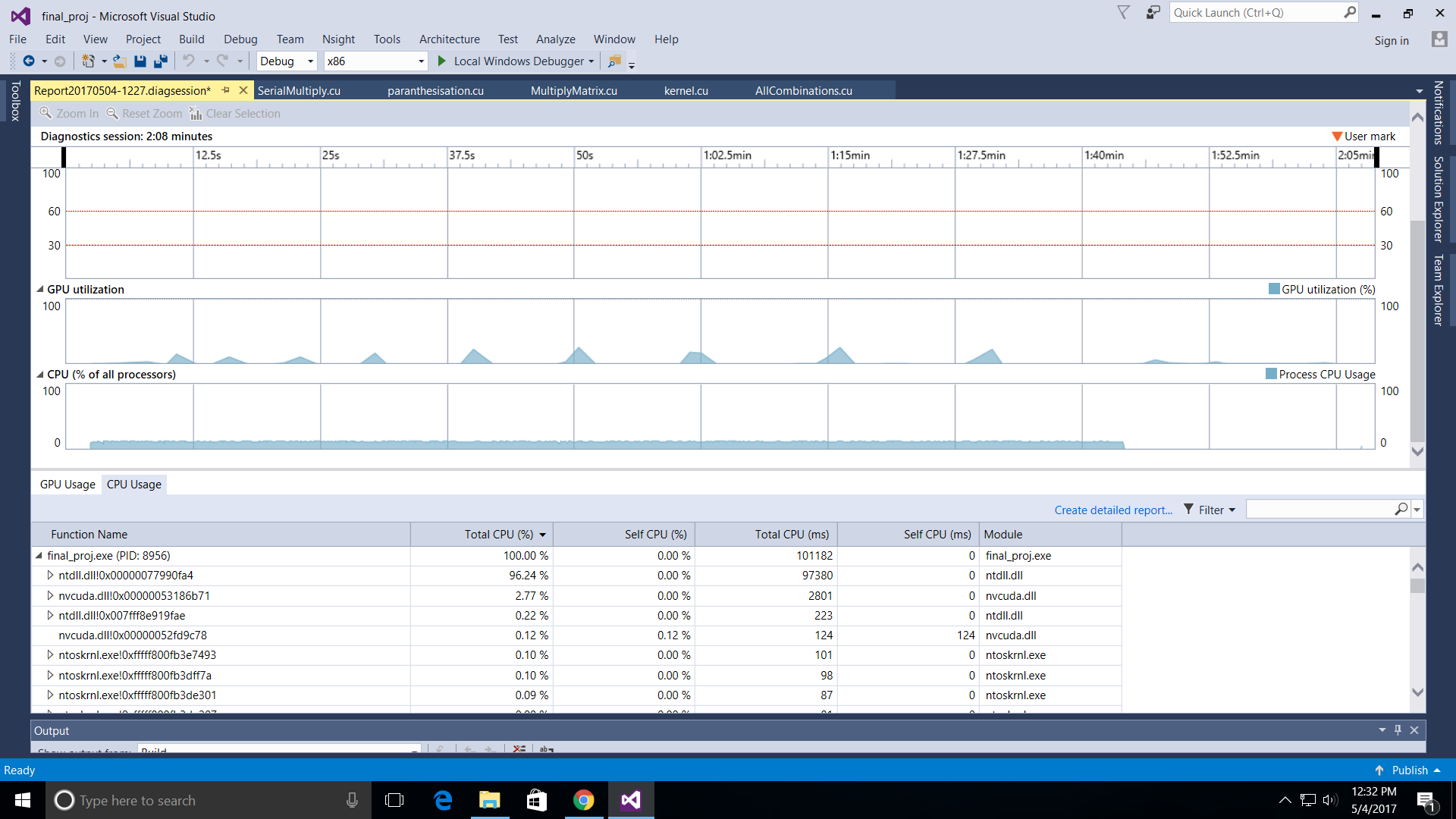
****

Fig 5.3: Usage statistics report for CPU vs GPU utilization

The peaks show the GPU utilisation each time the kernel is invoked for computing the matrix multiplication. The CPU utilisation remains constant during the kernel execution.

Figure 5.4 shows the detailed GPU utilization report from 74.31s to 77.11s.

