Fast and Memory Efficient Strassen's Matrix Multiplication on GPU Cluster

Arjun Gopala Krishnan

A Thesis

in

The Department of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

August 2021

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Arjun Gopala Krishnan

Entitled: Fast and Memory Efficient Strassen's Matrix Multiplication on GPU Cluster

and submitted in partial fulfillment of the requirements for the degree of

M. Comp. Sc.

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________________ Chair

Dr. Hovhannes Harutyunyan

______________________________________ Examiner

Dr. Hovhannes Harutyunyan

______________________________________ Examiner

Dr. Rajagopalan Jayakumar

______________________________________ Supervisor

Dr. Dhrubajyoti Goswami

Approved by ______________________________________

Chair of Department or Graduate Program Director

____

Dr. Mourad Debbabi, Dean

Faculty of Engineering and Computer Science

Date ______________________________________

Abstract

Fast and Memory Efficient Strassen's Matrix Multiplication on GPU Cluster

Arjun Gopala Krishnan

Prior implementations of Strassen's matrix multiplication algorithm on GPUs traded additional workspace in the form of global memory or registers for time. Although Strassen's algorithm offers a reduction in computational complexity as compared to the classical algorithm, the memory overhead associated with the algorithm limits its practical utility. While there were past attempts at reducing the memory footprint of Strassen's algorithm by compromising parallelism, no prior implementation, to our knowledge, was able to hide the workspace requirement successfully. This thesis presents an implementation of Strassen's matrix multiplication in CUDA, titled Multi-Stage Memory Efficient Strassen (MSMES), that eliminates additional workspace requirements by reusing and recovering input matrices. MSMES organizes the steps involved in Strassen's algorithm into five stages where multiple steps in the same stage can be executed in parallel. Two additional stages are also discussed in the thesis that allows the recovery of the input matrices. Unlike previous works, MSMES has no additional memory requirements irrespective of the level of recursion of Strassen's algorithm. Experiments performed with MSMES (with the recovery stages) on NVIDIA Tesla V100 GPU and NVIDIA GTX 1660ti GPU yielded higher compute performance and lower memory requirements as compared to the NVIDIA library function for double precision matrix multiplication, cublasDgemm. In the multi-GPU adaptation of matrix multiplication, we explore the performance of a Strassen-based and a tile-based global decomposition scheme. We also checked the performance of using MSMES and cublasDgemm for performing local matrix multiplication with each of the global decomposition schemes. From the experiments, it was identified that the combination of using Strassen-Winograd decomposition with MSMES yielded the highest speedup among all the tested combinations.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Dr. Dhrubajyoti Goswami. This work would have been impossible without his guidance, care, and encouragement.

I am thankful to my family, friends and colleagues for their support and encouragement.

Table of contents

List of Tables

List of Figures

List of Equations

Chapter 1 Introduction

Matrix multiplication is one of the most fundamental algorithmic problems and finds extensive use in the fields of simulation, machine learning, graphics, network theory, to name a few [2]. This prevalence of the multiplication operation makes any improvements on the time complexity of the matrix multiplication algorithm highly desirable. One of the earliest implementations of matrix multiplication on high performance devices came in the form of BLAS (Basic Linear Algebra Subprograms) library implemented in Fortran [3]. The interface of BLAS library was standardized by the BLAS Technical forum [4]. The specifications of BLAS became the de-facto standard for low-level implementations of matrix multiplication. Over the years, as computations started shifting towards GPGPU (General-Purpose computing on Graphics Processing Units) paradigm, BLAS was implemented for GPUs. The extensive research that went into adapting matrix multiplication on GPU architecture, algorithmic tools, theoretical approaches, and software engineering methods have resulted in faster and more efficient algorithms and implementations [2].

Until 1969 matrix multiplication was believed to be an operation of cubic complexity. However, the introduction of Strassen's algorithm [5], named after Volker Strassen, opened the possibility of numerous sub-cubic matrix multiplication algorithms. Even with the superior time complexity of Strassen's algorithm compared to naive algorithms, it was never widely used in any of the BLAS implementations due to its additional workspace requirements, higher memory operation, and numerical instability at higher levels of recursions. In this thesis, we present optimized implementations that addresses the additional workspace and memory operation requirements of Strassen's algorithm on GPUs.

1.1 Problem Statement and Motivation

The work on this thesis began as an exploration into GCNs (Graph Convolution Networks) [31] and possible techniques to improve the training phase performance of these networks. During the initial research, it was identified that GCNs use a matrix chain multiplication consisting of five matrices to compute the propagation function for a layer during the training phase [31]. Exploring opportunities to optimize this matrix chain multiplication resulted in the work presented in chapter 3 and 4. Although GCNs generally rely on sparse matrix multiplication algorithms due to the sparseness of the adjacency matrix used in the training phase, the multiplication explored in this thesis is dense matrix multiplication. This choice was made due to the wider range of applications possible with a dense matrix multiplication algorithm. Equation 1.1 represents the propagation function for the *(l + 1)th* layer of the GCN. Here, Â = A + I where A is the adjacency matrix of the graph, and I is the identity matrix. D is the diagonal node degree matrix. $H^{(l)}$is the propagation function for the *l-th* layer of the GCN. $W^{(l)}$is the weight matrix for the *l-th* layer and σ(.) is a non-linear activation function.

$$f\left(H^{(l)}, A\right) = \sigma(\widehat{D}^{-\frac{1}{2}}\hat{A}\widehat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)})$$

Equation 1.1: Propagation function for the layer l + 1 in a GCN.

The work presented by J. Huang et al. [1] showed the possibility to partially optimize the memory requirements of Strassen's algorithm. This provided the motivation to attempt an implementation of Strassen's algorithm on GPUs that was completely able to hide the additional workspace requirements.

The inability of [1] to eliminate the additional workspace requirement and the increasing register pressure with higher levels of recursion while using Strassen Reloaded algorithm [1] on GPUs was the first problem we wanted to address with the thesis. This led to the development of a Multi-Stage Memory Efficient Strassen's Algorithm (MSMES) which is discussed in detail in chapter 3.

The literature review conducted to explore multi-GPU implementations of Strassen's algorithm uncovered no prior attempts at implementing a kernel that used Strassen's algorithm to decompose the matrix as well as to perform the local matrix multiplications. Conversely, G. Ballard et al. had shown in [2] that a communication optimal implementation of Strassen decomposition followed by Strassen multiplication is feasible on a Cray XT4 [26] supercomputer. Hence, the second problem we wanted to address in this thesis was to implement an efficient multi-GPU matrix multiplication kernel using a global Strassen decomposition to split work among the participating GPUs which then uses MSMES to perform local matrix multiplications. This implementation is discussed in detail in chapter 4.

1.2 Challenges and contributions

Strassen's algorithm performs less computations per memory operations because it is a sub cubic matrix multiplication algorithm. This means that the performance of any implementation of Strassen's algorithm will heavily depend on memory consumption and data transfer latencies. NVIDIA GPUs use a complex memory hierarchy consisting of system memory, GPU global memory, shared memory and register memory [6]. While there are numerous CUDA (Compute Unified Device Architecture) library functions like mallocManaged [44] during testing we realized that these features are not optimized for the memory access patterns exhibited by Strassen's algorithm. Hence, one of our major challenges was to identify techniques to optimize memory access at all levels of the memory hierarchy. We used recommendations from CUTLASS (CUDA Templates for Linear Algebra Subroutines) and other custom caching and prefetching as well as load balancing techniques to optimize the memory footprint and memory access latencies of our implementations.

The following are the major contributions of our approach:

- Designed and implemented Strassen's algorithm called MSMES in CUDA which has no additional memory requirement compared to a naïve matrix multiplication algorithm.
- Our implementation of MSMES can perform any level of recursive Strassen with no additional memory requirements. To our knowledge, this is the only implementation of a recursive Strassen based GEMM (Generalized Matrix-matrix Multiplication) which has no additional workspace requirement.
- Formulated a performance model that allows us to predict the execution time of MSMES for any matrix size on any NVIDIA hardware.
- Designed and implemented a multi-GPU version of MSMES that uses Strassen-Winograd global decomposition to distribute work among the participating GPUs.
- Compared the performance of Strassen-Winograd-MSMES (Global decomposition using Strassen-Winograd algorithm followed by local MSMES multiplication) against other configurations mentioned in table 4.3.

1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 details background information that applies to the problems discussed in Chapters 3 and 4. Chapter 3 discusses the implementation of our memory efficient single GPU implementation of Strassen's algorithm. In chapter 4 the details of multi-GPU implementation of Strassen's algorithm are discussed. In addition to Chapter 2, relevant related works are also discussed separately in

Chapters 3 and 4. Finally, in chapter 5 we discuss some recommendations for future augmentations that can be added to the implementations in chapter 3 and 4.

Chapter 2 Background

2.1 BLAS

Basic Linear Algebra Subprograms (BLAS) is a specification that defines functions for commonly used operations in numerical programming like vector addition, dot products, matrix multiplication etc. [35]. BLAS originated as a Fortran library in 1979 and was eventually standardized by the BLAS technical forum [4].

Since BLAS is just a specification, most of the commonly used languages, frameworks, and architectures have their own implementations of BLAS specifications which are highly optimized for the respective hardware and compiler used with the language. BLAS allow users to develop programs that are independent of the hardware and libraries being used. The extensive work that goes into optimizing BLAS implementations also mean that the users are guaranteed superior performance compared to custom implementation a user might program [35].

BLAS functions are organized into three levels which corresponds to the chronological order of publication, as well as the degree of polynomial complexity of the algorithm [35].

Level 1

This level defines operations that typically take linear time, $O(N)$, for completion. Hence, routines defined in this level generally corresponds to vector operations like dot products, vector additions etc.

Level 2

This level defines matrix-vector operations like a generalized matrix-vector multiplication. The operations defined in this level generally have quadratic complexity, $O(N^2)$.

Level 3

This level defines matrix-matrix operations like a generalized matrix-matrix multiplication (GEMM). The operations defined in this level generally have cubic complexity, $O(N^3)$.

Given, matrices A, B, and C where A is $M \times K$, B is $K \times N$, and C is an $M \times N$ matrix, and where α and β are constants, GEMM computes:

$$C = \alpha A \times B + \beta C \tag{1}$$

In chapter 3 we implement a Level 3 BLAS standard matrix-matrix multiplication that uses Strassen's algorithm to compute the result.

2.2 Strassen's algorithm

Given matrices $A \in R^{M \, x \, K}$ and $B \in R^{K \, x \, N}$, Strassen's algorithm computes the BLAS matrix multiplication standard (equation (1)) by partitioning the matrices into 2 × 2 submatrices such that:

$$\begin{bmatrix} C_0 & C_1 \\ C_2 & C_3 \end{bmatrix} = \alpha \begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} + \beta \begin{bmatrix} C_0 & C_1 \\ C_2 & C_3 \end{bmatrix} \tag{2}$$

The algorithm rearranges the arithmetic operations such that equation (1) is computed with 7 sub matrix multiplications, rather than the 8 involved in the classical algorithm [5]. Assuming that dimensions of each matrix are N × N, applying the previous decomposition recursively would allow the computation to be performed at *$O(N^{2.81})$* [5]. The arithmetic operations involved in equation (1) are illustrated in table 2.1:

Step	Computation	Result
1	$(A_0 + A_3)$	P_0
2	$(A_2 + A_3)$	P_1
3	$(A_0 + A_1)$	P_2
4	$(A_2 - A_0)$	P_3
5	$(A_1 - A_3)$	P_4
6	$(B_0 + B_3)$	Q_0

7	$(B_1 - B_3)$	Q_1
8	$(B_2 - B_0)$	Q_2
9	$(B_0 + B_1)$	Q_3
10	$(B_2 + B_3)$	Q_4
11	$(P_0 \times Q_{3)}$	M_0
12	$(P_1 \times B_0)$	M_1
13	$(A_0 \times Q_1)$	M_2
14	$(A_3 \times Q_2)$	M_3
15	$(P_2 \times B_3)$	M_4
16	$(P_3 \times Q_3)$	M_5
17	$(P_4 \times Q_4)$	M_6
18	$M_0 + M_3 + M_6 - M_4$	C_0
19	$M_2 + M_4$	C_1
20	$M_1 + M_3$	C_2
21	$M_0 + M_2 + M_5 - M_1$	C_3

Table 2.1: Steps in Strassen's matrix multiplication.

Table 2.1 shows that Strassen's algorithm performs equation (1) with 7 sub matrix multiplications and 18 addition/subtraction operations (12 additions and 6 subtractions). While the complexity analysis of the algorithm shows clear advantages, practical use of the algorithm suffers from poor performance due to additional workspace demands for storing intermediate results, especially in a memory-constrained GPU.

2.3 GPU Architecture

NVIDIA's Tesla V100 and GTX 1660ti are examples of graphics processors which can perform GPGPU operations at high levels of parallelism. The Tesla V100 GPUs are enterprise level offerings that finds extensive use in research, GPU clusters, and professional workflows [6]. The GTX 1660ti on the other hand is predominantly used in personal and enthusiast consumer applications. Table 2.2 illustrates the comparison of specifications of the two GPUs used in our experiments.

An NVIDIA GPU is comprised of multiple graphics processing units, texture processing controllers, streaming multiprocessors (SM), and memory controllers. Each streaming multi-processor has 64 FP32 (Single-precision floating-point format) cores, 64 INT32 (32-bit Integer format) cores, 32 FP64 (Double-precision floating-point format) cores and optional tensor cores. The GPUs have off chip global memory which supplies data to the relevant cores through on chip memory controllers. Level 2 caches are used by the memory controllers to hide memory transfer latencies between the cores and global memory. Similarly, each streaming multiprocessor has reserved Level 1 cache that can be used to hide memory latencies between Level 2 cache and cores. The streaming multiprocessor also has shared memory that can be used to transfer data between threads in a thread block. The CUDA cores have on chip registers that can be used to store frequently used values or intermediate results. Figure 2.2 describes the usage of memory hierarchy across threads, blocks, and grids.

Parameter	**Tesla V100**	**GTX 1600ti**
Architecture	GV100 (Volta)	TU116 (Turing)
SMs	80	24
FP64 Cores / SM	32	32
GPU Boost Clock	1530 MHz	1770 MHz
Peak FP64 FLOPS	7.8 TFLOPS	169.9 GFLOPS
Global Memory Size	16 GB	6 GB
L2 cache size	6144 KB	1536 KB
L1 cache / SM	128 KB	64 KB/SM
Shared Memory / SM	Configurable up to 96 KB / SM	48 KB / SM
Register file size / SM	256 KB / SM	64 KB / SM

Table 2.2: Comparison of the GPUs used in the experiments.

From figure 2.3 and 2.4 we can observe that there are some differences between the GV100 SM and the TU116 SM used in the Tesla V100 and GTX 1660ti respectively. Namely, GV100 has tensor cores which are not available on TU116. While, tensor cores can help with certain multiplication workloads, we do not leverage tensor cores in our implementations. Also, the CUDA GEMM implementation, cublasDgemm [45] used to compare the performance of our implementation also do not leverage tensor cores.

2.4 GPGPU Programming model

In GPGPU paradigm, the GPU takes the role of a coprocessor [7]. The CPU which takes the role of the host issues data and invokes device kernels or GPU kernels that need to operate on the data. Device kernels are functions that are executed by the GPU where a programmer specified number of GPU threads and thread blocks are used to execute the kernel. Once the processing is completed by the GPU, the data is transferred back to the system global memory which is accessible to the CPU. The advantage of using GPUs to perform matrix multiplication lies in the GPU's ability to efficiently parallelize certain workloads.

While trying to adapt a problem to the GPGPU paradigm, factors like amount of data, data movement latency, cache behavior, processing per data etc. must be considered to determine whether the workload should be assigned to the CPU or to the GPU. Generally, in a data intensive workload like matrix multiplication where there is a huge scope for parallel processing, the CPU takes the role of orchestration, and the GPU is responsible for the computations. There are GPGPU implementations, like [19], where it was observed that assigning certain operations to the CPU, where the memory transfer times are higher compared to computation time, resulted in better speedups. In these kinds of implementations, the CPU computes some workloads while managing the GPUs performing the remainder workloads. For adapting the problem in this thesis, it was observed that using CPU exclusively for orchestration while using GPUs to perform all the computations yielded better performance.

