論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Cache Blocking and Parallel Runtime Scheduling of Hierarchical Matrices
著者(和文)	Deshmukh Sameer Satish
Author(English)	Sameer Satish Deshmukh
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12558号, 授与年月日:2023年9月22日, 学位の種別:課程博士, 審査員:横田 理央,吉瀬 謙二,宮﨑 純,DEFAGO XAVIER,小野 峻佑
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12558号, Conferred date:2023/9/22, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Cache Blocking and Parallel Runtime Scheduling of Hierarchical Matrices

Sameer Satish DESHMUKH
Department of Computer Science
School of Computing
Tokyo Institute of Technology

A thesis submitted for the degree of
Doctor of Engineering

2023
Under the supervision of Prof. Rio Yokota

Tokyo Tech

Contents

List of Figures

Acronyms

BEM Boundary Element Method. 4, 84

BLACS Basic Linear Algebra Communication Subroutines. 5

BLAS Basic Linear Algebra Subroutines. 5, 8, 48

BLIS BLAS-like Library Instantiation Software. 100, 101

BLR Block Low Rank. 30, 84–86, 94, 95

CPU Central Processing Unit. vi, vii, 5, 45, 46, 48, 50, 53–59

DPLASMA Distributed Linear Algebra Software for Multicore Architectures. 5, 52

DTD Dynamic Task Discovery. 101–103

ECM Execution Cache Memory. ii, iii, vii, 8, 9, 61, 65, 67–72

FDM Finite Difference Method. 20

FEM Finite Element Method. 20

FMA Fused Multiply Addition. 67, 69, 70, 79

GEMM GEneral Matrix Multiplication. 25

GETRF GEneral TRiangular Factorization. 25

GFLOPS Giga Floating point Operations Per Second. vii, viii, 74–76, 79

HODLR Hierarchical Off-Diagonal Low Rank. 30

HSS Hierarchically Semi-Separable. iii, viii, 7, 84–92, 94–96, 98, 99

LAPACK Linear Algebra PACKage. 5

LU factorization Lower Upper factorization. 22

MKL Math Kernel Library. 6

MPI Message Passing Interface. 5, 50, 51

NUMA Non Uniform Memory Access. 45, 50

PBLAS Parallel Basic Linear Algebra Subroutines. 48

PDE Partial Differential Equation. 4, 9, 84

PTG Parameterized Task Graph. 102, 103

ScaLAPACK Scalable Linear Algebra PACKage. viii, 5, 7, 48, 49, 52, 84, 85, 87–91

SIMD Single Instruction Multiple Data. 45, 47, 59, 60, 62, 64, 67, 69, 70, 72, 76, 82

SLATE Software for Linear Algebra Targeting Exascale. 5

SMT Simultaneous Multi-Threading. 45

SPD Symmetric Positive Definite. 24

SSL Scientific Software Library. 100, 102

STRUMPACK STRUctured Matrix PACKage. iii, viii, 7, 49, 84–86, 88, 90–92, 94–102

TRSM TRiangular Solve Matrix. 25

ULV U basis-Lower triangular-V basis. 7–9, 34

Abstract

Important scientific problems in electrostatics, acoustics and statistics can be solved using the Boundary Element Method (BEM). The coefficient matrix from a BEM discretization results in a dense matrix, leading to $O(N^2)$ and $O(N^3)$ time complexity for the matrix-vector product and direct factorization algorithms, respectively. The underlying geometry from which the dense matrix is generated can be exploited in order to approximate the interactions between far points. These points are expressed in the dense matrix as off-diagonals blocks. Low rank approximation of the off-diagonal blocks of such a structured dense matrix can reduce the time complexity of the matrix-vector product and direct factorization algorithms to $O(N)$ by trading off accuracy for time. The accuracy of the algorithm can be tuned to match the required accuracy of the application.

Algorithmic developments of the compression, multiplication and factorization routines of such low rank approximated matrices have led to reduction in the time complexity and better accuracy for a wide variety of problems. However, the implementation of such routines on modern, highly parallel computer architectures for such routines still requires the use of numerical software that was originally written for computation of dense linear algebra. The small, irregular kernels that are prevalent in the algorithms involving low rank approximation result in inefficient execution on modern hardware.

In this thesis, we propose improvements to the efficiency of the matrix-vector product and direct factorization algorithms of low rank approximated dense matrices on shared and distributed memory machines. We show that our techniques are applicable to 2D problems for modeling acoustic waves, electrostatics and statistics with acceptable accuracy for the application in question. We target the block low rank and hierarchically semi-separable low rank matrix formats for this study since they have been shown to work well with 2D problems.

The first part of this thesis improves the efficiency of the matrix-vector product. The matrix-vector product is useful for problems arising from the wave equation. The low rank matrix multiplication algorithm is a key computational kernel of the matrix vector product. We design efficient cache blocking algorithms by leveraging the Execution-Cache-Memory performance model. Our portable implementation making use of parallel

loops written in a high level language with architecture-specific assembly micro-kernels. This preserves the portability of our method and allows us outperform vendor optimized BLAS libraries on the Fujitsu A64FX, Intel Xeon 6148 and AMD EPYC 7502 CPUs. We improve the matrix vector product of low rank matrices by obtaining a 2x performance improvement in the low rank matrix multiplication algorithm on all CPU architectures.

The second part of this thesis improves the distributed memory factorization of hierarchical matrices arising out of 2D problems from electrostatics and statistics. Specifically, we show that we achieve up to 2x improvement in weak scaling performance over the state-of-the-art in dense direct factorization on up to 128 nodes of Fugaku. We show that this improvement in speed can be achieved with similar or better accuracy than other state-of-the-art implementations making use of low rank approximation of structured dense matrices. The main reasons behind the improvement can be attributed to the use of the HSS-ULV algorithm with a distributed asynchronous runtime system. The combination of an algorithm needing $O(N)$ computation and communication with an asynchronous distributed runtime system results in better overlap of computation with communication and results in better performance.

