Parallel Divide-and-Conquer Algorithm for Generalized Matrix Multiplication

Nate Bean, John Eagan, Marc Herdman,
Sarah Kern, Christian Vaughn, *Matin Pirouz*

Abstract—Generalized matrix multiplication (GEMM) involves computing large data sets. The matrices in these data sets may be sparse or dense. Many matrix multiplication approaches specifically target either sparse or dense matrices and not both.We implemented a parallel divide and conquer general matrix multiplication (PDCGMM) algorithm in Python that displays similar running time results for both sparse and dense matrices. We ran experiments utilizing the GPU memory via CUDA. PDCGMM demonstrated 5-6 times speedup over python's NumPy built in GEMM algorithm for larger matrices and 70-90 times speed up for smaller matrices that fit entirely on the GPU.

Index Terms—

1 Introduction

One of the most ubiquitously used operations in computer graphics, network theory, science, and engineering is matrix multiplication. Despite it's prolific use, it is a heavily complex and time-consuming operation as the sizes of matrices used can grow rapidly out of control. It is no wonder that matrix multiplication is the focus of so many research papers, each seeking to squeeze every bit of optimization out of it. A large portion of matrix multiplication research has concentrated on algorithms tailored to specific components of matrix multiplication such as size, sparsity, and density. While Tomikj Gusev [1], Arrigoni et al. [2] and Ishiguro et al. [13] influenced our decision to take a concurrent approach to our algorithm, a multitude of related work [3, 4, 8, 11, 12] found a general, scalable, approach lacking. In light of this, a generalized matrix multiplication algorithm that is simultaneously scalable and highly optimized is the approach we developed when designing our Parallel Divide and Conquer Algorithm for Generalized Matrix Multiplication (PDCGMM).

1.1 Contributions

Our solution to a generalized matrix multiplication algorithm is to utilize a mix of a divide and conquer approach and GPU computation to improve performance compared to existing approaches. By dividing the matrices small enough to fit into the available GPU memory and then leveraging the multiple pipelines available, PDCGMM is able to vastly increase the throughput of the operations needed to complete the multiplication.

2 RELATED WORK

2.1 Description & Baseline

The field of research surrounding matrix multiplication and its practical optimizations is vast and has produced many viable options for improving the algorithm's time complexity and runtime. This paper and its related work primarily deal with Parallel Matrix Multiplication (PMM) and its areas for improvement.

To establish a baseline for PMM strategies without the division of matrices into blocks, a recent paper [1] performed a series of experiments. These experiments compared Row First (RF), Column first (CF), Row by Row (RR), and Matrix Transposing first (MT) algorithms using metrics of speed, bandwidth, cache usage, and more. They found that hyperthreading produced no appreciable benefit but that the number of physical threads in a CPU greatly impacted the acceleration of matrix multiplication through parallelism. Of the four algorithm types explored, they concluded that MT had the most potential for gain [1].

Arrigoni et al. [2] explored MT algorithms in the context of multiplying a matrix by its transpose (A^TA or ATA). They created their algorithm with the ability to be parallelized, which has been shown to be essentially necessary for the use of these algorithms in practical settings [1]. The algorithms, which Arrigoni et al. called ATA, ATA-S, and ATA-D, were all based on Strassen's algorithm.

ATA-S is intended for multi-threaded machines and ATA-D is intended for multi-processor machines. Runtime and GFLOPs were calculated and compared against Intel MKL dysrk and Intel MKL dgemm respectively. In both the time and effective GFLOPs, the author's implementations showed improved performance. ATA-S and ATA-D decreased runtime over their MKL counterparts. Additionally, ATA-D had significant speed up over ATA-S for larger matrices.

Arrigoni et al. [2] successfully parallelized their ATA algorithm across different types of machines including distributed and shared memory machines. The primary lim-

- *Nate Bean, John Eagan, Marc Herdman, Sarah Kern, Christian Vaughn with the Department of Computer Science, California State University, Fresno .*
  E-mail: teknikly1@mail.fresnostate.edu johnjeagan@mail.fresnostate.edu normal42@mail.fresnostate.edu sarahekern1@mail.fresnostate.edu christian_vaughn@mail.fresnostate.edu
- *M. Pirouz is a faculty in the Department of Computer Science, California State University, Fresno*
  E-mail: mpirouz@mail.fresnostate.edu

Manuscript received (FILL IN)

itation of this approach was its specific application to the case of a matrix being multiplied by its transpose and didn't clearly generalize to other cases.

