SECOND-GENERATION POLYALGORITHMS FOR PARALLEL DENSE-MATRIX MULTIPLICATION

by

Grace Nansamba

Anthony Skjellum
Professor of Computer Science
(Chair)

Farah Kandah
Professor of Computer Science
(Committee Member)

Michael Ward
Professor of Computer Science
(Committee Member)

SECOND-GENERATION POLYALGORITHMS

FOR PARALLEL DENSE-MATRIX

MULTIPLICATION

by

Grace Nansamba

A Thesis Submitted to the Faculty of the
University of Tennessee at Chattanooga
in Partial Fulfillment of the Requirements of the
Master of Science in Computer Science

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

December 2020

ABSTRACT

The polyalgorithm library, originally designed in 1991-1993 by Robert Falgout, Jin Li, and Anthony Skjellum, includes fourteen dense matrix multiplication algorithms mapped onto two-dimensional process grids using the Message Passing Interface (MPI). This thesis' goal is to achieve optimized performance of parallel, dense linear algebra algorithms by varying the algorithm as a function of problem size, shape, data layout, concurrency, and architecture. We integrate these algorithms with an intra-node BLAS DGEMM kernel designed by Thomas Hines (Tennessee Tech), which improves the BLAS DGEMM performance in fat-by-thin dense matrix multiplication region. We add a rank-k-based SUMMA algorithm, which performs better than rank-1-based SUMMA. We studied performance on two cluster systems and results show the performance and improvements achieved. We compare and contrast our results with COSMA, a recent, highly optimized approach and verify that COSMA, using optimal 3D grid decompositions, has significant advantages provided its preferred data layouts can be used.

DEDICATION

To my father, Mr. Eddie-Bosco Senoga, the priest of my life, God’s perfect image to me! For maama Nassimbwa, my best friend, the Proverbs 31 wife of Mr. Senoga, but above all my mother.

ACKNOWLEDGMENTS

I thank God, for watching over me and guiding me with His loving eye, Hallelujah!

Dr. Anthony Skjellum, thank you for all the great things you have done for me. For all the questions you have answered with patience and kindness, giving me a deeper understanding and the big picture of what high-performance computing should be. I am delighted that I have had the opportunity to continue learning more from you. You are the best advisor and professor. Thank you for showing me that good things take time and that the more hours spent on a piece of code or a paragraph, the more meaningful the results will be and the smarter I will become. Thank you for teaching me to look for the most interesting part in anything I do and focus on that to make it even better. Thank you for the financial support and for always asking whether everything else is okay. Thank you for free pizza and for coming to game night. You are awesome!

My A-team: Thomas Gorham for listening to all my ideas and for sharing with me whenever you discovered cool stuff and for proofreading and editing; Thomas Hines for teaching me how to think smart and how to program and to stop and ask myself if I am happy with that line of code; Justin Broaddus for always accepting to zoom meet and talk about the results and for running my simulations at any hour possible; and, Derek Schafer for teaching me to organise my tasks and have neat work and for fixing my bugs. Thank you all for always turning up for the zoom meetings; I am grateful.

I would like to extend my appreciation to my committee members: Dr. Farah Kandah and Dr. Michael Ward, you have encouraged me to keep going and that I will make it at every step of the way. Thank you for your time to read my thesis and for the useful feedback. All my professors throughout my course, thank you for the knowledge you have imparted while at UTC.

Maxwell Omwenga and Evelyn Namugwanya Vaaye, you are my best friends. I would not have made it without you. Thank you for always being here, in the highs and lows you carried me and I was always happy just to know you are with me cheering me on and believing in me because you are enough. To all Simcenter friends and staff, especially Amani Altarawneh for inspiring me and Kim Sapp for the care and love, thank you for holding my hand and for encouraging me always. Dr. Joseph Kizza and Dr. Immaculate Kizza, thank you for being my home away from home. Finally, to my family and friends, thank you for the prayers, the earnest presence and for loving me through it all. I love you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BLAS — Basic Linear Algebra Subprograms

BLIS — BLAS-like Library Instantiation Software

CARMA — Communication-Avoiding Recursive Matrix-multiplication Algorithm

COSMA — Communication Optimal S-partition-based Matrix multiplication Algorithm)

DDI — Data Distribution Independence

DGEMM — Double-precision Matrix-matrix Multiplication

KNL — Knight Landing

LAPACK —Linear Algebra Package

MKL — Math Kernel Library

MPI — Message Passing Interface

PUMMA — Parallel Universal Matrix Multiplication Algorithms

ScaLAPACK — Scalable LAPACK

SUMMA — Scalable Universal Matrix Multiplication Algorithm

CHAPTER 1

INTRODUCTION

Parallel dense matrix multiplication has been a popular research topic over the years. Several algorithms have been designed to solve this common linear algebra operation. The aim of polyalgorithms is to maximize performance by choosing which algorithm among many is most efficient for a particular situation (problem size, data distribution onto logical process topologies or grids, and specific layouts of the A, B, and C matrices). For example, COSMA [25] is nearly communication-optimal for all combinations of matrix dimensions, and CARMA [7], which is best for rectangular grids. The 2.5D algorithm [31] uses extra memory but has issues with non-square matrix shapes. Cannon's algorithm [4] was designed for square grid shapes and was extended by Mathur-Johnsson's algorithm [28] to deal with arbitrary process grid shapes. The PUMMA algorithm [6] was designed for non-square grids and is an extension of Fox's algorithm [10]. The SUMMA algorithm [36] enhances communication patterns and requires less work space.

The authors of these diverse algorithms suggest that their algorithms have the best performance for a particular situation or a variety of problems, sizes, shapes, and concurrencies. We present results that the general claim of "best performance" is not true for all cases where different process grid, matrix shapes, and problem sizes are considered. We restate the earlier premise that no single algorithm has the best performance at all times, and polyalgorithms are therefore inevitable in the design of high-performance linear algebra libraries. We also present an integration of the new version of the GEMM (General matrix multiplication), which is useful for fat-by-thin cases as compared to other BLAS libraries. A new version of the SUMMA algorithm that is rank-k-based (and so is much faster than the original SUMMA rank-1 algorithm in the original polyalgorithms library) is also included; this does away with a legacy limitation in the prior work upon which this work is based. A scientific re-validation of a new and highly optimized algorithm, COSMA [25], is also presented.

1.1 Background

Many powerful linear algebra libraries have been developed throughout the past four decades focused on efficient and scalable implementation of dense matrix operations. The BLAS API [8] is the origin of such libraries and software that are available for a variety of computer architectures. Current libraries include OpenBLAS [39, 40], BLIS [37], MKL [20], and LAPACK [2], all of which have implemented the General Matrix-Matrix multiplication (GEMM) efficiently for various machines and architectures.

Parallel matrix multiplication algorithms have been a major topic in the high performance computing field for many years. This is so because, behind significant computer science applications including machine learning, data visualization, data mining, Computational Fluid Dynamics (CFD) and the Finite Element Method (FEM), there is often a matrix multiplication taking place. Parallel algorithms speed up a given computation by performing many tasks of an operation concurrently, such as by using many cores provided on distributed systems of servers. A parallel system is one in which unrelated processes work together as peers using communication protocols to send and receive messages. A typical parallel system, a cluster computer, combines off-the-shelf multicore servers (each with their own cores, caches, main memory, busses, networking interfaces, and optional storage), optionally with GPU accelerators, and a network (10Gbit/s Ethernet and 100Gbit/s InfiniBand are typical cluster networks).

The Message Passing Interface (MPI) is a popular protocol for transmitting messages between peer processes that form a parallel job execution environment. It is a portable, efficient and flexible standard for the message passing model of parallel computing [9]. Parallel applications have routinely been written in MPI for the past 25+ years, and parallel libraries, including matrix multiplication libraries, are typically written in MPI notation to support their use in MPI-based applications.

The naive matrix multiplication has three loops involved in the multiplication and these can be ordered in six ways namely: ijk, ikj, jik, jki, kij, and kji loop orderings. This is the underlying technique behind the polyalgorithims, manipulated differently for the various algorithms. The orderings perform differently in terms of speed depending on matrix shapes and system architecture (e.g., cache). The technique reappears in the parallel solvers where they are sometimes made faster by enhancing data distributions and always strive to use suitable communication primitives (MPI functions). Goto et al. [12] explored the principles of highly optimized matrix multiplication for high performance; his work is a subset of the GotoBLAS library [11]. It uses five loops, which were implemented by Goto when he realized more opportunities for parallelism within the third loop and added two more inner loops. In his paper [12], smaller blocks and panels are created that can be loaded in smaller cache and, hence, yield faster performance.

Dense matrix multiplication is relevant today since the growth of data is rampant and the need to analyze it is also growing with every passing day. The physical storage in the computer devices is finite and eventually gets full. Virtual storage in the cloud is the new way to store data. This results in dense data types which must be manipulated for analysis. Dense matrix multiplication is used in a variety of applications and is the core of many scientific computations and data-analysis techniques. If we are able to make this operation faster and more scalable then we would have accomplished the goal of many applications, which is to have high performance in terms of speed of the application.

The polyalgorithm library was developed by Dr. Anthony Skjellum and his colleagues during the period 1991–1995. This library contains algorithms that apply to the general case of rectangular matrix multiplication on rectangular process grids and are classified according to the communication primitives used [26]. A polyalgorithm refers to a group of related algorithms that are grouped together to perform related operations. However none of them is the fastest algorithm for all input parameters; yet, each can achieve the best performance based on certain, given parameters. In the polyalgorithm library, the DGEMM kernel [21] was used for local matrix-matrix operations; this a routine that performs the double-precision matrix-matrix multiplication operation on subproblems that emerge during a parallel matrix-matrix multiplication within a single process address space. This is available as a highly optimized routine available in several libraries; here, we used the BLIS framework [37].

The general problem being solved in parallel is $C = \alpha A \times B + \beta C$, where A, B, and C are dense matrices that are dimensionally compatible. A number of studies, such as [13], show that the choice of algorithm in a given situation depends on the basic operations such as rank-1 update, and matrix-vector operations and how they are implemented by duplication and redistribution. When choosing an algorithm, one must consider the matrix with the most elements, then limit the data movement in that matrix. It is good to note that all the "big" operations that go on in the algorithms are a sequence of smaller operations. A complete matrix multiplication is a series of rank-1 updates. rank-1 updates are also referred to as dot product which means multiply a row with a column at any point of the multiplication. These can further be extended into matrix-vector operations, rank-k updates, and block matrices updates. Together with other operations such as reductions, and summing up of the C elements from the local matrices to get the final C result. The goal of various ways of performing the multiplication is to arrive at one that works the fastest for a given situation (matrix sizes and shapes and initial distribution across processes).

Fat-by-thin matrix multiplication refers to a matrix with fewer rows and comparatively more columns multiplied by a matrix with many rows and fewer columns. $C = A \times B$ (of shape $M \times K \times N$) is consequently one in which the A matrix is fat, and the B matrix is thin. That is, K is much greater than either M or N. This means that the resulting C matrix is small relative to both A and B. Many of the BLAS libraries are efficient for square matrices but their performance deteriorates as the matrices become non-square. However, these edge cases are inevitable in most tools that use BLAS, which inspired the research about performance of these libraries in fat-thin regions. A new implementation of GEMM, X_DGEMM was designed to improve the performance in these regions by Hines et al. [18].

Van de Geijn and Watts developed SUMMA [36], a collection of highly efficient and scalable matrix multiplication operations with the Message Passing Interface (MPI) as the parallelization model [9]. A rank-1-based version of the SUMMA matrix multiplication was prototyped in the polyalgorithms library [26]. rank-1 refers to a series of outer products, done one at a time, to update a given C element in a $C = A \times B$ multiplication. In this work, we prototyped a rank-k based SUMMA, SUMMA_K which is more efficient and is faster compared to rank-1. SUMMA_K performs a number of k dot products concurrently, and this is what makes it faster. The rank-k mode was evidently the original intent of Watts and Van de Geijn also in their work.

1.2 Motivation

Our motivation for this thesis is to prove that polyalgorithms are still relevant in the design of matrix multiplication algorithms and linear algebra libraries. This is because different algorithms have different ways of manipulating data especially the kind of data movements required to complete the multiplication. Matrix multiplication is not memory bound but computation bound. The data manipulation has different costs, both computational and communication-related. There is always a trade-off that has to be done between computation and communication and we aim to minimize the latter, in order to favor computation. Generally, the fewer the data movements, the faster the algorithm. Because of these dynamics in the performance of algorithms, there is a need to use a polyalgorithmic approach to maximize performance by having a set of algorithms solving the same problem and then choosing the best for a given situation. Li et al. proved that no single algorithm always achieves the best performance on different matrix and grid shapes [26]. Is this still true two decades later considering the new tools and software? Do we reproduce trade-offs using different algorithms in different use cases, or is one always the best? These are some of the questions that triggered the second generation polyalgorithm for dense matrix multiplication. This work aims to prove whether this is true for advanced technologies and tools. The difference in performance for these algorithms depends on the shape and size of matrices and grid, the storage type and the data mapping type. Sana et al. [23] acknowledged that the polyalgorithmic approach is a promising answer to the problem of choosing the best algorithm for the ever changing execution supports for parallel systems.

The second motivation is an implementation of a new version of BLAS called X_DGEMM, that aims to improve performance in the fat-by-thin regions in matrix-matrix multiplication. This algorithm was designed and prototyped by Thomas Hines [18]; it has been shown to achieve significantly increased performance in the fat-by-thin regions as compared to other BLAS libraries such as: MKL [20], OpenBLAS [39], and BLIS [37]. We would like to analyze how this newer version impacts the polyalgorithms. We also want to compare and contrast the performance of X_DGEMM vs. DGEMM while using the same matrix and grid dimensions. X_DGEMM could be ideal to test for polyalgorithmic behavior, which would further establishes the fact that polyalgorithms are indeed important in the design of linear algebra libraries. The importance of fat-by-thin dense matrix multiplication is their usefulness in many computer science applications; for instance, the extreme edge shapes of the matrices and grids are common in machine learning. As noted above, the term fat-by-thin means that the K-dimension has many values compared to the M and N dimensions in an $M \times K$ by $K \times N$ matrix multiplication.

We also compare the performance of the polyalgorithms set to COSMA [25], a recent, optimized matrix multiplication algorithms. This new algorithm will be included in the polyalgorithms library in the future for cases where its use is most effective and/or fastest. Expanding the library by adding new algorithms such as COSMA and CARMA [7] that are more efficient is a great motive for carrying out this research. With new algorithms, more arbitrary shapes of matrices will be included and this will cater to special/edge cases involved in matrix multiplication operations. This would greatly advance the functionality of the polyalgorithmic set and make it even more ready to use by the public as a standard dense matrix multiplication library.

1.3 Problem Statement and Objectives

Achieving scalability for an algorithm is key for any system to be efficient and to effectively utilize resources. In HPC, strong scaling (aka Amdahl's law [17]) refers to increasing the use of equivalent cores on a fixed size problem while weak scaling (aka Gustafson-Barsis' Law [15]) refers to increasing the problem size as the core count as problem size increase. For strong scaling, the goal is to minimize the time-to-solution while, for weak scaling, the goal is to achieve constant time-to-solution for larger problems respectively. Second generation polyalgorithms aims to leverage the available resources in a distributed system to achieve strong scalability. The objectives of this thesis are as follows:

1. Integrate, test and compare the new BLIS version (X_DGEMM) with the BLIS DGEMM.
2. Write a modern SUMMA algorithm that is rank-k based.
3. Test different use cases for different algorithms in the polyalgorithms library and to identify which one performs better.
4. Compare and contrast the performance of the polyalgorithms with the new, highly optimized matrix multiplication algorithm COSMA.

These are the research questions that we want to answer:

- Do we reproduce the trade-offs between using different algorithms in different use cases, or is one always best?
- Does the using of the basic BLIS DGEMM vs. new BLIS version, X_DGEMM, change performance of the algorithm(s) in a significant way? Which algorithm is best for a given "parameter sweep"?
- Does the new SUMMA algorithm outperform any of the older algorithms?
- Do the polyalgorithms set still achieve competitive performance when compared to newer algorithms on current technologies?

1.4 Contributions

The list below shows the contributions we made in our research toward performance improvement of dense matrix multiplication:

- Analyze whether a single algorithm performs best in all use cases of the polyalgorithm library. In this thesis, we design new test scenarios of different matrix sizes and grid shapes. The simulations were run on two powerful parallel computing clusters: 117 (OneSeventeen) and Stampede2 of varied node, memory, and network performance characteristics. This research offers extensive performance results for non-square matrix sizes and non-square process grid shapes and a comprehensive analysis to show the importance of polyalgorithms. In addition to the polyalgorithmic behavior demonstrated in the original polyalgorithms library built 25+ years ago, new dimensions of similar behavior are discovered from the new algorithms that we have added.

- Integrate a new version of BLAS, X_DGEMM and compare it to the DGEMM in the BLIS library to analyze whether it improves the performance of the algorithms especially for fat-by-thin regions. Extreme fat-by-thin shapes are used to expose maximum parallelism in the k-th dimension of the multiplication (within a multicore processor DGEMM operation) to meet the purpose of this algorithm. In this algorithm, we also noticed the three dimensional properties that are exposed in the shared memory of the multiple threads that perform the DGEMM operation. We also show how polyalgorithmic behavior occurs in context of the choice of BLAS and how it is necessary in the new BLAS version, X_DGEMM itself.

- Design, implement, and evaluate a rank-k based SUMMA. This version performs faster than the original SUMMA rank one based. It has the potential to perform faster than other algorithms in the set depending on the choice of the k value and sizes of grid and matrices. The algorithm will replace the rank-1 version which is abnormally slow because of its inefficient BLAS utilization.

- Verify that COSMA is a highly optimized matrix multiplication algorithm that has good performance for different combinations of grid shapes. We compare and contrast COSMA results with the the polyalgorithms set results. While experimenting with COSMA, we achieve scientific re-validation of the the research in the COSMA paper [25].

1.5 Outline

The remainder of the thesis is organized as follows: Chapter 2 includes the Background and Related work. Chapter 3 presents the methodology and the implementation of the algorithms.

In Chapter 4, we discuss the results from experiments that we carried out. We conclude the thesis in Chapter 5 by discussing our findings and by outlining future work.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

In this chapter, we discuss what has already been attempted in the parallel matrix-multiplication topic. First, we discuss the original polyalgorithm library made in 1993, specifically its design and results that were achieved from experiments and comparisons of the different matrix-matrix algorithms algorithms. Its suggested future work birthed this thesis. Second, we discuss the fat-by-thin scenario of matrix multiplication: its advantages and applications in the computation science. Third, we give a brief description of the SUMMA algorithm including its modifications and application over the years. The rest of this chapter is a further literature review, in which we discuss works related to matrix-multiplication algorithms plus their practical applications. We conclude this chapter with an description of the COSMA algorithm, a recent communication-optimal matrix-multiplication algorithm.

2.1 Polyalgorithms

The term polyalgorithms refers to having a group of related algorithms that perform equivalent operations but none of them are known always to perform best for arbitrary problem shapes, sizes, and/or concurrencies.. Our assertion is that polyalgorithms form a backbone in the building of high-performance linear algebra libraries. We assert that we cannot have a single algorithm in a library because its performance will deteriorate in different situations and excel in other scenarios. Different algorithms operate differently with regard to computation, communication, overheads, and potential overlaps, when one changes the dimensions of the grid or the matrices. This leads to some algorithms changing for better and some become worse in terms of performance. In the polyalgorithm paper by Jin Li et al., a two-dimensional logical grid topology of processes is used with a notation of G(P,Q) [26]. The grid layout enables mapping of matrix data to different processes in a parallel system. It also provides a platform to reference and manipulate data elements in a matrix operation. The elements in the grid then communicate to each other using MPI operations. The grid size aligns directly with the number of processes in the MPI communicator; for example, if we have $P = Q = 4$ then we have 4×4 (16) processors for that execution[1]. Grid shape is also polyalgorithmic in that different shapes result into varied performance of each algorithm and shape of matrices used. For example, Fox's algorithm is faster in terms of speed for square grids, yet the BB algorithm [26] is best for smaller size grids. Cannon's algorithm depends on the size of each of the matrices and is also good for square grid and matrix shapes.

The set of fourteen algorithms in the polyalgorithms library was designed and categorized into three groups: Cannon's, Fox's, and the Broadcast-Broadcast approach. A comprehensive taxonomy of each group was given in [26]:

- **Cannon's group**: Cannon C stationary, Cannon A stationary, Cannon B stationary, Cannon C general, Cannon A general, Cannon B general. In this approach, two matrices are shifted and it is preferable to leave the biggest matrix stationary. By doing this, communication overheads are reduced and the general algorithm performance improves. The Cannon's group algorithms are memory efficient because during the computation, limited additional memory is required.