Acknowledgements

I remember making my way to Tokyo as a speaker at the Ruby Kaigi conference, and sending a mail to Prof. Rio Yokota asking if he would have time to meet and talk about a Master's degree in High Performance Computing. He agreed and accepted me under his wing, and thus began my journey as a graduate student at Tokyo Institute of Technology. Through several ups and downs, I was able to steadily improve my understanding of computer science and research. The results of the work done in the past few years are in front of the reader as this thesis. I would like to express my deepest gratitude to Prof. Rio Yokota for his trust and the knowledge that he has imparted.

My family has been a constant source of strength and support throughout this journey. I would like thank my mother, Dr. Manisha Deshmukh and my father, Dr. Satish Deshmukh for being very open minded about my chosen path in life. My sister Mayuri for being a constant source of endurance. My grandfather Dr. Hemchandra Deshmukh for buying me my first laptop computer with which I spent many pleasant evenings, which eventually resulted in this thesis. My grandmother, Mrs. Sumati Deshmukh for being her kind, industrious and generous self. Along this journey as a graduate student, I also met my now wife Namita. Building dreams of the future with her has been a constant source of encouragement for me to continue working on my research.

This thesis would have been impossible without support from various collaborators and colleagues who have helped guide the way. I would like to thank Qinxiang Ma, Thomas Spendlhofer, Muhammed Ridwan, Hiroyuki Ootomoh, Edgar Martinez and Prof. Yasuhiro Matsumoto for their ideas, criticism and support. Prof. George Bosilca from Innovative Computing Laboratory at University of Tennessee at Knoxville for inviting me for a wonderful 4 months in the beautiful city of Knoxville. The collaboration with the staff and students in his lab significantly enhanced my research results and led to the building of some great relationships.

Chapter 1

Introduction

Important problems arising from physical phenomena such as electrostatics, electromagnetism and geostatistics can be studied using Partial Differential Equations (PDEs). The Boundary Element Method (BEM) is an important technique for solving such PDEs. The BEM results in the formation of a dense matrix A that represents the magnitude of the interaction between all elements obtained by discretizing the PDE. The matrix vector product and factorization algorithms are important algorithms that make use of matrix A. The matrix-vector product and factorization of dense matrices cost $O(N^2)$ and $O(N^3)$, respectively. This cost is prohibitive if the matrix gets too large. However, it can be brought down to $O(N)$ for both the matrix-vector product and factorization by trading the accuracy for speed. The error introduced in this case can be controlled according to the needs of the application. This can be done with the use of the structured low-rank representation of the matrix which approximates the far interactions between the points in the domain. Eq. 1.1 shows that vector b can be obtained using a matrix-vector product between A and x or the vector x can be obtained by solving $A^{-1}b$.

$$Ax = b \tag{1.1}$$

The seeds of the idea behind structured low-rank approximation were sown by Hackbusch [78] when he proposed the concept of "Hierarchical matrices". Hierarchical matrices recursively approximate the low-rank off-diagonal blocks of a dense matrix. Although the initial advancements in the field were confined to obtaining a matrix-vector product, later advancements paved the way for a diverse set of algorithms such as matrix-matrix multiplication and factorization. Over time, advancements in matrix formats and algorithms have managed to bring down the cost of storing and factorizing structured dense matrix to close to $O(N)$.

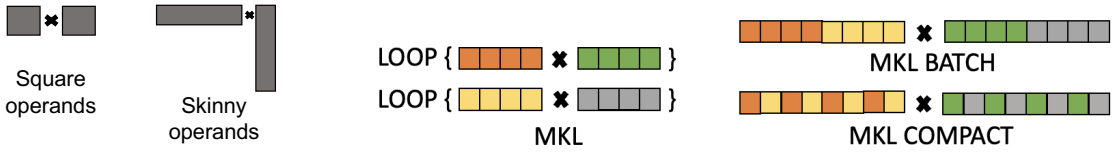
1.1 Historical background and significance

In many aspects, the contributions presented in this thesis go hand in hand with the history of software advancements done by the dense linear algebra community. Software for dense linear algebra has been so important to the development of high performance computing that Jack Dongarra, a pioneer of portability and performance in this field received the prestigious Turing award in 2022 for his work on the BLAS and LAPACK interfaces.

CPUs with multi-level cache hierarchies have become commonplace to overcome the limitations of memory bandwidth over processing speed. For example, the multiplication units of the A64FX can process almost 4 times more data per second than the peak bandwidth of the main memory. Given the wide variety of CPU vendors, portability and high performance became a priority for the community. Thus was born the idea of common interfaces for a core set of linear algebra routines which were useful across a wide variety of algorithms. This led to the standardization of the Basic Linear Algebra Subprograms (BLAS), which are the building blocks of all algorithms such as factorization and eigenvalue solvers. Any algorithm can be programmed as a combination of the BLAS routines in order to achieve high performance.

As the size of problems kept growing, so did the machines that were necessary to solve them. The development of distributed supercomputers by pioneers such as Seymour Cray prompted the community to respond with advancements in software for distributed dense linear algebra. The BLAS for distributed memory machines was recast as the Basic Linear Algebra Communication Subprograms (BLACS). The BLACS allow the creation of distributed algorithms with a focus on high performance and portability. The block cyclic data distribution was found to show the best trade-off between parallelism and load balance for all algorithms and was adopted as the standard data distribution. Therefore, the BLACS, and subsequently ScaLAPACK assume the block-cyclic data distribution for all algorithms. The BLACS serve an abstraction layer for the bulk synchronous communication employed by the Message Passing Interface (MPI). Other libraries such as Elemental were also proposed that made use of bulk synchronous distributed memory execution with different data distribution strategies.

The death of Moore's law and Dennard's scaling led to an increase in the available parallelism on modern computers. Rethinking the existing programming paradigms was necessary to take advantage of the new developments in hardware. Adaptability of the granularity of parallelism is very important to achieve portability and high performance given the diversity of modern hardware. This gave birth to several runtime system libraries such as openMP tasks, PaRSEC and starPU. These libraries are able to leverage the existing work done for the creation of BLAS and use it with asynchronous parallel execution. New linear algebra libraries such as SLATE, DPLASMA and Chameleon build on these runtime systems in order hide the communication and computation. This approach has been shown to be faster than the previously mentioned bulk synchronous approach.