There are numerous implementations of GPGPU paradigm and any language that can poll GPUs to perform some computation can implement a GPGPU framework [36]. OpenCL is a popular implementation of GPGPU that is actively supported on Intel, AMD, Nvidia and ARM platforms. CUDA is an NVIDIA proprietary framework that allows GPGPU implementations in C programming language with NVIDIA GPUs. CUDA was used for the implementations in this thesis due to its superior performance on NVIDIA hardware [37].

2.5 CUDA

CUDA is a parallel computing framework developed by NVIDIA for general purpose computing on NVIDIA GPUs. Introduced in 2007, this is the first framework that implemented general purpose computing APIs that did not require mapping computations to graphics primitives [36]. In CUDA, the function/code executed by the GPU is called a kernel. A kernel is parallelized by splitting the workload between threads which are grouped into blocks. The blocks are grouped into grids [8]. Hence, every kernel invocation requires the user to define the dimension of the thread block and the grid. These parameters are of the datatype dim3 [9]. The dim3 datatype can be used to define 1-, 2- or 3-dimensional thread blocks and grids. Figure 2.1 describes the organization of threads into blocks and grids.

The GPU has a block scheduler that dynamically assigns a block to a streaming multi-processor. A thread in a block can use shared memory of the streaming multiprocessor to exchange information with other threads in the same block. The threads in a block are arranged into groups of 32, called a warp [46]. Once the threads in a warp are ready for execution, the warp scheduler assigns these warps to streaming processors. If there is no warp divergence present in the threads that belong to a warp, all 32 threads in the warp get executed in a single step. If the shared memory and register demands of a kernel permit, multiple blocks maybe assigned to the same streaming multiprocessor.

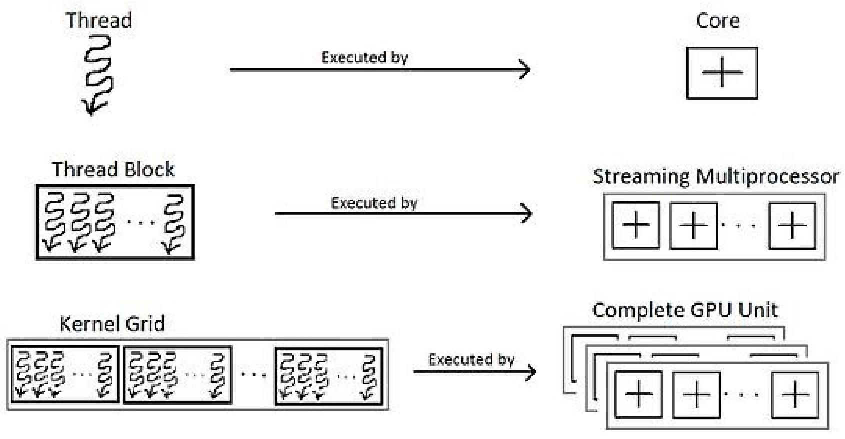


Figure 2.1: Organization of threads, blocks, and grids in CUDA [38]

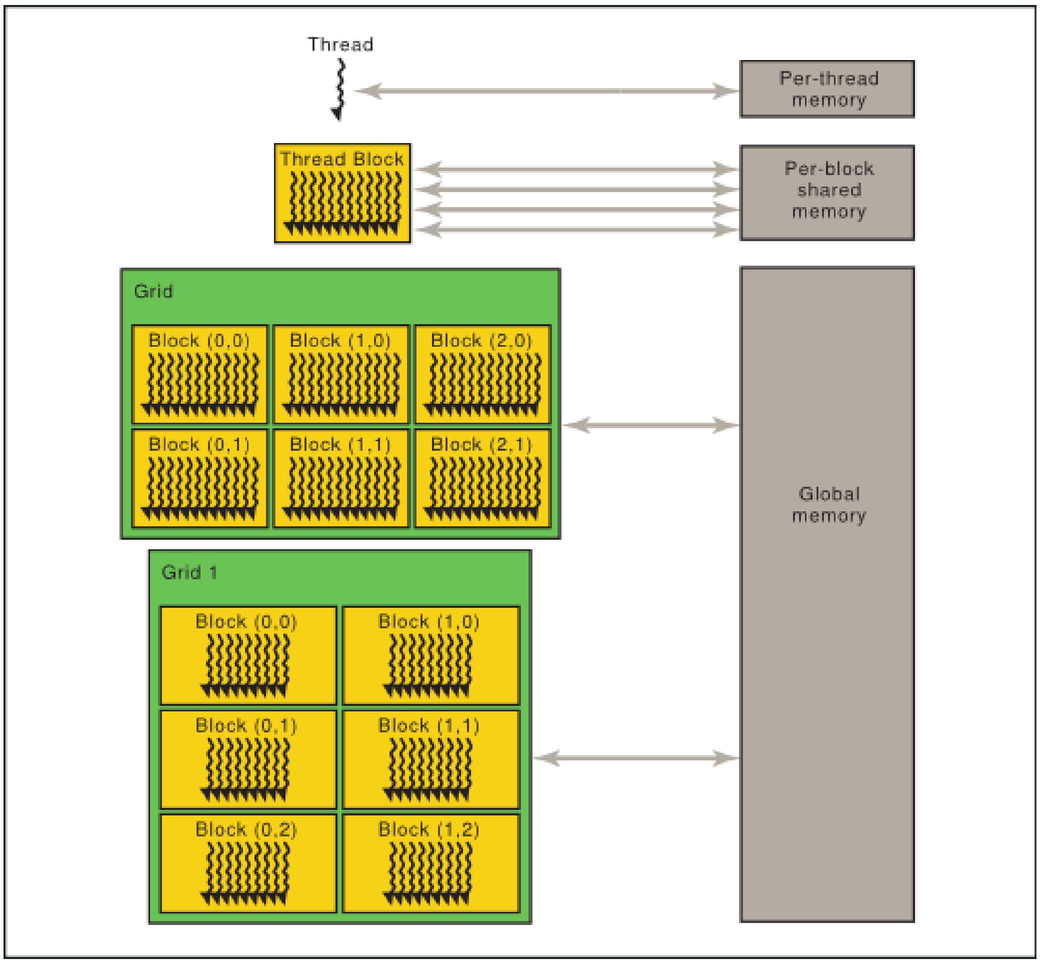


Figure 2.2: Memory hierarchy of NVIDIA GPUs [40].

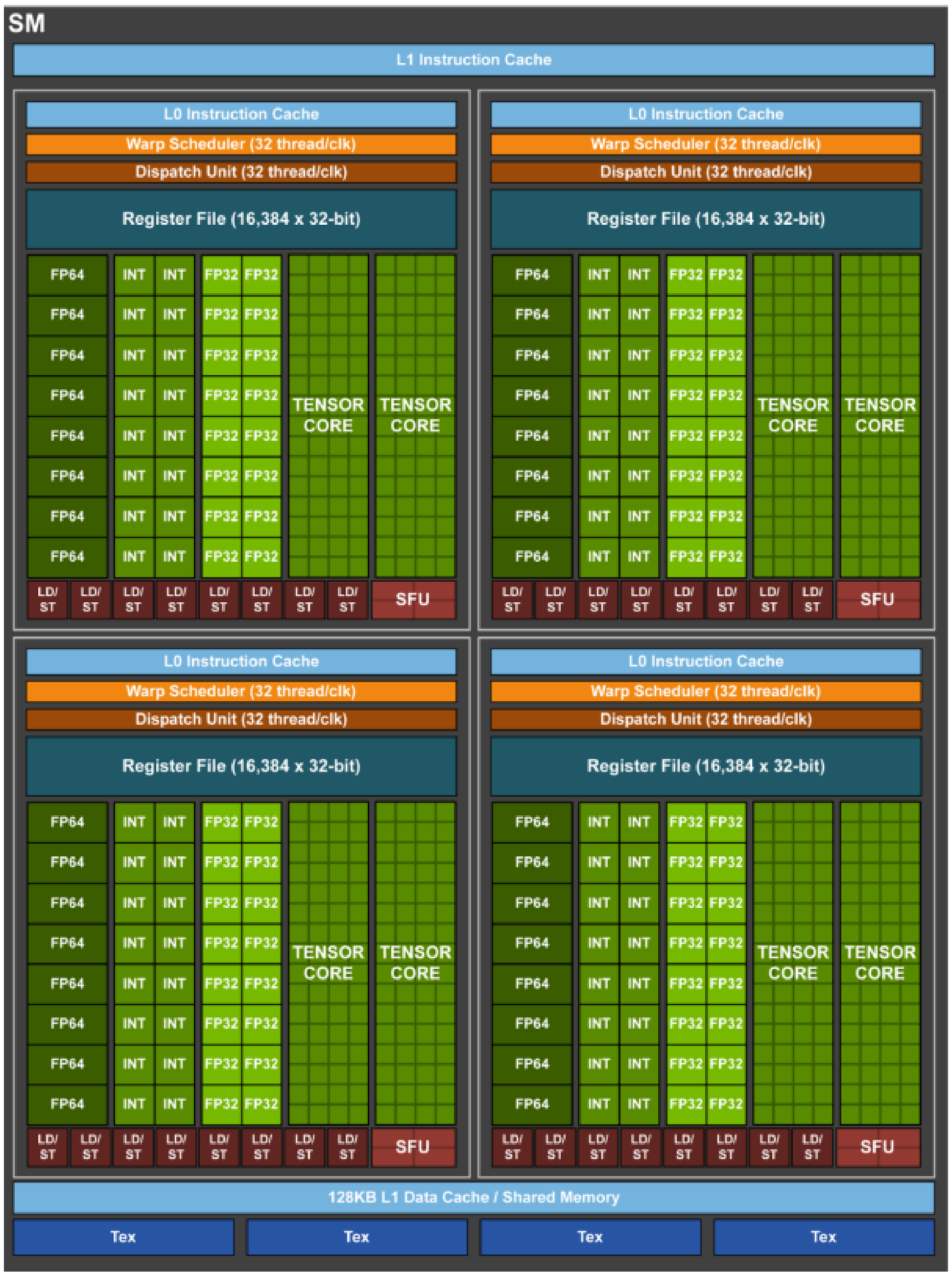

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Figure 2.3: Volta GV100 Streaming multiprocessor used in Tesla V100 [41]

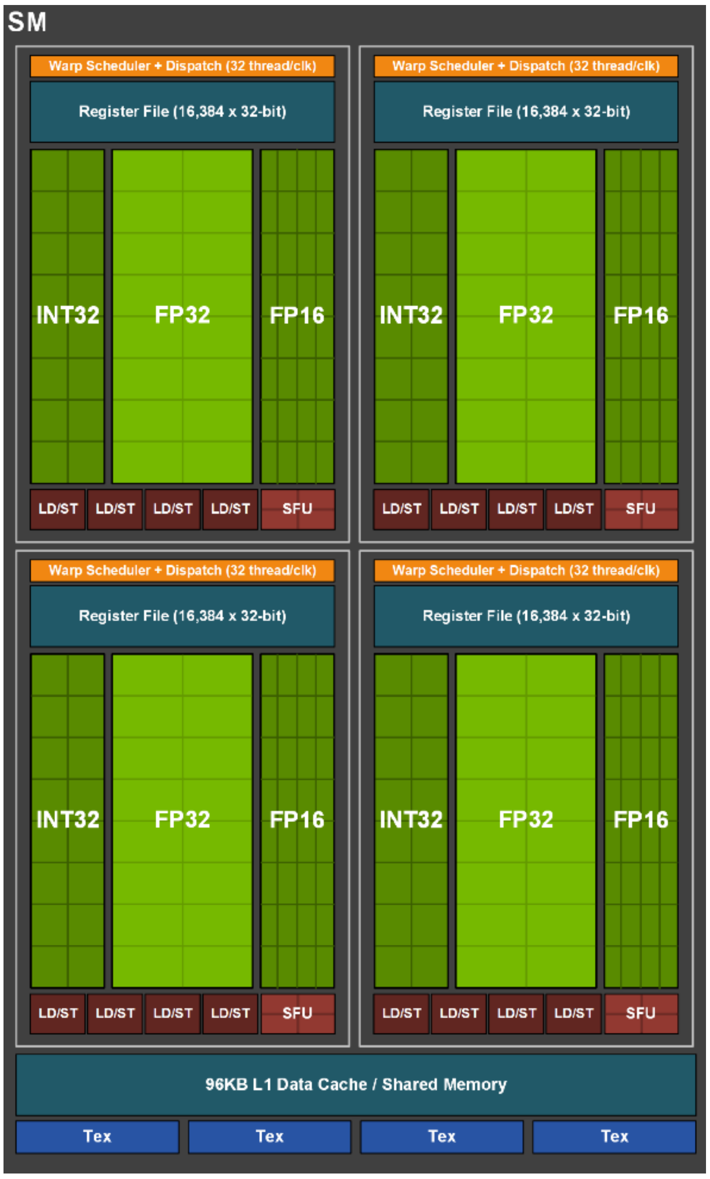

SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP16
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex

Figure 2.4: Turing TU116 Streaming Multiprocessor used in GTX 1660ti [42]

Chapter 3 Multi-Stage Memory Efficient Strassen

3.1 Introduction

Consider two matrices of size $N \times N$ each. While the traditional iterative algorithm for matrix multiplication performs the operation in $O(N^3)$, several sub-cubic algorithms have been formulated that improve the time complexity. Strassen's matrix multiplication is one such algorithm which when applied recursively can perform matrix multiplication at a time complexity of $O(N^{2.807})$ [5].

Although the improvement in time complexity of Strassen's algorithm is impressive from theoretical perspective, the algorithm demands higher workspace as compared to the traditional matrix multiplication algorithm and hence limits its practical utility, especially in a memory-constrained processors like GPUs.

Prior works (e.g., [1], [12], and [18]) attempted to lower the additional workspace requirement of Strassen's algorithm, but never managed to eliminate it completely. While [1] succeeded in removing the global memory requirements, this was done at the cost of increasing register memory requirements with each level of recursion of Strassen's algorithm. An ordering of the steps in Strassen's algorithm to lower the global memory requirement was formulated in [12], but it never managed to eliminate the additional workspace requirement entirely. While [18] successfully formulated a schedule that eliminates the global memory requirement for simple matrix multiplication, it could not eliminate the additional memory requirement for the BLAS standard matrix multiplication.

In this chapter, we present an improved implementation of Strassen's algorithm on CUDA that follows the CUTLASS guidelines. The implementation, titled Multi-Stage Memory Efficient Strassen (MSMES), eliminates the requirement of additional workspace associated with Strassen's algorithm by organizing and restructuring the Strassen's algorithm operations in stages, where multiple operations in the same stage can be executed in parallel, and reusing and eventually recovering the input matrices. MSMES can perform any depth of recursion of Strassen's algorithm without needing extra workspace in the form of global memory or registers. Though it comes at a cost of nominal additional computing requirement, overall, there is a noticeable performance gain with increasing level of recursion.

The chapter is organized as follows: section 3.2 discusses the background and related works. Section 3.3 describes the motivation and design of MSMES. Section 3.4 describes the implementation of MSMES, followed by a discussion on the experiments and results. Section 3.5 provides an analytical performance model of MSMES. Finally, section 3.6 concludes the chapter.

3.2 Background and Related Works

3.2.1 CUTLASS

GPUs use a multi-level memory hierarchy comprised of global, shared, and register memories. To efficiently use this hierarchy, it is essential to hide the data movement latencies between the different levels of memory. CUTLASS is a collection of CUDA C++ templates and abstractions for implementing high-performance GEMM computations at all levels and scales within CUDA kernels [10]. Detailed understanding of the high-performance implementation of GEMM based on CUTLASS was essential to design strategies for hierarchical partition and movement of data for MSMES.

Accumulating matrix product:

Accumulating matrix product is a set of loop optimizations recommended by NVIDIA in CUTLASS documentation to improve the memory access pattern associated with GEMM implementations on GPU.

For the remainder of the thesis, it is assumed that $\alpha = 1$ and $\beta = 0$ in equation (1) without loss of generality. The simplest iterative solution to the problem consists of three loops as follows:

```
for (int i = 0; i < M; ++i)
   for (int j = 0; j < N; ++j)
      for (int k = 0; k < K; ++k)
         C[i][j] += A[i][k] * B[k][j];
```

For simplicity, in the remainder of the thesis it is assumed that $M = N = K$. Thus, computational complexity of the previous loop-nest is $O(N^3)$ and the space requirement is $O(N^2)$. However, for the theoretical compute complexity to hold, every matrix element requires reusing $O(N)$ times [10]. Therefore, the above implementation depends on holding large working set of data in on-chip memory, which results in thrashing at higher values of *M*, *N*, and *K*.