- **Fox's group:** mm3_row, mm3_col, mm4_row, mm4_col, mm5_row, mm5_col versions. This consists of a one-to-all broadcast of one matrix and a shift of the other matrix. Blocks of the A matrix are broadcast using the one-to-all primitive in the row dimension and blocks of matrix B are shifted in the column dimension.

- **Broadcast-broadcast group**: BB version, SUMMA (1995 version). The only communication primitive is broadcast; it is simple to implement and is flexible.

Each of the algorithms are briefly defined here below.

[1] Multicore features of each process are handled strictly by the BLAS operations.

1. *mm3_row*: a direct extension of Fox's to deal with a non-square grid. Broadcast the rows of matrix A and shift rows of matrix B upward.

2. *mm3_col*: a direct extension of Fox's to deal with a non-square grid. Broadcast the columns of matrix A and shift rows of matrix B downward.

   The mm3 algorithms are loosely synchronous leading to a time lag delay in terms of idle time as it waits for unused rows and this makes the performance potentially sub-optimal.

3. *mm4_row*: it works similar to mm3_row. It is good for square grids and resolves the synchronous issues of the mm3 algorithms.

4. *mm4_col*: it is similar to mm3_col but without synchronization issues.

5. *BB*: this is a simple implementation of matrix multiplication involving only one communication primitive, broadcast. Matrix A broadcasts its rows and B broadcasts columns. No initial alignments are required, which eliminates extra alignment overheads. The BB algorithm works well when the process counts are small, since increased communication cost in both dimensions is expensive.

6. *cannon_c*: This is the original Cannon's algorithm.

7. *cannon_a*: Used when matrix A is much larger than matrix B; this reduces the communication costs.

8. *cannon_b*: Used when matrix B is much larger than matrix A.

9. *cannon_cg*: keep matrix C stationary.

10. *cannon_ag*: keep matrix A stationary.

11. *cannon_bg*: keep matrix B stationary.

12. *summa*: this algorithm is a broadcast-broadcast approach. It is a series of dot products of rows of matrix A and columns of matrix B. It is a simple implementation and flexible for

different matrix shapes and grid sizes. However, its performance deteriorates[2] when the process count increases and communication cost in both dimensions is expensive.

13. *mm5_row*: this algorithm works well when $P \geq Q$ grid shapes. It broadcasts the exact number of columns of A that can be locally multiplied by matrix B. It requires less memory than mm3 and mm4.

14. *mm5_col*: this algorithm works best when $P \leq Q$ grid shapes.

The authors in [26] discussed the communication cost analysis of the four communication primitives used in the library: shift, slide, align, and broadcast and with modeled these operations with predictive performance equations to show this [26]. A communication cost analysis for matrix multiplication algorithms was also discussed for each of the three approaches. Most interestingly, the expensive initial alignment of the Cannon group is explained. From this, a conclusion was drawn that the communication cost can be reduced by choosing the appropriate version of Cannon's algorithms that allows the matrix with the largest size to remain stationary [26]. The authors also showed situations where each of the MM versions performs best. mm5 is suitable for particular grid shapes cases (i.e., the row versions is used when $P \geq Q$ and the columns versions is used for situations where $Q \leq P$). Memory-requirement analysis was done for each algorithm to account for incremental memory overhead while doing the computation and can be summarised as follows: The Cannon group is memory efficient because it only has slide and shift primitives and thus requires no extra memory. Fox's group requires temporary memory to store the local matrix that is broadcast in either the row or column dimension and it was noted that mm5 requires less memory than mm3 and mm4. The BB algorithm requires more incremental temporary memory to store both the local matrix A and the local matrix B that are broadcast in the row and column dimensions respectively [26].

Jin Li et al. concluded that "A polyalgorithm is a practical approach to maximize performance by marrying multiple algorithms" [26]. No single algorithm discussed in their paper achieved the best performance for all situation of different matrix and grid shapes. The authors

[2] Our retrospective on this is that the rank-k SUMMA should have been implemented originally. That is why we did so in this work.

further found that for grid shapes where $P \geq Q$, one should use the row version of the Fox's group. The choice of algorithm from the Cannon group is largely dependent on the size of the matrices, and one should always choose to leave the largest matrix stationary. The BB algorithm is best for small process counts. The performance of these algorithms is dependent on both communication performance and memory.

Despite the many parallel algorithms that have been designed for different computer architectures and machines, the challenge that the user faces is to decide which one will perform better for a given system or application. Polyalgorithms is the solution to this problem. If we can automate the process of choosing the best option using machine learning tools, we could easily identify the best. In heterogeneous systems [3], one version of an algorithm will be suitable for configurations in only one machine. Heterogeneous computing refers to systems that use more than one kind of processor or cores. These systems gain performance or energy efficiency not just by adding the same type of processors, but also by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks such as massive floating-point operations in GPUs. Therefore, in polyalgorithms we obtain the best performance by having multiple algorithms and each can be the best in different contexts [23].

2.2 Fat-by-Thin Matrix Multiplication

A fat-by-thin matrix refers to multiplying a matrix that has fewer rows and comparatively many columns with a matrix that has more rows and comparatively fewer columns. The resulting matrix is a smaller size matrix as compared to either of the input matrices. Some applications of the fat-by-thin multiplication include; data mining algorithms such as linear regression, PCA, and highly optimised k-Means clustering, data analytics, Algorithm-Based Fault Tolerance (ABFT)[18]5. Fat-by-thin regions can be found in large data sets and a decline in performance is observed for edge cases while using the BLAS GEMM routine [18]. The solution is to devise a means that speed up performance in the fat-by-thin regions while minimizing the communication overhead. Notice that in a fat-by-thin multiplication, the size of matrix C (the output matrix) is small so the cost to sum up the C matrix can be accommodated compared to the speedup from parallelization in the K (inner multiplication) dimension. (We remind the reader that the entire question of BLAS performance is encapsulated in the DGEMM operation used and complements the choice of polyalgorithm applied at the global level on the $P \times Q$ process grid of MPI processes.)

While SGEMM/DGEMM operations are extremely well studied for square cases, GEMM operations with fat-by-thin matrices has evidently not been fully studied. A few researchers have implemented algorithms for tall-by-thin matrix multiplication [5], which can be closely compared to fat-by-thin matrix multiplication. Chen et al. in particular presented the challenges of optimizing GEMM for non-square matrix shapes and provided an algorithm for multiplying square matrices by thin matrices [5]. Cody et al. proposed two high-performance, irregular shape matrix-matrix multiplication algorithms on GPUs, with several optimization techniques focusing on GPU resource utilization [30]. Some optimizations include making each thread perform more work and interleaving the computation of each of the tiles.

2.3 SUMMA

SUMMA: The Scalable Universal Matrix Multiplication Algorithm [36] is a broadcast-broadcast algorithm of the form:

$$C = \alpha AB + \beta C \tag{2.1}$$

The elements of C are given canonically as follows:

$$C_{ij} := \alpha \sum_{k=0}^{K-1} A_{ik} B_{kj} + \beta C_{ij} \tag{2.2}$$

when size A ($M \times K$) and B ($K \times N$) are used. The Message Passing Interface (MPI) is used in this algorithm to provide the performance-portable message passing abstractions [36]. All the implementations of SUMMA are a generalization of the broadcast-broadcast approach. They require less work space and overcome the need for a square 2D grid of processes. It is used by PLAPACK [35] linear algebra packages. It's simpler and more flexible implementation gives it an advantage over other algorithms however, it is sensitive to communication overhead. The authors of SUMMA concluded that it is the best algorithm to use for general purpose implementations of matrix multiplication. They also concluded that SUMMA is suitable for the implementation of distributed BLAS matrix multiplications that incorporate different combinations of transposed and nontranposed matrices. Four different notations of the algorithms are presented in [36], where A and B are not transposed and cases where either A or B or both A and B are transposed[3]. This type of implementation shows the flexibility of the SUMMA algorithm. SUMMA algorithm uses pipelining and blocking and it allows larger problem sizes on each node. Blocking, the splitting of a matrix into smaller blocks, replaces the rank-one updates, which is slow (as noted above). Because of its dependency on the broadcast approach, SUMMA passes a message around a logical ring that forms the row or column; thus, it pipelines computation and communication. In the ring format, each process only communicates to its neighbor, no broadcasts are done; hence, fewer messages. Some other improvements have been made in the SUMMA algorithm; for example, instead of broadcasting single rows and columns, block rows and columns are used.

2.4 Additional Literature Review

Researchers have explored the concept of polyalgorithms for many years. Jin Li et al. [26] defined polyalgorithms as "the use of two or more algorithms to solve the same problem with a high level decision-making process determining which of a set of algorithms performs best in a given situation." A polyalgorithm is needed because what is best changes as a function of the input parameters. In Jeddi et.al. [23], the authors designed a set of solving algorithms that have varying behaviors and performances with an aim to derive the best one for a target parallel system. They showed that, as the systems are composed of collections of heterogeneous machines, it is difficult and nearly impossible for a user to choose an adequate algorithm because the execution requirements are continuously changing. One version will be well suited for a parallel configuration and not for another. They concluded that using the polyalgorithmic approach for a fast adaptation to the continuous changing of parallel and distributed systems is ideal. In 2004, Nasri et al. [29] proposed a polyalgorithm that takes advantages of standard and fast algorithms.

[3]This thesis, in following the Polyalgorithms work that preceded it, only covers the non-transposed cases.

It is able to choose the most suitable algorithm automatically for computing the matrix matrix multiplication of any dimension on a particular parallel system in a homogeneous cluster.

Several matrix multiplication algorithms have been developed, each designed to achieve speed, flexibility, and scalability. Cannon presented a systolic 2D approach [4] which has been extended over the years to accommodate a wide range of parameters ranging from square grids, non-square grids, and non-square matrices. The challenge that Cannon algorithms face is the initial and final alignments of the matrices, which was discussed in [26]. Fox's algorithm is also a popular algorithm for parallel dense matrix matrix multiplication [10]. Fox's algorithm has also been extended over the years to deal with non-square grids, row and column versions. The SUMMA algorithm [36], which uses broadcast broadcast approach, is simple and flexible, and it is used in linear algebra libraries such as [35]. Other matrix multiplication algorithms include; Mathur-Johnsson's algorithm [28] which extends Cannon's algorithm (the matrix C-stationary version) to deal with arbitrary matrices and grids. The PUMMA [6] extends Fox's algorithm to non-square grids but limits layouts ton block scattered data distributions in both dimensions. In [34], a recursive algorithm for square matrices was designed. Strassen's algorithm and its later variants (and parallel variants) can be faster than the naive multiplication algorithm (with sequential complexity of $(O(MNK)$ because of lower algorithmic complexity; it may also performs inefficiently for non-powers-of-two matrices and has lower accuracy. Its publication resulted in much additional research about matrix multiplication with better parameter tuning [16]. Further discussion of Strassen-type algorithms is beyond the scope of this thesis.

A 3D-approach to parallel matrix multiplication was introduced by Agarwal et al. in 1995, with an algorithm that was shown to be load balanced for both, communication and computation for parallel systems [1]. The advantage of 3D decomposition is that it reduces the communication overhead through reduced data movement. However, it requires more memory to replicate the matrices compared to the 2D algorithms; overall 3D algorithms have less total communication cost. This led to the 2.5D [31] approach, which takes advantage of the extra memory while reducing communication cost. In the paper [31], the goal was to minimize communication along the critical path. It is a balance between 2D and 3D since it maximizes the advantages of both approaches.

Scalability is an important subject in dense matrix multiplication algorithms. Scalability refers to the extent to which an algorithm's performance remains efficient as the problem size and/or the number of processes/cores is increased. Bryan et al. defines scalability as "the ability to retain high performance as the number of processors is increased" [27]. Gupta et al. presented a scalability analysis for classical matrix-multiplication algorithms [14]. The authors confirmed that none of the algorithms are superior because there are various factors that determine best performance. Some of the factors one must consider to determine scalability are communicating and computation speeds, a small communication overhead does not necessarily mean best performance.

COSMA is a recent, parallel matrix-matrix multiplication algorithm that was proposed by Kwasniewski et al. in 2019 [25]. Their idea was based on the Red-Blue Pebble game abstraction [24] to derive and model tight sequential and parallel I/O lower bounds. In this research, the authors proved that their algorithm had near communication optimal performance and that COSMA outperformed the best parallel BLAS libraries in all scenarios. COSMA aims to use local resources optimally in order to overcome the limitations of the top-down approach, which operates by taking the global problem and splits it into subsets that equal the number of ranks[4]. They used a sequential schedule that optimizes the re-use of its local memory by accumulating the outer product within the cache of a given tile. After that, they parallelize by extending the outer product across the global domain of the distributed system and reduction is done by the first process, called the bottom-up-approach. They form optimal square shape tiles and so the communication between ranks is minimized. They presented proofs for sequential and parallel I/O lower bounds and a heuristic for implementation optimizations. Their results show that COSMA is always has best performance and has least communication of data as compared to CARMA [7], ScaLAPACK [22], and CTF [32]. They created a COSMA miniapp in their COSMA library that is open source and can be easily used to test different run times for matrix multiplications and compare with other algorithms. We compared our polyalgorithms with COSMA since it is one of the most recent matrix multiplication algorithms to demonstrate whether it is faster in all cases as suggested by the authors. The results are presented in Chapter 4.

[4]This is how all the previously mentioned parallel operations are derived.

2.5 Summary

This chapter presented the design of the original polyalgorithm library and defined the 14 algorithms in the library. We also discussed fat-by-thin matrix multiplication, followed by a description of SUMMA algorithm. Literature review details polyalgorithms, matrix multiplication approaches, scalability and recent parallel matrix-matrix multiplication algorithm showing their advantages and constraints. From the reviews, matrix multiplication is a fundamental operation in solving computer science related problems. Extensive research has been done in this area in order to optimize the performance of matrix multiplication algorithms and in turn, speed up the applications in which these algorithms are used.

CHAPTER 3

METHODOLOGY

In this chapter, we discuss the approach that was used for this research. We used experimental data by controlling and manipulating variables to produce the desired results for analysis. The first section describes the design methodology, which shows the fundamental ideas of the polyalgorithm library: the logical grid and data distribution independence (DDI). The following sections describe the methods for data collection and measurements used to quantify the data. We also define the software and equipment and their architecture as used for the experiments.

3.1 Design Methodology

The fundamental idea for the design of the polyalgorithms library depends on two key factors: the Logical 2D process grid and data distribution independence. A logical grid is a collection of processes logically assigned a shape $P \times Q$; the processes are named p and q respectively. A logical grid allows the process to be referenced by its coordinates within a grid and this is how mapping of data occurs on a grid. Logical grids can be readily mapped to physical node topologies, which makes the design simple and flexible to apply in general applications. Figure 3.1 shows an example of a process grid and process mapping where eight processes are mapped to a 2×4 process grid. For a grid shape $P \times Q$, the grid has indexes $p, 0 \leq p < P$ and $q, 0 \leq q < Q$ respectively and the indexes are used for mapping the processes onto the logical grid. Data Distribution Independence (DDI) separates the organization of the data in parallel from the correctness of the answers; performance may change, but correctness does not change. DDI prevents explicit data redistributions, which is costly and useful for building scalable parallel libraries. DDI allows application steps before and after an algorithm to make choices of how

	0	1	2	3
0	0	1	2	3
1	4	5	6	7

Figure 3.1 Eight processes mapped to a 2×4 process grid

they store data, rather than simply conforming to a specific, rigid data layout. For instance, in SCALAPACK [22], block scatter in 2D is the only distribution, where you can set the blocksize. You can vary $P \times Q$, and blocksize, but arbitrary distributions aren't allowed. The blocksize in SCALAPACK might be related to the application or to the algorithm; both are possible, and generally would not be the same size, so this is about how big your checkerboard elements are. There are two types of blocking; algorithmic blocking and application blocking. Blocking is much more rigid than DDI, which could take advantage of blocking or not, but doesn't mandate it.

For any logical grid of processes $P \times Q$, a DDI linear algebra problem distributes matrix rows and columns that are "compatible" in which they store data with regard to each other. However, in both the row and column dimension of the grid of processes, any one-to-one, onto, and invertible mapping is allowed. This is the data distribution for each dimension of the matrix object. This means that one can map coefficients as linear, scatter, block scatter, etc. Once one has compatible data storage, then, regardless of what that is, the algorithm adapts to it and produces the right output. What is "compatible" can be "all in the right place from the start" or "the algorithm rearranges to make the right pieces be in the right places." In matrix multiplication in the polyalgorithms library, there is a requirement for basic compatibility of the objects when they start (A, B, C) and the library maintains that as the algorithms evolve step-by-step. This kind of dynamic compatibility is a super-set of what SCALAPACK [22] with PUMMA [6] does. A super-DDI algorithm would work for any original distribution of A, B, and C in the matrix multiplication situation, simply by moving the data around extra. In the polyalgorithms library, the types of DDI algorithms do not do that extra work.

3.2 Design of Experiments

One of the goals of this research is to determine whether the conclusions that were presented in the polyalgorithm library circa 1995 are still valid or not. We aim to show that the trade-offs between using different algorithms are still valid, but examine the potential superiority of using only one algorithm. The simulations were carried out the one-seventeen cluster (117) at the SimCenter, at the University of Tennessee at Chattanooga. This is a powerful cluster with 33 nodes and a login node. The compute nodes are configured as follows: two (dual-socket) Intel Xeon E5-2680 v4, 2.4GHz chips, each with 14 cores for a total of 28 cores per compute server. There is 128 GB of RAM per compute server for about 4.5 GB of RAM per core. One NVidia 16GB P100 GPU with 1792 double precision cores is available on each node (but was not used in this study). Theoretical peak performance of about 1 TFLOPs (CPU only) or roughly 5.7 TFLOPs (CPU/GPU).

The test runs design was based on the need to analyze the performance of the algorithms for different grid shapes and sizes and considering the capacity of the cluster (117). Three sweeps of data were designed while named according to the number of nodes (with one MPI process per server/node) involved in a given experiment: 16 , 24, and 30 nodes. Each of these was tested with grid shapes that were varied in terms of factors of the number of nodes. That is, a 16-nodes sweep includes: 1×16, 16×1, 8×2, 2×8, and 4×4. A 24-node sweep: 1×24, 24×1, 12×2, 2×12, 8×3, 3×8, 6×4, and 4×6. Finally, a 30- node sweep; 1×30, 30×1, 15×2 ,2×15, 5×6, and 6×5. These sweeps vary the grid shapes $P \times Q$ at constant total processes and this enables us to analyze how different algorithms perform when $P > Q$ or $P < Q$, because some of the algorithms were specifically designed to deal with particular grid shapes. This also shows how data distribution of the matrices is affected by different grid shapes, which in turn affects the overall performance of the algorithms. It was interesting to discover how the extreme grid shapes affect performance and this is applicable in the data (perfectly square matrix sizes on square grids) analysis world since it is rare to find perfect data in practical applications. Data almost always has extremes and edges that most researchers sometimes leave out.

We studied both square and fat-by-thin matrix shapes. The matrix sizes used were $20,000 \times 20,000 \times 20,000$: matrix A of size $20,000 \times 20,000$, matrix B of size $20,000 \times 20,000$ resulting into matrix C of size $20,000 \times 20,000$ for the square case. We used the shape $1,000 \times 1,000,000 \times 1,000$ for the fat-by-thin case; thus, matrix A is of size $1,000 \times 1,000,000$, matrix B is of size $1,000,000 \times 1,000$ resulting into a relatively small size square matrix C of size $1,000 \times 1,000$. The importance of the size choices used will be discussed in the following paragraphs. Table 3.1 is a summary of all the dimensions and parameters described in this paragraph and the previous paragraph. The number of nodes mentioned is the total number of nodes on the cluster that will be used for a set of runs. For each case, P is the row dimension of the process grid, Q is the columns dimension of the process grid, M is the number of rows of a matrix A, N is the number of columns of matrix B, while K is the number of columns of matrix A and rows of matrix B.

The experiments were designed in in such a way that each algorithm runs 10 times and the average run time (measured in seconds) is what we used to measure performance. The variance (sample standard deviation of the mean) is given for this value. (Similar mean and deviations are computed for the minimum mean runtimes for each case, but not used elsewhere in this thesis.) We also tested the original BLIS DGEMM and X_DGEMM (that leverages BLIS within itself) on the same test cases. Tables that show performance of the different algorithms and comparisons of the performance will be discussed in Chapter 4.