(a) Shapes of the operands of low rank operations. The small dimension is of size 2^2 to 2^6 and the long dimension is of size 2^{13} to 2^{15}.

(b) Memory layout of MKL, MKL BATCH and MKL COMPACT. Each color represents a different matrix that forms a discrete operand.

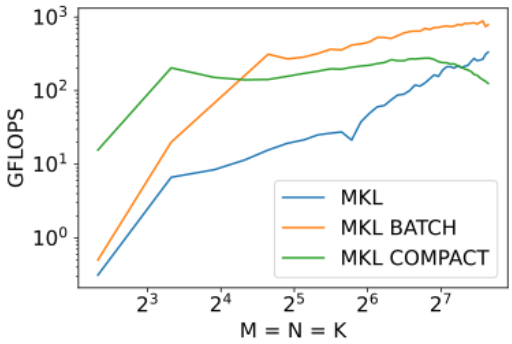
Figure 1.1: Shapes of the matrix operands found in low rank multiplication and various techniques of batching.

Scientific problems are getting larger than ever, and low rank approximation of the data is seen as a viable alternative to reducing the storage and compute costs of such large problems. Low rank computation promises drastic improvements in speed and storage costs by trading off accuracy. However, this accuracy is controllable according to the needs of the application, which makes low rank approximation an attractive alternative to full accuracy numerical calculations. The computations prevalent in low rank approximation typically involve computations between small and irregular matrices.

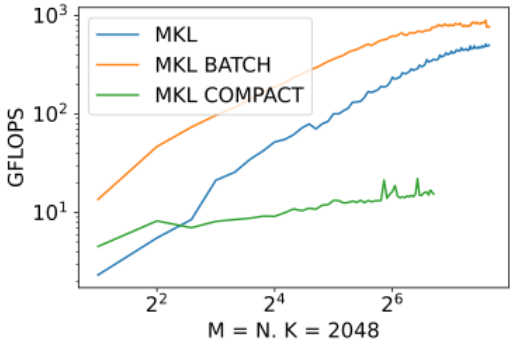
Batched computation [61] proposes to be a viable alternative for such matrices by performing many thousands of such computations together. Batching can improve the overlap between communication and computation in order to improve performance. Fig. 1.1(b) shows three ways of batching small and irregular matrix computations for the matrix multiplication operation involving four operands (each denoted by orange, green, yellow and grey) available in Intel's Math Kernel Library (MKL). Manual batching can be done by writing loops over calls to vanilla single-operand MKL functions. The memory of the operands in this case is kept in discrete arrays. MKL BATCH and MKL COMPACT are alternate implementations that interleave the memory of the operands into a contiguous array. MKL BATCH concatenates the arrays together for internal cache blocking whereas MKL COMPACT interleaves the operands into a contiguous array. Operations involving small and skinny matrix operands such as those shown in Fig. 1.1(a) are of particular interest to low rank computation (more discussion in Section 2.4).

Fig. 1.2 shows the performance of batched matrix computation on an Intel Xeon 6148 using 20 physical cores for the small and skinny matrices shown in Fig. 1.1(a). Our region of interest primarily lies in the region between 2^1 and 2^5 on the X axis for both Fig. 1.2(a) and Fig. 1.2(b). This region fails to achieve the peak performance for any of the batching techniques mentioned previously. This makes it necessary for the development of low rank multiplication routines shown in Chapter 4 that results in better performance of low rank multiplication.

The advancements in distributed dense linear algebra software have laid the foundation for extending low rank approximation algorithms on distributed memory machines. Factorization of large



(a) Batched small matrix computation.



(b) Batched skinny matrix computation.

Figure 1.2: Performance of small and skinny matrix multiplication using variants of MKL shown in Fig. 1.1 using 20 threads on Intel Xeon 6148. The peak performance of the CPU cannot be used for smaller matrix sizes.

dense matrices without low rank approximation can take several hours even on large supercomputers. Libraries implementing distributed factorization of low rank matrices have been shown to offer considerable speedup over their dense counterparts. For example, STRUMPACK implements the HSS-ULV factorization, an $O(N)$ factorization algorithm, using the ScaLAPACK library. As mentioned earlier, ScaLAPACK makes use of bulk synchronous execution. This reliance on ScaLAPACK hinders the parallel scalability of the HSS-ULV factorization, which motivates the development of a faster HSS-ULV algorithm using the PaRSEC runtime system.

1.2 Objectives

Algorithmic advancements have played a pivotal role in improving the usefulness of hierarchical matrices for a wide variety of real world problems, the numerical software available for efficient execution of these methods on modern parallel hardware has not been given adequate attention. Therefore, The hierarchical matrices community has been forced to fall back on linear algebra software that was originally designed for dense or sparse matrix computation. This has resulted in the inability of these methods to efficiently utilize hardware that is available on modern high performance computers. This thesis aims to address this problem of efficient execution on modern parallel hardware. The concrete objective of this thesis is as follows.

1. We develop a cache blocking technique for a batch of low-rank matrix multiplications, which is the core component in structured low-rank matrices.

2. We improve the parallelism of structured low-rank matrix factorization by utilizing a runtime system PaRSEC.

3. With these novel tools, we show that structured low-rank matrices can extract a larger portion of the capability of modern processors.

1.3 Contributions

This thesis makes contributions to improving the efficient implementation of algorithms that make use of hierarchical matrices.

1.3.1 Low rank matrix multiplication

Our first contribution, elaborated in Chapter 4 improves the matrix vector product by obtaining a 2x performance improvement in the low rank matrix multiplication algorithm. The matrix-vector product is useful for problems arising from the wave equation. The low rank matrix multiplication algorithm shown in Section 2.5.1 is a key computational kernel of the matrix vector product. We design efficient assembly code kernels that achieve the available machine efficiency by leveraging the ECM performance model (Section 3.3). Our portable implementation making use of parallel loops written in a high level language with architecture-specific assembly micro-kernels (Section 4.2) allows us to prove that our method out-performs vendor optimized BLAS libraries on the Fujitsu A64FX, Intel Xeon 6148 and AMD EPYC 7502. We show in Section 2.5 that our approach does indeed improve the performance of the matrix vector product for the block low rank matrix.