By applying loop inversion to the simple iterative solution, the *k-th* loop has been permuted outside the *i-th* and *j-th* loops as follows:

```
for (int k = 0; k < K; ++k)
  for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

This form loads the *k*-th column of *A* and *k*-th row of *B* once and performs the outer product on them and accumulates the result in *C*. After this step, the *k*-th column and *k*-th row of *A* and *B* respectively are never used again. However, this implementation requires the entire *C* matrix to be available in the on-chip memory and can again result in thrashing [10].

The memory requirement of this implementation can be reduced by partitioning *C* matrix into *M_tile* by *N_tile* that are guaranteed to fit in on-chip memory. Now, the outer product formulation can be applied on each tile. With this formulation, each row of *A* and each column of *B* are fetched only once.

```
for (int m = 0; m < M; m += M_tile)
  for (int n = 0; n < N; n += N_tile)
    for (int k = 0; k < K; ++k)
      for (int i = 0; i < M_tile; ++i)
        for (int j = 0; j < N_tile; ++j)
          C[m + i][n + j] += A[m + i][k] * B[k][n + j];
```

Here the outer loops can be trivially parallelized [10]. This technique of stepping through *K* dimension in memory optimized blocks while accumulating results on *C* partition is called accumulating matrix product [10].

Blocking Strategies:

Further improvements can be achieved by exploiting the hierarchical memory structure of GPUs. The matrices are decomposed into a hierarchy of thread block tiles, warp tiles, and thread tiles. This technique enables fine tuning of matrix tile dimensions at various levels of execution to better suit the available form of memory.

Device level:

Blocking at this level is performed for thread blocks [1]. The matrices *A*, *B*, and *C* are partitioned into *Ms* × *Ks*, *Ks* × *Ns*, and *Ms* × *Ns* blocks respectively. Each thread block is responsible for computing an *Ms* × *Ns* block of matrix *C*. The product of each *Ms* × *Ks* block of *A* with a corresponding *Ks* × *Ns* block of *B* is accumulated into the respective *Ms* × *Ns* block of *C*. Hence, the partitions of the *C* matrix are referred to as *C* accumulators. Since the *C* accumulators are updated numerous times during the computation, the *C* partitions are lifted to the register memory to reduce access latencies. In order to improve data locality, the partitions of *A* and *B* matrices are iteratively lifted to the shared memory where the data is accessible to all the threads in a thread block.

a) Thread block level: Blocking at this level is performed for warps [1]. At this level, the *C* accumulator is spatially partitioned across all the warps in a thread block. Each warp iteratively loads along the *K* dimension an *A* fragment (a sub-column of *A* partition of height *Mw*), and a *B* fragment (a sub row of *B* partition of width *Nw*) into registers. Then the outer product of these fragments are performed to compute the accumulation for the iteration.

b) Warp Level: Blocking is performed for the threads at this level [1]. Each thread in the warp computes an *Mr* × *Nr* partition of *C* accumulator by performing outer products of elements in an *Mr* fragment of *A* with an *Nr* fragment of *B* in "strip mining" (cyclic) pattern. Apart from register demand considerations, *Mr* and *Nr* are determined by the smallest granularity of vector load which is 128 bits.

c) Thread level: Threads issue a sequence of independent Fused Multiply Add instructions to the CUDA cores and computes accumulating matrix outer product of *Mr* subcolumns of *A* with *Nr* subrow of *B*.

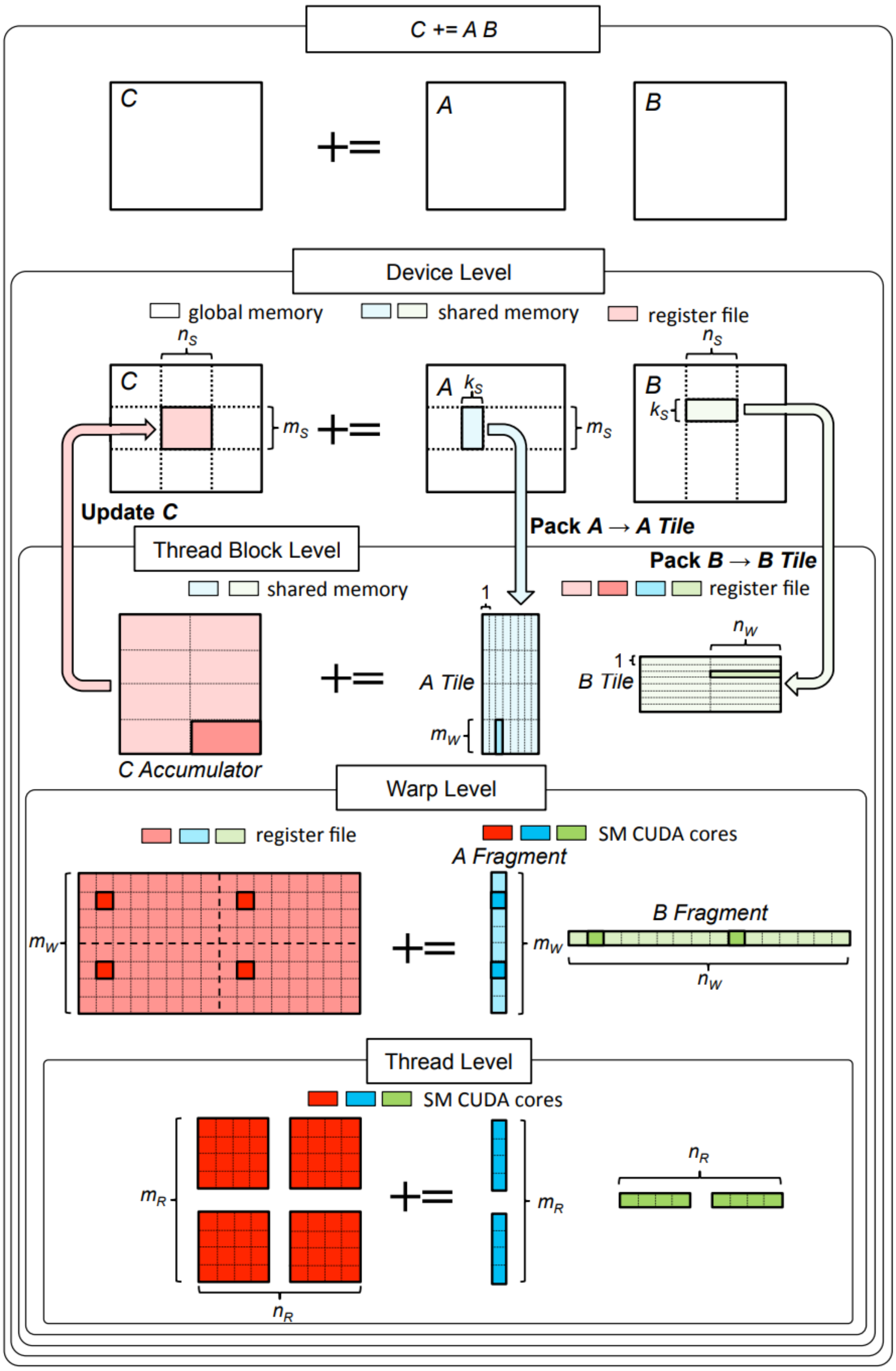


Figure 3.1: Blocking strategies and data movement for CUTLASS multiplication kernel [1].

Software prefetching:

CUTLASS uses global and local data prefetching to hide data movement latencies at various memory levels. The interleaving of prefetch instructions from global memory (lines 12 and 14 in Algorithm 3.1 below) and, from shared memory (lines 17 and 18 in Algorithm 3.1) with computations keep the SMs busy without having to wait for the next set of data to be loaded in. A synchronization step (line 24 in Algorithm 3.1) is required to ensure that all shared memory writes are completed before they get read in lines 12 and 14 in the next iteration.

Figure 3.1 describes the partition and movement of data in a CUTLASS matrix multiplication kernel.

```
01: Register: $frag_A[2][M_R]$, $frag_B[2][N_R]$
02: Register: $next_A[M_R]$, $next_B[N_R]$
03: NOP
04: Register: $accum_C[M_R \times N_R]$
05: Shared memory: $tile_A[K_S \times M_S]$, $tile_B[K_S \times N_S]$
06: Load one $M_S \times K_S$ block of A into $tile_A[K_S][M_S]$
07: Load one $K_S \times N_S$ block of B into $tile_B[K_S][N_S]$
08: __syncthreads()
09: Load first subvector of $tile_A$ into $frag_A[0][M_R]$
10: Load first subvector of $tile_B$ into $frag_B[0][N_R]$
11: **for** $block\_k = 0 : K_S : K$ **then**
12:     prefetch one subcolumn of next $M_S \times K_S$ block of A into $next_A$
13:     NOP
14:     prefetch one subrow of next $K_S \times N_S$ block of B into $next_B$
15:     NOP
16:     **for** $warp\_k = 0 : 1 : K_S$ **then**
17:         prefetch next subcolumn of $tile_A$ into $frag_A[(warp\_k + 1) \% 2][M_R]$
18:         prefetch next subrow of $tile_B$ into $frag_B[(warp\_k + 1) \% 2][N_R]$
19:         **for** $i = 0 : 1 : M_R$ **then**
20:             **for** $j = 0 : 1 : N_R$ **then**
21:                 $accum_C[i][j] \mathrel{+}= frag_A[warp\_k \% 2][i] \times frag_B[warp\_k \% 2][j]$
22:     store $next_A[M_R]$ into $tile_A[K_S][M_S]$
```