In contrast, Karstadt & Schwartz [3] sought to extend Bodrato's intermediate representation method for matrix squaring by supplying a basis transformation method to matrix multiplication, effectively modifying multiplication by transpose. The aim was to achieve a faster algorithm with the same base case and time complexity as Strassen's Algorithm by reducing the leading coefficient from 6 to 5. They succeeded in improving the communication and computation costs of a multitude of fast matrix multiplication algorithms by a notable constant factor. Their experiments showed that the sacrifice of slight asymptotic overhead, regarding communication cost, was well worth it to improve the arithmetic overhead via reduction of the leading coefficient. This resulted in improved variations of already existing fast matrix multiplication algorithms. The common recursive-bilinear algorithm approach can be applied up until a significantly small base case (typically 2x2), then classical MM methods should be utilized as they produce more computationally and temporally efficient solutions on smaller matrices. While this approach is not entirely novel, as linear transformations have been applied to matrix multiplication before, the alternative basis algorithm reduces both cache misses and communication overhead. This is the general use-case that was missing from the experiments above [2]. Despite this achievement the algorithm does not fare well on distributed systems. This leaves us with some commonly observed themes throughout this work.

The ideas behind how existing literature outperforms the baseline methods are twofold:

- Reducing communication cost with efficient element access and workload distribution.
- Optimizing for specialized matrix structures and/or degrees of sparseness.

2.2 Communication Cost

One of the challenges associated with matrix multiplication is the determination of how to access the elements of the matrix in a way that utilizes a CPU (or GPU [4]) cache architecture efficiently [1]. Another challenge inherent to parallel matrix multiplication is the complexity of assigning optimal workloads to multiple processing cores/threads [5]. Collectively, this can be thought of as the communication cost of matrix multiplication and its parallelization.

In 2021 Liao, Li, et al. evaluated standard methods of PMM as well as Strassen methods in their paper [6]. Their findings did not support any one general approach over another in all situations. Their findings did lead them to conclude that communication is the most important challenge in PMM today. They suggested that a bottom up approach to PMM may improve time-to-complete by minimizing communications.

Kwasniewski, Kabic, et al. did just that in their paper [5]. Using a computational directed acyclic graph they were able to establish a theoretical tight lower bound on communication. Then, by applying the Red-Blue pebble game they were able to divide the work to be done nearly I/O optimal before assigning it to individual processors. This is the bottom up approach Liao, Li, et al. mentioned in their paper [6].

Another improvement on the communication issue plaguing parallel matrix multiplication (PMM) is through the use of loop unrolling and Compressed Sparse Row (CSR) optimization. To improve the performance of the Sparse Dense Matrix Multiplications (SpDM), Soliman et al., looked at loop unrolling optimization and parallel processor application [7]. The authors achieved parallel processing with a shared memory model while employing the following optimization techniques: Compressed sparse row optimization and Multi-threaded Intel math kernel Library (MKL). The CSR algorithm was utilized to enable increased memory efficiency and improved rate of cache hits while MKL was utilized for it's enhancement of memory allocation as well as the added benefit of its memory alignment routines. The novelty of this approach is that Peano Curves allows for a cache oblivious method to tackling SpDM by reducing cache miss rates.

Comparisons were made utilizing both sparse and dense matrices sans data compression and code optimizations; essentially naïve versions of each. The results were measured via a three-stage method. Firstly, the CSR method was applied across a sparse matrix. Secondly, the authors applied the loop unrolling technique for optimization (LUTO). Lastly, they applied MKL method for increased cache hit rate. Execution times were measured after each successive method application. Blending CSR and loop unrolling improved application parallelism and led to an 86% decrease in execution time over standard naïve methods. The CSR is a 50% improvement over Peano Curve patterning due the fact that the Peano Curves are data reliant while CSR strictly focuses on cache miss reduction. Unfortunately, finding the sweet spot for the unrolling factor is problematic as a high unrolling factor can lead to an even higher incidence of cache misses, even more than not utilizing the unrolling algorithm at all. In fact, code unrolling and code optimizations, alone, did not lead to large improvements in execution time. These were largely a factor of CSR via reduction of cache misses. In the end, loop unrolling through Peano curves has some limited benefits in very specific use-cases but does not generalize well.

To improve utilization of CPU and GPU cache, Hong, Sukumaran-Rajam, et al. made the Adaptive Sparse-matrix Tiling (ASpT) algorithm to break matrices into rows, then make the rows into 2D tiles [8]. The algorithms worked best on dense matrix by sparse matrix multiplication, and dense times dense scaled by a sparse. The results showed the ASpT method was better than other existing ones. If there were significant amounts non-zero values, data could be reused through the cache. The downsides to this approach are that there is an overhead for tiling the matrices and the lack of reusing cache for sparse matrices [8].

2.3 Matrix Structures & Sparsity

Some of the most promising improvements in the time complexity and runtime of matrix multiplication have been produced by forgoing the concept of generalization and instead focusing on optimizing for specific matrix structures or degrees of sparseness. One common example is Strassen's

Algorithm, which achieves a time complexity of $\mathcal{O}(n^{log_2 7})$, but only for multiplying dense matrices [9].