1.3.2 Distributed memory factorization using runtime systems

Our second contribution, elaborated in Chapter 5 improves the distributed memory factorization of hierarchical matrices arising out of 2D problems from electrostatics and geostatistics. Specifically, we shown in Section 5.3 that we achieve up to 2x improvement in weak scaling performance over the state-of-the-art in dense direct factorization on up to 128 nodes of Fugaku. We show in Section 5.3.1 that this improvement in speed can be achieved with similar or better accuracy than other state-of-the-art implementations with low rank approximation. The main reasons behind the improvement can be attributed to the use of the HSS-ULV algorithm (Section 2.6.2) with the PaRSEC runtime system (Section 3.1.4). The combination of an algorithm needing $O(N)$ computation and communication with an asynchronous distributed runtime system results in faster resolution of dependencies and results in better performance.

1.4 Outline of this thesis

This thesis is organized as follows. Chapter 2 is dedicated to the development of a strong mathematical foundation to establish the principles of partial differential equations (Section 2.1.2) and low rank approximation (Section 2.5.1). Our original contribution depends heavily on these concepts. We show the various algorithms (sections 2.5 and 5.2.5) and matrix formats (Section 2.4) of low rank approximation that have been developed in the literature so far. Since these methods heavily draw from the literature of dense matrix computation, we provide a quick introduction to some key

concepts in Section 2.3. We also elaborate on the pros and cons of the boundary element method (2.1.3.4) and compare it with other techniques for solving PDEs (sections 2.1.3.1–2.1.3.3).

As outlined in Section 1.1, the development of numerical software by the dense linear algebra community is closely tied to the developments in software for low rank approximation. We dive deeper into the details of the software and the parallel computer architectures in Chapter 3. In particular, we survey the development of cache blocked and batched libraries in sections 3.1.1 and 3.1.2 and asynchronous runtime systems for modern highly parallel computer architectures in Section 3.1.4. Cache blocking and batching are background knowledge for Chapter 4 whereas runtime systems are important for Chapter 5. Performance modeling and prediction are an important prerequisite for understanding Chapter 4. Therefore, Section 3.2 provides an overview of theoretical performance modeling, and Section 3.3 provides a detailed understanding of the Execution-Cache-Memory (ECM) model, which is an important part of the development of our cache blocking method in Chapter 4.

Chapter 4 shows the first part of the original contribution of this thesis regarding speeding up of the low rank multiplication algorithm with the use of cache blocking on CPUs as elaborated in Section 1.3. The main idea is shown in Section 4.2, followed by details of optimization of the assembly micro-kernels using the ECM model in Section 4.3. The results as elaborated in Section 1.3 are shown in Section 4.4.

Chapter 5 shows the second major contribution of this thesis. This is about factorization of Hierarchically Semi-Separable matrices on distributed memory machines using the ULV factorization and asynchronous execution as elaborated in Section 1.3. We introduce the relevance of this work compared to other state-of-the-art implementations in Section 5.1. The various process distributions and distributed memory execution schemes of our work and also that of the other works that we compare with is outlined in 5.2. We finally end this chapter with results of weak scaling on Fugaku in Section 5.3.

Finally, the conclusion in Chapter 6 briefly summarizes the main results of this thesis in Section 6.1. This is followed by a brief discussion of the implications of this thesis on other methods and research directions in Section 6.2. The limitations of each major contribution seen in Section 1.3 are listed in Section 6.3. We end this thesis with a note on possible future directions of research in Section 6.4.

Chapter 2

Low Rank Approximation and Factorization of Dense Matrices

In this chapter, we will review the mathematics that is important for gaining an understanding of low rank approximation of structured dense matrices and their associated factorization algorithms. We first begin with a review of various scientific domains which can be successfully modeled with Partial Differential Equations (PDEs) in Section 2.1. We then see some of the common numerical methods for obtaining the solution of real physical problems that can be modeled with PDEs in Section 2.1.3. Section 2.2 goes over some of the common methods for solving the sparse and dense matrices that arise out of PDEs, including the various algorithms that have historically been used to solve them. Section 2.4 introduces the basic concept of low rank approximation that can be used for reducing the factorization time of dense matrices arising out of certain classes of PDE solvers. The main theory behind hierarchical matrices is explained in sections 2.6 and 2.7. This chapter concludes with a summary of the previous work in the field of factorization of low rank matrices in Section 2.8.

2.1 Scientific computing and Partial Differential Equations

2.1.1 Ordinary Differential Equations

Various physical phenomena can be studied with the help of equations that help humans understand their properties. The primary intention of gaining such an understanding is to know the state of the physical phenomenon after it has been subject to a stimuli. For example, if one wants to know whether a particular type of alloy used for making a car will be able to handle a crash, it is helpful to model the car design and simulate its behaviour to various kinds of forces before manufacturing the car.

Some of these physical phenomena, such as the velocity of a ball falling from a given height in a perfect vacuum can be modeled analytically. A simple equation such as 2.1 is sufficient to understand the velocity of the ball at time t, with v_0 given at the initial velocity at time $t = 0$ and $v(t)$ as the

velocity at time t. However, when the final state of the physical phenomena depends upon the interactions of multiple variables relevant to the medium, directly writing down the answer of this physical phenomena is not possible, unlike the case with the speed of the ball through a vacuum. Such problems, where the result depends upon the properties of an irregular physical medium and its interaction with external forces requires the use of numerical solutions. Numerical solutions typically involve performing many thousands of calculations, which is best done with a computer. This requirement of studying physical phenomena using numerical solutions has given birth to the field of scientific computing.

$$v(t) = v_0 - gt \tag{2.1}$$

Many physical phenomena such as the deformation of metals, electrostatic forces on molecules or the behaviour of radar waves bouncing off the surface of an aeroplane can be effectively studied using differential equations. Unlike simple laws of physics operating on point forces or mass where the entire domain is lumped together into a single point, differential equations allow studying large domains where the properties of the domain change between individual nodes of the domain [130, Chapter 3]. Differentiating the domain into tiny pieces allows for taking into account these changing properties and allows us to assemble the properties of the entire domain together. Finding a state of equilibrium for this domain allows us to precisely model the behaviour of this domain when it is stable for some condition.