```
23:     store $next_B$[$N_R$] into $tile_B$[$K_S$][$N_S$]
24:     __syncthreads()
25: write back $accum_C$ to corresponding block of C
```

Algorithm 3.1: CUTLASS GEMM algorithm (*adapted from [1]*).

3.2.2 Memory requirement of Strassen's algorithm

By analyzing the arithmetic operations involved in Strassen's algorithm, it can be shown that a naive implementation of the algorithm would require extra workspace to store the intermediate results P_0 through P_4, Q_0 through Q_4, and M_0 through M_6. Since each of these intermediate results are $\frac{N}{2} \times \frac{N}{2}$ in size, a naïve implementation would require an extra workspace of $16 \times \left(\frac{N}{2} \times \frac{N}{2}\right) = 4N^2$.

On limited global memory devices like GPUs, this need for extra workspace limits the maximum problem size that can be solved using Strassen's as compared to GEMM implementations. While global decomposition techniques can be used to improve the maximum problem size, the inability of naïve Strassen implementations to match the maximum problem size for GEMM would result in more steps after global decomposition, which can offset the improvements from computation complexity reduction of Strassen.

The increase in data movement on account of more steps in Strassen's algorithm results in higher mops (memory operations per second) to flops (floating point operations per second) ratio as compared to traditional matrix multiplication algorithms. Hence, a naive implementation of Strassen's algorithm would also be more susceptible to the effect of memory latency on execution time. This limits the possibility of using techniques like unified memory [11], offered by CUDA, to offset the limited global memory in GPUs.

The higher mops to flops ratio of Strassen's algorithm necessitates the need to use registers and shared memory to store frequently used values to reduce the effects of larger latency involved in load and store instructions to and from global memory [1]. Again, a non-memory optimized implementation of the Strassen's kernel with high register or shared memory requirement would stifle the concurrency and thread occupancy [47] of the kernel.

3.2.3 Recursive Strassen's algorithm

While 1-level Strassen (i.e., with no recursion applied) already reduces the number of submatrix multiplication from 8 to 7 as compared to the classical algorithm, recursive application of the algorithm is required to achieve the theoretical complexity of $O(N^{2.81})$ [5].

Strassen's algorithm is recursively applied by further decomposing each of the 7-submatrix multiplication and reapplying Strassen's algorithm to these operations. Although a multi-level Strassen reduces computational complexity, it has been observed that the extra workspace requirement gets amplified as the levels increase [1], [12]. A naive non-recursive Strassen's algorithm has an extra memory requirement of $4N^2$. But when the algorithm is applied recursively to create a 2-level Strassen, the workspace requirement increases to $7N^2$.

To effectively harness the reduction in computational complexity of multi-level Strassen's algorithm, the extra workspace requirement at all levels of the algorithm would have to be hidden efficiently.

3.2.4 Previous works implementing Strassen's algorithm.

A modified version of GPU8 algorithm [13] is used in [12] to implement a fast Strassen's multiplication algorithm. The GPU8 algorithm is adapted for 1-level Strassen, 1-level Winograd, multi-level Strassen, and multi-level Winograd. To our knowledge, this is the first attempt at implementing a Strassen based GEMM on GPUs. It uses temporary storage strategy formulated by Douglas et al [14] for the Winograd variant, which lowers the workspace requirement to $\frac{2}{3}N^2$. For the Strassen's implementation, it formulates an ordering which also lowers the workspace requirement to $\frac{2}{3}N^2$. For the multi-level algorithms, it uses strategies discussed by Huss-Lederman et al [15], [16] which use two temporary matrices at each level of recursion.

Huang et al. [1] implement the Strassen's algorithm on GPUs by following recommendations from CUTLASS for efficient data movement within the memory hierarchy of NVIDIA GPUs. It adapts the algorithm formulated by Lai et al [17] to hide memory requirements for intermediate results. It designs a new kernel by modifying the CUTLASS GEMM kernels, where additions of A and B submatrices are performed in the kernel during the packing phase. This eliminates the need for a temporary workspace to store this information, because the results are directly loaded to the registers and shared memory. It also develops a performance model for choosing the appropriate block sizes and predicting performance for various blocking configurations. While the global memory requirements are eliminated, the kernel's register requirements are higher as compared to the CUTLASS GEMM. The need for registers also increases with each level for the multi-level Strassen's implementation. This is in contrast to our work where the need for registers is a constant irrespective of the level of recursion.

The work by Boyer et al. [18] creates schedules for Strassen and Winograd variants of matrix multiplication, where workspace requirements are optimized for various constraints of execution. The authors are able to create a schedule for $C = A \times B$ with no extra memory requirements. They are also able to create schedules for equation (1), which can drop the additional memory requirement from N^2 to $\frac{2}{3}N^2$. We did not use the schedules from [18] because the variant, which does not consume any extra workspace, cannot be generalized to the BLAS standard for matrix multiplication denoted by equation (1) since it uses the output C matrix to store some of the temporary results. Also, this scheme will not be able to support multi-level Strassen's algorithm.

P. Lai et al. [43] implemented a Strassen-Winograd based GEMM on CUDA which used Strassen-Winograd algorithm to divide the matrix. Once a predicted cut-off dimension is crossed, CUDA implementations of GEMM like cublasSgemm or cublasDgemm was used to perform the actual computations. Unlike MSMES, this implementation did not address the additional workspace requirements of Strassen-Winograd algorithm.

3.3 Multi-Stage Memory Efficient Strassen

This section describes our work, MSMES, that eliminates the additional workspace requirement of Strassen's algorithm on GPUs without compromising parallelism. The extra workspace requirement of Strassen's algorithm arises from the need to store intermediate results. With a naïve implementation of an addition kernel followed by a multiplication kernel, the intermediate results P_0 through P_4, and Q_0 through Q_4 (refer to Table 2.1) would require temporary workspace. Similarly, when a naïve multiplication kernel followed by addition kernel is used, the intermediate results M_0 through M_6 would need additional workspace. While it is possible to reduce the additional workspace requirement by tackling each of the 25 steps (12 additions + 6 subtractions + 7 multiplications) in a serial fashion, this technique would hinder the ability to efficiently parallelize the algorithm.

The idea behind a memory efficient Strassen's algorithm on GPUs was inspired by the Strassen's kernel defined in [1] and from an observation on the concurrent submatrix utilization of each multiplication operation in Strassen's algorithm.

3.3.1 Concurrent submatrix utilization of Strassen's algorithm

Consider the following rearrangement of the steps in Strassen's algorithm:

$$M_0 = (A_0 + A_3) \times (B_0 + B_3);\ C_0 \mathrel{+}= M_0;\ C_3 \mathrel{+}= M_0;$$

$$M_1 = (A_2 + A_3) \times B_0;\ C_2 \mathrel{+}= M_1;\ C_3 \mathrel{-}= M_1;$$

$$M_2 = A_0 \times (B_1 - B_3);\ C_1 \mathrel{+}= M_2;\ C_3 \mathrel{+}= M_2;$$

$$M_3 = A_3 \times (B_2 - B_0);\ C_0 \mathrel{+}= M_3;\ C_2 \mathrel{+}= M_3;$$

$$M_4 = (A_0 + A_1) \times B_3;\ C_1 \mathrel{+}= M_4;\ C_0 \mathrel{-}= M_4;$$

$$M_5 = (A_2 - A_0) \times (B_0 + B_1);\ C_3 \mathrel{+}= M_5$$

$$M_6 = (A_1 - A_3) \times (B_2 + B_3);\ C_0 \mathrel{+}= M_6;$$

Each of these steps compute one of the submatrix multiplications involved in Strassen's algorithm. It can be observed that the steps access at most 2 submatrices each of A and B concurrently. These results are updated to at most 2 submatrices of C concurrently.

We design an adder kernel where, instead of consuming the extra workspace needed to store:

$$\begin{Bmatrix} A_i \pm A_j \\ B_k \pm B_l \end{Bmatrix} i, j, k, l \in \{0, 3\}$$

the results are stored in A_i and B_k respectively. If A_j and B_l are unaltered, the original values of A_i and B_k can be recovered by a simple matrix subtraction (or addition) of order $O(N^2)$ and can then be used for the next submatrix multiplication. This technique would eliminate the extra workspace needed to store P_0 through P_4, and Q_0 through Q_4.

3.3.2 Strassen multiplication kernel

We extend the CUTLASS GEMM kernel from [10] to accommodate the new Strassen primitive:

$$M_i = A_j \times B_k;\ C_l \pm= M_i;\ C_m \pm= M_i : i, j, k, l, m \,\epsilon\, \{0, 6\}$$

From Algorithm 3.1, it can be observed that the extra workspace needed to store *M₀* through *M₆* has been eliminated using C accumulator registers in the kernel (lines 04 and 21 in algorithm 3.1). The C accumulator was already used in the original CUTLASS GEMM kernel. Hence, our modified kernel does not use any extra registers to eliminate the additional workspace demand of Strassen's algorithm.

3.3.3 Rescheduling the stages to increase concurrency

From the adjustments discussed in subsections 3.3.1 and 3.3.2, although the extra workspace requirement is eliminated, it is achieved at the cost of concurrency of performing the submatrix multiplications. To identify an ordering of the steps to improve the concurrency of the algorithm, the following constraints are considered.

- A submatrix can be overwritten with the result of an addition or subtraction it was involved in as long as there is a way to recover the original values of the submatrix in the next stage.
- Overwrite submatrices in such a way that the recovery is simple and can be ideally done in a single stage.
- No C submatrices will be used for overwrites since equation (1) necessitates the retention of old values in the C matrix.

With these considerations, the following groupings are identified.

$$\left.\begin{cases} M_1 = (A_2 + A_3) \times B_0;\ C_2 += M_1;\ C_3 -= M_1; \\ M_2 = A_0 \times (B_1 - B_3);\ C_1 += M_2;\ C_3 += M_2; \\ M_3 = A_3 \times (B_2 - B_0);\ C_0 += M_3;\ C_2 += M_3; \\ M_4 = (A_0 + A_1) \times B_3;\ C_1 += M_4;\ C_0 -= M_4; \end{cases}\right\} \rightarrow Group\ 1$$

$$\left.\begin{cases} M_0 = (A_0 + A_3) \times (B_0 + B_3);\ C_0 += M_0;\ C_3 += M_0; \\ M_5 = (A_2 - A_0) \times (B_0 + B_1);\ C_3 += M_5 \\ M_6 = (A_1 - A_3) \times (B_2 + B_3);\ C_0 += M_6; \end{cases}\right\} \rightarrow Group\ 2$$

In the previous groupings, 2 submatrices of *A* and *B* each are overwritten while their respective partners remain intact. This would allow a maximum of 4 submatrix multiplication steps to be executed concurrently. The following outlines the stages involved in this grouping strategy:

Stage 1 (addition/subtraction stage):

$$A_2 = A_2 + A_3$$
$$B_1 = B_1 - B_3$$
$$B_2 = B_2 - B_0$$
$$A_1 = A_0 + A_1$$

Stage 2 (Multiplication stage):

$$M_1 = A_2 \times B_0;\ C_2 += M_1;\ C_3 -= M_1;$$
$$M_2 = A_0 \times B_1;\ C_1 += M_2;\ C_3 += M_2;$$
$$M_3 = A_3 \times B_2;\ C_0 += M_3;\ C_2 += M_3;$$
$$M_4 = A_1 \times B_3;\ C_1 += M_4;\ C_0 -= M_4;$$

Stage 3 (addition stage):

$$A_0 = A_0 + A_3$$
$$B_0 = B_0 + B_3$$

Stage 4 (addition/subtraction stage):

$$A_2 = A_2 - A_0\ Equivalent\ to\ (A_2 + A_3 - A_3 - A_0)$$
$$B_0 = B_1 + B_0\ Equivalent\ to\ (B_1 - B_3 + B_0 + B_3)$$
$$A_1 = A_1 - A_0\ Equivalent\ to\ (A_1 + A_0 - A_0 - A_3)$$
$$B_2 = B_2 + B_0\ Equivalent\ to\ (B_2 - B_0 + B_0 + B_3)$$

Stage 5 (Multiplication stage):

$M_0 = A_0 \times B_0;\ C_0 += M_0;\ C_3 += M_0;$

$M_5 = A_2 \times B_1;\ C_3 += M_5;$

$M_6 = A_1 \times B_2;\ C_0 += M_6;$

All steps in a stage can be executed concurrently. During the multiplication stages, where multiple threads might be updating the same *C* submatrix values, atomicAdd function [9] is used to avoid race conditions.

Though stage 3 in this scheduling, with only 2 concurrent steps, might seem like a bottleneck, the embarrassingly parallel nature of matrix addition allows the stage to be completed with the same thread density and without adding any extra delays to the workflow. This observation is confirmed in subsection 3.5.1.

3.3.4 Reconstruction of the Input Matrices