Hossain and Mahmud [10] utilized diagonally structured matrices to perform matrix multiplication and matrix vector multiplication, which can be thought of as matrix multiplication where the second matrix has only one column. Their paper showed how specialized structures of dense matrices, such as banded and triangular matrices, could be represented using their diagonals to provide efficient cache usage and reduced storage requirements. They devised a novel scheme for storing matrices using their diagonals inside a one-dimensional array when the nonzero elements are densely packed.

The novelty of this approach is that previous storage schemes utilized a two-dimensional array to store the diagonals with padded, non-referenced elements filling in the missing space for shorter diagonals, whereas this paper presented a one-dimensional alternative that doesn't require padding. They tested their matrix multiplication algorithm on multiple bandwidths with matrix dimension $n = 100000$ and 2 to 16 threads. The results indicate that as the bandwidth of the matrices increase, the speedup and efficiency of the algorithm increases with peak values around a bandwidth of 1600. Also, as the number of threads increases, the speedup increases drastically while the efficiency drops relatively slowly. This shows that their algorithm using diagonal matrix structuring performs well when parallelizing the independent diagonal calculations and scales efficiently as the bandwidth increases.

The primary advantage of this approach was the ability to store these specific types of matrices in a scheme that is both space-efficient and provides stride-1 access to the matrix elements without any form of indirect referencing [10]. The limitation of this storage scheme is that it doesn't apply to matrices that do not have well defined nonzero diagonals. In these cases where the algorithm cannot take advantage of only storing the nonzero elements, the storage scheme must store all of the diagonals of the matrices. It cannot miss any nonzero elements that may be scattered throughout the matrices. This also presents an issue during parallelization of the algorithm since dividing a matrix into smaller matrices or blocks may not provide the same structure.

Another paper [4] re-examines the belief that naive implementations of sparse matrix vector multiplication (SpMV) cannot achieve the same performance as more complex, state of the art implementations when parallelized. The authors implemented naive algorithms for both SpMV and the transpose multiplication case (SpMVT). They utilized the compressed sparse rows (CSR) format for a matrix to force elements in the same row of a matrix to be stored consecutively in memory for lower cache communication costs. The authors also developed the parallel versions of these two algorithms by parallelizing the loop that is operating over all of the matrix rows.

The novelty of this paper is not the parallel implementations of matrix vector multiplication themselves, but rather the evaluation of these naive algorithms compared to more state of the art implementations on modern GPU hardware, which could provide counterintuitive results due to atomic operations being supported directly in hardware and superior cache performance. This paper is novel in that it suggests elements of naive matrix multiplication could provide better runtime performance for GPU-based SpMV.

The authors compared their naive algorithms with state of the art algorithms (cuSparse, CUSP, and bhSparse) on a variety of matrix sizes and densities. The naive approach and CUSP achieved the best performance in five tests, cuSparse in three tests, and bhSparse in one. The naive implementations showed their best performance with large matrices that have very few nonzero elements that are distributed uniformly. The authors found that the naive approach can outperform more complicated algorithms when under these specific matrix conditions, even when not accounting for the conversion costs. When the matrices are less regular, the state of the art algorithms performed far better.

The advantages of this naive approach are the elimination of format conversion overhead required for state of the art implementations and better performance with large, regular matrices. In terms of time complexity, the conversion overhead presents itself as an increase in the constant factor. Overhead reduction is especially important when a matrix is only used for one matrix vector multiplication operation because the costs cannot be amortized. The limitation of this approach is that it doesn't generate as many uniform workloads as other state of the art algorithms when working with smaller, less regular matrices due to load imbalances and divergence.

2.4 Generalized Matrix Multiplication

A problem found in many matrix multiplication implementations is the inability to be efficient for sparse and dense matrices alike. Rasouli et al. [11] applied parallel D&C techniques to the general matrix to matrix multiplication (GEMM) algorithm. Additionally, they introduced new methods for data compression of matrices and overlapping communication. The overlapping communication allows for parallelization by computing matrix multiplication as the divide and conquer recursion takes place. Through experiments, they concluded their recursive GEMM algorithm has approximately 2.2 times speed up over the Portable Extensible Toolkit for Scientific Computation's (PETSc) implementation [11]. Extensive experiments over varying sized and types of matrices proved that their GEMM algorithm effectively computes matrix multiplication despite the level of density. In addition to efficiency over sparse and dense matrices, a novelty of the authors' work was the compression method they designed, based on Golomb-Rice encoding, to reduce communication cost. The greatest limitation of their work is that the recursive GEMM algorithm fails to perform well for diagonal matrices due to the recursive nature of their GEMM implementation. PETSc accounts for this specific case, therefore, it performs better compared to the authors' algorithm. [11]

3 OUR APPROACH

This section details our approach to creating PDCGMM, a parallel D&C implementation of matrix multiplication.