Finding such solutions to differential equations is much harder than being able to write down the solutions for purely analytical equations such as Eq. 2.1. Gilbert Strang [133, Section 3.1] calls this akin to drilling for oil vs. picking up gold. For example, if you want to find the solution of quadratic equation $ax^2 + bx + c = 0$, it is a simple matter of plugging in the values of the variables into the formula for the sum and product of the roots of the equation. However, if you are faced with a differential equation of the form $dy/dt = sin(yt)$, this means that the solution of this equation is another function with a slope of $sin(yt)$. Finding this function is a non-trivial task.

The solution of the differential equation will always be another function. This function must have two conditions:

1. It must exist and be continuous at a given initial value.

2. It must be unique for all values including the initial value.

Any function that satisfies the above two conditions for a given differential equation can be said to a solution of the differential equation. Of course, there are cases where the differential equation must be limited by some value in order to ensure that the solution is a continuous function. Now the question remains - how does one go about finding an appropriate function that yields a satisfactory solution to a given differential equation?

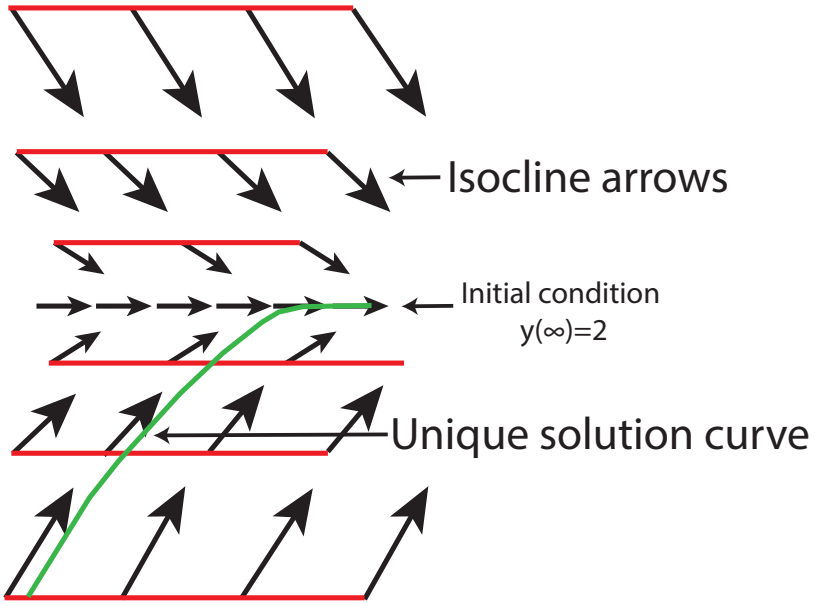


Figure 2.1: Obtaining a graphical solution for a simple differential equation in Eq. 2.2.

One way to think about this is via an analogy of points and curves. Since the slope of the solution is already known at every point, you can draw arrows in the solution space that have the slope of the solution function. The initial condition of the differential equation is where the solution will have a constant slope since this is where the equation is most stable. All the arrows will be pointing towards this initial state.

$$f(t, y) = \frac{dy}{dt} = 2 - y \tag{2.2}$$

For example, consider the differential equation in Eq. 2.2, whose solution is shown in Fig. 2.1. We want a solution vector which will have a length of dt along the t axis and $(2 - y)dt$ along the y axis in the $t - y$ plane so that the resulting vector will have a slope of $2 - y$. Multiple vectors (or arrows) can be drawn on a plane in the direction of the solution. The constant term in the solution of a differential equation is reflected in the fact that there are multiple arrows along an isocline surface (or a line with the same slope shown in red). A unique curve (or function) shown in green running through successive such arrows forms the solution of the differential equation. All the arrows move toward the initial condition, which is the state of equilibrium of this equation. Therefore, the solution of Eq. 2.2 is given by $y = 2 + Ce^{-t}$ where $y(\infty) = 2$. So in order to solve a differential equation, a graphical technique is to first define an initial condition where the solution will be constant. Then you draw curves using points and arrows such that a unique solution passes through that curve.

Certain physical domains can be modeled well with initial values. These are phenomena where the domain depends on only a single condition and its future state only depends on this initial

value. However, there are other physical phenomena arising out steady-state problems that only attain equilibrium at a boundary and not an initial state [133, 7.3]. This requires extending the definition of initial values for differential equations and also include boundary conditions. Boundary conditions state that the state of the system is constant at a given boundary condition. The boundary condition can either be an essential boundary condition (or 'Dirichlet boundary condition') or natural boundary condition (or 'Neumann boundary condition'). The Dirichlet boundary condition allows us to specify at what condition a node of the medium is grounded. The Neumann boundary condition specifies the value of the derivative of a normal to the surface. A state of equilibrium of the domain can then be specified using the Dirichlet or Neumann boundary conditions which allows us to find a stable state of the resulting differential equation of the domain. This state of equilibrium then yields solvable equations which reveal the properties of the domain.

Therefore, studying a physical medium via a differential equation requires first defining the medium, then defining a differential equation that demonstrates the forces on the medium, and finally defining boundary conditions that show when the system is in a state of equilibrium. The reaction of this physical medium to an external force or potential can be calculated after finding the solution of the differential equation. Unlike equations that deal with matter lumped together into a single node such as the aforementioned calculation of the speed of a falling ball in a perfect vacuum, solutions to differential equations must be obtained by fitting a suitable curve on a set of points that corresponds to the solution of the differential equation [133, Section 3.1].

2.1.2 Partial Differential Equations

The Ordinary Differential Equations described in Section 2.1.1 are useful for physical phenomena that are one dimensional. However, nature frequently throws more complex problems at us involving more than one dimension, which makes it essential to have mathematical models involving multiple dimensions. Partial Differential Equations (PDEs) [133, Section 7.4] become a useful tool in this case, which allows one to differentiate across a dimension by keeping other dimensions constant. Partial Differential Equations can be found pretty much everywhere in nature and are used extensively for modeling phenomena ranging from weather simulations to drug discovery.