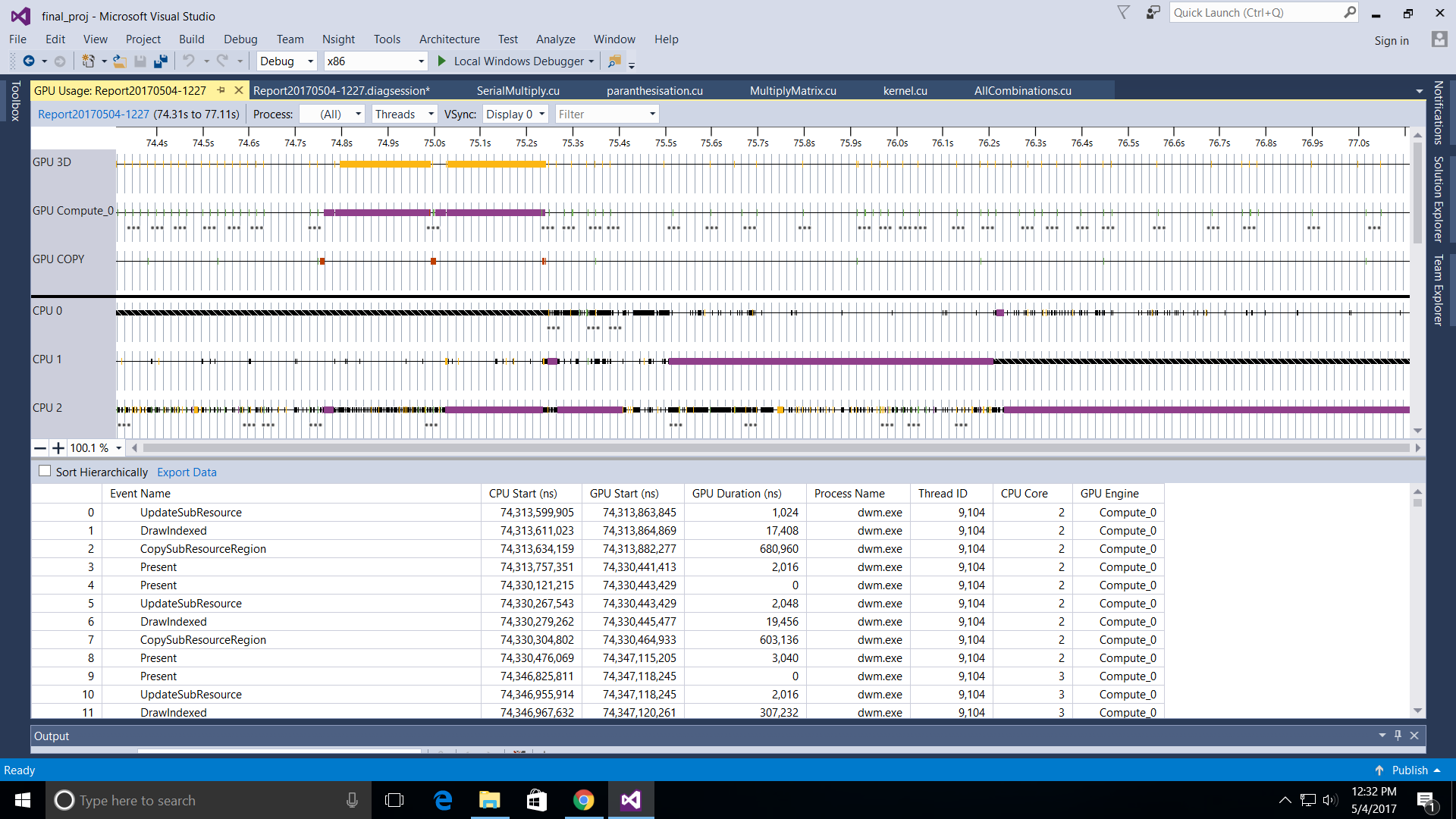
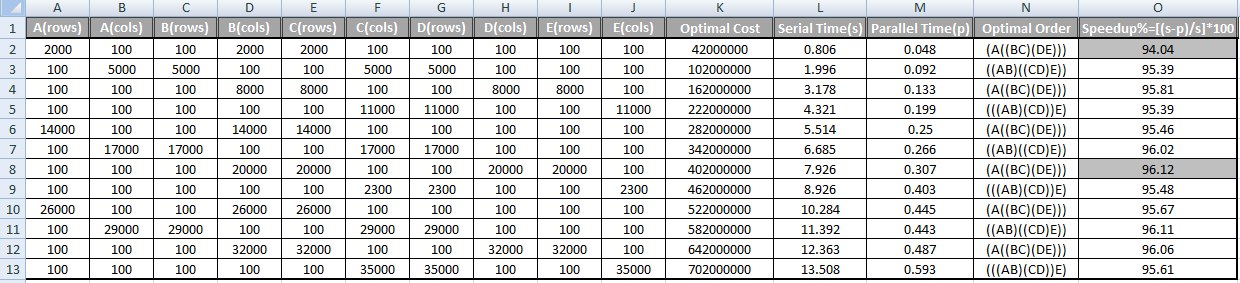