TABLE 1: Notations

Symbol	Meaning
A	First Matrix
B	Second Matrix
C	Resultant Matrix from AB
m	Number of rows in A
n	Number of columns in A, Number of rows in B
p	Number of columns in B
c	Number of cores in a machine
pt	Partitioning threshold
0	Matrix filled with all zeroes (Zero matrix)
b	Number of total available bytes of memory
parfor	For-loop executed in parallel

3.1 Definitions

3.2 Partitioning Threshold

PDCGMM divides A and B into matrices with smaller dimensions using block partitioning. This reduces cache storage requirements because there are fewer elements to store and access in smaller matrices.

By reducing matrix dimensions, communication costs incurred by cache misses and slow memory accesses will be reduced.

A and B will be considered sufficiently partitioned when $m \leq c$. The intention of this partitioning threshold is to divide A until the CPU can run all m row calculations in parallel. In other words, the CPU will assign one core to each row of A and perform a dot product of that row and each column of B.

Given $A_{m\times n}$ and c, the following values can be calculated:

The total number of partitions performed, P:

$$P(m,n,c) = \begin{cases} 0, & \text{if } m \leq c \\ 1 + P(m, \lfloor \frac{n}{2} \rfloor, c), & \text{if } m > c \text{ and } m \leq n \\ 1 + P(\lfloor \frac{m}{2} \rfloor, n, c), & \text{if } m > c \text{ and } m > n \end{cases}$$

The number of times m is partitioned, P_m:

$$P_m(m,c) = \begin{cases} 0, & \text{if } m \leq c \\ 1 + P_m(\lfloor \frac{m}{2} \rfloor, c), & \text{if } m > c \end{cases}$$

The number of times n is partitioned, P_n:

$$P_n(m,n,c) = \begin{cases} 0, & \text{if } m \leq c \\ 1 + P_n(m, \lfloor \frac{n}{2} \rfloor, c), & \text{if } m > c \text{ and } m \leq n \\ P_n(\lfloor \frac{m}{2} \rfloor, n, c), & \text{if } m > c \text{ and } m > n \end{cases}$$

The number of rows after partitioning, m_P:

$$m_P = \left\lfloor \frac{m}{2^{P_m(m,c)}} \right\rfloor$$

The number of columns after partitioning, n_P:

$$n_P = \left\lfloor \frac{n}{2^{P_n(m,n,c)}} \right\rfloor$$

3.2.1 GPU-based Partitioning

The partitioning described above focuses on optimizing the partitioning threshold for matrix multiplication on a CPU. PDCGMM also considers a partitioning threshold for GPU-based matrix multiplication. Modern GPUs can perform parallel computing of simple operations far more quickly than the CPU because of their hundreds or even thousands of processing cores.

The primary concern for performing matrix multiplication on a GPU is the storage of the matrices in the GPU's relatively limited memory. This memory limitation is handled by PDCGMM through the use of block partitioning to perform the matrix multiplication on smaller submatrices. Our partitioning threshold takes into account the amount of GPU memory available and partitions A and B until the submatrices are small enough to fit entirely on the GPU.

Given a GPU that can store b bytes and matrix elements of type double-precision floating-point (float64), the maximum number of rows in A and B that can be stored is:

$$\sqrt{b/8}$$

In terms of our algorithm's base case and partitioning threshold parameterized with m:

$$m \leq \frac{\sqrt{b/8}}{2}$$

3.3 Parallel D&C Matrix Multiplication

In PDCGMM, $A_{m\times n}$ will be multiplied with $B_{n\times p}$ producing $C_{m\times p}$.

PDCGMM recursively divides A and B into submatrices until $m \leq \frac{\sqrt{b/8}}{2}$. Partitioning is halted and matrix multiplication is performed on the subproblem. The submatrices will be small enough for the GPU to process each row in parallel.

Algorithm 1: MatrixMultiply(A, B, m, n, p, pt)

```
Input: Matrices $A_{m\times n}$, $B_{n\times p}$. Partitioning threshold, $pt$.
Output: Resultant Matrix $C_{m\times p}$
if $m \leq pt$ then // Base Case
    $C \leftarrow 0_{m\times p}$
    parfor $i = 0$ to $m$ do
        for $j = 0$ to $p$ do
            for $k = 0$ to $n$ do
                $C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj}$
    return $C$
return Partition$(A, B, m, n, p, pt)$
```

Our partitioning algorithm has two cases:

Case 1: $m \leq n$

A will be split in half by its columns into (A_1, A_2).

B will be split in half by its rows into (B_1, B_2).

$$A = \begin{bmatrix} A_1 & \mid & A_2 \end{bmatrix} B = \begin{bmatrix} B_1 \\ \hline B_2 \end{bmatrix}$$

C will be calculated as $A_1B_1 + A_2B_2$.

$$C = \begin{bmatrix} A_1B_1 + A_2B_2 \end{bmatrix}$$

Case 2: $m > n$

A will be split in half by its rows into (A_1, A_2).