Further, we performed test runs on XSEDE stampede2 [33]. Stampede2 is one of the Texas Advanced Computing Center (TACC) systems, and is one of the University of Texas at Austin's flagship supercomputers. The first phase of the Stampede2 roll out featured the second generation of processors based on Intel's Many Integrated Core (MIC) architecture. Stampede2's 4,200 Knights Landing (KNL) nodes represent a radical break with the first-generation Knights Corner (KNC) MIC coprocessor. Unlike the legacy KNC, a Stampede2 KNL is not a coprocessor: each 68-core KNL is a standalone, self-booting processor that is the sole processor in its node.

Phase 2 added to Stampede2 a total of 1,736 Intel Xeon Skylake (SKX) nodes. SKX nodes are what we used for experiments in this research.

The measurement metric used for this study was time taken for an algorithm to run, which was measured in seconds. The smaller the number of seconds that an algorithm takes to run, the faster it's performance is. Different algorithms had varying run times for the arbitrary situations considering the type of test, whether square or fat-by-thin. The analysis in this thesis was primarily based on numbers (quantitative) and a focus on fastest times (time to solution), while noting uncertainty in fastest times where significant. The use of quantitative data analysis methods was the best option to use. This is because we are dealing with numbers, run times and performance measurement. It is easy to compare, contrast, and accurately note all the variations in the data outputs from the test runs. From these numbers we were able to determine which algorithm was more efficient for a particular situation. We drew conclusions about how the current performance of algorithms is similar or different from those reported in the earlier work from the 1990's. The implementations of the new versions of the BLAS brought about performance changes and the the measure of performance as stated earlier was fastest run time in seconds. Tables, graphs and visualizations were used for analysis in Jin Li's paper [26]. The results presented enabled the replication of their work to perform similar tests. This prompted us to use similar tools to what they evidently employed.

3.3 Ideal Performance

From the experiments we designed for this work, we expected to observe polyalgorithmic behavior from the simulations, meaning that that not one single algorithm should perform best for all cases. This had been shown in [26]. Like the earlier work, we varied the test parameters in terms of grid shapes and matrix sizes and therefore, we expected the results to differ from those made in the original paper. The new version of the BLIS X_DGEMM was also expected to deliver better performance as compared to the original BLIS DGEMM for fat-by-thin matrix size cases. This improvement had been proposed and demonstrated in [18]; they compared the new algorithm with optimized baseline BLAS libraries and the results showed the significant improvement with the new algorithm. For us, these performance variations suggested that X_DGEMM would change the best algorithm's performance, and potentially change which algorithm was fastest by reducing inefficiencies of on-node BLAS for the fat-by-thin case.

Md Mosharaf Hossain et.al. in [19], showed that performance degradation is higher for fat-by-thin matrix multiplication for edge cases in big data sets. Likewise, the original DGEMM performs better than the X_DGEMM for square size matrix cases, according to the numerous experiments made in this research. So, we did not expect to use X_DGEMM in all situations. Further, the problem sizes are different in each process given non-square grids and/or non-square processes, meaning that choice of DGEMM vs. X_DGEMM actually is a process-by-process and subproblem-by-subproblem issues. However, in this thesis, we only addressed the all-or-none case of either using normal DGEMM or else X_DGEMM in all processes of a parallel multiplication. We left as future work an internal selection mechanism and tradeoff of DGEMM vs. X_DGEMM modes of operation, which we considered a polyalgorithmic optimization for the BLAS library itself.

Ideally, the SUMMA_K algorithm added should perform faster than the original rank-1-based SUMMA. rank-1-based SUMMA is slow, due to the series of dot products that are done one at a time, which is inefficient given the memory hierarchy of a node. From the polyalgorithm set, we expect mm5_row to work best for cases where $P > Q$ and mm5_col to work best for cases where $P < Q$, based on the prior results from Li et al. Likewise, based on the earlier work, we expect Fox's group algorithms: mm3, mm4, mm5 should have similar performance [26].

3.4 Implementation

In this subsection, we discuss how the polyalgorithm library works. The design of the new algorithms is described and sample pseudo-code is presented.

3.4.1 Description of the polyalgorithm library

The parallel dense matrix multiplication algorithms were designed in C. As with other libraries, the Message Passing Interface (MPI), a standardized application programmer interface

(API) for the message-passing model of parallel computing among unrelated processes, was used to send and receive messages. In particular, we employed Open MPI version 3.1.0. The OpenMP notation for on-node concurrency was used to implement the new algorithms X_DGEMM extension for shared memory inside the BLAS. The BLAS library used for our study is BLIS [37].

A brief discussion of the test programs is given in Appendix C.

3.4.2 X_DGEMM algorithm

The new algorithm was motivated by the fact that the the General Matrix-Matrix Multiplication (GEMM) is slow for fat-thin regions. X_DGEMM was designed by Thomas Hines in his research work at the Tennessee Technological University [18]. He analyzed the performance of DGEMM using three BLAS libraries and confirmed that it was slow for the fat-by-thin region across all three libraries. The idea is that many single threaded DGEMMs can be run in the fat-by- thin regions with high performance. As noted above, fat-by-thin multiplication refers to a matrix with fewer rows and comparatively columns multiplied by a matrix with many rows and comparatively fewer columns. The advantage is that we now have a small size of C so the sum reduction over partial C sums will not be expensive in terms of memory (because C is replicated). The algorithm was specifically designed for fat-by-thin regions and is not practical otherwise. Using a large K value will typically yield good performance taking into account the available computing resources (such as extra memory). The X_DGEMM also works with sub-block matrices ($m \neq lda \neq ldc$, $k \neq ldb$), and this supports sub-blocking compatibility in the DGEMM call within the X_DGEMM.

The X_DGEMM algorithm works as described in [18]: "The main idea of the algorithm is to split A and B in the k dimension. A becomes a 1-by-t block matrix, and B becomes a t-by-1 block matrix. C has no k dimension and so becomes a 1-by-1 block matrix. The multiplication turns into a dot product where the t elements of A and B are multiplied pairwise and the resulting 1 by 1 block matrices are summed to form C. The t block matrix multiplications are performed by calling single-threaded GEMMs. Each GEMM writes to the entire C matrix. As we want all the

Require: A, an m by k matrix (input)
Require: B, a k by n matrix (input)
Require: C, an m by n matrix (input / output)
Require: $nthreads$, the number of threads to use
1: Partition A and B in the k dimension into $nthreads$ submatrices: A_0, A_1, ...
2: Allocate space for $nthreads$ C matrices: C_0, C_1...
3: Call GEMM $nthreads$ times with A_i, B_i, and C_i as parameters
4: Sum reduce C_is to C

Figure 3.2 X_DGEMM algorithm (adapted from [18])

GEMMs to run at the same time, each is given scratch space to write its own partial C. Finally, all the partial Cs are sum reduced to the full C [1]" [18]; this algorithm has been shown in Figure 3.2.

Again, note that X_DGEMM is a superstructure over the underlying BLIS kernel for DGEMM; the performance tuning effect of X_DGEMM derives from re-parametrizing the total distribution of concurrent OpenMP threads as compared to BLIS' default mechanism [18].

3.4.3 Modern rank-k based SUMMA

The idea is to improve the efficiency of the rank-1 SUMMA previously coded in the polyalgorithm library by increasing the opportunity for on-node parallel efficiency. rank-1 refers to a series of outer products done one at a time to update a given C element in a $A \times B = C$ multiplication. Rank-k, on the other hand, carries out k number of column or row broadcasts at the same time (followed by rank-k type BLAS operations locally in each process). SUMMA is a broadcast-broadcast approach. In the broadcast approach, we use local matrices $\hat{A}$ and $\hat{B}$. The number of columns in $\hat{A}$ and the number of rows in $\hat{B}$ is what determines the rank of the algorithm. Matrix $\hat{A}$ and $\hat{B}$ are temporary buffers for storing the local matrices broadcast. Each process computes as much of the matrix multiplication as can locally after each communication phase.

In rank-1 SUMMA, each local matrix $\hat{A}$ has size of $m\hat{A} \times 1$ and local matrix $\hat{B}$ has size of $1 \times n\hat{B}$ rows and columns. So during the MPI broadcast, a given root process on the grid will

[1]Note that the block dimension referred to is nominally $\lfloor \frac{k}{t} \rfloor$.

broadcast rows of matrix $\hat{A}$ with size of $m\hat{A}$. A root process in matrix $\hat{B}$ will broadcast columns of size $n\hat{B}$. BLAS performance on each node is limited for this case.

Figure 3.3 shows the rank-k SUMMA algorithm (SUMMA_K) where the size of each local matrix $\hat{A}$ is $m\hat{A} \times k$ and the local matrix B is of size $k \times n\hat{B}$. So at every process, the process at root will broadcast a count of $m\hat{A} \times k$ rows for matrix $\hat{A}$. The root process in $\hat{B}$ will broadcast a count of $k \times n\hat{B}$ columns. The mapping and data distribution functions are also adjusted in the rank-k version, instead of incrementing once at every iteration in the global indexes, we increment by a factor of k. When all data distributions are complete, we use the local DGEMM (or X_DGEMM) matrix multiplication to multiply the matrices and update the local C.

$C := \beta C;$ // Scale matrix C (input/output)
$k := g;$ // Initialization of k (input)
$m\hat{A} := \hat{A} \rightarrow m;$
$m\hat{B} := \hat{B} \rightarrow n;$ // Initialization of local matrix sizes
$kA := 0;$
$kB := 0;$ //Initialization of the process communicator
for $I = 0;\ I < K;\ I += K$ **do**
  if $kA == q$ **then**
    $\hat{A} \leftarrow row\ broadcast(kA)A;$ // Row broadcast matrix A at root kA
    // Broadcast $m\hat{A} \times k$ rows
    $kA := kA + 1;$ // For all in process grid
  end if
  if $kB == p$ **then**
    $\hat{B} \leftarrow col\ broadcast(kB)B;$ // Col broadcast matrix B at root kB
    // Broadcast $k \times n\hat{B}$ columns
  end if
  // Actually use the DGEMM or X_DGEMM multiplication multiply
  for $i = 0;\ i < m\hat{A};\ i{+}{+}$ **do**
    for $j = 0;\ j < n\hat{B};\ j{+}{+}$ **do**
      for $ik = 0;\ ik < k;\ ik{+}{+}$ **do**
        /* Update local matrix C*/
        $C(m\hat{A} \times j + i)\ := C + a \times \hat{A}(m\hat{A} \times ik + i) \times \hat{B}(k \times j + ik);$
      end for
    end for
  end for
end for

Figure 3.3 rank-k SUMMA algorithm

Table 3.1 Summary of parameters and dimensions used for experiments. **Number of nodes** is the total number of nodes on the cluster that are used for a set of runs, *P* is the row dimension of the process grid, *Q* is the columns dimension of the process grid, *M* is the number of rows of a matrix *A*, *N* is the number of columns of matrix *B*, and *K* is the number of columns of matrix *A* and rows of matrix *B*

Number of nodes	**P**	**Q**	**M**	**K**	**N**
16 nodes			**square**		
	16	1	20,000	20,000	20,000
	8	2	20,000	20,000	20,000
	4	4	20,000	20,000	20,000
	2	8	20,000	20,000	20,000
	1	16	20,000	20,000	20,000
			fat-by-thin		
	16	1	1,000	1,000,0000	1,000
	8	2	1,000	1,000,0000	1,000
	4	4	1,000	1,000,0000	1,000
	2	8	1,000	1,000,0000	1,000
	1	16	1,000	1,000,0000	1,000
24 nodes			**fat-by-thin**		
	24	1	1,000	1,000,0000	1,000
	12	2	1,000	1,000,0000	1,000
	8	3	1,000	1,000,0000	1,000
	6	4	1,000	1,000,0000	1,000
	4	6	1,000	1,000,0000	1,000
	3	8	1,000	1,000,0000	1,000
	2	12	1,000	1,000,0000	1,000
	1	24	1,000	1,000,0000	1,000
30 nodes			**fat-by-thin**		
	30	1	1,000	1,000,0000	1,000
	15	2	1,000	1,000,0000	1,000
	6	5	1,000	1,000,0000	1,000
	5	6	1,000	1,000,0000	1,000
	2	15	1,000	1,000,0000	1,000
	1	30	1,000	1,000,0000	1,000
	10	3	1,000	1,000,0000	1,000
	3	10	1,000	1,000,0000	1,000

3.5 Summary

We discussed the design methodology of the polyalgorithm library, which is made of a logical grid and data distribution independence. The design of experiments was described for future reference. The architecture of the servers (nodes) used for the study were defined with their limitations and advantages. We also presented the ideal performance for the library and the new algorithms (rank-*k* SUMMA), suggesting reasons for the expected results. A detailed flow of operations and test programs that make up the polymath library was indicated with a forward reference to Appendix C. Pseudo-code depicting the specific parameters, X_DGEMM design, SUMMA_K design was provided.

CHAPTER 4

PERFORMANCE EVALUATION

In this section, we present the results from the experiments that were performed for our research to compare performance of various parallel matrix-multiplication algorithms using two GEMM multiplication kernels (DGEMM and X_DGEMM) on two computer clusters. The algorithms were tested on the "117" and "Stampede2" clusters. We have arrived at the following discoveries and reaffirmations of previous results. We are able to conclude that:

- The Fox's (MM) group of algorithms perform better than other algorithms for arbitrary grid shapes and matrix sizes. These algorithms in the MM group show polyalgorithmic behavior for some cases but experiments show that the mm5 version predominantly has the best performance for grid shapes where $P < Q$ or $P > Q$. Usually, non-square grid shapes were used in the experiments and therefore, the mm5 group had the best performance primarily where the grid was extremely non-square. Other MM algorithm had the best performance in cases where the grid shape was square or nearly square.

- In some of the tests that were carried out on square grids, the polyalgorithmic behavior was reproduced for different situations. In such cases, just like in the [26], no single algorithm had best performance for all the different matrix shapes.

- X_DGEMM is significantly faster than the original DGEMM for fat-by-thin matrix multiplication. The difference is smaller when the grid shape is square or nearly square.

- X_DGEMM is also polyalgorithmic for special non-square matrix cases. X_DGEMM has similar trends of performance with DGEMM and the MM group had best performance with polyalgorithmic behavior based on the grid shape.

- The new algorithm Rank_k SUMMA is faster than the rank-1 SUMMA; it sometimes outperforms than other algorithms in the library. We also noted that the SUMMA_K algorithm showed similar performance to the Broadcast-Broadcast (BB) algorithm due to Li et al. This is because both algorithm use the same operations (broadcast primitives) and both are outer product based. The BB algorithm had competitive performance in range with other algorithms, there was no case where it was the best.

- The Cannon group had competitive performance with the other algorithms for square matrix size cases. Most of the Cannon group algorithms executed successfully for small grid sizes (16 nodes) but the performance deteriorated as the grid size was increased and we had to eliminate most of the algorithms that failed to execute. The only times that an algorithm from the Cannon group had best performance was where the grid shape was square: 4×4, 5×5. This is not unexpected given prior knowledge.

Some of the algorithms that were tested in 1993 have not been included in some test scenarios. This is because of our discovery of some constraints in those algorithms in terms of what matrix and grid shapes plus sizes. Some of the algorithms in the Cannon group were excluded for certain experiments. All result tables have been included in Appendices A and B for reference. Table 4.1 presents a few of the cases that will be discussed in the remainder of this chapter. The cases are for both fat-by-thin matrix shapes $(1,000 \times 1,000,000 \times 1,000)$ and square shapes $(20,000 \times 20,000 \times 20,000)$. The results presented are for the 16 nodes sweep; $1 \times 16, 16 \times 1, 2 \times 8, 8 \times 2$ and 4×4. More data including the other two sweeps are given in the aforementioned appendices.

4.1 Running on 117

The specifications of the 117 cluster were discussed in Chapter 3. The results show that the algorithms have different performance for various grid shapes and matrix sizes. The following graphs show the performance changes among algorithms for the two GEMM versions. When we change the dimensions of the matrix, it will evidently also behave differently in terms of performance for arbitrary grid sizes. Figure 4.1 depicts a bar graph for the average run times in seconds for DGEMM vs X_DGEMM for a fat-by-thin $(1k \times 1m \times 1k)$ matrix shape on a 16×1

Table 4.1 A mapping of case number to specific parameter set

Case	M	K	N	P	Q
1	1k	1m	1k	16	1
2	20k	20k	20k	16	1
3	1k	1m	1k	8	2
4	20k	20k	20k	8	2
5	1k	1m	1k	2	8
6	20k	20k	20k	2	8
7	1k	1m	1k	4	4
8	20k	20k	20k	4	4
9	1k	1m	1k	1	16
10	20k	20k	20k	1	16

process grid. We can see that the difference in performance that X_DGEMM indeed performs better than DGEMM for the fat-by-thin cases. On the other hand, Figure 4.2 shows that DGEMM has better performance than X_DGEMM for square matrix shapes. Figure 4.3 shows that DGEMM and X_DGEMM achieved comparable performance. Overall X_DGEMM performed better except when used in the BB and SUMMA_K algorithms. Comparing Figure 4.3 with Figure 4.1, we realized that for extremely non-square grid shapes, in which one of the dimensions is such as 16×1 or 1×16, X_DGEMM highly outperformed DGEMM. This is because the local matrices become more non-square as well, further taking advantage of the k-dimension in X_DGEMM. Figure 4.4 shows a situation where a different grid shape was used on the same matrix size, the fact that DGEMM performs better than X_DGEMM for square matrix shapes is thus affirmed. Figure 4.5 is a case that shows that mm5_col algorithm achieved best performance when a $P \leq Q$ grid shape was used, this is because this algorithm was particularly designed for such grid shapes whereas mm5_row is best for $P \geq Q$ as showed in Figure 4.4. Figure 4.6, shows that DGEMM and X_DGEMM can sometimes have same performance for square matrix grids. This is because the local matrices are less non-square. Figure 4.6 shows results from the SUMMA_K runs where different K-factors were used. For some algorithms, DGEMM performed better than X_DGEMM for fat-by-thin regions .

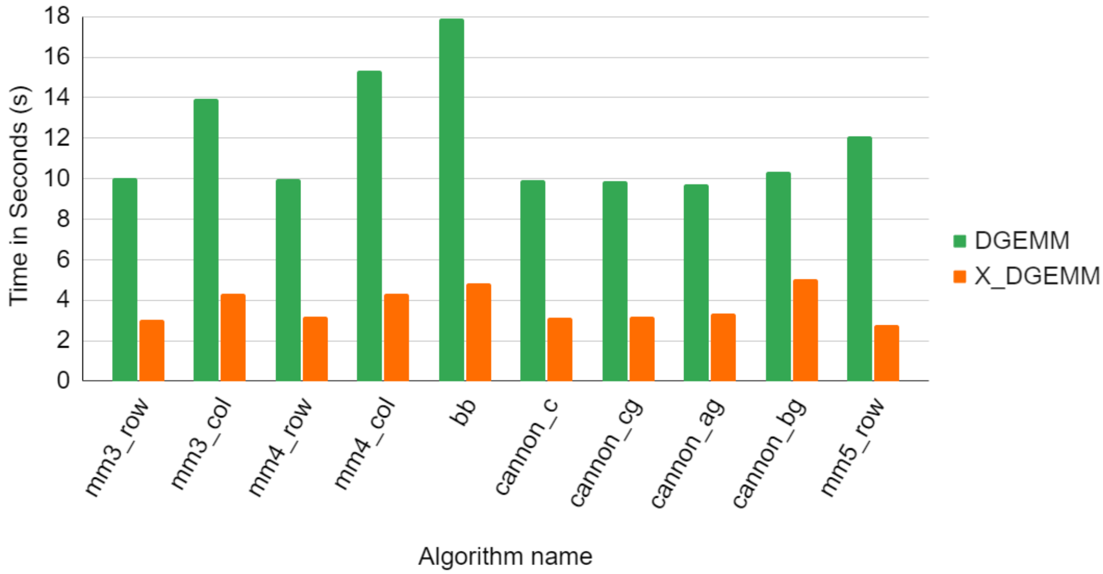

Figure 4.1 Average run time of DGEMM vs. X_DGEMM for Case 1. X_DGEMM performs better than DGEMM as expected for fat-by-thin cases. mm3_row achieved best performance for X_DGEMM while cannon_ag was best for DGEMM which shows polyalgorithmic behavior when different version of GEMM are used

We also confirmed that X_DGEMM algorithm performs significantly better than the BLIS DGEMM for fat-by-thin matrix multiplication. It puts all parallelization in the K dimension, so K must be large and M and N must be small. X_DGEMM also has polyalgorithmic characteristics because there is no time when a single algorithm is best for all cases. For square cases of matrix sizes, the BLIS DGEMM outperforms X_DGEMM.