At it most basic form, a PDE used for modeling 3D phenomena can be shown as Eq. 2.3. The expression u is an equation in x, y and z and the ∇ is a shorthand for showing partial differentiation in each dimension. The dot product is then adding these partial differentiations together. The expanded form of Eq. 2.3 is shown in Eq. 2.4. The u_x, u_y and u_z show that we are taking a partial differentiation x, y and z whereas the other variables are kept constant, respectively.

$$\nabla \cdot u = 0 \tag{2.3}$$

$$\left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right) \cdot (u_x, u_y, u_z) = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} \tag{2.4}$$

Physical phenomena such as electrostatic fields cannot be studied when they are in a state of motion. They can only be studied in a state of equilibrium. This means that the system is being studied in a state where energy is conserved. The boundary conditions and initial condition are useful for imposing this steady state equilibrium on the PDEs describing the system. In order to express the Dirichlet and Neumann boundary conditions, a second order PDE becomes a necessity [131, p240]. Examples of the second order PDE are Laplace and Poisson equations shown in Eq. 2.5 and Eq. 2.6, respectively. The Laplace equation is extensively used for studying electrostatic fields where the source or sink is zero. Another such PDE is the Poisson's equation shown in which is useful when the source or sink is non-zero.

$$\nabla^2 \cdot u = \left(\frac{\partial^2}{\partial x^2}, \frac{\partial^2}{\partial y^2}, \frac{\partial^2}{\partial z^2}\right) \cdot (u_x, u_y, u_z) = \frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_z}{\partial z^2} = 0 \tag{2.5}$$

$$\nabla^2 \cdot u = f \tag{2.6}$$

If the equation involves time, initial conditions are also involved. An example of a time-dependent PDE is the heat equation [130, 6.4] shown in Eq. 2.7. Various other time-dependent PDEs such as Maxwell's equations, Schrodinger's equation, and the Wave equation have been developed in the past for studying a large variety of physical phenomena.

$$u_t = \nabla^2 \cdot u \tag{2.7}$$

PDEs such as Laplace's equation in Eq. 2.5 show solutions for certain simple classes of boundary conditions [133, 3.4]. For example, the solution $u(x, y) = x^2 + y^2$ will exist for the Laplace equation where the domain is a simple quadratic curve. The solution can also be obtained by fitting an analytical expansion such as a Taylor series onto the real solution as long as the sum converges. While analytical solutions are mathematically exact and very fast to compute, they only work for very simple problems. Real world problems show far more complex geometry than what can be modeled with a continuous curve. Analytical solutions using a Taylor series will frequently not converge for such problems. This makes it necessary to have numerical solutions to partial differential equations. Numerical solutions allow us to obtain the properties of each specific point in the domain and compute the approximate solution of the PDE from these real observations. Although numerical methods are generally less accurate and require more computational power than analytical methods, they have been consistently found to be more useful in practice.

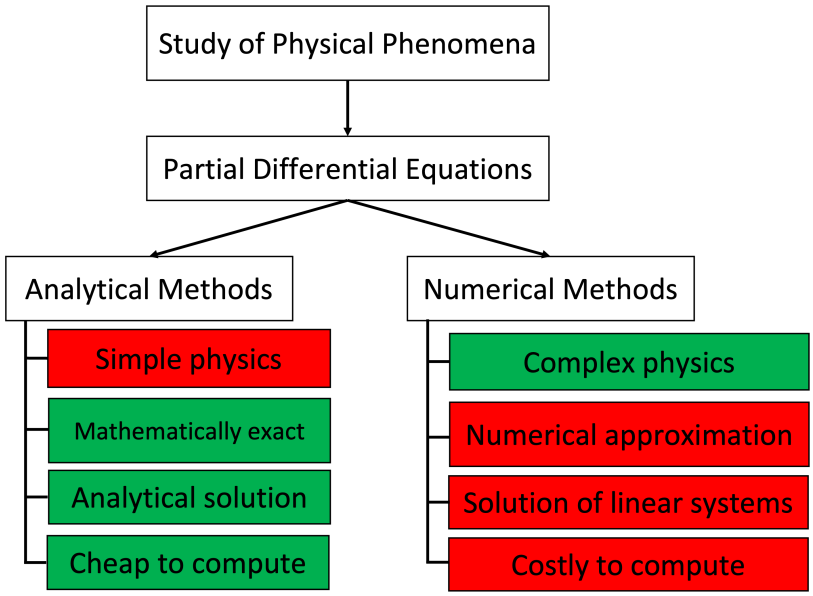


Figure 2.2: The techniques for obtaining solutions to PDEs can be broadly categorized into analytical and numerical methods. While analytical solutions are very accurate and fast, they are inflexible and usually not applicable to real world problems. Numerical solutions are most useful for real world problems in spite of requiring more compute power and time for calculation.

Fig. 2.2 summarizes the above discussion broadly and shows the trade-offs between analytical and numerical solutions to PDEs. The next section discusses some of the important methods for obtaining numerical solution of PDEs and their various pros and cons.

2.1.3 Numerical solutions of Partial Differential Equations

Discretization is the first step for obtaining a numerical solution of a PDE. It reduces the continuous domain of the PDE into a discrete and bounded domain that can be solved with the help of a computer. Discretization can be thought of as assigning magnitudes to each vector in the vector field of the domain in the direction of the vector at that point. The Finite Element Method, Finite Difference Method, Spectral Method [137] and Boundary Element Method [97] are some of the popular discretization strategies for obtaining the numerical solution of PDEs. All the numerical techniques result in the generation of a matrix A that quantifies the partial derivative of a single point with respect to other connected points in the domain. The matrix A is a system of linear equations of N points where N is the number of points in the domain. The state (or magnitude) of each point in the domain can then be expressed in the form of a vector x. When the matrix A is applied to this vector x, the new state of the domain can be found in the result vector b. This operation can be expressed as Eq. 1.1.