In a GPU based matrix multiplication algorithm, the input matrices *A* and *B* are copied to the global memory of the GPU and are duplicates of the input matrices. Hence, in most scenarios, recovery of the input matrix can be omitted for a very small improvement in runtime. But, for the sake of completeness, the following two stages have been designed to recover the input matrices. While recovery is optional for non-recursive Strassen's, most workflows utilizing matrices would apply multiple operations on the input matrices. Hence, all the experiments conducted are performed with the recovery stages.

Stage 6 (addition/subtraction: recuperation stage 1):

$A_0 = A_0 - A_3\ Equivalent\ to\ (A_0 + A_3 - A_3)$

$B_0 = B_0 - B_3\ Equivalent\ to\ (B_0 + B_3 - B_3)$

Stage 7 (addition/subtraction: Recuperation stage 2):

$A_1 = A_1 + A_3\ Equivalent\ to\ (A_1 - A_3 + A_3)$

$B_1 = B_1 - B_0\ Equivalent\ to\ (B_1 + B_0 - B_0)$

$A_0 = A_0 - A_3\ Equivalent\ to\ (A_0 + A_3 - A_3)$

$B_0 = B_0 - B_3\ Equivalent\ to\ (B_0 + B_3 - B_3)$

3.3.5 Recursive Strassen (Multi-level Strassen)

In a multi-level Strassen's algorithm, each of the 7-submatrix multiplication is further decomposed by applying Strassen's algorithm on them. In MSMES, using the modified multiplication kernel and by using input matrices to store intermediate results, no extra global memory, shared memory, or registers are consumed irrespective of the number of levels of recursion.

The problem is recursively decomposed by applying Strassen's algorithm until the submatrix dimension reaches the cutoff length, at which point no more recursion is applied and the matrices are multiplied directly using the modified Strassen's multiplication kernel.

In recursive Strassen, it is imperative to use the recovery stages to revert the changes made to the input matrices so that the subsequent steps in the prior level of recursion will not be affected. Since all the additional operations performed at each level of recursion is $O(N^2)$, the theoretical time complexity of Strassen's algorithm remains unchanged.

3.4 Implementation and experiments

All the kernels of MSMES are implemented in CUDA, designed for NVIDIA GPUs. The kernels closely replicate the memory management and data transfer strategies used in CUTLASS GEMM to utilize all levels of GPU memory hierarchy. This allows the kernels to efficiently hide memory transfer latencies.

The kernels are designed in such a way that each stage of the algorithm can be completed by invoking a single device kernel. The stages in the algorithm belong to one of two fundamental types discussed in the following. A stage is either an addition/subtraction stage or a multiplication stage. Therefore, two kernels are developed which can be configured to fit the steps in any of the stages.

3.4.1 Addition/subtraction kernel

Through analysis of the stages of the proposed reschedule in the previous section, it can be observed that there are 2 types of addition/subtraction stages as follows:

1. Type 3.4.1.1: stages 1, 4, and 7 where 4 submatrix addition/subtraction operations are performed.
2. Type 3.4.1.2: stages 3 and 6 where 2 submatrix addition/subtraction operations are performed.

The kernel splits the threads into either 4 groups or 2 groups for stages of type 3.4.1.1 and 3.4.1.2, respectively. Each thread group is assigned the pointers corresponding to the submatrices in their respective step in the stage.

The addition kernel performs one level of prefetching of $Ms \times Ks$ chunk to the shared memory to interleave computations with load instructions from global memory.

3.4.2 Multiplication kernel

There are two multiplication stages in the proposed reschedule as follows:

1. Type 3.4.2.1: stage 2 which consists of 4 submatrix multiplications with each result getting updated to 2 submatrices of C.
2. Type 3.4.2.2: stage 5 which consists of 3 submatrix multiplications where one of the results is added to 2 submatrices of C and the other 2 results are added to one submatrix each of C.

The kernel can be configured to split the threads into 4 groups for type 3.4.2.1 stage and into 3 groups for type 3.4.2.2 stage. Each thread group is assigned the pointer to the C submatrix to which they are intended to update the results of multiplication.

The kernel performs prefetches from global memory to shared memory and from shared memory to registers to hide memory latencies and overlap load operations with computations.

3.4.3 Optimizing the kernel launch parameters

As discussed in section 2.5, every kernel launched in CUDA requires the block dimensions and grid dimensions to be supplied in dim3 data type. These parameters define the launch configuration of the kernel. [1] has conducted experiments to determine the ideal launch configurations for their implementations.

During testing it was realized that the recommendations from [1] are unbefitting for MSMES. By profiling the kernels using nvprof [48], it was determined that the usage of launch configurations recommended by [1] resulted in higher register usage by each thread performing MSMES. This higher register consumption exhausted the available register per SM (Streaming Multiprocessor) and limited the number of threads that could run concurrently thereby reducing the thread concurrency of MSMES.

NVIDIA's CUDA occupancy calculator [39] was used to fine tune the launch parameters as well as the values of *Mr*, *Nr,* and *Ks* mentioned in section 3.2.1. By setting *Mr* = *Nr* = *Ks* = 8 and by setting count of threads in the block to 128, the register consumption was lowered, and the warp concurrency results shown in figures 3.2, 3.3 and 3.4 were obtained. The results shown are for an NVIDIA GPU with compute capability 7.5 using CUDA version 11.1 which has a shared memory size of 65536.

Future adaptations to a GPU with a different compute capability or with a different CUDA version should go through similar analysis to determine the configurations as the register file size, shared memory size and max warp occupancy of the multiprocessor can vary from GPU to GPU.

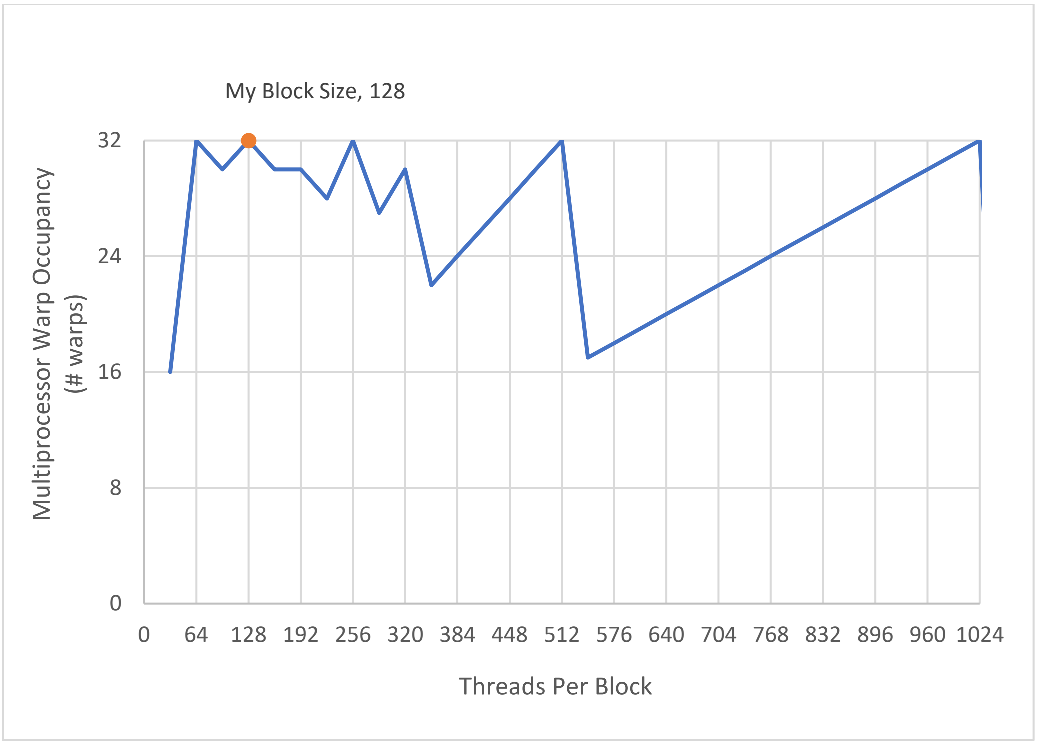


Figure 3.2: Variation of warp occupancy with threads per block.

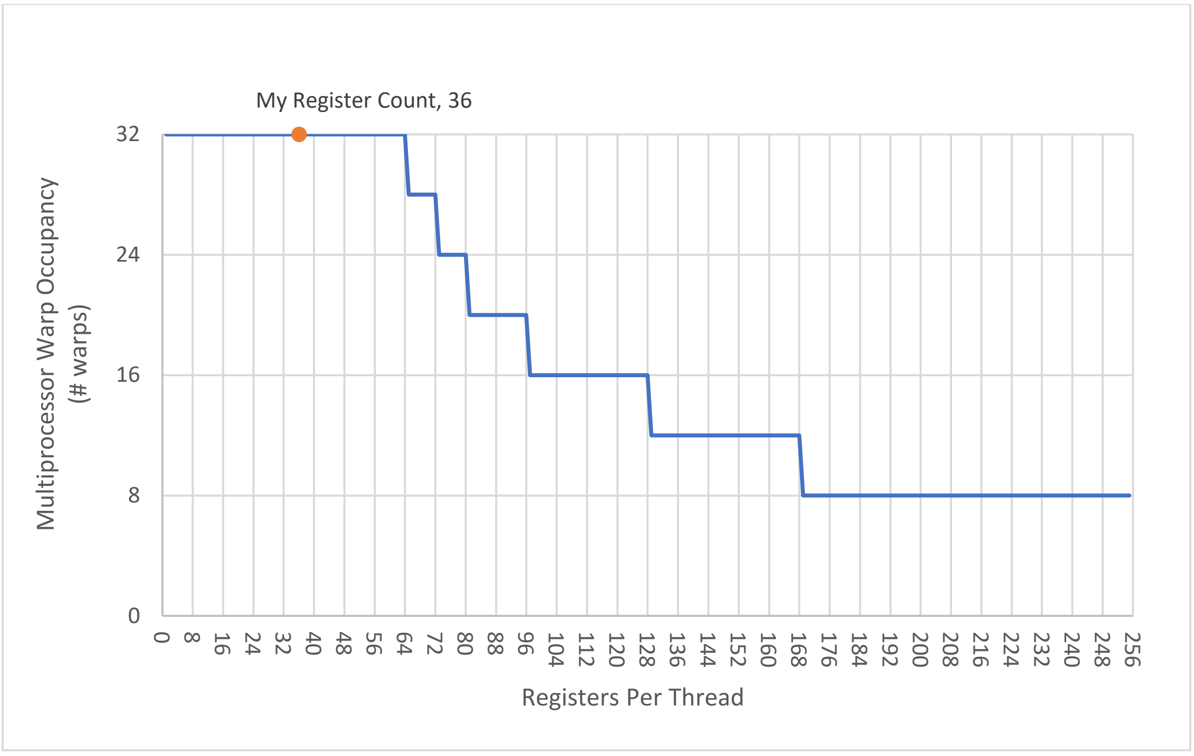


Figure 3.3: Variation of warp occupancy with registers per thread.

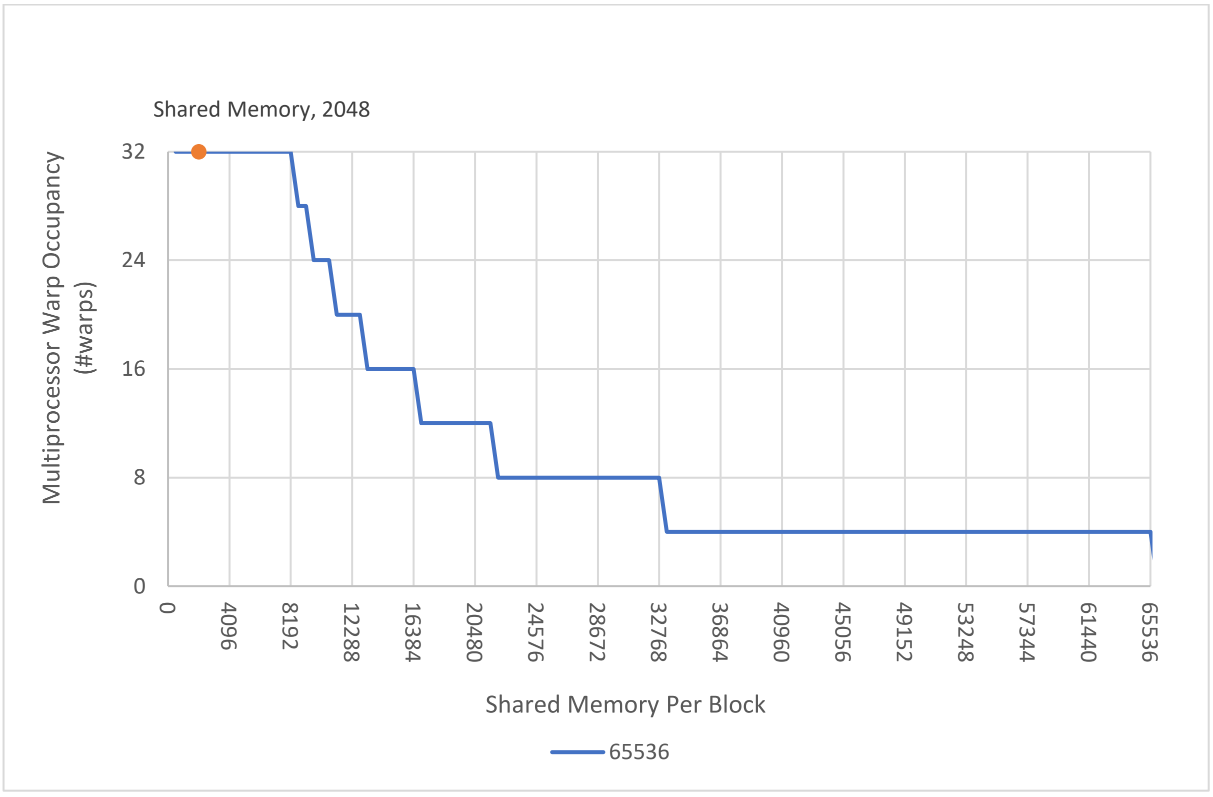


Figure 3.4: Variation of warp occupancy with shared memory per block.

3.4.4 *Experiments*

Experiments were performed with the following configurations:

Parameter	**Configuration (a)**	**Configuration (b)**
CPU	Intel Xeon Gold 6140 × 2	Intel Core i5 9400
GPU	Tesla V100 × 11	GTX 1660ti
RAM	383GB	16GB
OS	Scientific Linux 7.9 (Nitrogen)	Windows 10
CUDA version	11.0	11.1
C compiler version	GCC 4.8.5	Microsoft C/C++ 19.27.29112

Table 3.1: Test Configurations.

Both cublasDgemm and MSMES were provided the same double precision input matrices generated using curandGenerateUniformDouble [49]. Execution times were measured using CUDA events which has a resolution of approximately half a microsecond [9]. The effective throughput and memory consumption are considered as the primary metric to evaluate performance.

Effective throughput computed using equation (3) gives actual throughput for classical cubic matrix multiplication algorithms which performs $2N^3$floating point operations. For sub-cubic algorithms like Strassen's algorithm, equation (3) gives performance relative to classical algorithms.

$$Effective\ throughput\ in\ TFLOPS = \frac{2 \times N^3}{time\ in\ s} \times 10^{-12} \quad (3)$$

Efficacy of global memory usage was evaluated by running the kernels with successively larger matrices until the kernel ran out of space. Three experiments were conducted to evaluate the performance of MSMES:

1) The performance of MSMES in double precision multiplication was compared against the cublasDgemm kernel [45]. On configuration (b), MSMES outperformed the

cublasDgemm kernel for matrices of size as low as 896 (N=896 for an N × N matrix) (Figure 3.5). Beyond the crossover point, MSMES outperformed cublasDgemm kernel by 7.73% on the average. On configuration (a) beyond matrix of size 1152 (N=1152 for an N × N matrix), MSMES outperformed cublasDgemm kernel by 7.12% on the average (Figure 3.6).

2) MSMES has memory consumption comparable to cublasDgemm. This is an expected outcome because the multiplication kernel used in our implementation is a modified version of the CUTLASS GEMM kernel. On configuration (a) with 6GB of global memory, MSMES computed double precision matrix multiplication for square matrices as large as 14976 (N=14976 for an N × N matrix), whereas cublasDgemm failed to initialize beyond square matrix dimension of 14720 (N=14720 for an N × N matrix). On configuration (b) with 16GB of global memory, MSMES computed double precision square matrix multiplication for matrices of dimension as large as 24576, whereas cublasDgemm failed to initialize beyond 24320.
3) N-level MSMES outperformed 1-level MSMES and cublasDgemm over the test range with no extra memory consumption. On configuration (a), 2-level MSMES yielded a maximum speedup of 1.125 against 1-level (i.e. non-recursive) MSMES at N = 2048. 3-level MSMES yielded a maximum speedup of 1.21 against 1-level MSMES at N = 16384 (Figures 3.7 and 3.8).
4) When compared to cublasDgemm, 2-level MSMES yielded a maximum speedup of 1.23 at N = 7168 and 3-level MSMES yielded a maximum speedup of 1.25 at N = 7168 (Figures 3.7 and 3.8).

While MSMES can perform deeper recursions for multi-level Strassen's algorithm with no extra memory demands, experiments were limited to 3-levels due to the known numerical instability of Strassen's algorithm at higher levels of recursion.

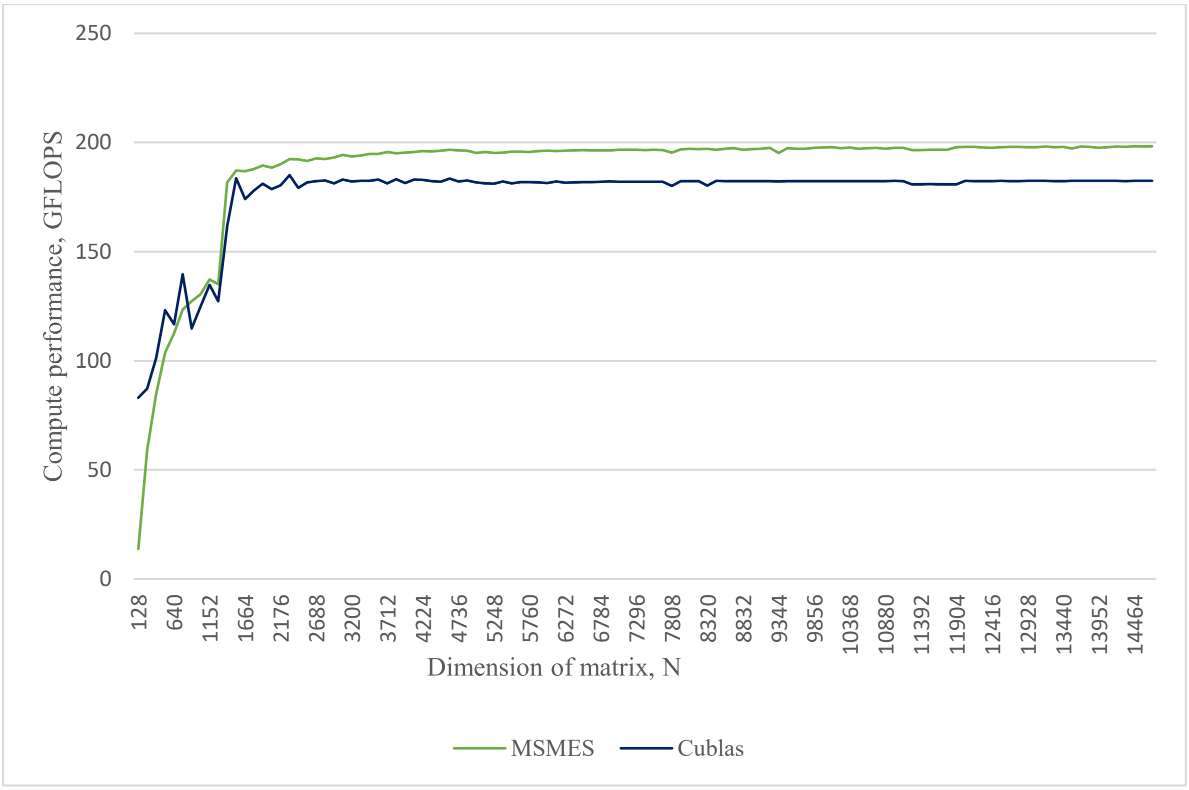


Figure 3.5. MSMES v/s CUBLAS double precision performance in configuration (b)

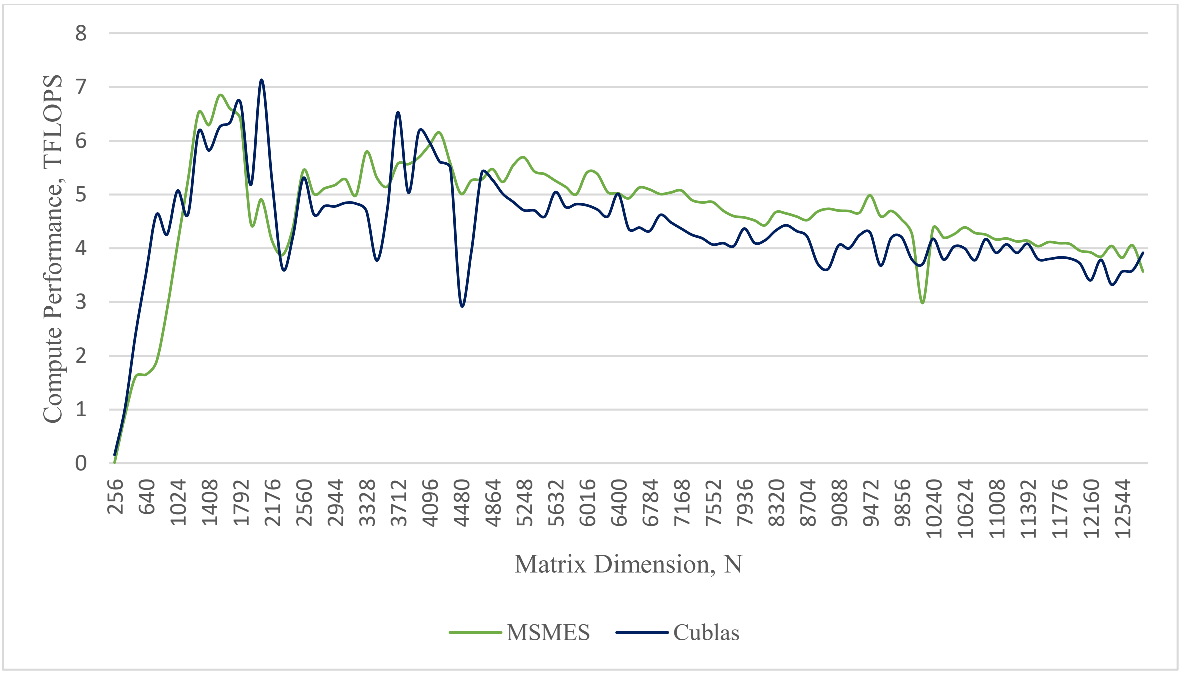


Figure 3.6. MSMES v/s CUBLAS double precision performance in configuration (a)

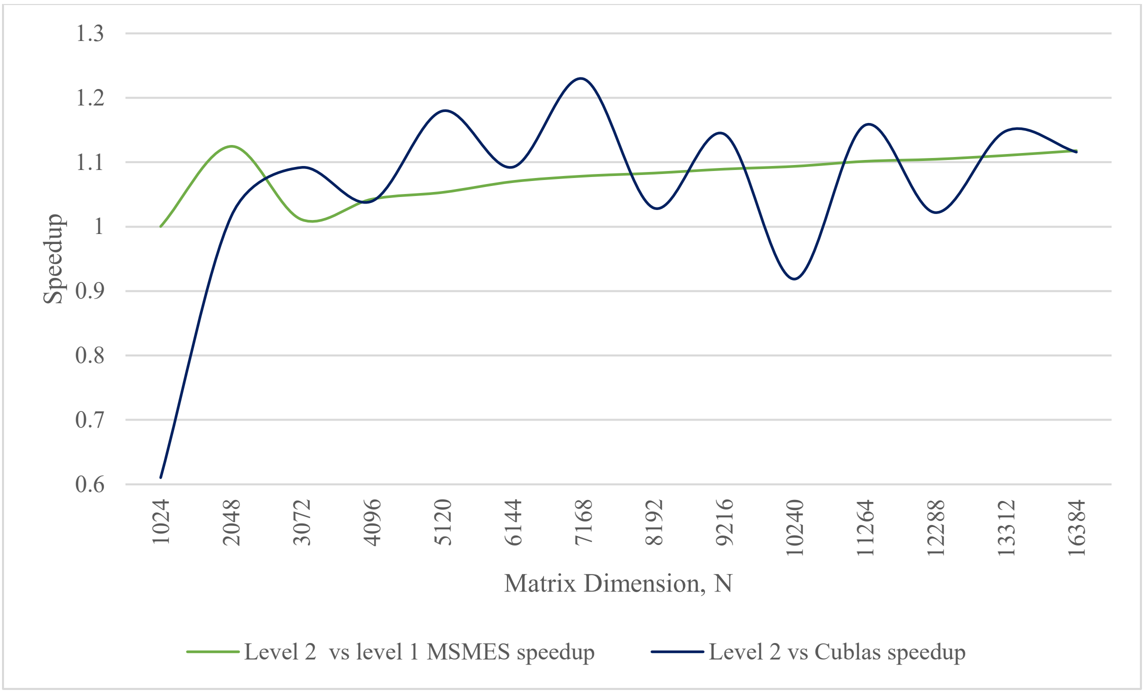


Figure 3.7. Speedup of Level 2 MSMES against Level 1 MSMES v/s Level 2 MSMES against CUBLAS in configuration (a)

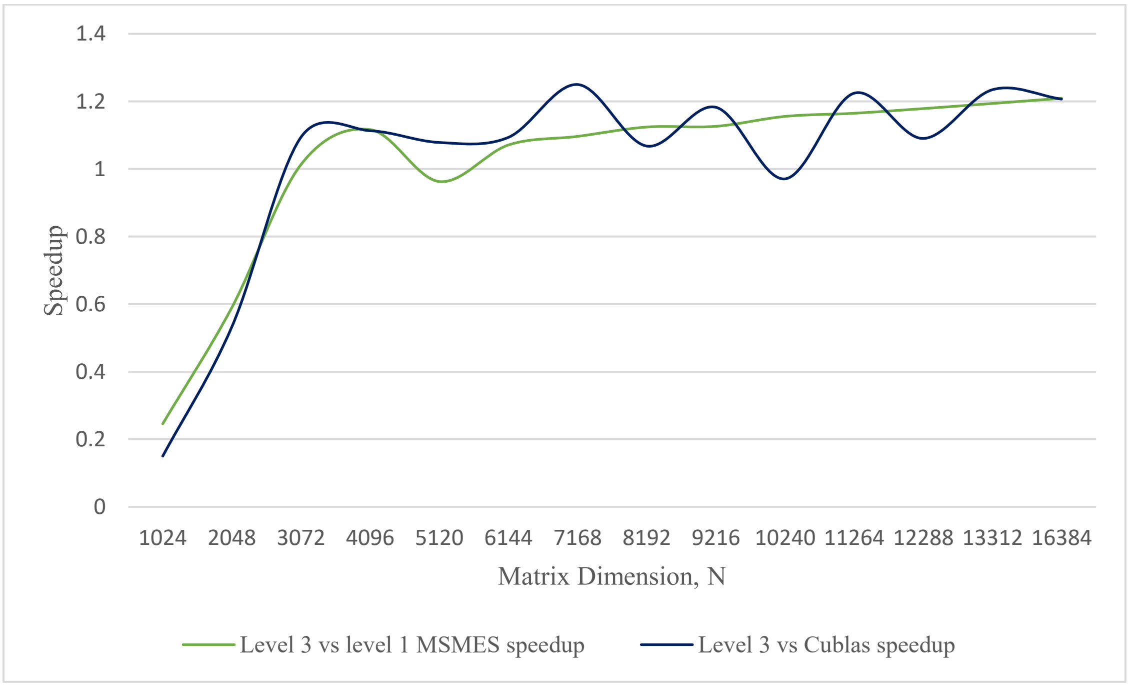


Figure 3.8. Speedup of Level 3 MSMES to Level 1 MSMES v/s Level 3 MSMES to CUBLAS in configuration (a)

3.5 Performance Analysis

In this section, the performance of MSMES is analyzed by inspecting the contribution of each stage to the total execution time. Such an analysis would aid in the discovery of bottlenecks, if any, and would also allow to quantify the impact of performing recovery stages 6 and 7.

Later in the section, a performance model for MSMES is presented by analyzing the computation and communication involved in the implementation.

3.5.1 Stage-wise breakdown of exectuion time

The time consumed by each kernel launch associated with a stage was measured using CUDA events. The results obtained from CUDA events were further validated by profiling the kernel with nvprof [49] using the same launch configurations. The timing data is illustrated in Table 3.2.

N	**Contributions as percentage of total execution time**						
	Stage 1	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**	**Stage 6**	**Stage 7**
256	0.79	54.38	0.39	0.78	42.48	0.4	0.78
1792	0.72	54.9	0.38	0.74	42.14	0.38	0.74
3328	0.64	55.31	0.32	0.62	42.18	0.31	0.62
4864	0.52	56.2	0.29	0.55	41.62	0.3	0.52
6400	0.41	56.68	0.21	0.41	41.67	0.21	0.41
7936	0.39	56.81	0.2	0.39	41.63	0.2	0.38
9472	0.36	57.14	0.19	0.36	41.41	0.18	0.36
11008	0.33	57.33	0.18	0.34	41.29	0.19	0.34
12544	0.33	57.8	0.18	0.33	40.83	0.18	0.35
Average	0.5	56.3	0.26	0.5	41.7	0.26	0.5

Table 3.2: Stagewise contribution to the total execution time.

It was observed from the data that the addition/subtraction kernels of type 3.4.1.1 (refer to section 3.4) are each responsible for roughly 0.5% (average contribution from all the rows in table 3.2) of the total execution time. Addition/subtraction kernels of type 3.4.1.2 (refer to section 3.4) are each responsible for approximately 0.26% (average contribution from all the rows in table 3.2) of the total execution time. This difference in contributions between 3.4.1.1 and 3.4.1.2 of approximately 48% arises because 3.4.1.1 performs twice as many computations as 3.4.1.2 per kernel launch.

Multiplication kernel of type 3.4.2.1 and 3.4.2.2 (refer to section 3.4) contributes 56.3% and 41.7% (average contribution from table 3.2) respectively to the total execution time. The difference in contributions between 3.4.2.1 and 3.4.2.2 of roughly 30% arises because 3.4.2.2 only performs $\frac{3}{4}$th of the number of computations performed by 3.4.2.1 per kernel launch.

It was also observed from the data that the contributions of the addition/subtraction kernels decrease with the increase in problem size. This trend was expected since each of the addition/subtraction kernels has a time complexity of $O(n^2)$, where n is the dimension of the $n \times n$ submatrix under consideration at the current level of recursion. In contrast, the multiplication kernels have time complexity of $O(n^3)$.