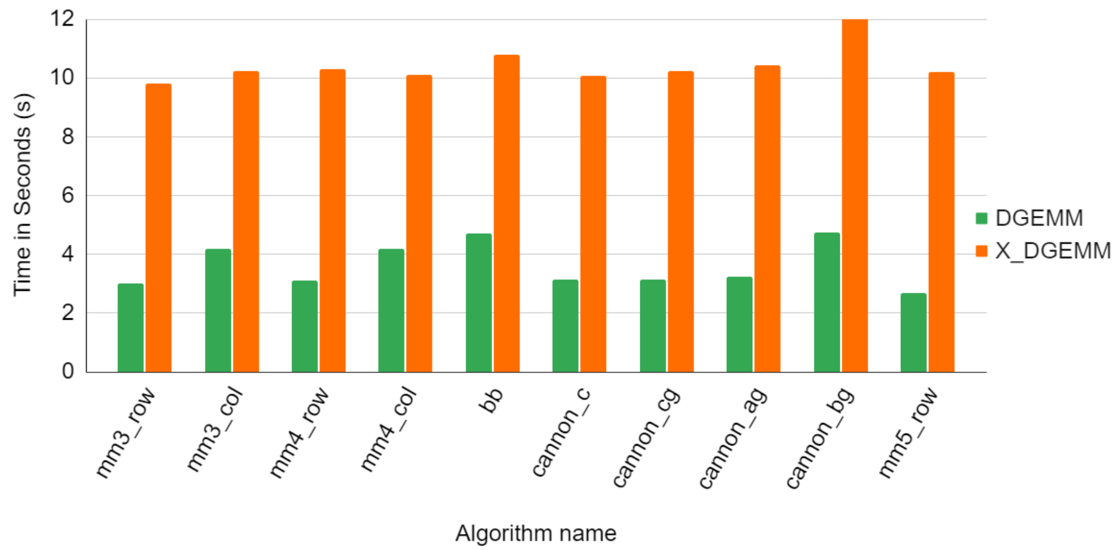

Figure 4.2 Average run time of DGEMM vs. X_DGEMM for Case 2. DGEMM performs better than X_DGEMM as expected for square cases

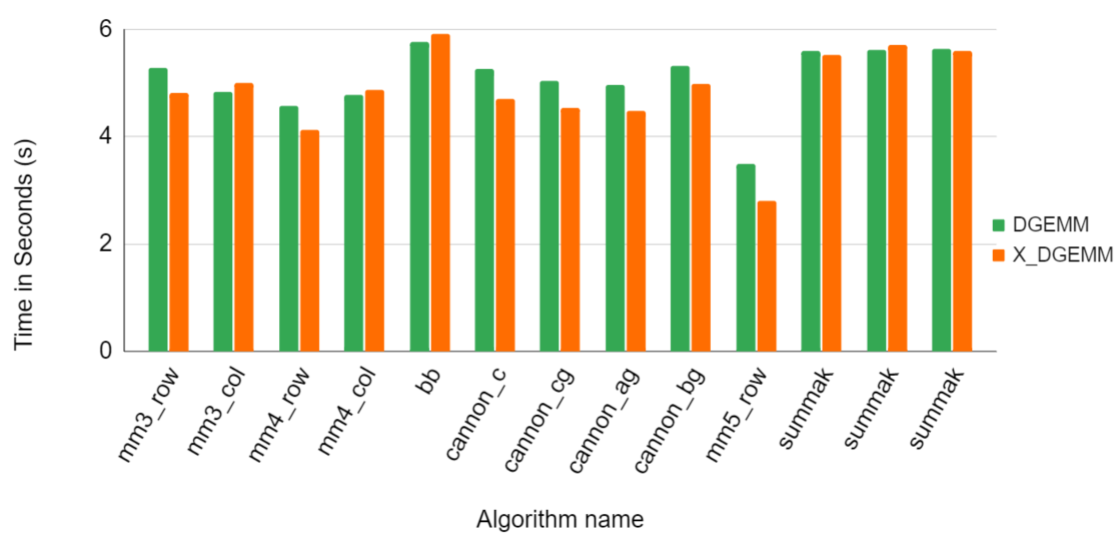

Figure 4.3 Average run time of DGEMM vs. X_DGEMM for Case 3. DGEMM and X_DGEMM achieved similar performance for some algorithms for some fat-by-thin cases on extremely non-square grid shapes

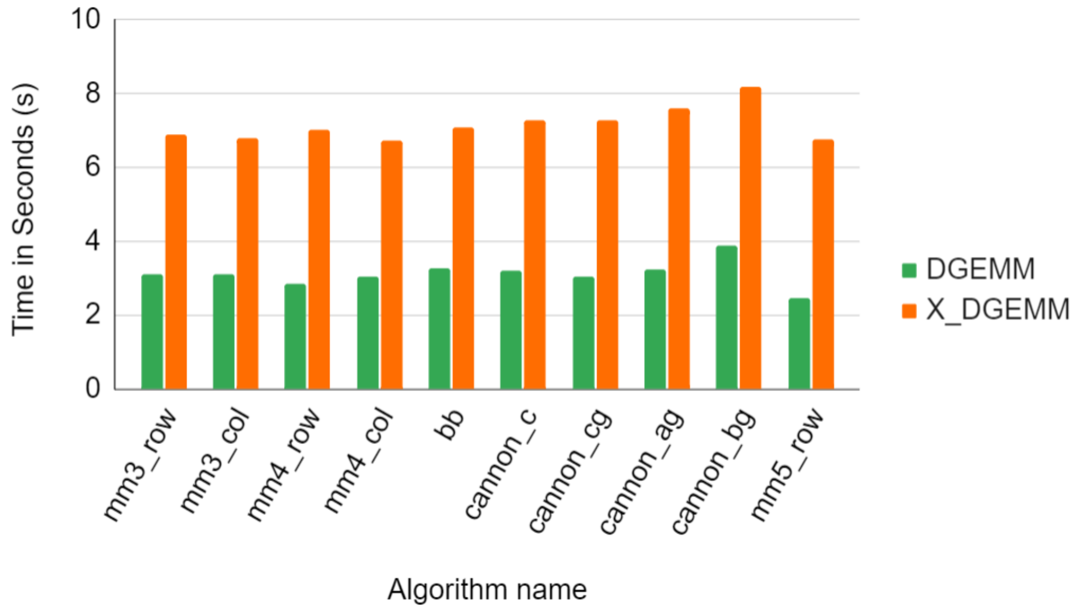

Figure 4.4 Average run time of DGEMM vs. X_DGEMM for Case 4. DGEMM performs better than X_DGEMM as expected for square cases

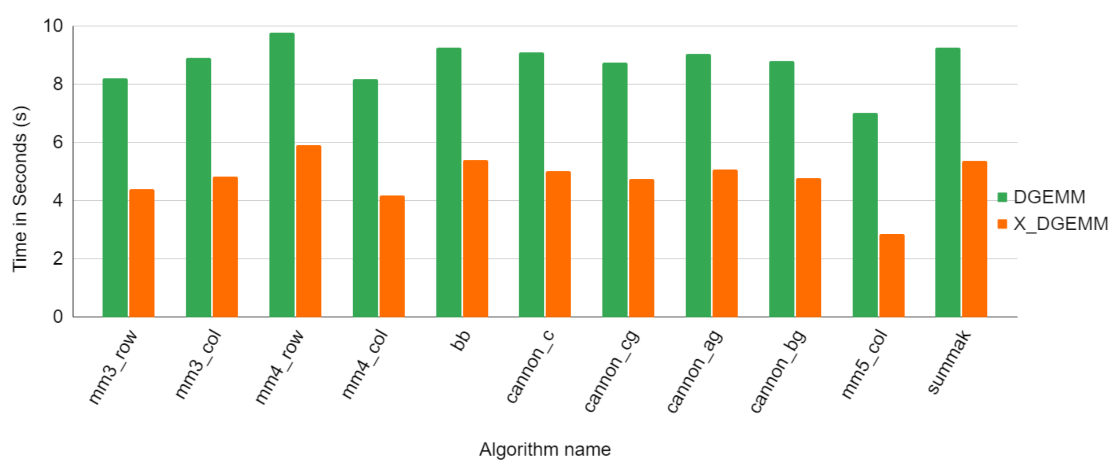

Figure 4.5 Average run time of DGEMM vs. X_DGEMM for Case 5. mm5_col achieved best performance for both DGEMM and X_DGEMM due to the $P \leq Q$ grid shape

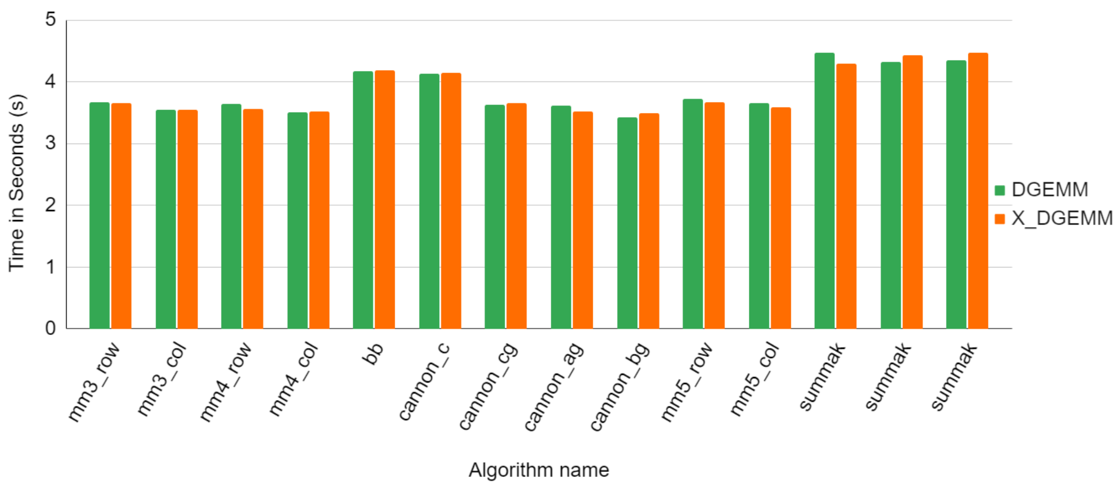

Figure 4.6 Average run time of DGEMM vs. X_DGEMM for Case 7. X_DGEMM and DGEMM had the same performance for most of the algorithms, evidently because of to the square grid shape, which leads to similar sizes of the local matrices involved in the multiplications

4.2 Running on Stampede2

Stampede2 results have quite similar trends of performance to the 117 cluster. The only difference is that Stampede2 is faster in terms of time that the algorithms take to run. This is because stampede2 has better processors (nodes) than 117. Stampede2 also has more cores per processor than 117: 1,736 SKX compute nodes each with 48 cores per node compared to 117 which has 33 nodes each with 28 cores per node. The Stampede 2 processor is an Intel Xeon Platinum 8160 ("Skylake") while 117 is equipped with Intel Xeon E5-2680 v4. Some of the advantages of the Intel Xeon Platinum 8160 over Intel Xeon E5-2680 v4 include faster core speed, higher RAM speed, more CPU cores[1], bigger L2 cache, more memory channels, faster bus transfer rate, and higher turbo clock speed [38]. All these are some of the factors that made the performance of the algorithms better on Stampede2. The specifications of these servers are detailed in Chapter 3. X_DGEMM and DGEMM versions have similar trends of performance such that the best algorithm is the same for both versions. Generally, the mm5 algorithm performed best for most cases. The Fox's (MM) group most often has the best performance throughout the experiments. Figure 4.7 and show the performance of DGEMM vs. X_DGEMM on a 1x16 grid shape. From the figures, we confirm that indeed X_DGEMM performs better for fat-by-thin cases than DGEMM. DGEMM performs better for square matrix shapes on Stampede2. This is evidence that the polyalgorithms achieve good performance on current systems architectures and technologies. The trend of the algorithms performance is similar on both 117 and Stampede2. DGEMM performs better than X_DGEMM as expected for square cases. Individual performance of the algorithms is significantly better on Stampede2 as compared to performance than 117. Compare Table A.1 and Table B.1 for the difference in performance of the two clusters.

4.3 BB vs. rank-k SUMMA vs. rank-1 SUMMA

In this research, we discovered that both BB and SUMMA_K algorithms use an outer product during the matrix multiplication. This is based on the format in which these two algorithms

[1] the DGEMM and X_DGEMM algorithms transparently access this multicore concurrency. Note that all the configurations done in this work use one MPI process per node.

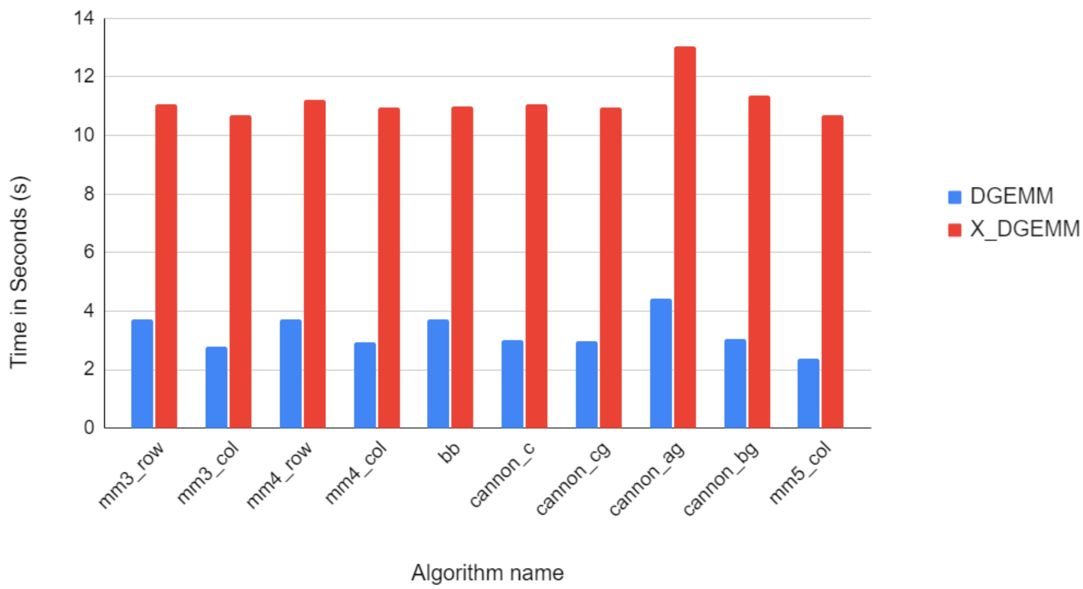

Figure 4.7 Average run time for DGEMM vs. X_DGEMM for Case 10 on Stampede2. DGEMM performs better than X_DGEMM as expected for square cases

broadcast the matrix elements. The difference between the two algorithms is that for SUMMA_K, we are able to choose the k factor. In this way, performance is improved by not maximizing the k value during matrix multiplication. Therefore, the SUMMA_K algorithm can have better performance than BB for particular cases of the k value. Another interesting observation is that the rank-k SUMMA_K algorithm also performs differently with different values of k. This is a polyalgorithmic behavior within the SUMMA_K. Therefore, we need to be careful while choosing the value of k in order to achieve maximum performance from this algorithm. The BB and SUMMA_K algorithms are similar and their performance was also similar from the experiments. The results for SUMMA_K were as expected to be faster than the rank-1 SUMMA algorithm. We carried out tests of the original SUMMA algorithm in the polymath library on 16 nodes: 8×2 and 4×4 grid shapes. We compared the results with the new rank-k-based SUMMA_K algorithm. There is a significant increase of speed in performance for the SUMMA_K algorithm as shown in Table 4.2. From Table 4.2, we noted the similarities and differences between the BB and SUMMA_K algorithms. For a fat-by-thin multiplication on an 8×2 grid, SUMMA_K performed better than BB with a k value of 62,500. While with the same matrix dimension, on a 4x4 grid, BB performed better than SUMMA_K. This shows how the k factor makes SUMMA_K polyalgorithmic for different matrix shapes and that when BB chooses the largest value of k, its performance may deteriorate. A small k gives poor performance for a large size matrix since it will be performing relatively inefficient GEMM-type operations. A large k also gives poor performance; this could be because SUMMA is communication overhead sensitive; more study may be warranted in future work. Table 4.3 shows a case where SUMMA_K achieved the best performance out of all the polyalgorithms on a 2×8 grid and $20,000 \times 20,000 \times 20,000$ matrix size using a k factor of 250. This further highlights the importance of using the appropriate k factor while using the SUMMA_K algorithm. The performance of SUMMA_K with other k factors 1250 and 625 is significantly slower than when $k = 250$ but competitive with the rest of the polyalgorithms.

Table 4.2 Comparison of BB, SUMMA_K and SUMMA algorithms performance (seconds). BB and SUMMA_K have competitive performance because they both perform rank-k updates. SUMMA is extremely slow compared to the SUMMA_K and BB because it performs rank-1 updates

P	**Q**	**M**	**K**	**N**	**k-factor-SUMMA_K**	**Agorithm name**	**avg max**	**avg min**
8	2	10^3	10^6	10^3		SUMMA	$43.39 \pm .05$	$43.39 \pm .05$
					62,500	SUMMA_K	$5.61 \pm .15$	$5.55 \pm .15$
						BB	$5.77 \pm .02$	$5.73 \pm .02$
4	4	10^3	10^6	10^3		SUMMA	$38.48 \pm .02$	$38.48 \pm .02$
					62,500	SUMMA_K	$4.47 \pm .11$	$4.43 \pm .11$
						BB	$4.16 \pm .05$	$4.08 \pm .00$

4.4 COSMA vs. Polyalgorithms

From our experiments, we validated that COSMA [25] is indeed an optimal algorithm for different matrix multiplication situations. In the experiments, we tested a square matrix shape on 16 nodes. Table 4.4 represents the performance of polyalgorithms and COSMA on a square grid

Table 4.3 Stampede2: DGEMM run time(seconds) $20,000 \times 20,000 \times 20,000$, on 2×8 grid. SUMMA_K achieved best performance when a k-factor of 250 was used. An appropriate selection of the k-factor can significantly improve the performance of SUMMA_K. (Bold numbers represent the lowest time(s) observed.)

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	2.178206	0.018087	2.160626	0.018022
mm3_col	2.177041	0.020087	2.110852	0.006659
mm4_row	2.149056	0.02052	2.128406	0.022695
mm4_col	2.049804	0.064312	1.799479	0.058719
bb	2.213366	0.014942	2.170275	0.017982
cannon_c	2.258626	0.03154	1.926514	0.030959
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.184387	0.041438	1.863977	0.040397
cannon_ag	2.975107	0.061201	2.660054	0.057914
cannon_bg	2.36045	0.044266	2.039952	0.042943
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	1.720906	0.100718	1.667862	0.084922
summak= 1,250	2.163689	0.037726	2.152244	0.027705
summak=625	2.374226	0.083487	2.363193	0.074538
summak=250	**1.653983**	0.059084	1.648238	0.056313

and square matrix shape. Ultimately, competitive performance between the polyalgorithms and COSMA is achieved. Algorithms from the Fox's group achieved the best performance; COSMA was better than some algorithms from the Cannon group. In Table 4.5, COSMA performed significantly faster than the polyalgorithms. This may result from the high level of optimizations that the COSMA algorithms does, meaning that it also works extremely well for non-square grid shapes; further study may be warranted. The good performance of COSMA for non-square grids is further presented in Table 4.6, where it still significantly outperforms the polyalgorithms. As the grid gets less non-square, as displayed in Table 4.7 and Table 4.8, COSMA's performance deteriorates since we force it to indirectly operate in 2D rather than its default 3D decomposition strategy. COSMA was still competitive with the polyalgorithms in Table 4.8 but slow for the case in Table 4.7. For these case the mm5_row performed best for the case where $P \geq Q$ while mm5_col was best for $P \leq Q$.

It is important to report that, 25 years later, the polyalgorithms set is still competitive with current algorithms such as COSMA [25]. COSMA can thus be considered polyalgorithmic as the rest of the algorithms in the polymath library. The set of runs in this section exhibited perfect polyalgorithmic behavior. The shapes we used are suboptimal given the fact that they do not explore the 3D dimensional capability of the COSMA algorithm. We determine the grid shape; that is, the 2D grid shape which is polymath based. In this case, the COSMA algorithm is forced to adopt a nearly 2D distribution, which will not fully take advantage of the 3D functionalities of COSMA. For a fixed number of processor $S = P \times Q \times R$, the third dimension R for COSMA is forced to be $R = 1$ in such cases. For a 16 node sweep, the divisions strategy could be $8 \times 2 \times 1$, $4 \times 2 \times 1$, or $1 \times 16 \times 1$. If COSMA's default is used where the algorithm implicitly chooses the grid shape, COSMA achieved best performance of approximately 1.08s. COSMA uses a division strategy depending on the number of assigned processors and in this way, chooses the best grid shape dimensions to perform the matrix multiplication. For the experiment used in our research on 16 nodes, the default division strategy was $2 \times 2 \times 4$ for $P \times Q \times R$ as represented as a 3D logical grid topology. The authors in of the COSMA paper [25] concluded that their algorithm is superior to other algorithms when it is possible to work on the problem in the optimal configuration and that they apparently discount as lower-order work the costs of pre- and post- reorganizations and replications of data needed to work in their optimal 3D layout of the matrices. These results offer proof that polyalgorithms are still important even when compared to recent, innovative algorithms such as COSMA.