For problems involving time, the state of the system at a given time can be expressed as the vector x, and the final state of the system can be expressed as vector b. For problems involving

structural mechanics, the vector b represents forces and x represents displacement of the points in the domain. Calculating the vector x using Eq. 2.8 allows for obtaining the final magnitudes of each point as a result of the behaviour of the physical domain. However, factorization of A or using an iterative solver is a much cheaper and numerically stable approach for calculating x. Finding solutions to PDEs using such matrix factorization techniques is at the heart of scientific computing.

$$x = A^{-1}b \tag{2.8}$$

The structure of the matrix changes according to the method used for discretization, and so does its stability and accuracy. Each method has its own pros and cons, and choosing the right method is a matter of experience, the geometry of the domain, the required accuracy and permissible use of computational resources. We will now see some of the popular techniques of solving PDEs and their pros and cons.

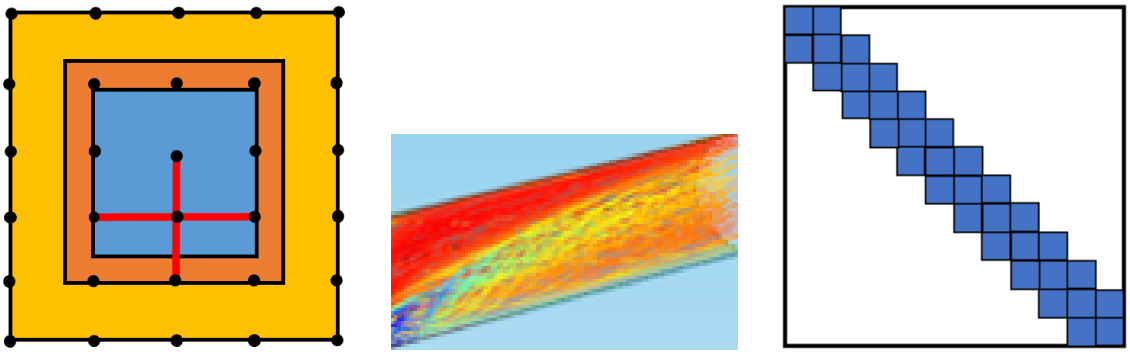
2.1.3.1 Finite Difference Method

The Finite Difference Method (FDM) is the most simple discretization method, where derivatives are approximated through differences between the values on the grid points. Fig. 2.3(a) shows how a simple 2D domain with varying density (the different density is shown in different colors) that can be modeled using the FDM. The domain is first sub-divided into discrete points (the black dots) which are uniformly spaced apart. All points are connected to the points above and below them, with the exception of the points on the boundary, which do not have any connection inside the domain. Such graph structures are called a "grid" or "mesh", and the nodes for this graph are called "grid points" or "mesh points". Applications such as incompressible fluids shown in Fig. 2.3(b), which show uniform distribution of the domain without notches, depressions or other deformities are suitable applications of the FDM.

Consider the point in the middle of red cross in Fig. 2.3(a). The derivatives at this point in the x direction can be calculated using the forward and backward difference between the points ahead and behind this point, respectively[130, 5.1]. The same can be performed in the y direction by taking the difference between the point above and below. If we name this point $u(x)$ and use a uniform distance h between all the points, the second order derivative in the X direction can be denoted using Eq. 2.9. A similar equation can be derived for the Y axis.

$$\frac{\partial^2 u}{\partial x^2} = \frac{(x+h) - 2u(x) + u(x-h)}{h^2} \tag{2.9}$$

Obtaining the second order differentiation as shown in Eq. 2.9 results in the generation of a band diagonal sparse matrix as shown in Fig. 2.3(c). For a two dimensional problem, the structure is that of a '5-point stencil', where each row of the matrix which is not close to the edge has exactly 5 elements - one for the point in question and 4 for the points in its 4 unique directions. Although this



(a) The physical domain expressed as uniformly spaced points for discretization with the Finite Difference Method.

(b) A sample application of the Finite Difference Method for modeling fluid flows.

(c) Structure of the matrix resulting from the discretization of the Finite Difference Method (FDM). This is a band diagonal sparse matrix.

Figure 2.3: Expressing FDM with uniformly distributed points and sample application used in modeling incompressible fluids. It results in a matrix structure as shown on the right.

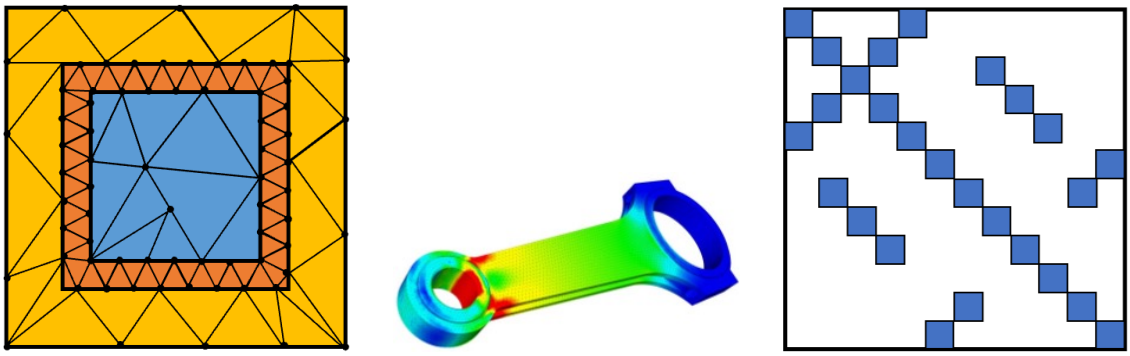
matrix can get very large due to the presence of millions of points in the domain, various techniques such as nested dissection shown in Section 2.2 can be used for efficiently building solvers for such matrices.

2.1.3.2 Finite Element Method

The Finite Element Method (FEM) [130, 5.4] [133] is one of the most popular and extensively used methods for many problems in structural mechanics and fluid dynamics. It is more flexible to use than FDM since it can discretize the domain using triangular meshes, which allows it to be used with complex-shaped boundaries, and hence finds more adoption in industry. Many important software products such as ANSYS and openFOAM have been specifically designed for providing engineers and designers with flexible ways of developing solvers using FEM.

Fig. 2.4 shows the sub-divisions of a sample domain made by the FEM in Fig. 2.4(a) into triangles with black dots at each corner. The triangles have different shapes depending on the location of the domain that they are discretizing. This is different from the equally spaced points throughout the domain in the FDM as seen in Fig. 2.3(a). The added flexibility allows the FEM to change the density of points for different parts of the domain, making it useful in a wider variety of applications than FDM.