B will be split in half by its columns into (B_1, B_2).

$$A = \left[\frac{A_1}{A_2}\right] B = \left[B_1 \;\middle|\; B_2\right]$$

C will be calculated as the appending of four matrices:

$$C_1 = A_1B_1, C_2 = A_1B_2,$$

$$C_3 = A_2B_1, C_4 = A_2B_2.$$

$$C = \left[\begin{array}{c|c} A_1B_1 & A_1B_2 \\ \hline A_2B_1 & A_2B_2 \end{array}\right] = \left[\begin{array}{c|c} C_1 & C_2 \\ \hline C_3 & C_4 \end{array}\right] = \left[\frac{C_{12}}{C_{34}}\right] = \left[C_{\substack{12\\34}}\right]$$

```
Algorithm 2: Partition($A$, $B$, $m$, $n$, $p$, $pt$)
  Input: Matrices $A_{m\times n}$, $B_{n\times p}$. Partitioning threshold,
         $pt$.
  Output: Resultant Matrix $C_{m\times p}$
  if $m \le n$ then // Case 1
      $(A_1, A_2)$ ←SplitColumns($A$, $m$, $n$)
      $(B_1, B_2)$ ←SplitRows($B$, $n$, $p$)
      $C$ ←MatrixMultiply($A_1, B_1, m, \lfloor\frac{n}{2}\rfloor, p, pt$)+
            MatrixMultiply($A_2, B_2, m, \lceil\frac{n}{2}\rceil, p, pt$)
      return $C$
  else // $m > n$ - Case 2
      $(A_1, A_2)$ ←SplitRows($A$, $m$, $n$)
      $(B_1, B_2)$ ←SplitColumns($B$, $n$, $p$)
      $C_1$ ←MatrixMultiply($A_1, B_1, \lfloor\frac{m}{2}\rfloor, n, \lfloor\frac{p}{2}\rfloor, pt$)
      $C_2$ ←MatrixMultiply($A_1, B_2, \lfloor\frac{m}{2}\rfloor, n, \lceil\frac{p}{2}\rceil, pt$)
      $C_3$ ←MatrixMultiply($A_2, B_1, \lceil\frac{m}{2}\rceil, n, \lfloor\frac{p}{2}\rfloor, pt$)
      $C_4$ ←MatrixMultiply($A_2, B_2, \lceil\frac{m}{2}\rceil, n, \lceil\frac{p}{2}\rceil, pt$)
      $C_{12}$ ←AppendHorizontal($C_1, C_2$)
      $C_{34}$ ←AppendHorizontal($C_3, C_4$)
      $C$ ←AppendVertical($C_{12}, C_{34}$)
      return $C$
```

3.4 Splitting a Matrix

```
Algorithm 3: SplitColumns($M$, $row$, $col$)
  Input: Matrix $M_{row\times col}$
  Output: Matrices $M1_{row\times\lfloor col/2\rfloor}$ and $M2_{row\times\lceil col/2\rceil}$
  $M_1 \leftarrow 0_{row\times\lfloor col/2\rfloor}$
  $M_2 \leftarrow 0_{row\times\lceil col/2\rceil}$
  $M_1Columns$ = $\lfloor col/2\rfloor$
  $M_2Columns$ = $\lceil col/2\rceil$
  for $i = 0$ to $col$ do
      if $i < M_1Columns$ then
          $M_{1_i} \leftarrow M_i$
      else
          $M_{2_i} \leftarrow M_i$
  return $(M_1, M_2)$
```

```
Algorithm 4: SplitRows($M$, $row$, $col$)
  Input: Matrix $M_{row\times col}$
  Output: Matrices $M1_{\lfloor row/2\rfloor\times col}$ and $M2_{\lfloor row/2\rfloor\times col}$
  $M_1 \leftarrow 0_{\lfloor row/2\rfloor\times col}$
  $M_2 \leftarrow 0_{\lfloor row/2\rfloor\times col}$
  $M_1Rows$ = $\lfloor row/2\rfloor$
  $M_2Rows$ = $\lceil row/2\rceil$
  for $i = 0$ to $col$ do
      for $j = 0$ to $row$ do
          if $j < M_1Rows$ then
              $M_{1_{ji}} \leftarrow M_{ji}$
          else
              $M_{2_{ji}} \leftarrow M_{ji}$
  return $(M_1, M_2)$
```

3.5 Lemmas

Lemma 3.1. *For any two matrices A and B that are known to have valid dimensions for multiplication, the partitioned matrices A_1, A_2 will also have valid dimensions for multiplication with partitioned matrices B_1, B_2.*

Proof. Given A and B, the matrix multiplication operation AB can only be performed if the number of columns in A and number of rows in B are equal.

Assume that A and B have valid dimensions for matrix multiplication. Let $m \times n$ represent the dimensions of A, and $n \times p$ represent the dimensions of B.
There are two cases for partitioning.

- Case 1: $m \le n$
  A will be split in half by column producing A_1 and A_2 with dimensions $m \times \lfloor n/2\rfloor$ and $m \times \lceil n/2\rceil$ respectively.
  B will be split in half by row producing B_1 and B_2 with dimensions $\lfloor n/2\rfloor \times p$ and $\lceil n/2\rceil \times p$ respectively.