Table 4.4 COSMA vs. polyalgorithms average run time in seconds for $M = K = N = 20,000$ matrix shape on a 4×4 process grid. In this table the Fox' group (mm3 and mm4 both row and columns versions) algorithms achieved the same and best performance considering the error bar. This is expected because for square grid there is no synchronous problem for mm3 and no initial slides for mm4 [26]. (Bold numbers represent the lowest time(s) observed.)

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	**1.597341**	0.024794	1.533709	0.019031
mm3_col	**1.563012**	0.036022	1.512657	0.023392
mm4_row	**1.58353**	0.040723	1.521707	0.028928
mm4_col	**1.545319**	0.022784	1.500303	0.01224
bb	1.891934	0.036561	1.813445	0.033924
cannon_c	1.733719	0.067762	1.491295	0.05684
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	1.784764	0.064621	1.567013	0.057309
cannon_ag	2.277589	0.099263	2.067456	0.105484
cannon_bg	2.270355	0.063714	2.064399	0.063468
summa	N/A	N/A	N/A	N/A
mm5_row	1.743822	0.051897	1.655398	0.022168
mm5_col	1.751378	0.034258	1.647067	0.035906
COSMA	1.8126	N/A	N/A	N/A

Table 4.5 COSMA vs. polyalgorithms average run time in seconds for $M = K = N = 20,000$ matrix shape on a 16×1 process grid. For extremely non-square shapes where only one dimension is indirectly used, COSMA is better than the polyalgorithms. (Bold numbers represent the lowest time(s) observed.)

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	2.585334	0.045147	2.451067	0.046238
mm3_col	3.463056	0.015206	3.444748	0.015015
mm4_row	2.688545	0.032001	2.462185	0.031552
mm4_col	3.455902	0.016743	3.439662	0.017589
bb	3.436845	0.009315	3.42145	0.008023
cannon_c	2.682052	0.045513	2.444983	0.029263
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.65708	0.030479	2.439806	0.028526
cannon_ag	2.793409	0.036987	2.576428	0.036979
cannon_bg	4.154851	0.10849	3.940715	0.108349
summa	N/A	N/A	N/A	N/A
mm5_row	2.292658	0.053389	2.216903	0.052696
mm5_col	N/A	N/A	N/A	N/A
COSMA	**1.2047**	N/A	N/A	N/A

Table 4.6 COSMA vs. polyalgorithms average run time in seconds for $M = K = N = 20,000$ matrix shape on a 1×16 process grid. For extremely non-square shapes where only one dimension is indirectly used, COSMA is better than the polyalgorithms. (Bold numbers represent the lowest time(s) observed.)

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.578719	0.029591	3.562803	0.030027
mm3_col	2.755198	0.038175	2.616753	0.046191
mm4_row	3.592684	0.030402	3.572687	0.027991
mm4_col	2.877741	0.082514	2.604477	0.043191
bb	3.556445	0.011625	3.542672	0.00963
cannon_c	2.867763	0.032626	2.644708	0.031319
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.843202	0.024139	2.624548	0.023815
cannon_ag	4.226014	0.118736	4.014871	0.118009
cannon_bg	2.902392	0.030229	2.68472	0.028154
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	2.332504	0.066513	2.266316	0.067785
COSMA	**1.4025**	N/A	N/A	N/A

Table 4.7 COSMA vs. polyalgorithms average run time in seconds for $M = K = N = 20,000$ matrix shape on a 2×8 process grid. mm5_col achieved best performance because of the $P \leq Q$ grid shape. (Bold numbers represent the lowest time(s) observed.)

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	2.111144	0.030962	2.094617	0.028245
mm3_col	2.118654	0.020739	2.053215	0.008508
mm4_row	2.07584	0.013178	2.057578	0.010599
mm4_col	2.043327	0.067366	1.816637	0.066054
bb	2.170646	0.017821	2.144723	0.019418
cannon_c	2.211638	0.045994	1.857015	0.044505
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.161477	0.041353	1.848058	0.041646
cannon_ag	3.044329	0.081283	2.71306	0.068561
cannon_bg	2.363414	0.030135	2.044465	0.02871
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**1.773298**	0.03927	1.698227	0.038807
COSMA	3.0456	N/A	N/A	N/A

Table 4.8 COSMA vs. polyalgorithms average run time in seconds for $M = K = N = 20,000$ matrix shape on a 8×2 process grid. mm5_row achieved best performance because of the $P \geq Q$ grid shape. (Bold numbers represent the lowest time(s) observed.)

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	2.005457	0.029538	1.945231	0.025401
mm3_col	1.971702	0.022117	1.949778	0.006275
mm4_row	1.899726	0.040621	1.657638	0.036981
mm4_col	1.972492	0.018618	1.954382	0.016725
bb	2.036383	0.008788	2.017463	0.008488
cannon_c	2.023054	0.037908	1.712544	0.034213
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.028953	0.041459	1.717536	0.043512
cannon_ag	2.239083	0.045515	1.924992	0.046112
cannon_bg	2.805638	0.083675	2.492211	0.08282
summa	N/A	N/A	N/A	N/A
mm5_row	**1.593167**	0.051841	1.543465	0.051492
mm5_col	N/A	N/A	N/A	N/A
COSMA	1.8394	N/A	N/A	N/A

4.5 Comparisons

We made these comparisons by testing the matrix multiplication algorithms with different matrix sizes and grid shapes:

- For square matrix shapes, DGEMM had best performance as compared to X_DGEMM for tested the different grid shapes. We did a $20,000 \times 20,000 \times 20,000$ matrix size on 16 nodes and a significant difference in performance was noticed. This is because the DGEMM operations efficiently works for square and nearly square matrices. It was also noted that performance of algorithms on the square grids is better than on non-square grids. This results from the non-square shapes of the local matrices that become more non-square as the grid becomes extremely non-square.

- Fox's group (MM algorithms) are the best for square matrix shapes on non-square grids. For DGEMM, mm5 performs best for all grid shapes (mm5_row when $P > Q$ and mm5_col when $P < Q$). For the 4×4 square grid mm4_col performs best.

- Runs with X_DGEMM as the local BLAS operation shows more polyalgorithmic behavior than with DGEMM (this is unsurprising because X_DGEMM is only beneficial when most local subproblems are non-square within the totality of the matrix operations done in the process grid over the course of a single parallel multiplication). Different MM algorithms exhibit their best performance for different grid shapes and the Cannon group has competitive performance with the MM group considering the error bar for (4×4 and 1×16 grid) cases.

- Runs with X_DGEMM as the local BLAS operation performs better than DGEMM for fat-by-thin cases except for square matrices one square grids.

- Stampede2 generally performs better than the 117 cluster because it uses more advanced technologies in terms of CPU, memory, cache, etc.

- The performance trend on Stampede2 is similar to 117; the MM group is the best, and Cannon is not competitive in any case. Further, we noted that DGEMM and X_DGEMM-enabled runs yield the same algorithm performing best for most grid shapes (that is, mm5) except for 4×4, where mm3_row is the best. 16×1 is the only case that where the best algorithm varies for the two DGEMM options.

- COSMA vs. polyalgorithm results on stampede2 show that COSMA is not always faster than polyalgorithms for square matrices on 2×8, 8×2 and 4×4, 1×16, and 16×1 grids.

- SUMMA_K is competitive with other algorithms and has comparable performance to the BB algorithm. It also achieved best performance for particular cases when the appropriate K value is used, an example is shown in Table 4.3 Interesting, BB was among the best algorithms for several scenarios in [26].

4.6 Summary

The experimental results from our research were analyzed and discussion was presented regarding why different algorithms performed differently in certain situations. Generally, the mm5 algorithm performed best for non-square grid shapes. Other algorithms were competitive with each other in their respective groups. The BB algorithm and the SUMMA_K algorithm have similar performance but the latter allows the user to choose the K-factor and this can be optimized and so sometimes is better than the BB algorithm, which always maximizes the k value. The results of our experiments compare the performance of polyalgorithms with COSMA showed that the poly algorithms are still competitive, except if COSMA's optimal 3D layouts are supported in terms of matrix redistribution and memory replication. That is, we had better performance when we predefined the grid, but when we allowed the new algorithm COSMA to maximize its default setting, it had better performance[2].

[2]Further study is needed for operational sequences to determine if the cost of moving between optimal and near optimal COSMA layouts and application-relevant layouts will yield the minimum time to solution. A operational sequence computes A and B, redistributes A and B into the COSMA 3D grid format, computes with COSMA with the lowest runtime, then redistributes C into the application-relevant layout. If redistirbution costs can be overlooked, then COSMA will be the algorithm of choice for that scenario. The ability to do some partial matrix multiplication work

The new algorithm, X_DGEMM, yielded better parallel run times for most fat-by-thin cases as expected, whereas DGEMM did well for the square matrix shapes. The trends of performance of these two algorithms on the both clusters used was similar, but Stampede2 has faster performance, as expected based on its newer and higher-end node architecture.

while redistributing into the COSMA format could further reduce the overheads noted of such reorganizations. All these matters are left for future work.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This chapter concludes the research presented in this thesis. The new findings are affirmed and summarized, thereby presenting the usefulness of polyalgorithms. Future work and areas that need further investigation are also suggested.

5.1 Summary

We suumarized the work done in this thesis by answering the following research question:

- **Do we reproduce the trade-offs between using different algorithms in different use cases, or is one always best?**

  Polyalgorithms are still a core in building linear algebra libraries; we do not choose to design a library with just one algorithm[1]. This fact is reinforced by the results presented with the diversity of algorithmic performance. However, for this particular study, we mostly used non-square grid shapes and fat-by-thin matrix shapes. For this reason, one specific algorithm, mm5, showed best performance for most of the cases presented. The mm5 algorithm was specifically designed to deal with cases where $P < Q$ or $P > Q$. The conclusions from Li et al. [26] remain valid for square matrix grid shapes and no single algorithm performs best for all cases of arbitrary matrix shapes and grid sizes. We also explored the relative impacts of two systems of different performance by performing the same set of experiments on both. Some of the differences of the two clusters are the CPU speed, RAM bandwidth, CPU threads, cache size, size of memory channels, bus transfer rate, and turbo clock speed. It is important

[1] We compared with COSMA quite a bit in this thesis. Generally speaking, COSMA should be one of several options in a next-generation polyalgorithms library, not be considered as a sole alternative.

to note the system specifications when comparing performance of various algorithms across the experiments presented here.

- **Does the using of the basic BLIS DGEMM vs. new BLIS version, X_DGEMM, change performance of the algorithm(s) in a significant way? Which algorithm is best for a given "parameter sweep"?**

  As one might expect, using different versions of DGEMM may significantly change the performance of all the parallel algorithms. The original BLIS DGEMM version is best for square matrix shapes while the new X_DGEMM version is best for fat-by-thin matrix shapes. It is therefore important to choose the right DGEMM version in order to achieve the best performance for arbitrary grid shapes and matrix sizes. The DGEMM versions in this research are polyalgorithmic themselves since they achieve distinct levels of performance for different grid shapes and matrix sizes. None of the parallel algorithms had the best performance based on the "sweep"; best performance mostly relied on the grid shapes as explained earlier in the thesis.

- **Does the new SUMMA algorithm outperform any of the older algorithms?**

  The SUMMA_K algorithm performed better than the rank-1 SUMMA algorithm and its primary advantage is that we can choose the k-factor that will achieve best performance for a particular situation rather than maximizing K which could decrease performance. SUMMA_K also outperformed some of the old algorithms, especially algorithms in the Cannon group. It achieved best performance for a case presented in Table 4.3. SUMMA_K performance was particularly similar to the BB algorithm for most cases where an appropriate k-value was used. We noted that BB is a limiting case of the rank-k SUMMA algorithm.

- **Does the polyalgorithms set still achieve competitive performance when compared to newer algorithms on current technologies?**

  The polyalgorithms that were designed in 1991–95 and describewd in Li et al. [26] are still competitive with newer algorithms and achieve good performance on current systems architectures and technologies. This is a strong indication that polyalgorithms are still practical and important for maximizing performance of parallel dense matrix-matrix multiplications. The COSMA algorithm that was compared to the polyalgorithms in this work achieved best performance when optimal 3D decomposition are used. COSMA is also found to be competitive for non-optimal 2D decompositions and this validates the research in the COSMA paper [25]. In a future polyalgorithms library, we would include both COSMA and 2.5/3D decompositions of the algorithms presently offered in 2D form in the polyalgorithms library, but with the added guarantee of the DDI behavior described in this thesis and reported previously in Jin et al.'s paper.

5.2 Future Work

In addition to those ideas and extensions already mentioned, in the future, we hope to use machine learning to select the best algorithm for given situations automatically, including both the algorithmic cost, and the cost of moving the data to/from its application-relevant organization as needed by steps before and after the matrix multiplication (the cost of redistribution is only lower order work asymptotically, and can matter). Another step would be to reformulate the library in C++, taking advantage of modern C++ for flexibility, compile-time optimizations (including potential for algorithmic selection), and supporting data-distribution and layout decisions at compile-time where possible. Further, we can envisage using a parallel backend to a deep-learning algorithm (that uses parallel dense matrix-matrix multiplication) that provides the optimization parameters for its own deep learning use cases, creating a closed-loop deep-learning that solves application problems while tuning its own runtime performance. As of now, all matrices are handled as they are allocated by the test program or application; we would prefer to be able to reorganize (remap) them also, when appropriate, especially since we will also be including the COSMA algorithm moving forward. We would also like to determine the actual cost of redistribution and have a high quality matrix (data distribution and grid shape) re-organizer for the library.

REFERENCES

[1] Agarwal, R. C., Balle, S. M., Gustavson, F. G., Joshi, M., and Palkar, P. (1995). A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582. 17

[2] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., et al. (1999). *LAPACK Users' guide*. SIAM. 2

[3] Braun, T., Siegel, H., and Maciejewski, A. (2001). Heterogeneous computing: Goals, methods, and open problems. pages 307–320. 14

[4] Cannon, L. E. (1969). *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University-Bozeman, College of Engineering. AAI7010025. 1, 17

[5] Chen, J., Xiong, N., Liang, X., Tao, D., Li, S., Ouyang, K., Zhao, K., DeBardeleben, N., Guan, Q., and Chen, Z. (2019). Tsm2: Optimizing tall-and-skinny matrix-matrix multiplication on gpus. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 106–116, New York, NY, USA. Association for Computing Machinery. 15

[6] Choi, J., Walker, D. W., and Dongarra, J. J. (1994). Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570. 1, 17, 21

[7] Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O., and Spillinger, O. (2013). Communication-optimal parallel recursive rectangular matrix multiplication. *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 261–272. 1, 6, 18

[8] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. (1990). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17. 2

[9] Forum, M. P. I. (2015). *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart. 2, 4

[10] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D., and White, R. L. (1989). Solving problems on concurrent processors vol. 1: General techniques and regular problems. *Computers in Physics*, 3(1):83–84. 1, 17

[11] Goto, K. (2005). Tacc® gotoblas library. https://www.tacc.utexas.edu/systems/software. Accessed: 09.16.2020. 3

[12] Goto, K. and Geijn, R. A. v. d. (2008). Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3). 3

[13] Gunnels, J., Lin, C., Morrow, G., and van de Geijn, R. (1998). A flexible class of parallel matrix multiplication algorithms. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 110–116. 4

[14] Gupta, A. and Kumar, V. (1993). Scalability of parallel algorithms for matrix multiplication. In *1993 International Conference on Parallel Processing - ICPP'93*, volume 3, pages 115–123. 18

[15] Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533. 6

[16] Hedtke, I. (2010). Strassen's matrix multiplication algorithm for matrices of arbitrary order. *ArXiv*, abs/1007.2117. 17

[17] Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38. 6

[18] Hossain, M. M., Hines, T. M., Ghafoor, S. K., Islam, S. R., Kannan, R., and Sukumar, S. R. (Dec 2018). A flexible-blocking based approach for performance tuning of matrix multiplication routines for large matrices with edge cases. *2018 IEEE International Conference on Big Data (Big Data)*. xii, 4, 6, 14, 24, 26, 27

[19] Hossain, M. M., Hines, T. M., Ghafoor, S. K., Rabiul Islam, S., Kannan, R., and Sukumar, S. R. (2018). A flexible-blocking based approach for performance tuning of matrix multiplication routines for large matrices with edge cases. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3853–3862. 25

[20] Intel (2003). Intel® math kernel library. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. Accessed: 08.18.2020. 2, 6

[21] Intel (2020). Multiplying matrices using dgemm. https://software.intel.com/content/www/us/en/develop/documentation/mkl-tutorial-c/top/multiplying-matrices-using-dgemm.html. Accessed: 09.16.2020. 3

[22] Jaeyoung Choi and Dongarra, J. J. (1995). Scalable linear algebra software libraries for distributed memory concurrent computers. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 170–177. 18, 21

[23] Jeddi, S. and Nasri, W. (2005). A poly-algorithm for efficient parallel matrix multiplication on metacomputing platforms. In *2005 IEEE International Conference on Cluster Computing*, pages 1–9. IEEE. 5, 14, 16

[24] Jia-Wei, H. and Kung, H.-T. (1981). I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. 18

[25] Kwasniewski, G., Kabić, M., Besta, M., VandeVondele, J., Solcà, R., and Hoefler, T. (2019). Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and*

Analysis, SC '19, New York, NY, USA. Association for Computing Machinery. 1, 2, 6, 8, 18, 41, 43, 54

[26] Li, J., Skjellum, A., and Falgout, R. D. (1995). A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. x, 3, 5, 11, 13, 16, 17, 24, 25, 32, 44, 50, 52, 53

[27] Marker, B., Van Zee, F. G., Goto, K., Quintana-Ortí, G., and van de Geijn, R. A. (2007). Toward scalable matrix multiply on multithreaded architectures. In Kermarrec, A.-M., Bougé, L., and Priol, T., editors, *Euro-Par 2007 Parallel Processing*, pages 748–757, Berlin, Heidelberg. Springer Berlin Heidelberg. 18

[28] Mathur, K. K. and Johnsson, S. L. (1994). Multiplication of matrices of arbitrary shape on a data parallel computer. *Parallel Computing*, 20:919951. 1, 17

[29] Nasri, W. and Trystram, D. (2004). A polyalgorithmic approach applied for fast matrix multiplication on clusters. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 234. IEEE. 16

[30] Rivera, C., Chen, J., Xiong, N., Song, S., and Tao, D. (2020). Ism2: Optimizing irregular-shaped matrix-matrix multiplication on gpus. *ArXiv*, abs/2002.03258. 15

[31] Solomonik, E. and Demmel, J. (2011). Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, page 90–109, Berlin, Heidelberg. Springer-Verlag. 1, 17

[32] Solomonik, E., Matthews, D., Hammond, J., and Demmel, J. (2013). Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 813–824. 18

[33] Stanzione, D., Barth, B., Gaffney, N., Gaither, K., Hempel, C., Minyard, T., Mehringer, S., Wernert, E., Tufo, H., Panda, D., and Teller, P. (2017). Stampede 2: The evolution of an xsede supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, New York, NY, USA. Association for Computing Machinery. 23

[34] Strassen, V. (1969). Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356. 17

[35] van de Geijn, R. A. (1997). *Using PLAPACK: Parallel Linear Algebra Package*. MIT Press, Cambridge, MA, USA. 15, 17

[36] Van De Geijn, R. A. and Watts, J. (1997). Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274. 1, 4, 15, 16, 17

[37] Van Zee, F. G. and van de Geijn, R. A. (2015). BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33. 2, 4, 6, 26

[38] Versus (2020). Intel xeon e5-2690 v4 vs intel xeon platinum 8160. https://versus.com/en/intel-xeon-e5-2690-v4-vs-intel-xeon-platinum-8160. Accessed: 09.16.2020. 39

[39] Wang, Q., Zhang, X., Zhang, Y., and Yi, Q. (2013). Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE. 2, 6

[40] Xianyi, Z., Qian, W., and Yunquan, Z. (2012). Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691. IEEE. 2

APPENDIX A

RAW DATA FROM 117 CLUSTER RUNS

In this section, we have included all the data from the run that were done. We made the runs on two servers: 117 and Stampede2. The graphs include all results for the different nominal nodes using both the DGEMM and X_DGEMM algorithms. Performance is measured as run-time in seconds. The first set of tables are for runs made of 117, followed by tables of runs on Stampede2.