It can be calculated from the average contributions of stages 6 and 7 (across all the rows of table 3.2) to the total execution time that the recovery of the input matrices adds to approximately 0.76% to the total execution time, which is in fact quite negligible.

3.5.2 Performance Modeling

MSMES has three time-consuming operations: performing arithmetic operations on the matrix elements (addition, subtraction, or multiplication), loading matrix elements from global memory to shared memory, and loading matrix elements from shared memory to registers. The implementation presented here overlaps memory operations with computations. Therefore, the lower bound of total execution time, T, can be represented as:

$$T \geq max(T_{flop}, T_{gmop}, T_{smop})$$

Here, T_{flop} is the time taken for performing the arithmetic operations. T_{gmop} is the time taken for global memory loads and stores, and T_{smop} is the time taken to perform shared memory operations.

The following assumptions are made in the prediction model.

1. Peak FP64 performance of the GPU, τ_{flops}, is assumed to be available throughout the execution, and this performance is uniformly distributed across all the active threads.
2. Peak global memory bandwidth, τ_{gmops}, is available throughout the execution and is uniformly distributed among the active threads.
3. Peak shared memory bandwidth, τ_{smops}, is available throughout the executions and is uniformly distributed among the active threads.

In real-life scenario, where multiple processes might be requesting GPU resources at the same time, these assumptions may not hold. Hence, the model provided here gives the theoretical best-case performance of MSMES.

The MSMES implementation with recovery stages activated has three kernel invocations of type 3.4.1.1 (stages 1, 4 and 7), two kernel invocations of type 3.4.1.2 (stages 3 and 6), one kernel invocation of type 3.4.2.1 (stage 2), and one kernel invocation of type 3.4.2.2 (stage 5). Therefore, the total values being computed in the following sections are aggregates of the contributions of the seven kernels we just outlined.

1) Arithmetic operations time

Kernels of type 3.4.1.1 perform four matrix addition operations on sub matrices of dimension $\frac{N}{2} \times \frac{N}{2}$ and hence all together perform N^2 computations. Kernels of type 3.4.1.2 perform two matrix addition operations on sub matrices of dimension $\frac{N}{2} \times \frac{N}{2}$ and hence all together perform $\frac{N^2}{2}$ computations. Kernel of type 3.4.2.1 performs four matrix multiplication operations followed by eight matrix additions on sub matrices of dimension $\frac{N}{2} \times \frac{N}{2}$ and hence all together perform $\left(\frac{8N^3}{8} + 2N^2\right)$ computations. Kernel of type 3.4.2.2 performs three matrix multiplication operations followed by four matrix additions on sub matrices of dimension $\frac{N}{2} \times \frac{N}{2}$ and hence all together perform $\left(\frac{6N^3}{8} + N^2\right)$ computations.

Therefore, time for total arithmetic operations can be calculated as (3 × contribution of 3.4.1.1 + 2 × contribution of 3.4.1.2 + contribution of 3.4.2.1 + contribution of 3.4.2.2):

$$T_{flop} = \frac{\frac{7N^3}{4} + 7N^2}{\tau_{flops}}$$

2) Global memory operations time

Kernels of type 3.4.1.1 access eight submatrices of dimension $\frac{N}{2} \times \frac{N}{2}$ and hence perform $(2N^2)$ global memory operations. Similarly, kernels of type 3.4.2.1 access four sub matrices of dimension $\frac{N}{2} \times \frac{N}{2}$ and hence perform (N^2) global memory operations.

For computing global memory operations of kernels of type 3.4.2, we calculate the memory operations of an individual thread block. This value is then multiplied by the total number of thread-blocks per kernel launch to compute the total global memory operations performed by the kernel launch. Given operand matrices of size $N \times N$ (a kernel is provided the entire operand matrix whereas the individual stage being addressed in the kernel uses only sub matrices of dimension $\frac{N}{2} \times \frac{N}{2}$), kernel of type 3.4.2.1 uses $\frac{N^2}{N_S^2}$ thread blocks per launch, where each thread block computes an $(N_S \times N_S)$ tile of the output matrix. With the reduced computations in kernel 3.4.2.2, we have $\frac{3N^2}{4N_S^2}$ thread-blocks per launch of kernel. In kernel type 3.4.2.1, each thread block accesses $N_S \times \frac{N}{2}$ blocks of each of the input matrices for matrix multiplication and later accesses two tiles of size $N_S \times N_S$ to perform matrix addition/subtraction to update the results. Therefore, 3.4.2.1 performs $\frac{N^2}{N_S^2} \times (2N_S^2 + N_S \times N)$ global memory operations. In kernel type 3.4.2.2, each thread block accesses $N_S \times \frac{N}{2}$ blocks of each of the input matrices for matrix multiplication and later a third of the thread-blocks access two tiles of size $N_S \times N_S$ to perform matrix addition/subtraction to update the results; the remaining thread-blocks access one tile of $N_S \times N_S$ to perform updates. Therefore, 3.4.2.2 performs $\frac{3N^2}{4N_S^2} \times (N_S \times N) + \frac{N^2}{4N_S^2} \times (2N_S^2) + \frac{N^2}{2N_S^2} \times (N_S^2)$ global memory operations. Summing up, total time for global memory operations can be calculated as:

$$T_{gmop} = \frac{11N^2 + \frac{7N^3}{4N_S}}{\tau_{gmops}}$$

3) Shared memory operations time

Kernels of type 3.4.1.1 and 3.4.1.2 perform $(2N^2)$ and (N^2) shared memory operations, respectively. Given thread block dimensions t_x and t_y, kernel of type 3.4.2.1 has $\frac{N^2 \times (t_x \times t_y)}{N_S^2}$ threads. Kernel of type 3.4.2.2 has $\frac{3N^2(t_x \times t_y)}{4N_S^2}$ threads. The shared memory operation of a kernel is calculated by multiplying the shared memory operation of one of its threads with the thread count of the kernel. In kernel 3.4.2.1, all the threads access $N_R \times N$ row of A and $N_R \times N$ column of B. The values computed are then written back to two $N_R \times N_R$ tiles of C. In kernel 3.4.2.2, all the threads access $N_R \times N$ row of A and $N_R \times N$ column of B. Later a third of the threads in 3.4.2.2 accesses two $N_R \times N_R$ tiles of C to update the results to. The remaining threads in 3.4.2.2 accesses only one $N_R \times N_R$ tile of C to update the results to. Therefore, Kernels of type 3.4.2.1 and 3.4.2.2 perform $\frac{N^2 \times (t_x \times t_y)}{N_S^2} \times (2N_R^2 + N_R \times N)$ and $\frac{3N^2(t_x \times t_y)}{4N_S^2} \times (N_R \times N) + \frac{N^2(t_x \times t_y)}{4N_S^2} \times (2N_R^2) + \frac{N^2}{2N_S^2} \times (N_R^2)$ shared memory operations. Hence, time for shared memory operations can be calculated as:

$$T_{smop} = \frac{11N^2 + \frac{7N^3(t_x \times t_y)}{4N_S^2} \times N_R}{\tau_{smops}}$$

3.6 Conclusion

The chapter presents an implementation of Strassen's matrix multiplication algorithm in CUDA, titled Multi-Stage Memory Efficient Strassen (MSMES), that eliminates additional workspace requirements by reusing and recovering input matrices. MSMES organizes the steps involved in Strassen's algorithm into five stages where multiple steps in the same stage can be executed in parallel. Two additional stages are also discussed in the chapter that allows the recovery of input matrices. MSMES has no additional memory requirements, irrespective of the depth of recursions of Strassen's algorithm. Tests were performed to measure compute performance and memory utilization on two hardware configurations. On configuration (a) which consists of a Tesla V100 GPU, MSMES on average outperformed the CUDA library matrix multiplication function cublasDgemm beyond matrices of dimension 1152. On configuration (b) which consists of a GTX 1660ti, MSMES outperformed cublasDgemm for matrices of dimension as small as 896. On either configuration, MSMES was able to accommodate larger matrices than cublasDgemm, thereby proving the lower memory requirements of MSMES. The ability of MSMES to perform multi-level Strassen with no additional global, shared, and register memory demands make it suitable for applications where numerical stability is not essential.

Chapter 4 Multi-GPU Strassen's algorithm

4.1 Introduction:

With the shift of parallel computing to GPGPU paradigm, supercomputing clusters are increasingly depending on the compute performance of GPUs to improve its Rpeak and Rmax scores [23]. In fact, Six out of ten of the most performant supercomputers in the world uses GPUs to derive more than 90% of its compute performance [24]. This shift makes it necessary to adapt fundamental operations like matrix multiplication to multi-GPU architectures.

While there were previous studies into adapting Strassen's algorithm to multi-GPU architectures [19, 20], which decomposed the matrix using Strassen's algorithm followed by using CUBLAS to perform the actual multiplication, there were no previous attempts at using Strasen to decompose the matrix followed by using Strassen's algorithm to perform the actual matrix multiplication. The work in [2] has already explored a global Strassen decomposition followed by local Strassen multiplication on a distributed memory multi-CPU cluster and has clearly shown the communication and computation advantages of a similar scheme. With the performance benefits of MSMES that we have shown in the previous chapter, it was evident that there is a need for a multi-GPU implementation of MSMES following the recommendations of [2].

4.2 Background:

Owing to the variations of tests being performed to analyze the impact of various multi-GPU matrix multiplication schemes, we use a modularized approach for our multi-GPU discussions. Each multi-GPU matrix multiplication scheme consists of two modules.

4.2.1 Module 1: Global decomposition module

This module is responsible for the global decomposition of the input matrix. The algorithm used in the decomposition of the input matrix generates subtasks which are distributed across the participating GPUs to perform computations defined in the subtasks. Through analysis of the data movement patterns and computation complexity associated with the subtasks, it is possible to identify optimizations that improve GPU utilization, amount of data movement, bottleneck subtasks etc. The following decomposition schemes were used in our work. All the decomposition techniques used here decomposes the matrix as shown in figure 4.1.

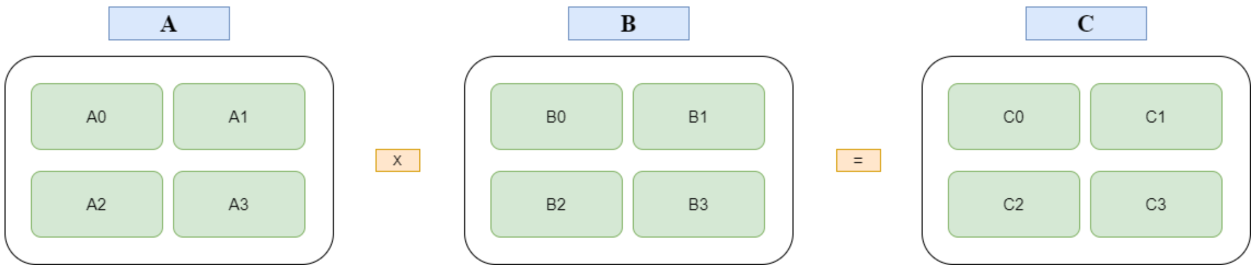


Figure 4.1: Decomposition of matrices for multi-GPU multiplication.

Naïve decomposition

Naïve or 2D scheme decomposes the matrix multiplication into subtasks listed in table 4.1. The decomposition technique is called 2D due to its ability to map to a two-dimensional mesh topology of processors. From the table, it can be observed that for a 2 × 2 mesh topology, the scheme would decompose the matrix multiplication into 8 submatrix multiplications followed by 4 submatrix additions. If the dimension N of the input matrices (for N × N matrices) is a multiple of P for a P × P mesh topology, it can be observed that the subtasks would evenly distribute among the processors.

Operation	**Result**
$A_0 \times B_0$	R_1
$A_1 \times B_2$	R_2
$A_0 \times B_1$	R_3
$A_1 \times B_3$	R_4
$A_2 \times B_0$	R_5
$A_3 \times B_2$	R_6
$A_2 \times B_1$	R_7
$A_3 \times B_3$	R_8
$R_1 + R_2$	C_0
$R_3 + R_4$	C_1
$R_5 + R_6$	C_2

$R_7 + R_8$	C_3

Table 4.1: Subtasks in naïve global decomposition.

Strassen's decomposition.

Strassen's scheme decomposes the matrix multiplication by applying Strassen's algorithm on the input matrices. The tasks generated after this decomposition are listed in table 2.1. From the table it can be observed that one level of Strassen decomposition generates 7 submatrix multiplications and 18 submatrix additions/subtractions. With 7 GPUs used for the submatrix multiplication it can be observed that while the submatrix multiplications map evenly, the addition/subtraction subtasks will be mapped unevenly (18 tasks mapped to 7 processors). Even though the cost associated with the addition/subtraction stages are observed to be nominal in section 3.5.1, the existence of the Strassen-Winograd algorithm made this scheme unbefitting for our evaluations.

Strassen-Winograd decomposition.

Strassen-Winograd algorithm is an improvement on Strassen's algorithm where the number of additions/subtractions are reduced to 14 from the 18 in Strassen's algorithm. The tasks generated by applying 1 level of Strassen-Winograd scheme are listed in table 4.2. From the table it can be observed that this scheme generates 7 submatrix multiplication and 15 submatrix additions/subtractions. With 7 GPUs used in the operation, the 7 submatrix multiplications map evenly. Out of the 15 submatrix additions/subtractions, 8 are performed prior to the submatrix multiplication and the remaining 7 are performed after the submatrix multiplication. Even though, the submatrix additions/subtractions do not evenly map to the number of processors used, a technique is discussed in section 4.4.1 to improve the load distribution.