  In this case of partitioning, A_1B_1 and A_2B_2 are required matrix multiplications.
  Therefore, A_1 must have valid dimensions with B_1 and A_2 must have valid dimensions with B_2.

  A_1 and B_1 have the same inner dimensions $\lfloor n/2\rfloor$ and A_2 and B_2 have the same inner dimensions $\lceil n/2\rceil$.

  Therefore, the matrix multiplications A_1B_1 and A_2B_2 can be performed and A_1, A_2, B_1, and B_2 have valid dimensions for Case 1.
- Case 2: $m > n$
  A is split by its rows and B is split by its columns. This leaves the "inner" dimension n for both matrices unaffected.

  Therefore the validity of the submatrices' dimensions with respect to matrix multiplication is maintained.

  A_1 and A_2 will both have n columns and B_1 and B_2 will both have n rows so matrix multiplication between any A submatrix and B submatrix will be valid.

All required matrix multiplications can be performed with the partitioned matrices A_1, A_2, B_1, and B_2 for Case 2.

For both cases of partitioning, A_1, A_2, B_1, and B_2 have valid dimensions for the required matrix multiplications. Therefore, PDCGMM provides valid submatrices for matrix multiplication of A and B. ∎

Lemma 3.2. *For any two matrices A and B, that are known to be valid for multiplication, the partitioned matrices D, E, F, & G will also be qualitatively valid while maintaining dimensional rules for matrix multiplication.*

Proof. Suppose we have matrices A and B such that:

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \\ A_5 & A_6 \end{bmatrix} B = \begin{bmatrix} B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 \end{bmatrix}$$

Further suppose that A and B have been partitioned into submatrices D, E, F, and G.

$$A = \begin{bmatrix} D_1 & D_2 \\ D_3 & D_4 \\ E_1 & E_2 \end{bmatrix} B = \begin{bmatrix} F_1 & F_3 & G_1 \\ F_2 & F_4 & G_2 \end{bmatrix}$$

This yields:

$$\{D_1, D_2, D_3, D_4\} \;\; \& \;\; \{E_1, E_2\} \in A$$
$$\{F_1, F_2, F_3, F_4\} \;\; \& \;\; \{G_1, G_2\} \in B$$

where,

$$D^{2\times 2} F^{2\times 2} = (DF)^{2\times 2}$$
$$E^{1\times 2} G^{2\times 1} = (EG)^{1\times 1}$$

Therefore:

$$D \cdot F^T = \{D_1F_1 + D_2F_2 + D_3F_3 + D_4F_4\}$$
$$E \cdot G^T = \{E_1G_1 + E_2G_2\}$$
$$A \cdot B^T = \{A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 + A_5B_5 + A_6B_6\}$$

$$D_1F_1 + D_2F_2 = A_1B_1 + A_2B_2$$
$$D_3F_3 + D_4F_4 = A_3B_3 + A_4B_4$$
$$E_1G_1 + E_2G_2 = A_5B_5 + A_6B_6$$

By the law of associative addition:

$$DF^T + EG^T = AB^T$$
$$DFEG = AB$$

Matrix Dimensions:

$$DFEG = 3 \times 3$$
$$AB = 3 \times 3 \quad \checkmark$$

∎

4 EXPERIMENT

4.1 Setup

We tested PDCGMM using a single workstation with specifications detailed in Table 2. PDCGMM was implemented in Python using the CuPy library to facilitate processing on the GPU through the Compute Unified Device Architecture (CUDA). Specifically, we implemented Algorithm 1 with c now representing a given partitioning threshold for the GPU (described in 3.2.1) and Algorithm 2 as two different partitioning functions depending on where the matrices are currently being stored and manipulated, CPU or GPU.

Data was gathered by running PDCGMM and the NumPy implementation of matrix multiplication using Basic Linear Algebra Subprograms (BLAS) across a number of pre-generated matrices. For each test, the execution time of both algorithms were measured 10 times.

TABLE 2: Machine Specifications

CPU	Intel i5-9600k @ 4.8GHz
GPU	Nvidia GPU GTX 1660 Super
GPU Memory	6GB
Sys. Memory	16GB
OS	Windows 10

4.2 Data-Sets

Sparse matrices were collected from the SuiteSparse Matrix Collection (SSMC) and are detailed in Table 3 [12]. We chose these specific sparse matrices because they are typically used in other literature. They range from 10,000 to 20,000 rows and columns. A 10,000 squared matrix is used for the lower bound of data-sets because it is the largest that can fully fit on our GPU without partitioning. Any smaller and the results would be too small for comparison. A 20,000 squared matrix is the upper bound because larger matrices took significantly longer with the NumPy approach and made our results difficult to observe. Dense matrices were randomly generated prior to testing and are detailed in Table 4. The dense matrices were generated to be the same size as the SSMC data-sets for proper comparison. For our purposes, we define density as:

$$\frac{\text{\# of non-zeros}}{\text{Rows} * \text{Columns}}$$

A matrix with a greater density has more non-zero elements and sparse matrix formats, such as CSR, become less useful. Table 4 details the dimensions of the generated dense matrices. As stated before, these matrices are 100% dense. Therefore they only contain non-zero elements.