Table A.1 117 cluster: DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 16×1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.004114	0.089586	2.852661	0.091707
mm3_col	4.174975	0.34167	4.150421	0.323331
mm4_row	3.102586	0.133188	2.863519	0.123914
mm4_col	4.179767	0.279268	4.157584	0.281229
bb	4.719917	0.038954	4.690012	0.040476
cannon_c	3.138984	0.093306	2.864152	0.112676
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.145616	0.132011	2.905035	0.132137
cannon_ag	3.245361	0.102395	3.005661	0.102403
cannon_bg	4.756179	0.107676	4.51,2508	0.107822
summa	N/A	N/A	N/A	N/A
mm5_row	2.684945	0.126424	2.627342	0.094087
mm5_col	N/A	N/A	N/A	N/A

Table A.2 117 cluster: X_DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 16×1 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	9.807986	0.091137	9.339222	0.115536
mm3_col	10.234802	0.297784	10.123398	0.298362
mm4_row	10.292248	0.07337	9.715777	0.096508
mm4_col	10.117461	0.106569	9.995225	0.105703
bb	10.792354	10.792354	10.666155	0.078611
cannon_c	10.083978	0.116911	9.533392	0.15164
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	10.223631	0.149714	9.983826	0.149744
cannon_ag	10.42875	0.103419	10.205293	0.101851
cannon_bg	12.062024	0.066946	11.826327	0.066841
summa	N/A	N/A	N/A	N/A
mm5_row	10.220091	0.043336	9.956149	0.021391
mm5_col	N/A	N/A	N/A	N/A

Table A.3 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 16×1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	10.057969	0.227202	9.74641	0.183809
mm3_col	13.925253	1.262522	13.869742	1.25967
mm4_row	10.004802	0.302435	9.370544	0.273511
mm4_col	15.352479	0.70173	15.296557	0.701991
bb	17.914256	0.008055	17.870258	0.006057
cannon_c	9.913032	0.206929	9.221466	0.223969
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	9.892918	0.170405	9.274711	0.170651
cannon_ag	9.728389	0.205295	9.096664	0.205768
cannon_bg	10.358781			
summa	N/A	N/A	N/A	N/A
mm5_row	12.104181	0.186454	10.073063	0.247597
mm5_col	N/A	N/A	N/A	N/A

Table A.4 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 16×1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.026592	0.16276	2.896384	0.16687
mm3_col	4.333003	0.394342	4.304782	0.393918
mm4_row	3.190416	0.11956	2.917259	0.088837
mm4_col	4.304849	0.274472	4.260647	0.276986
bb	4.826139	0.099824	4.80445	0.105569
cannon_c	3.158663	0.119964	2.903084	0.119718
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.203345	0.220134	2.961556	0.220144
cannon_ag	3.328331	0.22169	3.084653	0.221633
cannon_bg	5.016202	0.156771	4.762207	0.156939
summa	N/A	N/A	N/A	N/A
mm5_row	2.794082	0.160895	2.668053	0.093165
mm5_col	N/A	N/A	N/A	N/A

Table A.5 117 cluster: DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 8×2 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.104391	0.123473	3.004158	0.082401
mm3_col	3.105573	0.068623	3.054157	0.073608
mm4_row	2.843934	0.120926	2.466518	0.092181
mm4_col	3.05586	0.06514	2.988244	0.065338
bb	3.2842	0.063683	3.204386	0.020394
cannon_c	3.197274	0.143192	2.672129	0.118622
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.030174	0.153434	2.592266	0.153556
cannon_ag	3.237907	0.078029	2.799402	0.078119
cannon_bg	3.879122	0.116756	3.498527	0.116427
summa	N/A	N/A	N/A	N/A
mm5_row	2.456145	0.125266	2.363079	0.095889
mm5_col	N/A	N/A	N/A	N/A

Table A.6 117 cluster: X_DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 8×2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	6.880494	0.066767	6.572668	0.048774
mm3_col	6.798455	0.056455	6.66033	0.069793
mm4_row	7.038601	0.022697	6.610789	0.038731
mm4_col	6.737182	0.042547	6.622553	0.059701
bb	7.089602	0.021556	6.923108	0.027314
cannon_c	7.295701	0.105666	6.790563	0.065147
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	7.295074	0.138681	6.853461	0.138832
cannon_ag	7.608736	0.049439	7.15834	0.048941
cannon_bg	8.193423	0.051755	7.749879	0.051545
summa	N/A	N/A	N/A	N/A
mm5_row	6.775231	0.023891	6.66992	0.035763
mm5_col	N/A	N/A	N/A	N/A

Table A.7 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 8×2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	5.289764	0.094543	5.17076	0.082301
mm3_col	4.843809	0.28812	4.728267	0.264463
mm4_row	4.572714	0.069358	3.759876	0.125709
mm4_col	4.771813	0.162862	4.664777	0.115745
bb	5.773333	0.024624	5.729975	0.011002
cannon_c	5.267049	0.128561	4.152796	0.133519
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.037954	0.147177	3.950869	0.141284
cannon_ag	4.961636	0.094926	3.875407	0.096832
cannon_bg	5.316073	0.113255	4.208152	0.111541
summa	43.393309	0.051926	43.393223	0.051949
mm5_row	3.493115	0.111571	3.308793	0.083013
mm5_col	N/A	N/A	N/A	N/A
summak= 62,500	5.60572	0.149145	5.550417	0.145323
summak=31,250	5.625967	0.037938	5.563215	0.03324
summak=15,625	5.635896	0.170954	5.574486	0.176853

Table A.8 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 8×2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	4.827699	0.059626	4.780027	0.068865
mm3_col	4.997269	0.287585	4.971626	0.291044
mm4_row	4.125989	0.039004	3.228847	0.082819
mm4_col	4.872022	0.041889	4.859593	0.041686
bb	5.912459	0.01025	5.507978	0.010381
cannon_c	4.708409	0.128773	3.744426	0.087419
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.535183	0.035756	3.440054	0.036128
cannon_ag	4.490902	0.013836	3.437396	0.011911
cannon_bg	4.9859	0.075267	3.889889	0.061019
summa	N/A	N/A	N/A	N/A
mm5_row	2.806208	0.184184	2.691557	0.133161
mm5_col	N/A	N/A	N/A	N/A
summak= 62,500	5.530408	0.034883	5.474745	0.035216
summak=31,250	5.719671	0.193283	5.658157	0.193626
summak=15,625	5.593211	0.118783	5.529109	0.107637

Table A.9 117 cluster: DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 4×4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	2.554772	0.067991	2.440048	0.080905
mm3_col	2.586245	0.131499	2.485543	0.118272
mm4_row	2.533918	0.112229	2.417407	0.095751
mm4_col	2.538205	0.11661	2.432643	0.080889
bb	2.743331	0.107562	2.675613	0.087649
cannon_c	2.824683	0.187615	2.492293	0.154594
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.523293	0.179543	2.308209	0.179549
cannon_ag	2.83235	0.109847	2.617463	0.10925
cannon_bg	3.076794	0.129853	2.751346	0.129767
summa	N/A	N/A	N/A	N/A
mm5_row	2.600132	0.104624	2.465986	0.087082
mm5_col	2.610751	0.148734	2.497223	0.101187

Table A.10 117 cluster: X_DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 4×4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.905616	0.107547	3.774039	0.087028
mm3_col	4.001572	0.038902	3.898343	0.03627
mm4_row	3.983841	0.017431	3.886098	0.009403
mm4_col	4.013215	0.018047	3.869312	0.010206
bb	4.239633	0.014753	4.071826	0.016979
cannon_c	4.435158	0.010509	4.068065	0.009844
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.079392	0.016737	3.858012	0.016998
cannon_ag	4.403749	0.022573	4.175929	0.022419
cannon_bg	4.514618	0.024273	4.194499	0.023726
summa	N/A	N/A	N/A	N/A
mm5_row	4.070637	0.026807	3.985425	0.017176
mm5_col	4.060671	0.025365	3.98565	0.018975
summak= 62,500	N/A	N/A	N/A	N/A
summak=31,250	N/A	N/A	N/A	N/A
summak=15,625	N/A	N/A	N/A	N/A

Table A.11 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 4×4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.672814	0.224535	3.560813	0.215449
mm3_col	3.544421	0.150348	3.43488	111300
mm4_row	3.640113	0.171183	3.47388	0.097796
mm4_col	3.506616	0.139265	3.381051	0.115133
bb	4.175733	0.045755	4.084926	0.003978
cannon_c	4.133527	0.045718	3.45612	0.086638
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.626133	0.101716	2.962833	0.1,00094
cannon_ag	3.613001	0.153531	3.054591	0.126627
cannon_bg	**3.421078**	0.098859	2.889669	0.095378
summa	38.477298	0.024092	38.477116	0.024091
mm5_row	3.728349	0.130747	3.590537	0.06835
mm5_col	3.660328	0.158836	3.477348	0.107694
summak= 62,500	4.46597	0.112223	4.429396	0.112805
summak=31,250	4.325335	0.124371	4.29311	0.119803
summak=15,625	4.348232	0.183979	4.306223	0.189218

Table A.12 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 4×4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.651921	0.111527	3.529917	0.105842
mm3_col	3.552332	0.075631	3.442905	0.084738
mm4_row	3.559412	0.062854	3.454921	0.061221
mm4_col	3.526929	0.094232	3.437655	0.073984
bb	4.185282	0.064527	4.131923	0.033066
cannon_c	4.146038	0.159305	3.471973	0.183963
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.65848	0.106228	3.004872	0.105518
cannon_ag	**3.52477**	0.107053	2.970659	0.09432
cannon_bg	3.494098	0.116045	2.969172	0.115908
summa	N/A	N/A	N/A	N/A
mm5_row	3.667567	0.144066	3.50564	0.084442
mm5_col	3.589096	0.094507	3.444496	0.050592
summak= 62,500	4.29752	0.109586	4.262886	0.108901
summak=31,250	4.425282	0.190869	4.401204	0.190316
summak=15,625	4.472903	0.179782	4.434156	0.179585

Table A.13 117 cluster: DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 2×8 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.050797	0.105518	2.95816	0.087967
mm3_col	3.152321	0.074975	3.226733	0.051303
mm4_row	3.324212	0.074975	3.226733	0.051303
mm4_col	2.941664	0.187384	2.60006	0.093016
bb	3.06895	0.07682	2.955071	0.028471
cannon_c	3.268867	0.120419	2.795767	0.101205
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.15243	0.129845	2.682396	0.129126
cannon_ag	4.030698	0.09531	3.499185	0.094703
cannon_bg	3.289265	0.112578	2.819897	0.113076
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	2.535425	0.151602	2.434492	0.114446
summak= 62,500	N/A	N/A	N/A	N/A
summak=31,250	N/A	N/A	N/A	N/A
summak=15,625	N/A	N/A	N/A	N/A

Table A.14 117 cluster: X_DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 2×8 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	7.679111	0.223376	7.578926	0.223872
mm3_col	7.947279	0.072991	7.83734	0.06894
mm4_row	8.13212	0.020043	8.04218	0.02286
mm4_col	7.853115	0.027801	7.521231	0.03284
bb	7.878829	0.007786	7.759382	0.009535
cannon_c	8.372337	0.203652	7.892582	0.191184
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	8.40751	0.017652	8.019497	0.017506
cannon_ag	9.031702	0.037362	8.635826	0.038829
cannon_bg	8.588519	0.040684	8.127759	0.039795
summa	N/A	N/A	N/A	N/A
mm5_row	0.000002	0.000003	0.000001	0.000002
mm5_col	7.703817	0.029472	7.411596	0.025598

Table A.15 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 2×8 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	8.195272	0.181465	8.127514	0.187312
mm3_col	8.89789	0.049263	8.778078	0.052295
mm4_row	9.757232	0.068386	9.633832	0.03858
mm4_col	8.177215	0.165181	7.478362	0.174695
bb	9.256326	0.006383	9.230341	0.003973
cannon_c	9.098165	0.132656	7.764621	0.149583
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	8.753082	0.1102	7.625635	0.110741
cannon_ag	9.043917	0.100804	7.874272	0.075587
cannon_bg	8.786284	0.047815	7.665578	0.047224
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**7.001529**	0.135105	6.887903	0.094991
summak= 62,500	9.25885	0.028569	9.175043	0.028565
summak=31,250	9.366887	0.034752	9.279534	0.034711
summak=15,625	9.146521	0.030841	9.062226	0.030768

Table A.16 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 2×8 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.385591	0.141031	4.306348	0.160715
mm3_col	4.823953	0.048917	4.783931	0.049543
mm4_row	5.898224	0.087832	5.838839	0.109524
mm4_col	4.183099	0.05283	3.475151	0.09389
bb	5.386932	0.007	5.37966	0.006266
cannon_c	5.012098	0.125997	3.791895	0.128165
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.734167	0.071588	3.573558	0.071842
cannon_ag	5.071166	0.07473	3.940617	0.070804
cannon_bg	4.766935	0.053921	3.605236	0.054509
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.860369**	0.19526	2.763426	0.145105
summak= 62,500	5.375917	0.140651	5.303192	0.144338
summak=31,250	N/A	N/A	N/A	N/A
summak=15,625	N/A	N/A	N/A	N/A

Table A.17 117 cluster: DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 1×16 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.24695	0.036424	4.230039	0.035559
mm3_col	3.065601	0.049501	2.96687	0.035908
mm4_row	4.906686	0.117064	4.88042	0.113211
mm4_col	3.243534	0.049167	2.973007	0.04682
bb	4.800987	0.018126	4.785862	0.015149
cannon_c	3.261526	0.046843	2.973649	0.040279
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.304488	0.04595	3.047362	0.045891
cannon_ag	4.984504	0.065223	4.723674	0.065061
cannon_bg	3.378881	0.055228	3.118533	0.05615
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	2.7797	0.050615	2.708755	0.056394
summak= 62,500	N/A	N/A	N/A	N/A
summak=31,250	N/A	N/A	N/A	N/A
summak=15,625	N/A	N/A	N/A	N/A

Table A.18 117 cluster: X_DGEMM Run-Time(seconds) $20,000 \times 20,000 \times 20,000$, on 1×16 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	10.954598	0.074044	10.83657	0.079228
mm3_col	10.499467	0.029476	10.228963	0.016863
mm4_row	11.494188	0.148509	11.380996	0.148396
mm4_col	10.866928	0.035813	10.558586	0.074165
bb	11.490831	0.091332	11.383701	0.091168
cannon_c	10.925632	0.035783	10.360635	0.155385
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	10.912119	0.041656	10.679809	0.041
cannon_ag	12.64365	0.09125	12.405948	0.091045
cannon_bg	11.001981	0.054501	10.767641	0.056371
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	10.584375	0.029371	10.245091	0.068257
summak= 62,500	N/A	N/A	N/A	N/A
summak=31,250	N/A	N/A	N/A	N/A
summak=15,625	N/A	N/A	N/A	N/A

Table A.19 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 1×16 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	9.506201	0.470391	9.488248	0.465892
mm3_col	4.849476	0.093473	4.642501	0.056113
mm4_row	12.58785	0.011255	12.570569	0.009831
mm4_col	5.170553	0.009201	4.563987	0.008543
bb	10.825191	0.010449	10.819677	0.010589
cannon_c	5.183785	0.007139	4.580853	0.006809
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.379653	0.123209	4.750989	0.118898
cannon_ag	5.81584			
cannon_bg	10.228605	0.123475	9.621917	0.128256
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	9.027007	0.152136	8.797246	0.113245

Table A.20 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 24×1 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	8.277652	0.183249	7.719422	0.145028
mm3_col	12.614204	1.11824	12.551237	1.113973
mm4_row	8.219189	0.03588	7.718527	0.039594
mm4_col	13.14542	1.469705	13.083617	1.469467
bb	15.667921	0.012626	15.623418	0.012763
cannon_c	8.432319	0.290238	7.790356	0.09091
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	8.269445	0.069254	7.855751	0.06973
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A		
mm5_row	**7.637168**	0.177512	7.20609	0.10901
mm5_col	N/A	N/A	N/A	N/A

Table A.21 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 24×1 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	7.905346	0.304585	5.880218	0.20412
mm3_col	11.937141	0.633023	11.790512	0.666229
mm4_row	8.593247	0.351341	6.385729	0.203765
mm4_col	12.110588	0.864861	11.955695	0.884422
bb	11.603471	0.152037	11.450136	0.129502
cannon_c	8.827907	0.361374	6.574008	0.231057
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	8.872749	0.246963	8.453835	0.246962
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**7.841697**	0.140351	5.851113	0.160187
mm5_col	N/A	N/A	N/A	N/A

Table A.22 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 1×24 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	12.708139	0.476209	12.6852	0.473325
mm3_col	8.810144	0.075449	8.374924	0.116966
mm4_row	14.271543	0.122454	14.249858	0.122386
mm4_col	8.261561	0.057774	7.832586	0.040741
bb	13.398971	0.018287	13.386314	0.018401
cannon_c	8.296713	0.075067	7.855906	0.068286
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	8.317732	0.03611	7.906559	0.03616
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**7.592804**	0.121829	7.42827	0.067271

Table A.23 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 1×24 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	8.436811	0.358767	8.37875	0.355653
mm3_col	5.21341	0.201451	4.742025	0.100803
mm4_row	10.104417	0.260126	10.045261	0.22356
mm4_col	5.836928	0.379183	4.86	0.146479
bb	10.906115	0.274895	10.881611	0.239607
cannon_c	5.595362	0.355432	4.639839	0.105265
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.01676	0.366017	5.575004	0.366056
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**4.96719**	0.110307	4.317125	0.121891

Table A.24 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 12×2 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	8.155617	0.152833	7.753132	0.076672
mm3_col	9.084359	0.055306	9.054605	0.055197
mm4_row	7.794312	0.015147	6.665349	0.053644
mm4_col	8.78163	0.369829	8.533531	0.369237
bb	9.759138	0.19609	9.258754	0.096375
cannon_c	8.843081	0.055106	7.369898	0.068586
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**7.256081**	0.031664	6.486028	0.018025
mm5_col	N/A	N/A	N/A	N/A

Table A.25 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 12×2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	5.016872	0.079693	4.619336	0.064571
mm3_col	5.823837	0.419396	5.789203	0.419848
mm4_row	3.705507	0.048792	3.145052	0.018939
mm4_col	5.851084	0.189386	5.645621	0.192158
bb	6.419063	0.015455	6.07164	0.096508
cannon_c	4.439845	0.172308	3.547339	0.148953
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.833241**	0.194516	2.643571	0.193441
mm5_col	N/A	N/A	N/A	N/A

Table A.26 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 2×12 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	8.360173	0.061364	8.274832	0.061033
mm3_col	7.836562	0.074348	7.505397	0.069345
mm4_row	8.636001	0.427579	8.498593	0.42726
mm4_col	7.567457	0.039951	6.591008	0.077796
bb	9.752465	0.029003	9.360372	0.016875
cannon_c	8.625536	0.047929	7.255965	0.025839
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	6.731072	0.086537	6.136918	0.068075

Table A.27 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 2×12 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	5.39587	0.463774	5.299252	0.439574
mm3_col	5.068792	0.13206	4.720074	0.100202
mm4_row	6.497521	0.224668	6.371469	0.224907
mm4_col	3.987239	0.129534	3.257971	0.062737
bb	6.522373	0.357037	6.362599	0.294475
cannon_c	5.093367	0.254834	4.18314	0.231113
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**3.471539**	0.240552	3.023848	0.202893

Table A.28 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 6×4 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.23908	0.07192	3.057954	0.079454
mm3_col	3.746472	0.153754	3.71399	0.12483
mm4_row	3.779406	0.068145	3.337772	0.065403
mm4_col	3.973444	0.14231	3.643815	0.12389
bb	3.956565	0.010578	3.890766	0.007096
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	N/A	N/A	N/A	N/A

Table A.29 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 6×4 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	**3.411982**	0.153692	3.118854	0.115694
mm3_col	3.692251	0.182783	3.635881	0.153468
mm4_row	3.530008	0.066448	3.108971	0.048179
mm4_col	3.98254	0.161504	3.549455	0.135286
bb	4.118643	0.096264	3.936409	0.061849
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**3.444333**	0.164456	3.323492	0.139378
mm5_col	N/A	N/A	N/A	N/A

Table A.30 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 4×6 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	6.042612	0.088612	5.985112	0.072089
mm3_col	**5.584795**	0.079407	5.355801	0.073593
mm4_row	6.309155	0.02956	5.950634	0.037071
mm4_col	5.946651	0.053852	5.50302	0.030063
bb	5.882264	0.026684	5.850024	0.022049
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	5.843155	0.048247	5.621361	0.048374