Fig 5.4: Detailed Usage statistics report for GPU Utilisation

Fig 5.4 shows the following details:

1. CPU Start: Shows the invocation time for a particular instruction (in nano sec)
2. GPU Start: Shows the invocation time of kernel.
3. ThreadId: Shows the kernel thread executing that particular event.
4. CPU Core: Shows the CPU core number being used.

**Serial Multiplication Vs Parallel Multiplication in Optimal Order:**

****Fig 5.5: Time Comparison Chart

From the fig 5.5. In each iteration, size of matrices is increased by 3000 alternatively.

Speedup percentage:

Min : 94.04%

Max : 96.12%

In the figure 5.6 the execution time comparison graph for serial and parallel executions. X-axis shows optimal multiplication cost for the 12 test cases which are incremented 3000 size per case in alternative order and the y-axis shows the execution time in seconds.

Fig 5.6: Serial and optimal order time comparison graph

Inferences from the graph:

1. Linearly increasing execution time for serial implementation.
2. Maintains below 0.6 seconds in Parallel implementation.
3. Increase in speedup percentage is 96.12%

**6. CONCLUSION**

The main aim of the project is to improve the time and space complexities of applications using matrix chain multiplication by using GPU and CUDA as the implementation language

.

This has been achieved by covering the following:

1. Finding out the optimal order of chain matrix multiplication

2. Multiplying the matrices using data parallelism, shared memory and tiled approach in the optimal order obtained from the 1st step.

GPU-based matrix chain multiplication is proposed to accelerate the speedup. The proposed work is implemented in CUDA and tested with matrix sizes varying from 100 to 35000 for three, four and five matrices. The experimental results show that, the speed up achieved through GPU implementation is 96.12% when compared with Serial implementation of the matrix chain multiplication. Finally, the proposed technique should be viewed more as a complement than an alternative to existing methods, to be used in all those cases where the other techniques cannot be employed efficiently.

**References**

[1] Raghav Lakhotia, Sanjeev Kumar, Rishabh Sood on “Matrix Chain Multiplication Using Greedy and Divide-Conquer approach”, IJCTT.

[2][CUDA C Programming Guide](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html" \l "abstract)**-**

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#abstract>

# [3]Dynamic Programming | Set 8 (Matrix Chain Multiplication)-

http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/

[4] Matrix chain multiplication- From Wikipedia, the free encyclopedia

[5] Yong Dou, S. Vassiliadis , G. K. Kuzmanov, G. N. Gaydadjiev “64-bit Floating-Point FPGA Matrix Multiplication”

[6] John Cheng, Max Grossman, Ty McKercher “Professional CUDA C Programming”

[7] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull, “Implementation of strassen’s algorithm for matrix multiplication,” in Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on. IEEE, 1996, pp. 32–32.

[8] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith, “Gemmw: a portable level 3 blas winograd variant of strassen’s matrix-matrix multiply algorithm,” Journal of Computational Physics, vol. 110, no. 1, pp. 1–10, 1994.

[9] Q. Luo and J. B. Drake, “A scalable parallel strassen’s matrix multiplication algorithm for distributed-memory computers,” in Proceedings of the 1995 ACM symposium on Applied computing. ACM, 1995, pp. 221–226.

[10] F. Desprez and F. Suter, “Impact of mixed-parallelism on parallel implementations of the strassen and winograd matrix multiplication algorithms,” Concurrency and Computation: practice and experience, vol. 16, no. 8, pp. 771–797, 2004.