TABLE 3: Test matrix info from SSMC.

ID	Name	Rows	Columns	Density
S1	bcsstk17	10,974	10,974	0.36%
S2	ex11	16,614	16,614	0.40%
S3	gupta3	16,783	16,783	3.31%
S4	human_gene1	22,283	22,283	4.97%
S5	human_gene2	14,340	14,340	8.79%

TABLE 4: Synthesized dense matrices.

ID	Rows	Columns
D1	10,974	10,974
D2	16,614	16,614
D3	16,783	16,783
D4	22,283	22,283
D5	14,340	14,340

4.3 Results and Discussion

Tables 5 and 6 detail the results from our experiments. After running PDCGMM and NumPy matrix multiplication over sparse and dense matrices 10 times each, we calculated the average run time of each algorithm and annotated this time (in seconds) in Tables 5 (sparse matrices) and 6 (dense matrices). Based on these run times, we calculated and annotated the speed up of PDCGMM over NumPy.

PDCGMM performed significantly faster than NumPy for matrices S1 and D1 in particular. This performance increase can be attributed to the fact that the matrices were small enough to be stored entirely on the GPU and matrix multiplication was performed without any partitioning.

$$\text{GPU Memory: 6GB} = 6,000,000,000 \text{ bytes}$$

$$\text{Partitioning Threshold: } \frac{\sqrt{6,000,000,000/8}}{2} \approx 13,693 \text{ rows}$$

This partitioning threshold represents how many rows two square matrices can have and still be stored on 6GB of memory. S1 and D1 were the only matrices tested that had less than 13,693 rows so no communication costs were incurred by moving the matrices to and from the CPU during partitioning. When the matrix fit entirely with the GPU's memory, PDCGMM performed 69 times faster than NumPy's built-in matrix multiplier for a sparse matrix and 92 times faster for a dense matrix.

This is not a surprising result as much of the overhead of PDCGMM comes from sending data from the CPU to the GPU. Larger matrices provide more realistic comparative results where matrices are too large to be stored and computed entirely on the GPU.

On sparse matrices, PDCGMM achieved results between 4.833 and 5.907 times faster than NumPy. Results were similar for dense matrices with speed-ups between 4.985 and 6.064 times. Figure 1 visually organizes our results and performance improvement over NumPy. PDCGMM outperformed the well established algorithms used by numPy in every test by taking advantage of the numerous CUDA cores and threads present in most computers with modern GPUs.

TABLE 5: Execution Time on Sparse Matrices

ID	PDCGMM	NumPy	Speedup
S1	0.166s	11.477s	69.139
S2	6.649s	36.533s	5.495
S3	6.952s	39.395s	5.667
S4	21.433s	103.592s	4.833
S5	4.466s	26.380s	5.907

TABLE 6: Execution Time on Dense Matrices

ID	PDCGMM	NumPy	Speedup
D1	0.142s	12.999s	91.542
D2	6.901s	39.349s	5.702
D3	7.057s	39.943s	5.660
D4	20.928s	104.327s	4.985
D5	4.368s	26.489s	6.064

The performance of PDCGMM was almost indistinguishable across sparse and dense matrices, demonstrating potential as a generalized matrix multiplication algorithm. This density-oblivious property is important for matrix multiplication algorithms that utilize block partitioning since blocks may have widely different densities.

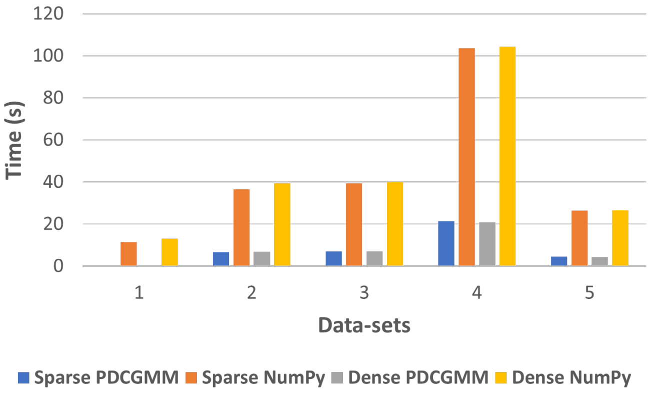

Fig. 1: Graph comparing running time results of dense/sparse matrices of PDCGMM vs NumPy.