Table A.31 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 4×6 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.769065	0.105776	3.636135	0.100905
mm3_col	3.288956	0.058488	3.045613	0.054742
mm4_row	3.97135	0.066274	3.645849	0.06438
mm4_col	3.737667	0.1097	3.24509	0.061775
bb	3.8943	0.109047	3.762724	0.063711
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**3.045159**	0.124608	2.864296	0.109519

Table A.32 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 3×8 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	5.941311	0.004607	5.917854	0.003593
mm3_col	6.427264	0.067619	6.277239	0.075211
mm4_row	7.201848	0.009072	6.926737	0.009201
mm4_col	6.009803	0.067411	5.533805	0.092317
bb	6.419483	0.012063	6.283853	0.010096
cannon_c	6.920871	0.055174	5.862715	0.053662
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.880652	0.066198	5.863518	0.066857
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**5.690901**	0.047681	5.479689	0.044332

Table A.33 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 3×8 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.345468	0.04407	3.325504	0.027403
mm3_col	3.691979	0.076939	3.556967	0.103921
mm4_row	4.459966	0.010466	4.199915	0.010118
mm4_col	3.236577	0.050663	2.750708	0.060178
bb	3.737314	0.02033	3.589997	0.017811
cannon_c	4.178907	0.060371	3.134632	0.037055
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.123989	0.051	3.11014	0.051083
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.767081**	0.079505	2.610261	0.077018

Table A.34 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 8×3 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.433816	0.064044	4.207472	0.044913
mm3_col	3.952358	0.215372	3.900937	0.172892
mm4_row	3.672156	0.148774	3.240419	0.124684
mm4_col	4.17466	0.104854	3.763696	0.106383
bb	4.513156	0.123794	4.415134	0.128925
cannon_c	4.601369	0.181123	3.518478	0.158133
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.567851	0.184818	3.471512	0.187373
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**3.174379**	0.178214	3.035684	0.152473
mm5_col	N/A	N/A	N/A	N/A

Table A.35 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 8×3 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.089301	0.094796	3.836033	0.044396
mm3_col	3.63176	0.154435	3.571395	0.130299
mm4_row	3.467076	0.044583	3.004129	0.081941
mm4_col	4.097665	0.025381	3.733532	0.025412
bb	4.149715	0.115838	3.990766	0.048797
cannon_c	4.333116	0.11522	3.29889	0.112673
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.472289	0.110709	3.36948	0.110719
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.92707**	0.08835	2.762622	0.063144
mm5_col	N/A	N/A	N/A	N/A

Table A.36 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 3×10 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	5.658576	0.030364	5.516253	0.029879
mm3_col	6.044017	0.107095	5.830446	0.096565
mm4_row	5.968206	0.032444	5.578032	0.031064
mm4_col	5.621032	0.068459	4.970533	0.068837
bb	6.195687	0.024764	5.914449	0.023563
cannon_c	6.527976	0.104975	5.560402	0.075452
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.42601	0.129621	5.626838	0.129384
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**5.38035**	0.067028	4.942287	0.057666

Table A.37 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 3×10 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.361506	0.057846	3.308901	0.059096
mm3_col	3.773656	0.116328	3.52887	0.116931
mm4_row	3.769395	0.089326	3.525667	0.059297
mm4_col	3.118827	0.146727	2.60058	0.140186
bb	4.193947	0.075422	3.899466	0.067344
cannon_c	3.976602	0.183455	2.957747	0.196558
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.702559	0.118242	2.910946	0.117193
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.647606**	0.150038	2.39658	0.111943

Table A.38 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 10×3 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.331042	0.040055	4.044473	0.065154
mm3_col	3.772255	0.112424	3.695666	0.112225
mm4_row	3.22704	0.060704	2.879021	0.055365
mm4_col	4.028443	0.137934	3.700095	0.160385
bb	4.524471	0.151264	4.30107	0.090595
cannon_c	4.08308	0.203586	3.215353	0.198393
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.076479	0.09697	3.225377	0.097085
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.723728**	0.135088	2.582524	0.135852
mm5_col	N/A	N/A	N/A	N/A

Table A.39 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 10×3 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.16118	0.166135	3.767824	0.064417
mm3_col	3.602342	0.204186	3.534657	0.197787
mm4_row	2.960542	0.062001	2.558269	0.052308
mm4_col	3.708714	0.179388	3.488363	0.18055
bb	4.429548	0.068823	4.10144	0.03835
cannon_c	3.736268	0.13508	2.876797	0.143757
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.686949	0.180508	2.826036	0.181002
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.369775**	0.096946	2.224609	0.035516
mm5_col	N/A	N/A	N/A	N/A

Table A.40 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 30×1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	7.74168	0.135983	6.934071	0.093455
mm3_col	12.017695	0.85497	11.968479	0.850446
mm4_row	7.832086	0.112914	7.126988	0.108305
mm4_col	13.593331	0.806504	13.545818	0.806467
bb	15.937639	0.018766	15.908537	0.018672
cannon_c	7.860598	0.136744	7.117615	0.104309
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	7.992033	0.069894	7.64791	0.069932
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**7.507258**	0.085654	6.744218	0.070468
mm5_col	N/A	N/A	N/A	N/A

Table A.41 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 30×1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	6.443398	0.377536	5.245961	0.20251
mm3_col	11.250942	1.303781	11.152174	1.325466
mm4_row	7.125105	0.520599	5.634596	0.254473
mm4_col	11.111163	0.734538	11.062897	0.713675
bb	10.543812	0.226271	10.451876	0.228575
cannon_c	7.029017	0.464902	5.41941	0.44225
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.9942	0.575133	6.646455	0.57524
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**6.577775**	0.306185	5.074156	0.378611
mm5_col	N/A	N/A	N/A	N/A

Table A.42 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 2×15 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	7.532195	0.033244	7.427399	0.034702
mm3_col	7.367456	0.076981	7.078931	0.064433
mm4_row	8.897254	0.038259	8.636502	0.038356
mm4_col	6.20054	0.056742	5.510332	0.046038
bb	9.069572	0.006403	8.614552	0.003564
cannon_c	7.025849	0.0637	6.067	0.06312
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.994801	0.103069	6.187791	0.103312
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**5.584336**	0.0815	5.321612	0.067384

Table A.43 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 2×15 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	5.818474	0.148632	5.787106	0.1808
mm3_col	5.000154	0.105162	4.621051	0.058237
mm4_row	7.176419	0.099827	6.945173	0.09979
mm4_col	3.458848	0.179643	2.851203	0.042788
bb	6.823433	0.063346	6.618808	0.116469
cannon_c	4.402799	0.24331	3.311483	0.133038
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.315871	0.44215	3.51601	0.440841
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.796528**	0.119844	2.600186	0.101171

Table A.44 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 5×6 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	6.261006	0.101686	5.777143	0.102417
mm3_col	5.433641	0.071195	5.138297	0.062578
mm4_row	6.159344	0.041186	5.709461	0.046373
mm4_col	**5.994861**	0.017517	5.361392	0.017172
bb	6.035153	0.004102	5.807901	0.00428
cannon_c	6.92523	0.067782	5.847134	0.071656
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.748876	0.071624	5.93346	0.072681
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**5.944865**	0.068752	5.656613	0.030769

Table A.45 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 5×6 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.691431	0.17162	3.375068	0.168859
mm3_col	2.732705	0.095267	2.509271	0.065791
mm4_row	3.428537	0.218373	3.022396	0.143852
mm4_col	**3.082165**	0.029336	2.729782	0.051363
bb	3.428679	0.094941	3.330151	0.089533
cannon_c	3.752579	0.328614	2.679432	0.214604
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.658458	0.300035	2.849253	0.289468
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	3.323145	0.12203	3.081725	0.066143

Table A.46 117 cluster: DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 6×6 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	**5.127599**	0.07795	4.955492	0.068715
mm3_col	5.919727	0.210218	5.467323	0.210379
mm4_row	5.729786	0.036668	5.284339	0.047631
mm4_col	5.763377	0.058944	5.341638	0.053793
bb	5.839115	0.020458	5.67865	0.016797
cannon_c	6.16325	0.047117	5.47865	0.048904
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.226938	0.056079	5.411826	0.037845
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	5.661105	0.003186	5.472099	0.003339
mm5_col	N/A	N/A	N/A	N/A

Table A.47 117 cluster: X_DGEMM Run-Time(seconds) $1,000 \times 1,000,000 \times 1,000$, on 6×5 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	**2.835786**	0.139896	2.647434	0.102929
mm3_col	3.393552	0.160402	3.083157	0.156762
mm4_row	3.15113	0.04096	2.830466	0.02848
mm4_col	3.337536	0.091479	3.009792	0.094253
bb	3.735427	0.056979	3.510252	0.057275
cannon_c	3.587058	0.161621	2.733039	0.179476
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.436548	0.182387	2.617134	0.190478
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	3.133371	0.123768	2.888467	0.173353
mm5_col	N/A	N/A	N/A	N/A

APPENDIX B

RAW DATA FROM STAMPEDE2 RUNS

Table B.1 Stampede2: DGEMM Run-Time(seconds) 20kx20kx20k, on 16x1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	2.781787	0.090894	2.67702	0.048309
mm3_col	3.641537	0.01537	3.627309	0.018168
mm4_row	2.825459	0.061571	2.589572	0.046353
mm4_col	3.644668	0.010152	3.631606	0.009818
bb	3.653826	0.012297	3.640809	0.011967
cannon_c	2.858778	0.034343	2.639316	0.022007
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.84269	0.053122	2.62709	0.056352
cannon_ag	2.959602	0.044541	2.746206	0.044116
cannon_bg	4.316988	0.140226	4.106365	0.13864
summa	N/A	N/A	N/A	N/A
mm5_row	**2.307429**	0.103681	2.248955	0.09019
mm5_col	N/A	N/A	N/A	N/A
summak=1250	3.535665	0.037091	3.518732	0.035377
summak=625	3.768682	0.038324	3.757743	0.039015
summak=250	4.183685	0.052424	4.178481	0.053159

Table B.2 Stampede2: X_DGEMM Run-Time(seconds) 20kx20kx20k, on 16x1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	**10.5638**	0.26663	10.033047	0.09836
mm3_col	11.04146	0.069297	10.903602	0.068812
mm4_row	11.09142	0.142112	10.385731	0.128191
mm4_col	11.038767	0.060433	10.889493	0.053762
bb	10.9865	0.115236	10.85584	0.112233
cannon_c	10.833055	0.122747	10.322224	0.158513
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	11.094455	0.038426	10.861638	0.007348
cannon_ag	11.255783	0.125771	11.038024	0.125063
cannon_bg	12.979559	0.063691	12.767189	0.062854
summa	N/A	N/A	N/A	N/A
mm5_row	10.839376	0.189131	10.226774	0.137236
mm5_col	N/A	N/A	N/A	N/A
summak= 1250	**10.501177**	0.112763	10.381507	0.112231
summak=625	18.847259	0.262214	18.714434	0.254911
summak=250	43.209114	1.008515	43.088213	0.996716

Table B.3 Stampede2: DGEMM Run-Time(seconds) 20kx20kx20k, on 1x16 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.721197	0.023215	3.70298	0.026661
mm3_col	2.793188	0.052383	2.682351	0.02295
mm4_row	3.705667	0.01519	3.690697	0.014622
mm4_col	2.952792	0.063358	2.725152	0.06392
bb	3.707692	0.011358	3.693089	0.010659
cannon_c	2.992992	0.068722	2.759803	0.038073
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.960102	0.057623	2.740712	0.057444
cannon_ag	4.434341	0.134848	4.221224	0.133209
cannon_bg	3.040574	0.042223	2.817793	0.041794
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.380452**	0.116267	2.32298	0.09519
summak= 1250	3.592225	0.033082	3.574372	0.035315
summak=625	3.736233	0.029571	3.720949	0.031025
summak=250	4.090763	0.033732	4.085094	0.033356

Table B.4 Stampede2: X_DGEMM Run-Time(seconds) 20kx20kx20k, on 1x16 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	11.073586	0.121923	10.922636	0.124183
mm3_col	10.704742	0.179754	10.195957	0.114066
mm4_row	11.211824	0.102536	11.05454	0.104986
mm4_col	10.935694	0.223104	10.338617	0.158416
bb	10.98566	0.092076	10.831358	0.075502
cannon_c	11.063379	0.199083	10.421793	0.127362
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	10.952076	0.157104	10.738861	0.157171
cannon_ag	13.047051	0.124595	12.827471	0.128155
cannon_bg	11.374654	0.085352	11.157551	0.085269
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**10.683153**	0.21005	10.11201	0.148845
summak= 1250	10.757767	0.162068	10.633830	0.159782
summak=625	18.970190	0.365688	18.835942	0.359578

Table B.5 Stampede2: DGEMM Run-Time(seconds) 20kx20kx20k, on 8x2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	2.160356	0.046074	2.08393	0.025741
mm3_col	2.113326	0.021485	2.088769	0.021455
mm4_row	1.976474	0.03739	1.735596	0.031953
mm4_col	2.103251	0.019088	2.077999	0.011975
bb	2.177195	0.037186	2.141646	0.006661
cannon_c	2.091218	0.037345	1.773757	0.049114
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.107715	0.044901	1.797311	0.042125
cannon_ag	2.267914	0.042042	1.957826	0.044525
cannon_bg	2.883409	0.090229	2.556079	0.086415
summa	N/A	N/A	N/A	N/A
mm5_row	**1.593792**	0.046396	1.561728	0.041971
mm5_col	N/A	N/A	N/A	N/A
summak= 1250	2.123396	0.025102	2.101459	0.027603
summak=625	2.269564	0.032824	2.261642	0.032603
summak=250	1.685542	0.074503	1.678678	0.074444

Table B.6 Stampede2: X_DGEMM Run-Time(seconds) 20kx20kx20k, on 8x2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	5.829654	0.086489	5.638389	0.08703
mm3_col	5.925174	0.042597	5.795448	0.037617
mm4_row	5.867925	0.100847	5.474437	0.067227
mm4_col	5.898397	0.024651	5.75964	0.035594
bb	5.901189	0.015819	5.603319	0.04061
cannon_c	6.219219	0.006134	5.715006	0.029968
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.10593	0.064629	5.782994	0.066513
cannon_ag	6.503421	0.11164	6.187174	0.113738
cannon_bg	7.217602	0.046582	6.89713	0.04789
summa	N/A	N/A	N/A	N/A
mm5_row	**5.708056**	0.153206	5.467709	0.169103
mm5_col	N/A	N/A	N/A	N/A
summak=1250	9.848178	0.104170	9.718495	0.100391
summak=625	18.581733	0.079641	18.447612	0.075655

Table B.7 Stampede2: DGEMM Run-Time(seconds) 20kx20kx20k, on 2x8 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	2.178206	0.018087	2.160626	0.018022
mm3_col	2.177041	0.020087	2.110852	0.006659
mm4_row	2.149056	0.02052	2.128406	0.022695
mm4_col	2.049804	0.064312	1.799479	0.058719
bb	2.213366	0.014942	2.170275	0.017982
cannon_c	2.258626	0.03154	1.926514	0.030959
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	2.184387	0.041438	1.863977	0.040397
cannon_ag	2.975107	0.061201	2.660054	0.057914
cannon_bg	2.36045	0.044266	2.039952	0.042943
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	1.720906	0.100718	1.667862	0.084922
summak= 1250	2.163689	0.037726	2.152244	0.027705
summak=625	2.374226	0.083487	2.363193	0.074538
summak=250	**1.653983**	0.059084	1.648238	0.056313

Table B.8 Stampede2: X_DGEMM Run-Time(seconds) 20kx20kx20k, on 2x8 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	5.766913	0.097106	5.657572	0.103068
mm3_col	**5.75177**	0.055777	5.575617	0.045228
mm4_row	5.762985	0.022461	5.636158	0.030075
mm4_col	5.966553	0.031713	5.612762	0.036581
bb	5.880814	0.029139	5.601784	0.034195
cannon_c	6.209049	0.022052	5.707688	0.053382
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	6.00845	0.103226	5.693742	0.10474
cannon_ag	7.169963	0.064818	6.852157	0.06436
cannon_bg	6.438192	0.051365	6.114714	0.051761
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**5.754381**	0.008952	5.510686	0.049643
summak= 1250	9.815227	0.140657	9.692203	0.137320

Table B.9 Stampede2: DGEMM Run-Time(seconds) 20kx20kx20k, on 4x4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	**1.715549**	0.036488	1.658283	0.028918
mm3_col	1.721091	0.047677	1.652185	0.025321
mm4_row	1.739423	0.045115	1.665969	0.032649
mm4_col	**1.719584**	0.049705	1.65757	0.039481
bb	1.950508	0.037679	1.904439	0.014983
cannon_c	1.873396	0.061284	1.643583	0.056254
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	1.949031	0.041982	1.729718	0.047018
cannon_ag	2.395361	0.040062	2.200311	0.046517
cannon_bg	2.359316	0.064383	2.150774	0.06178
summa	N/A	N/A	N/A	N/A
mm5_row	1.895056	0.076066	1.763404	0.049523
mm5_col	1.899374	0.040792	1.787484	0.033205
summak= 1250	2.191787	0.160748	2.131948	0.127206
summak=625	2.383149	0.136051	2.379411	0.136246
summak=250	2.596985	0.103635	2.587019	0.098707

Table B.10 Stampede2: X_DGEMM Run-Time(seconds) 20kx20kx20k, on 4x4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.510406	0.057565	3.384522	0.088097
mm3_col	3.538849	0.052219	3.44696	0.046102
mm4_row	**3.560631**	0.04339	3.466854	0.042327
mm4_col	**3.59904**	0.037082	3.47895	0.036957
bb	3.732406	0.017893	3.572639	0.00919
cannon_c	3.741176	0.045123	3.521032	0.035665
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.744007	0.030398	3.522951	0.029137
cannon_ag	4.18289	0.03432	3.980191	0.036256
cannon_bg	4.196543	0.041242	3.975273	0.039116
summa	N/A	N/A	N/A	N/A
mm5_row	3.69321	0.053846	3.575891	0.076973
mm5_col	3.684001	0.074108	3.449531	0.051415
summak= 1250	9.678787	0.150045	9.550333	0.147053

Table B.11 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 16x1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	**5.738962**	0.083517	5.433079	0.045086
mm3_col	7.48876	0.012003	7.453662	0.012118
mm4_row	5.833786	0.046033	5.265165	0.045266
mm4_col	7.488633	0.011174	7.456309	0.010868
bb	7.444492	0.011646	7.411649	0.011439
cannon_c	5.877859	0.150466	5.301826	0.132116
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.820165	0.052336	5.272106	0.072784
cannon_ag	5.851473	0.056287	5.300519	0.056182
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	N/A	N/A	N/A	N/A

Table B.12 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 16x1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	**5.653682**	0.104966	5.371229	0.091355
mm3_col	7.521545	0.010601	7.491058	0.012902
mm4_row	5.889552	0.091338	5.317088	0.063364
mm4_col	7.52251	0.019253	7.493305	0.021095
bb	7.488498	0.009259	7.463464	0.010165
cannon_c	5.901989	0.080636	5.33691	0.075043
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.869044	0.069127	5.314036	0.069242
cannon_ag	5.807341	0.096016	5.245784	0.095304
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	N/A	N/A	N/A	N/A

Table B.13 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 1X16 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	7.735438	0.021647	7.703688	0.022421
mm3_col	**5.608723**	0.048987	5.42666	0.055741
mm4_row	7.687203	0.01673	7.657675	0.016033
mm4_col	6.026921	0.057329	5.449322	0.064139
bb	7.640521	0.021457	7.611523	0.019517
cannon_c	5.981446	0.061259	5.435711	0.060641
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.956073	0.063972	5.416753	0.063945
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	N/A	N/A	N/A	N/A

Table B.14 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 1X16 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	7.452418	0.044339	7.423282	0.043084
mm3_col	5.487094	0.141911	5.256505	0.117795
mm4_row	7.43297	0.014037	7.400243	0.013732
mm4_col	5.756797	0.080023	5.217998	0.077223
bb	7.422669	0.013424	7.39153	0.013252
cannon_c	5.706446	0.101514	5.162247	0.083754
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.667892	0.066134	5.140944	0.065195
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	N/A	N/A	N/A	N/A