Task No.	Operation	Result
1	A_0	T_0
2	A_1	T_1
3	$A_2 + A_3$	T_2
4	$T_2 - A_1$	T_3
5	$A_0 - A_2$	T_4
6	$A_1 + T_3$	T_5
7	A_3	T_6

8	B_0	S_0
9	B_2	S_1
10	$B_1 + B_0$	S_2
11	$B_3 - S_2$	S_3
12	$B_3 - B_1$	S_4
13	B_3	S_5
14	$S_3 - B_2$	S_6
15	$T_0 \times S_0$	Q_0
16	$T_1 \times S_1$	Q_1
17	$T_2 \times S_2$	Q_2
18	$T_3 \times S_3$	Q_3
19	$T_4 \times S_4$	Q_4
20	$T_5 \times S_5$	Q_5
21	$T_6 \times S_6$	Q_6
22	$Q_0 + Q_3$	U_1
23	$U_1 + Q_4$	U_2
24	$U_1 + Q_2$	U_3
25	$Q_0 + Q_1$	C_0
26	$U_3 + Q_5$	C_1
27	$U_2 - Q_6$	C_2
28	$U_2 + Q_2$	C_3

Table 4.2: Tasks in Strassen-Winograd decomposition.

4.2.2 Module 2: Local multiplication module.

This is the module that performs the actual multiplication operation locally on the participating processors.

cublasDgemm:

cublasDgemm is the CUDA implementation of BLAS multiplication operation for double precision floating point numbers [45].

MSMES (Multi-Stage Memory Efficient Strassen's algorithm):

MSMES is our implementation of a memory efficient Strassen's matrix multiplication algorithm on CUDA that was explored in detail in the previous chapter.

With the ability to combine different options for each of the modules, the configurations outlined in table 4.3 are explored in this chapter

Sl. No.	Global decomposition Scheme	Local multiplication algorithm
1	2D	cublasDgemm
2	2D	MSMES
3	Strassen-Winograd	cublasDgemm
4	Strassen-Winograd	MSMES

Table 4.3: Module combinations evaluated.

4.3 Previous Works:

The work in [3] designed a generic matrix-matrix multiplication algorithm for C = A x B for multi-GPU accelerated distributed memory platforms. They were able to overcome the limitations of SLATE [https://icl.bitbucket.io/slate/] library where C matrix must fit in the global memory of the GPUs. The work designed an algorithm around tiled matrix outer products with numerous optimizations to realize their results. They used data prefetching at various levels of memories like our work to overlap computations with communications. The work performs an implementation of GEMM in PARSEC [25] to perform matrix multiplications.

Zhang et al [19] implemented a multi-GPU version of matrix multiplication by decomposing the global matrix multiplication using Strassen's algorithm followed by using CUBLAS implementation of matrix multiplication at local levels to perform the actual multiplication. This corresponds to the technique outlined in entry number 3 of table 4.3 here. [19] investigated the performance of distributed matrix multiplication by leveraging GPUs to perform the addition and multiplication stages of Strassen's algorithm as well as using CPUs for the addition stages and GPUs for multiplication stages. Based on their implementation, [19] arrived at the conclusion that naïve matrix multiplication solutions outperform Strassen's algorithm on multi-GPU architectures due to the communication overheads associated with Strassen's algorithm.

Zhang et al [20] designed a middleware for scheduling tasks in operations with task dependencies on multi-GPU architectures. Strassen's matrix multiplication was used as one of their example implementations to measure the performance of their hierarchical scheduler middleware. The scheduler assigns tasks to either CPUs or GPUs depending on the computational complexity of the task. In the example of Strassen's matrix multiplication, addition/subtraction tasks were assigned to CPU cores which performs single and double precision addition/subtraction admirably compared to GPUs when the cost of submatrix communication to GPU global memory is taken into consideration. On the other hand, multiplication operations are assigned to GPUs which can outperform CPUs on the operation due to the higher parallelism associated with GPUs. This implementation is again an example of global Strassen decomposition of matrices followed by using CUBLAS implementation of matrix multiplication at local levels to perform the actual multiplication.

Ballard et al. [2] has extensively documented the communication cost of matrix multiplication on Cray XT4, a distributed memory MIMD supercomputer [26]. They analyzed and modeled the communication costs involved with various decomposition strategies as well as multiplication strategies. The work performed in multi-GPU implementation of MSMES is an adaptation of the Communication Optimal Parallel Strassen Multiplication algorithm [2] on a multi-GPU CUDA architecture.

The following schemes were explored in [2]:

1. Classical Algorithm on memory bound architectures: This algorithm decomposes the input matrices into tiles that can be fit into the global memory of the CPU. These tiles are then mapped to the available CPUs which loads the corresponding matrices into their respective global memory and performs a classical (cubic) matrix multiplication algorithm on them.
2. 2D global decomposition with local Strassen's matrix multiplication: These algorithms also decompose the input matrices into tiles that fit into the global memory of the CPUs and maps the tiles to the participating CPUs. Once the tile pair is loaded into the CPU, it performs Strassen-Winograd algorithm on the pair of tiles.
3. Strassen decomposition with local classical matrix multiplication: This algorithm traverses sufficient DFS steps in a multi-level Strassen decomposition tree such that

the operand matrices after DFS traversal will fit in the global memory allocated to the CPU. These decomposed matrices are then assigned to CPUs which perform a classical matrix multiplication on the submatrix.

4. Strassen decomposition with local Strassen's multiplication: This algorithm traverses sufficient DFS steps in a multi-level Strassen decomposition tree such that the operand matrices after DFS traversal will fit in the global memory allocated to the CPU. These decomposed matrices are then assigned to CPUs which perform Strassen-Winograd algorithm on the submatrix.

4.5 Implementation:

This section covers the specifics of implementing the schemes explored under Module 1 of our multi-GPU matrix multiplication implementation. The details of implementing MSMES has already been documented in the previous chapter and shall not be repeated here. We also document some of the atypical behavior we observed while using some additional features offered by CUDA which should have theoretically improved our implementation.

4.4.1 Strassen-Winograd decomposition.

The Strassen-Winograd decomposition essentially consists of three stages. The pre-multiplication stage that consists of addition/subtraction tasks, the multiplication stage that performs 7 submatrix multiplications and the post multiplication addition/subtraction stage. There are 8 addition/subtraction operations in the pre multiplication stage with only 7 GPUs to process them. Here, data parallelism was leveraged by splitting each of the operand submatrix involved in the pre multiplication operations into 7 tiles. Similar optimizations were applied to the post multiplication stage as well. Figure 4.2 shows an example of the split. Next, subtasks were grouped together to improve task parallelism. Table 4.4 shows the details of the groups that were created. Choosing to use data parallelism in groups 1, 2 and 4 not only allowed improved load balancing but it also eliminated the inter task dependencies from slowing down execution. For example, from Table 4.2 task 4 is dependent on the results of task 3, had these tasks been parallelized leveraging task parallelism, the GPU performing task 4 would have had to wait for the results from the GPU performing task 3.

Initially all the global memory allocations on the GPU were made using cudaMallocManaged, a CUDA library implementation for unified memory [27] and atomidAddSystem [28] directive was used to avoid race conditions. This implementation could do away with the entire group 4 in table 4.4 as the computed results from group 3 could be atomically added to their respective C accumulator values. Unfortunately, the frequent memory transfers associated with a device managed memory resulted in subpar performance during group 3 subtasks. Later implementations used cudaMallocManaged only with groups 1, 2 and 4.

While tests were only conducted with matrices whose dimensions were multiples of 7, padding a matrix to make its dimension a multiple of 7 is a trivial problem and could therefore, theoretically allow any matrix to be decomposed using the described Strassen-Winograd decomposition scheme.

Group No.	Operations	Type of parallelism used
Group 1	$A_2 + A_3$ $T_2 - A_1$ $A_0 - A_2$ $A_1 + T_3$	Data parallelism (Data is evenly distributed among GPUs and all the GPUs perform all the tasks)
Group 2	$B_1 + B_0$ $B_3 - S_2$ $B_3 - B_1$ $S_3 - B_2$	Data parallelism (Data is evenly distributed among GPUs and all the GPUs perform all the tasks)
Group 3	$T_0 \times S_0$ $T_1 \times S_1$ $T_2 \times S_2$ $T_3 \times S_3$ $T_4 \times S_4$ $T_5 \times S_5$ $T_6 \times S_6$	Task parallelism (Each task is assigned to one GPU)
Group 4	$Q_0 + Q_3$ $U_1 + Q_4$ $U_1 + Q_2$ $Q_0 + Q_1$ $U_3 + Q_5$ $U_2 - Q_6$ $U_2 + Q_2$	Data Parallelism (Data is evenly distributed among GPUs and all the GPUs perform all the tasks)

Table 4.4: Grouping subtasks in Strassen-Winograd decomposition.

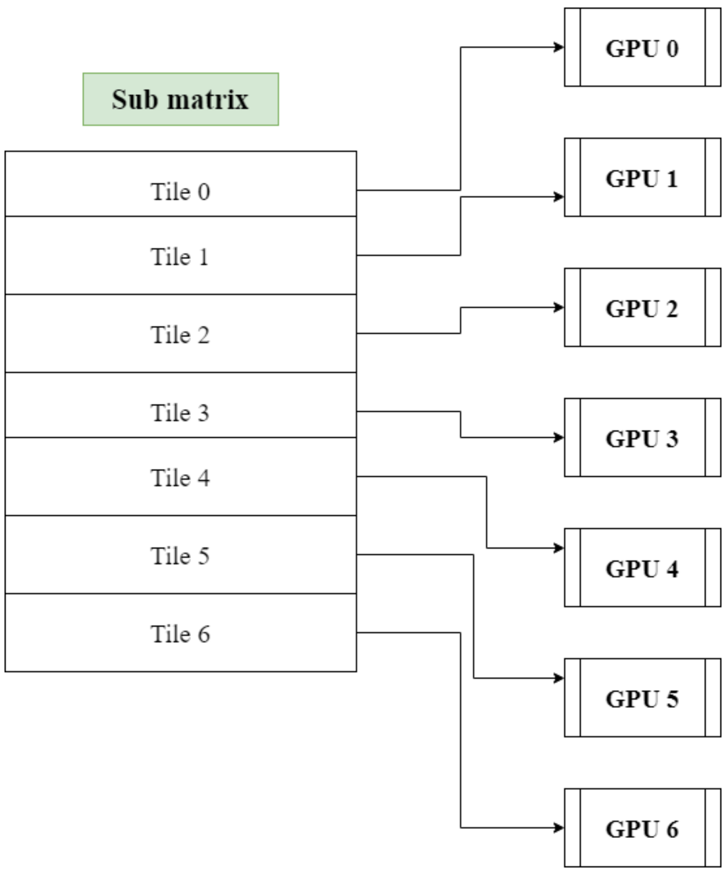
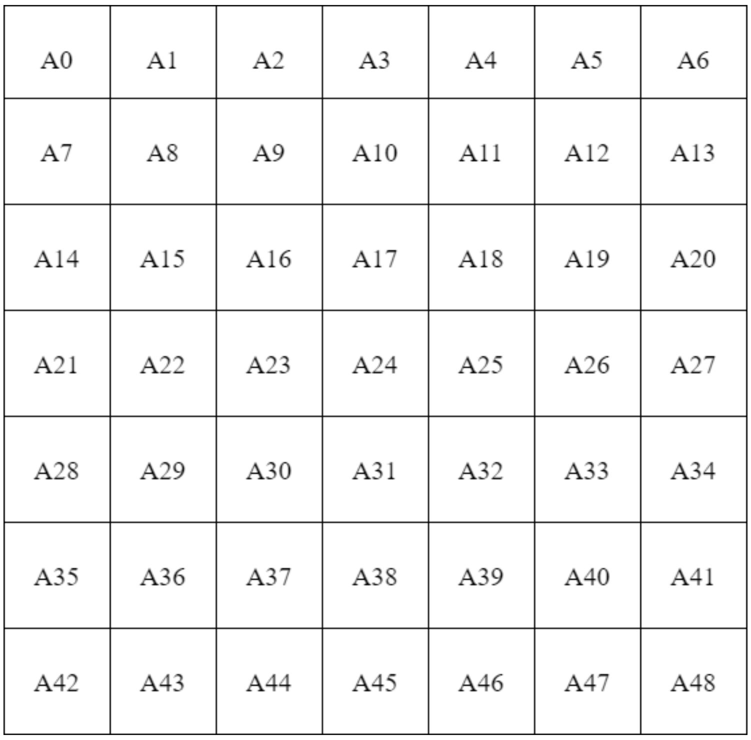


Figure 4.2: Splitting the data in submatrix across 7 GPUs.

4.4.2 Naïve/2D decomposition:

The naïve decomposition was optimized to be used with 7 GPUs to maintain the similitude of the results from Strassen-Winograd decomposition. The input matrices were decomposed as shown in Figure 7. While the 2D decomposition strategy maps more conveniently to 2D mesh topology of processors, it was not possible in our test environment since the maximum number of GPUs that could be used concurrently on environment(a) listed in table 2.3 were limited to 8. Still, the configuration we used with 7 GPUs have efficiencies comparable to tests run using a (2 × 2) mesh topology with 4 GPUs. The 7 GPU configuration performed at an average efficiency of 50.1% compared to 54.8% for the (2 × 2) mesh configuration. The subtasks involved with this decomposition were distributed statically across the participating GPUs.



A0	A1	A2	A3	A4	A5	A6
A7	A8	A9	A10	A11	A12	A13
A14	A15	A16	A17	A18	A19	A20
A21	A22	A23	A24	A25	A26	A27
A28	A29	A30	A31	A32	A33	A34
A35	A36	A37	A38	A39	A40	A41
A42	A43	A44	A45	A46	A47	A48

Figure 4.3: Decomposition strategy for matrices for 2D scheme with 7 GPUs.

In both the implementations, each GPU was assigned a separate CPU thread to orchestrate communications and computations.

4.6 Experiments:

Experiments were conducted on various size of input matrices (the matrices were square and had dimensions which were multiples of 7) containing double precision floating point numbers generated using the CUDA library function curandGenerateUniformDouble [29]. The time it took to complete matrix multiplications were calculated using the C++ library function time defined in time header file [30]. The speedups of the configurations listed in Table 4.3 compared to MSMES running on a single GPU were used to measure the performance of each of the configuration.

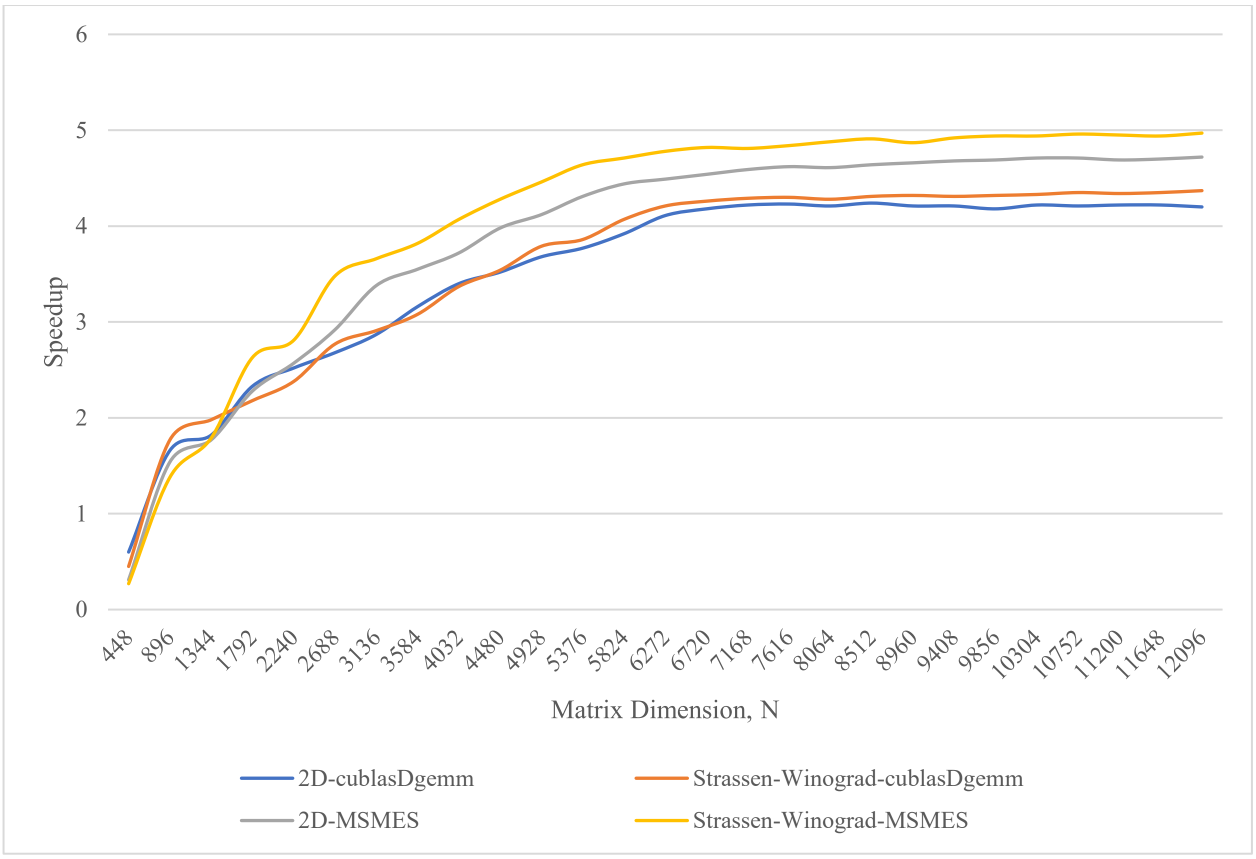


Figure 4.4: Speedup v/s Number of elements graph for the tested configurations

From figure 4.4 it can be observed that Strassen-Winograd-MSMES configuration has the highest speedup of 4.97 out of all the tested configuration. This configuration also outperformed all the other configurations beyond N = 1792. Below N = 1792, Strassen-Winograd-cublasDgemm exhibited superior performance.

4.7 Conclusion:

With the exponential increase in the consumption and generation of data, there is a compelling need to explore algorithms that can improve the processing speed of the data. Matrix multiplication is a fundamental linear algebra operation that finds extensive use in data intensive workflows like deep learning, computer graphics, scientific computing etc.

In this section we explored and implemented different configurations for multi-GPU matrix multiplication. Naïve and Strassen-Winograd decomposition techniques were implemented using cublasDgemm and MSMES and their performances were compared. From the experiments, it was observed that the configuration that used Strassen-Winograd scheme for decomposition with MSMES for local matrix multiplication yielded the best performance.

Chapter 5 Future Work and Conclusion

5.1 Introduction:

This chapter discusses the optimizations and solutions that can be augmented to the multi-GPU implementations discussed in the previous chapter. Since the initial motivation for the thesis came from exploring the chain matrix multiplication involved in the training phase of GCNs, one of the sections in this chapter is our recommendations for chained matrix multiplication.