4.4 Comparative Results

Ishiguro et al. evaluated the performance increase from utilizing the GPU compared to using the CPU in sparse matrix - sparse matrix multiplication (SpMxSpM) and sparse matrix - dense matrix multiplication (SpMM) [13]. In particular, their GPU-based matrix multiplication algorithm achieved a 3.24x speedup for SpMxSpM and a maximum speedup of 8.44x for SpMM compared a CPU-based implementation. However, their experiments had a maximum matrix size of 10,000 × 10,000 while our experiments tested matrices with double that size.

As stated previously, PDCGMM performed about 5 to 6 times faster than the CPU-based NumPy implementation when the matrices were too large to fit on the GPU. When considering matrices that could fit entirely on GPU memory, matrix S1 (Table 3) had dimensions around 11,000 × 11,000 and had a speedup of 70 to 90 times over NumPy, which far surpasses the maximum speedup Ishiguro et al. achieved. In terms of execution time, PDCGMM also performed much faster with an execution time of 0.142 to 0.166 seconds (Tables 5 & 6) on matrix S1, whereas their algorithm took around 50 seconds for a 10,000 × 10,000 matrix [13]. Even for matrix S4 (22,283 × 22,283), PDCGMM only took 20.928 seconds.

This execution time difference cannot be explained as the result of hardware limitations since Ishiguro et al. utilized two NVIDIA Tesla P100 GPUs, which operate at 4.7 teraflops (TFLOPS) when operating on double-precision floating-point numbers. This exceeds our machine's GPU (Table 2) performance of 157 gigaflops (GFLOPS) for double-precision floats by about 30 times.

5 CONCLUSION

We developed PDCGMM to be a D&C approach and CUDA-based parallel implementation for generalized matrix multiplication to improve performance and scalability for both sparse and dense matrix multiplication. PDCGMM scales effectively due to our partitioning scheme, which breaks up the matrices to fit on the GPU without running out of memory which allows PDCGMM to take advantage of the parallel performance of the GPU. PDCGMM performed better than existing work but is not fully optimized for sparse matrices which could be compressed for better performance and scalability.

For future work, the partitioning algorithm of PDCGMM could be improved. For smaller matrix sizes that are below the partitioning threshold, they can be directly sent to the GPU for matrix multiplication. Larger matrices above this threshold must be broken down by the CPU and sent to the GPU in blocks. Further improvements to the partition algorithm could limit the overhead from communication between the CPU and GPU by taking advantage of shared memory, reducing runtimes significantly.

REFERENCES

[1] Tomikj, N. & Gusev, M. Parallel matrix multiplication. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 0204–0209 (IEEE, 2018). URL https://doi.org/10.23919/MIPRO.2018.8400039.

[2] Arrigoni, V., Maggioli, F., Massini, A. & Rodolà, E. Efficiently parallelizable strassen-based multiplication of a matrix by its transpose. In *50th International Conference on Parallel Processing* (ACM, 2021). URL https://doi.org/10.1145/3472456.3473519.

[3] Karstadt, E. & Schwartz, O. Matrix multiplication, a little faster. *Journal of the ACM* **67** (2020). URL https://doi-org.hmlproxy.lib.csufresno.edu/10.1145/3364504.

[4] Steinberger, M., Derlery, A., Zayer, R. & Seidel, H.-P. How naive is naive spmv on the gpu? In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–8 (IEEE, 2016).

[5] Kwasniewski, G. *et al.* Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, 2019). URL https://doi-org/10.1145/3295500.3356181.

[6] Liao, X., Li, S., Yu, W. & Lu, Y. Parallel matrix multiplication algorithms in supercomputing. In *2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP)*, 1–4 (IEEE, 2021).

[7] Soliman, K., Shenawy, M. E. & Farag, A. A. E. Loop unrolling effect on parallel code optimization. In *Proceedings of the 2nd International Conference on Future Networks and Distributed Systems* (ACM, 2018). URL https://doi.org/10.1145/3231053.3231060.

[8] Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K. & Sadayappan, P. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 300–314 (ACM, 2019). URL https://doi.org/10.1145/3293883.3295712.

[9] Stein, C., Leiserson, C. E., Cormen, T. H. & Rivest, R. L. *Introduction to algorithms* (The MIT Press, 2009).

[10] Hossain, S. & Mahmud, M. S. On computing with diagonally structured matrices. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–6 (IEEE, 2019).

[11] Rasouli, M., Kirby, R. M. & Sundar, H. A compressed, divide and conquer algorithm for scalable distributed matrix-matrix multiplication. In *The International Conference on High Performance Computing in Asia-Pacific Region*, 110–119 (ACM, 2021). URL https://doi.org/10.1145/3432261.3432271.

[12] Davis, T. A. & Hu, Y. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.* **38** (2011). URL https://doi.org/10.1145/2049662.2049663.

[13] Ishiguro, F., Katagiri, T., Ohshima, S. & Nagai, T. Performance evaluation of accurate matrix-matrix multiplication on gpu using sparse matrix multiplications. In *2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)*, 178–184 (2020).