Table B.15 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 2X8 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.262806	0.03065	4.213576	0.03464
mm3_col	4.197872	0.033265	4.083128	0.032126
mm4_row	4.228498	0.018209	4.202092	0.022274
mm4_col	3.958016	0.043948	3.406046	0.03462
bb	4.422545	0.015353	4.377503	0.009953
cannon_c	4.394054	0.038217	3.583541	0.030055
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.301309	0.031906	3.517384	0.031933
cannon_ag	4.278885	0.050151	3.494111	0.057449
cannon_bg	4.321825	0.084361	3.537836	0.084096
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.931949**	0.080208	2.86153	0.072125
summak= 62500	4.394717	0.033610	4.361812	0.033293
summak=31250	4.356114	0.017053	4.338945	0.018001
summak=15625	4.515287	0.036462	4.503864	0.030831

Table B.16 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 2X8 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.933289	0.039866	3.90542	0.041735
mm3_col	3.970964	0.01829	3.863822	0.024559
mm4_row	3.953631	0.027084	3.931302	0.025789
mm4_col	3.630889	0.050354	3.108206	0.057096
bb	4.140487	0.007097	4.103479	0.00589
cannon_c	4.064698	0.019351	3.260979	0.016716
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.91861	0.029584	3.14584	0.027402
cannon_ag	3.831314	0.03503	3.039961	0.029134
cannon_bg	3.921933	0.040472	3.143978	0.040751
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.5369**	0.054973	2.494047	0.036845
summak= 62500	4.153626	0.064796	4.134559	0.064476
summak=31250	4.081104	0.066222	4.072584	0.066876
summak=15625	4.331057	0.096921	4.324720	0.097092

Table B.17 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 8x2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	4.214909	0.015194	4.136341	0.023237
mm3_col	4.073732	0.014319	4.040215	0.015117
mm4_row	3.729855	0.054938	3.168406	0.042156
mm4_col	4.102313	0.038124	4.062709	0.035695
bb	4.302904	0.016558	4.254546	0.012367
cannon_c	4.205115	0.042712	0.042712	0.033712
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.136069	0.042626	3.332678	0.044775
cannon_ag	4.019952	0.048972	3.239578	0.04012
cannon_bg	3.98165	0.043959	3.199116	0.047832
summa	N/A	N/A	N/A	N/A
mm5_row	**2.720745**	0.081657	2.668047	0.067295
mm5_col	N/A	N/A	N/A	N/A
summak= 62500	4.277147	0.030453	4.258347	0.019875
summak=31250	4.148420	0.024456	4.135920	0.023636
summak=15625	4.309977	0.022043	4.299258	0.022856

Table B.18 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 8x2 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.055317	0.033084	3.911126	0.037647
mm3_col	3.973925	0.012593	3.946825	0.013045
mm4_row	3.63912	0.053744	3.1194	0.039113
mm4_col	3.959473	0.008977	3.932838	0.010365
bb	4.126256	0.008871	4.078065	0.006156
cannon_c	4.075613	0.056711	3.272461	0.058197
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	4.026461	0.046892	3.243728	0.043026
cannon_ag	3.907203	0.037716	3.13685	0.037493
cannon_bg	3.842685	0.032123	3.060884	0.047861
summa	N/A	N/A	N/A	N/A
mm5_row	**2.649031**	0.082122	2.59096	0.065382
mm5_col	N/A	N/A	N/A	N/A
summak= 62500	4.175258	0.034787	4.149201	0.029637
summak=31250	4.056620	4.056620	4.043206	0.025781
summak=15625	4.330239	0.018880	4.321053	0.017460

Table B.19 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 4x4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.203153	0.068965	3.096638	0.066479
mm3_col	**3.129**	0.033202	2.99895	0.037072
mm4_row	3.173233	0.052567	3.072323	0.044028
mm4_col	**3.107457**	0.045969	3.004562	0.031968
bb	3.876474	0.017606	3.840092	0.014238
cannon_c	3.542897	0.074237	3.053601	0.071775
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.57502	0.041449	3.084943	0.038046
cannon_ag	3.541813	0.074767	3.033528	0.101516
cannon_bg	3.575321	0.060357	3.077167	0.063835
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	3.387456	0.045475	3.256848	3.256848
summak= 62500	3.901732	0.041087	3.850946	0.036295
summak=31250	3.772787	0.017295	3.761962	0.015786
summak=15625	3.959038	0.013488	3.946298	0.013273

Table B.20 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 4x4 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	2.839411	0.041068	2.762465	0.032706
mm3_col	**2.793943**	0.014931	2.715991	0.028313
mm4_row	2.82084	0.031039	2.740449	0.023687
mm4_col	2.821432	0.045188	2.730391	0.027149
bb	3.618108	0.019769	3.564933	0.0063
cannon_c	3.129582	0.042002	2.603955	0.054401
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.209889	0.050087	2.691529	0.057724
cannon_ag	3.144318	0.040328	2.659598	0.047086
cannon_bg	3.188132	0.062252	2.675017	0.062166
summa	N/A	N/A	N/A	N/A
mm5_row	2.945723	0.035036	2.864223	0.037891
mm5_col	2.929245	0.038861	2.833816	0.035598
summak= 62500	3.635329	0.034364	3.589823	0.037431
summak=31250	3.614295	0.027068	3.607060	0.027223
summak=15625	3.832672	0.021941	3.828678	0.021785

Table B.21 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 1x24 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	8.45839	0.021523	8.427345	0.025684
mm3_col	5.726837	0.085555	5.360642	0.03012
mm4_row	8.442294	0.018277	8.410518	0.018394
mm4_col	5.460098	0.028983	5.107415	0.030643
bb	8.445527	0.034651	8.422815	0.034973
cannon_c	5.495547	0.035806	5.139393	0.035265
cannon_a	5.488725	0.019606	5.123699	0.019726
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**4.396079**	0.110864	4.261504	0.092716
mm5_col	N/A	N/A	N/A	N/A

Table B.22 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 1x24 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	8.38215	0.044878	8.356134	0.043363
mm3_col	5.59121	0.11584	5.379425	0.074502
mm4_row	8.319037	0.028694	8.29219	0.028812
mm4_col	5.577472	0.042847	5.222875	0.04289
bb	8.313715	0.020968	8.293889	0.020852
cannon_c	5.625993	0.066275	5.270311	0.039199
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.622802	0.05763	5.262799	0.057476
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**4.31915**	0.274214	4.171234	0.255951

Table B.23 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 24X1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	5.65601	0.080213	5.270823	0.030611
mm3_col	8.344213	0.038761	8.311609	0.043717
mm4_row	5.279027	0.040525	4.903033	0.043972
mm4_col	8.319632	0.057649	8.293763	0.05798
bb	8.275923	0.037704	8.255105	0.038211
cannon_c	5.363099	0.064787	4.988217	0.062832
cannon_a	N/A	N/A	N/A	N/A
cannon_b	5.321795	0.031881	4.958703	0.037603
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**4.337421**	0.15214	4.231628	0.126008
mm5_col	N/A	N/A	N/A	N/A

Table B.24 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 24X1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	5.840285	0.076161	5.61341	0.039287
mm3_col	8.377076	0.02276	8.347141	0.019958
mm4_row	5.79778	0.0713	5.417092	0.047292
mm4_col	8.389939	0.050351	8.347916	0.015609
bb	8.370216	0.008956	8.338602	0.010537
cannon_c	5.792052	0.046904	5.426762	0.028081
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.783374	0.034093	5.393122	0.034435
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**4.515002**	0.206727	4.361538	0.165978
mm5_col	N/A	N/A	N/A	N/A

Table B.25 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 12X2 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.328413	0.018401	3.8539	0.048035
mm3_col	4.401844	0.014249	4.380015	0.01248
mm4_row	3.558425	0.049529	3.176455	0.064476
mm4_col	4.82329	0.013667	4.683989	0.013227
bb	4.545437	0.010635	4.51123	0.007552
cannon_c	4.056724	0.034728	3.39884	0.054532
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.838548**	0.035863	2.76039	0.035247
mm5_col	N/A	N/A	N/A	N/A

Table B.26 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 12X2 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.283733	0.047612	3.855587	0.042021
mm3_col	4.339678	0.011752	4.321973	0.010727
mm4_row	3.577238	0.042983	3.220233	0.057073
mm4_col	4.7731	0.017513	4.616497	0.017315
bb	4.478846	0.013113	4.439388	0.010069
cannon_c	4.106531	0.042369	3.429956	0.050785
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.698222**	0.075327	2.638715	0.064311
mm5_col	N/A	N/A	N/A	N/A

Table B.27 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 3X8 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.352797	0.017312	3.312307	0.014975
mm3_col	3.535287	0.01873	3.430297	0.008898
mm4_row	3.651113	0.035717	3.329517	0.019782
mm4_col	3.147572	0.028107	2.720499	0.027057
bb	3.503153	0.023311	3.434125	0.023503
cannon_c	3.960935	0.056582	3.023022	0.051233
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.947293	0.077164	3.02448	0.090275
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.866903**	0.015111	2.770972	0.01393

Table B.28 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 3X8 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.179909	0.030605	3.133902	0.031569
mm3_col	3.429871	0.020838	3.33795	0.008213
mm4_row	3.478981	0.033973	3.16107	0.008101
mm4_col	3.032952	0.054987	2.619552	0.050194
bb	3.322138	0.010813	3.259055	0.012552
cannon_c	3.834265	0.065851	2.893359	0.058547
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.779584	0.059361	2.886051	0.058027
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.663021**	0.02219	2.590837	0.01452

Table B.29 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 8X3 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.505393	0.034106	3.407168	0.031964
mm3_col	3.248058	0.021183	3.21024	0.024634
mm4_row	3.103884	0.035893	2.707429	0.028878
mm4_col	3.571994	0.031927	3.256261	0.018786
bb	3.398381	0.0227	3.323218	0.012216
cannon_c	3.905596	0.089938	2.977605	0.06427
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.888331	0.063813	2.966737	0.060206
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.767815**	0.021237	2.679015	0.018975
mm5_col	N/A	N/A	N/A	N/A

Table B.30 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 8X3 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.513424	0.029809	3.387817	0.03705
mm3_col	3.187358	0.013369	3.131347	0.010076
mm4_row	3.032882	0.064184	2.632228	0.053707
mm4_col	3.508774	0.022052	3.195344	0.020867
bb	3.308673	0.006467	3.234233	0.007019
cannon_c	3.855943	0.088553	2.925804	0.07412
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.836439	0.064748	2.914107	0.066995
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.646421**	0.023586	2.573228	0.022015
mm5_col	N/A	N/A	N/A	N/A

Table B.31 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 6X4 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.160832	0.037778	2.976554	0.026889
mm3_col	3.248382	0.027495	3.166304	0.02357
mm4_row	3.396989	0.057084	3.079719	0.049937
mm4_col	3.617156	0.03322	3.25897	0.023269
bb	3.344351	0.019605	3.299733	0.010001
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**3.007754**	0.035207	2.882488	0.020396
mm5_col	N/A	N/A	N/A	N/A

Table B.32 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 6X4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.046412	0.039844	2.887525	0.02951
mm3_col	3.155118	0.026347	3.094892	0.025028
mm4_row	3.254924	0.020002	2.919847	0.083947
mm4_col	3.485395	0.03544	3.119913	0.016384
bb	3.237799	0.017807	3.184951	0.010161
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.835384**	0.055644	2.716264	0.031296
mm5_col	N/A	N/A	N/A	N/A

Table B.33 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 4X6 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.345744	0.032026	3.275283	0.020191
mm3_col	3.138826	0.02008	2.957825	0.015805
mm4_row	3.676762	0.030009	3.314785	0.01393
mm4_col	3.440351	0.025458	3.120354	0.013242
bb	3.376627	0.021567	3.332475	0.021953
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**3.037468**	0.052676	2.909494	0.041564

Table B.34 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 4X6 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.218744	0.032966	3.158233	0.034832
mm3_col	3.017851	0.018722	2.850026	0.011723
mm4_row	3.462503	0.018531	3.117569	0.012224
mm4_col	3.299377	0.049833	2.920159	0.079418
bb	3.229824	0.014196	3.19323	0.010022
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.894791**	0.063406	2.772078	0.063643

Table B.35 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 10X3 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.514858	0.01367	3.325621	0.008122
mm3_col	3.581997	0.010092	3.487422	0.007226
mm4_row	2.870717	0.041385	2.520075	0.061059
mm4_col	3.658376	0.020346	3.39813	0.013543
bb	3.488661	0.014617	3.450064	0.013549
cannon_c	3.40215	0.076734	2.647186	0.07138
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.318689	0.078509	2.60855	0.067382
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.510632**	0.045239	2.427399	0.031088
mm5_col	N/A	N/A	N/A	N/A

Table B.36 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 10X3 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.500899	0.035638	3.260365	0.033977
mm3_col	3.533815	0.008569	3.445624	0.009695
mm4_row	2.777297	0.068878	2.404114	0.03959
mm4_col	3.612043	0.026721	3.342159	0.012045
bb	3.409142	0.015286	3.367717	0.014057
cannon_c	3.157822	0.060354	2.452814	0.075366
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.160184	0.058897	2.438898	0.057878
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.335101**	0.046256	2.27162	0.038967
mm5_col	N/A	N/A	N/A	N/A

Table B.37 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 3X10 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.757077	0.018986	3.671671	0.012994
mm3_col	3.536374	0.007454	3.361788	0.022038
mm4_row	3.801957	0.033509	3.547532	0.025433
mm4_col	3.011143	0.050239	2.663315	0.05965
bb	3.673503	0.016808	3.617787	0.010663
cannon_c	3.478287	0.065792	2.75625	0.04395
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.473586	0.053429	2.791566	0.048048
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.650887**	0.027816	2.560581	0.019219

Table B.38 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 3X10 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.587176	0.037604	3.489033	0.030261
mm3_col	3.468597	0.013486	3.225098	0.017994
mm4_row	3.622106	0.041971	3.365743	3.365743
mm4_col	2.756187	0.058557	2.388885	0.033558
bb	3.472019	3.472019	3.472019	0.009364
cannon_c	3.309094	0.05307	2.612884	0.034711
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.328692	0.064763	2.609425	0.061198
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.341024**	0.041335	2.280672	2.280672

Table B.39 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 15X2 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.29837	0.051705	3.920812	3.920812
mm3_col	3.920812	0.018768	4.421479	0.015555
mm4_row	3.218701	0.078583	2.859497	0.046316
mm4_col	4.705823	0.019068	4.47726	0.013508
bb	4.472857	0.221796	4.439948	0.228622
cannon_c	3.70359	0.061336	3.009066	0.058045
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.716928	0.052876	3.022234	0.051743
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.831777**	0.047748	2.747272	0.035346
mm5_col	N/A	N/A	N/A	N/A

Table B.40 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 15X2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	4.368235	0.043347	3.959199	0.045686
mm3_col	4.489708	0.021787	4.465323	0.020755
mm4_row	3.184423	0.135599	2.836097	0.126174
mm4_col	4.732523	4.732523	4.498371	4.498371
bb	4.374367	0.032356	4.326145	0.024157
cannon_c	3.749523	0.099634	3.056503	0.09932
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.735474	0.081331	3.048426	0.085474
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**2.7764**	0.057209	2.716735	0.051187
mm5_col	N/A	N/A	N/A	N/A

Table B.41 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 2X15 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	4.537496	0.018019	4.517639	0.015055
mm3_col	4.283722	0.023754	3.965412	0.017423
mm4_row	4.843161	0.020365	4.619049	0.022802
mm4_col	3.331805	0.087465	2.971812	0.05567
bb	4.539502	0.01738	4.474962	0.009298
cannon_c	3.953567	0.083517	3.268627	0.079354
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.947715	0.097082	3.268721	0.075353
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.898471**	2.898471	2.826505	0.07025

Table B.42 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 2X15 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	4.506144	0.018922	4.486898	0.019619
mm3_col	4.22307	0.0139	3.926386	0.031005
mm4_row	4.720304	0.020659	4.497234	0.014164
mm4_col	3.20567	0.108612	2.829852	0.09955
bb	4.440573	0.019602	4.395036	0.010968
cannon_c	3.89975	0.061623	3.245693	0.06778
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.833345	0.052451	3.171206	0.050631
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**2.831264**	0.041963	2.776026	0.033707

Table B.43 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 1X30 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	8.852186	0.096911	5.398056	0.029981
mm3_col	8.861799	0.043633	8.819533	0.019454
mm4_row	8.861799	0.043633	8.819533	0.019454
mm4_col	5.389181	0.044735	5.088852	0.040424
bb	8.838018	0.016282	8.803336	0.014641
cannon_c	5.391435	0.017308	5.081901	0.017725
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.368639	0.018572	5.068666	0.018196
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**4.532408**	0.099982	4.399734	0.097491

Table B.44 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 1X30 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	8.897819	0.044805	8.866161	0.041318
mm3_col	5.650716	0.089897	5.439874	0.026054
mm4_row	8.898598	0.036385	8.847434	0.015697
mm4_col	5.538136	0.056598	5.240385	0.051751
bb	8.814915	0.019445	8.782449	0.018156
cannon_c	5.534651	0.075329	5.249275	0.066419
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.546148	0.046314	5.248003	0.047168
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**4.620335**	0.144842	4.436738	0.133359

Table B.45 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 30x1 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	5.776788	0.130887	5.305695	0.044333
mm3_col	8.772705	0.017406	8.739611	0.01666
mm4_row	5.223377	0.055667	4.892938	0.030422
mm4_col	8.749091	0.019171	8.716653	0.019039
bb	8.691961	0.01368	8.674242	0.014123
cannon_c	5.186697	5.186697	4.87013	4.87013
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.190609	0.030463	4.88004	0.033113
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**4.4046**	0.033113	4.263045	0.126637
mm5_col	N/A	N/A	N/A	N/A

Table B.46 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 30x1 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	5.999449	0.063811	5.689894	0.037963
mm3_col	8.926476	0.03225	8.88464	0.021936
mm4_row	5.648814	0.037875	5.344519	0.02944
mm4_col	8.897074	0.018547	8.859861	0.015481
bb	8.882847	0.021721	8.854683	0.022376
cannon_c	5.68718	0.079879	5.373852	0.061781
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	5.699088	0.041086	5.378356	0.031133
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**4.700531**	0.222084	4.523871	0.201087
mm5_col	N/A	N/A	N/A	N/A

Table B.47 Stampede2: DGEMM Run-Time(seconds) 1kx1mx1k, on 5X6 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	3.411531	0.019788	3.103697	0.013191
mm3_col	**2.68632**	0.019168	2.550471	2.550471
mm4_row	3.105214	0.028839	2.770949	0.043365
mm4_col	2.919297	0.02479	2.602031	0.031491
bb	3.198404	0.012501	3.112315	0.012311
cannon_c	3.28702	0.094339	2.520919	0.083752
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.219742	0.026986	2.509193	0.026738
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	2.922301	0.032711	2.838288	0.034756

Table B.48 Stampede2: X_DGEMM Run-Time(seconds) 1kx1mx1k, on 5X6 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	3.2025	0.04168	2.901067	0.033822
mm3_col	**2.532934**	0.028159	2.402534	0.035942
mm4_row	2.827535	0.031552	2.543495	0.022738
mm4_col	2.757272	0.048684	2.428585	0.032532
bb	2.956305	0.015096	2.914598	0.010835
cannon_c	3.016148	0.055457	2.320354	0.034354
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	3.055042	0.070141	2.3453	0.067721
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	2.681195	0.020289	2.595165	0.022055

APPENDIX C

SUMMARY OF TEST PROGRAMS

Once the library is built, a couple of test programs are used to generate experimental data files and to run the tests as described in the methodology. The gen1.c program is used to generate the data file with all the required parameters as defined by the user. The program provides a prompt and the user is able to to select from the available options as listed below:

- Select one or more algorithms from the polyalgorithms set.
- Input the mesh dimension: grid shape or size in the form of $P \times Q$.
- Select the storage type: row major or column major.
- Input the dimensions of matrix A and B.
- Select the type of row and columns mapping: linear, scatter, block linear, block scatter, virtual scatter, virtual block scatter.
- Input alpha and beta for scaling.
- Select the data type: all one, uniform or random.

The above steps generate the data file called *data.* We then run the main test program which uses the generated data file. A simple test run with four processors can be: *mpirun -np 4 ./main1 data.*

VITA

Grace Nansamba was born in 1993 and raised in Kiwoko, Uganda. She graduated in 2015 from Makerere University with a Bachelor of Information Technology. She came to America in 2018 to pursue a Master's degree in Computer Science at the University of Tennessee at Chattanooga; She worked as a graduate research assistant for Dr. Anthony Skjellum, focusing on high performance computing. She also worked as a teaching Assistant at UTC, assisting the instructor in the preparation of course materials and lecturing. She won a ChaTech (Chattanooga Technology) Scholarship in 2019. Grace plans to advance her education further through a PhD. program in Computational Science at the University of Tennessee at Chattanooga.