The next section discusses the details for a theoretical load balancer that would allow the multi-GPU implementation to perform better in an architecture with heterogeneous GPUs or a shared architecture where some of the GPUs are behaving like stragglers due to load from other processes.

5.2 Matrix chain multiplication

The task of computing a matrix chain multiplication on a multi-GPU architecture can be decomposed in two different ways.

1. Each of the GPUs can be assigned a separate matrix multiplication involved in the chain.
2. The individual matrix multiplications in the chain can be assigned to multiple GPUs using Strassen-Winograd-MSMES decomposition discussed in the previous chapter.

While each technique has its pros and cons, this section takes into consideration the efficiency of algorithm and the limitations of a multi-GPU architecture to recommend a strategy for solving the matrix chain multiplication problem.

Our implementation of multi-GPU matrix multiplication using Strassen-Winograd decomposition followed by MSMES local multiplication yields a peak speedup of 4.97 while using 7 GPUs. While this configuration outperforms all the other configurations we tested, the multi-GPU implementations have lackluster efficiencies compared to single-GPU MSMES implementation. Given speedup of S using N GPUs, the efficiency (E) of a parallel algorithm can be calculated using the following formula:

$$E = \frac{S}{N}$$

This yields a peak efficiency of 71% for our Strassen-Winograd-MSMES implementation.

Consider the problem of multiplying a matrix chain of M matrices performed on a compute cluster with N processors. In supercomputing architectures like Cray where processing cores in the range of hundreds of thousands are available [33], up to a reasonable degree, it is possible to keep N > M. This flexibility on the number of processors allows these architectures to use many processors to leverage the speedups offered by even inefficient implementations to reduce the computation times of the matrix chain multiplication problem. The Virya compute cluster, that was used to design the implementations in this thesis, on the other hand has a hard limit of 8 GPUs per process [32]. This limits our ability to maintain N > M for all problem sizes. In such an architecture, as soon as M becomes greater than 2N (there is more than N independent matrix multiplications), we must consider the efficiency of an implementation before deciding on a technique to solve the problem.

Finally, the results of the optimal matrix parenthesization problem [34] for the given matrix chain also determines the technique that needs to be employed to solve the problem at a given stage.

With these constraints in mind, we recommend the following steps to perform a matrix chain multiplication on multi-GPU architectures.

1. Identify the optimal parenthesization for the given matrix chain.
2. If the number of optimal independent matrix multiplication is more than or equal to N, use the technique Strassen-Winograd-MSMES to perform the multiplication. Equation 5.1 provides an example of this when N = 3.
3. If the number of independent matrix multiplication is less than N, then compare the efficiency of performing an inefficient parenthesization to the efficiency of a Strassen-Winograd-MSMES decomposition to determine which strategy to apply. Equation 5.2 shows an example of this.
4. Recursively apply steps 1 to 3 until the chain is completed.

$$A1 \times (A2 \times A3) \times (A4 \times A5) \times (A6 \times A7)$$

Equation 5.1: This shows an example of a parenthesization where 3 GPUs will have 3 optimal independent matrix multiplications to solve parallelly. The independent multiplications are highlighted in green.

$$\Big(\big((A1 \times A2) \times A3\big) \times A4\Big) \times A5) \times (A6 \times A7)$$

Equation 5.2: This shows an example of a parenthesization where 3 GPUs will not have optimal independent matrix multiplications to solve parallelly.

5.3 Dynamic Load balancing

A fundamental assumption in our current implementation of Strassen-Winograd-MSMES is that the participating GPUs are of similar specifications and have similar performances. This assumption unfortunately does not hold true in a heterogeneous GPU configuration and or on a cluster where multiple processes might be competing for GPU time. This results in certain GPUs becoming stragglers thereby causing bottlenecks.

A proposed solution to this scenario is to further decompose each of the operations outlined in Table 4.4. By decomposing each of the operations into smaller chunks we can send the input matrices in a “streaming” fashion to GPUs. This would not only improve the overlap of computations with communications, but also allows the implementation to identify stragglers and divert their workloads to other available GPUs. Like our decomposition discussions in section 4.2.1, experiments need to be conducted for Strassen-Winograd vs naïve decomposition strategies to see the more performant solution for “streaming” the input matrices for the multiplication operations listed in table 4.4.

5.4 Conclusion

In this thesis, we present a novel implementation of Strassen's algorithm devoid of additional workspace requirements. This novel implementation is also capable of recursive Strassen's algorithm with no additional memory demands. The tests performed that compared the performance of the implementation to cublasDgemm show superior memory consumption and throughput. Performance models created allow the implementation to be optimized on various hardware configurations.

The single GPU implementation coined, MSMES, was leveraged to implement a multi-GPU version of Strassen's algorithm that outperformed other tested configurations.

With the augmentations we have mentioned in this section, the solution we present could be compelling for heterogeneous/shared multi-GPU matrix chain multiplication use cases like machine learning, deep learning, etc.

References

[1] J. Huang, C. Yu, and R. Van de Geijn, “Strassen's algorithm reloaded on GPUs,” Strassen's Algorithm Reloaded on GPUs | ACM Transactions on Mathematical Software, 01-Apr-2020.

[2] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication", Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12, 2012.

[3] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage", ACM Transactions on Mathematical Software, vol. 5, no. 3, pp. 308-323, 1979.

[4] "BLAS Techical Forum", Netlib.org. [Online]. Available: http://netlib.org/blas/blast-forum/.

[5] V. Strassen, "Gaussian elimination is not optimal", Numerische Mathematik, vol. 13, no. 4, pp. 354-356, 1969.

[6] Nvidia, "NVIDIA TESLA V100 GPU ARCHITECTURE", Nvidia, 2017.

[7] J. Fung, F. Tang and S. Mann, "Mediated Reality using Computer Graphics Hardware for Computer Vision", in International Symposium on Wearable Computing, Seattle, Washington, USA, 2002, pp. 7-10.

[8] M. Romero and R. Urra, "CUDA Overview", Cuda.ce.rit.edu, 2012.

[9] "Programming Guide :: CUDA Toolkit Documentation", Docs.nvidia.com, 2021.

[10] A. Kerr, D. Merrill, J. Demouth and J. Tran, "CUTLASS: Fast Linear Algebra in CUDA C++ | NVIDIA Developer Blog", NVIDIA Developer Blog, 2018.

[11] M. Harris, "Unified Memory for CUDA Beginners | NVIDIA Developer Blog", NVIDIA Developer Blog, 2017.

[12] J. Li, S. Ranka and S. Sahni, "Strassen's Matrix Multiplication on GPUs", 2011 IEEE 17th International Conference on Parallel and Distributed Systems, 2011.

[13] S. Ranka and S. Sahni, "SIMD Matrix Multiplication", Bilkent University Lecture Series, pp. 95-110, 1990.

[14] C. Douglas, M. Heroux, G. Slishman and R. 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.

[15] S. Huss-Lederman, E. Jacobson, A. Tsao, T. Turnbull and J. Johnson, "Implementation of Strassen's algorithm for matrix multiplication", Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '96, 1996.

[16] S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao and T. Turnbull, Strassen’s algorithm for matrix multiplication: Modeling, analysis, and implementation, CCS-TR-96-17, Center for Computing Sciences, 1996.

[17] P. Lai, H. Arafat, V. Elango and P. Sadayappan, "Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs", 20th Annual International Conference on High Performance Computing, 2013.

[18] B. Boyer, J. Dumas, C. Pernet and W. Zhou, "Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm", Proceedings of the 2009 international symposium on Symbolic and algebraic computation - ISSAC '09, 2009.

[19] P. Zhang and Y. Gao, “Matrix Multiplication on High-Density Multi-GPU Architectures: Theoretical and Experimental Investigations,” Lecture Notes in Computer Science, pp. 17–30, 2015.

[20] P. Zhang, Y. Gao and M. Qiu, "A Data-Oriented Method for Scheduling Dependent Tasks on High-Density Multi-GPU Systems," 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, 2015, pp. 694-699, doi: 10.1109/HPCC-CSS-ICESS.2015.314.

[21] T. Herault, Y. Robert, G. Bosilca and J. Dongarra, "Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC," 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2019, pp. 33-41, doi: 10.1109/ScalA49573.2019.00010.

[22] D. Coppersmith and S. Winograd, On the asymptotic complexity of matrix multiplication. SIAM Journal on Computing 11, 3 (1982), 472–492.

[23] In high-performance computing, what are Rmax and Rpeak? [Online]. Available: https://kb.iu.edu/d/bbzo.

[24] June 2021 | TOP500, Top500.org, 2021. [Online]. Available: https://www.top500.org/lists/top500/2021/06/.

[25] Parsec (parser) - Wikipedia, En.wikipedia.org, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Parsec_(parser).

[26] "Cray XT4," Wikipedia, 15-Sep-2020. [Online]. Available: https://en.wikipedia.org/wiki/Cray_XT4.

[27] "Memory management," NVIDIA Developer Documentation. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html.

[28] "CUDA C++ programming guide," NVIDIA Developer Documentation. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[29] "Host api," NVIDIA Developer Documentation. [Online]. Available: https://docs.nvidia.com/cuda/curand/group__HOST.html.

[30] "std::time," cppreference.com. [Online]. Available: https://en.cppreference.com/w/cpp/chrono/c/time.

[31] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," arXiv.org, 22-Feb-2017. [Online]. Available: https://arxiv.org/abs/1609.02907.

[32] Virya: CSSE GPU server user manual, edition 2.5, Concordia University, 2019, pp. 14

[33] T. Bakker, "Supercomputer," ECMWF, 07-Jan-2019. [Online]. Available: https://www.ecmwf.int/en/computing/our-facilities/supercomputer.

[34] A. Grama, "The Optimal Matrix-Parenthesization Problem," in Introduction to parallel computing, Harlow, England: Addison-Wesley, 2013.

[35] "Basic linear algebra subprograms," Wikipedia, 31-Jul-2021. [Online]. Available: https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms.

[36] "General-purpose computing on graphics processing units," Wikipedia, 15-Jun-2021. [Online]. Available: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units#Implementations.

[37] J. Fang, A. L. Varbanescu and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," 2011 International Conference on Parallel Processing, 2011, pp. 216-225, doi: 10.1109/ICPP.2011.45.

[38] "Thread block (CUDA programming)," Wikipedia, 25-Jun-2021. [Online]. Available: https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)#cite_note-6.

[39] "CUDA occupancy Calculator," NVIDIA Developer Documentation. [Online]. Available: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

[40] "Parallel thread Execution Isa Version 7.4," NVIDIA Developer Documentation. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[41] "Volta Architecture Whitepaper." NVIDIA. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[42] R. Smith and N. Oh, "The NVIDIA GeForce Gtx 1660 Ti Review, Feat. Evga XC GAMING: Turing Sheds RTX for the mainstream market," RSS, 22-Feb-2019. [Online]. Available: https://www.anandtech.com/show/13973/nvidia-gtx-1660-ti-review-feat-evga-xc-gaming/2.

[43] P. Lai, H. Arafat, V. Elango and P. Sadayappan, "Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs," 20th Annual International Conference on High Performance Computing, 2013, pp. 139-148, doi: 10.1109/HiPC.2013.6799109.

[44] "Unified memory for cuda beginners," NVIDIA Developer Blog, 05-Jan-2021. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[45] "cublas," NVIDIA Developer Documentation. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html.

[46] "Thread block (CUDA programming)," Wikipedia, 25-Jun-2021. [Online]. Available: https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming).

[47] "Achieved Occupancy," NVIDIA docs. [Online]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[48] "Profiler user's Guide," NVIDIA Developer Documentation. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[49] "Host api," NVIDIA Developer Documentation. [Online]. Available: https://docs.nvidia.com/cuda/curand/group__HOST.html.