The basic principle of the FEM begins with the observation that approximation of a PDE on a domain need not consider all the points in the domain if one resorts to using the weak form of the PDE over the strong form. The strong form of the PDE, such as that shown in Eq. 2.5 or Eq. 2.24 requires that the solution of the PDE is satisfied at every point in the domain. However, this is challenging for real world scenarios due to the need of computation and complexity of modeling every point in the domain. The weak form of the PDE on the other hand, is able to relax this requirement



(a) Grid of FEM on a domain with varying densities. The use of irregular polygons permits the admission of piecewise polynomials for discretization of the domain.

(b) A sample application of FEM used for studying the impact of stresses on a structure.

(c) Structure of the matrix arising out of the discretization using FEM. This is a sparse matrix.

Figure 2.4: The grid resulting from FEM for a sample application. The rightmost figure shows the structure of the resulting stiffness matrix.

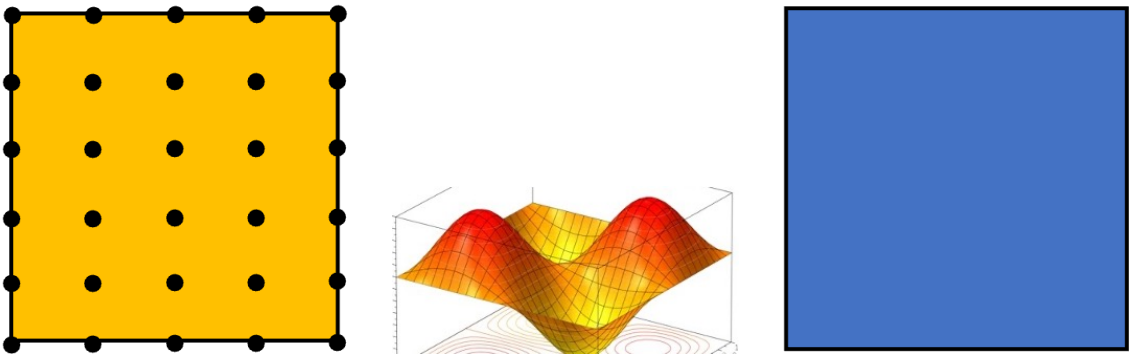
by making use of *piecewise polynomials* instead of points. The polynomials serve as a substitute for approximation of every point in the domain. This means that the FEM can use polynomials for performing interpolation at irregular intervals and therefore adapt better to structures which do not exhibit regularly spaced points. This is better suited for many industrial applications.

Two main principles are used for obtaining the finite element discretization [132, 23]: Rayleigh-Ritz [126, 3.10] discretization and the Galerkin discretization. Both principles work on the concept of combining together polynomials of the form $U(x)$ as shown in Eq. 2.10. The idea is to approximate $U(x)$ using the polynomials $\phi_i(x)$ and 'weights' U_i in order to obtain the best approximation of the curve $u(x)$ that defines the solution of the PDE. The polynomials $\phi_i(x)$ are called 'trial' functions and can vary depending on the domain and the accuracy that one desires.

$$U(x) = U_0 \cdot \phi_0(x) + \cdots + U_n \cdot \phi_n(x) \tag{2.10}$$

The Rayleigh-Ritz criterion was proposed before Galerkin's method and works on the principle of minimization of energy of the system. Each 'piece' in the curve is approximated using a separate equation that must be determined manually. The Rayleigh-Ritz method requires the generation of a polynomial for each 'piece' that connects the curve. This has to be typically done separately for every piece. The Galerkin method, in contrast, is more flexible and allows both the generation of the polynomial and the subsequent solution by the computer.

Typically using linear trial functions is sufficient in most real world problems, however they can also be extended to n^{th} degree polynomials. The linear piecewise polynomials are typically triangles of the form $a + bx + cy$ for 2D domains. These triangles can be seen in the sample domain in Fig.



(a) Grid of the spectral method. Note that the domain must be uniform.

(b) Application of the spectral method in the study of wave propagation.

(c) Dense matrix arising out the spectral method.

Figure 2.5: The grid resulting from Spectral Method and a sample application.

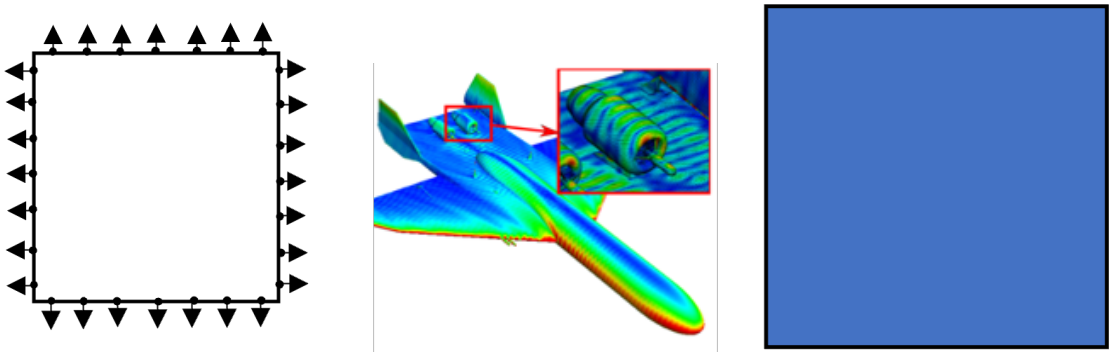
2.4(a). However, they can also be straight lines or parallelograms. It really depends on how much complexity you're willing to trade-off for speed or accuracy.

2.1.3.3 The Spectral Method

The spectral method are yet another type of discretization method that is different from both FDM and FEM. As the name suggests, it depends on the use of the Fourier series to discretize the partial differential equation. Due to the nature of the Fourier series, the discretization error of spectral methods have an exponential convergence rate. This is a huge advantage over FDM and FEM, which typically have first or second order convergence of the discretization error. First order convergence means that if you half the distance between the points, the error decreases by 1/2. Second order convergence means that if you half the distance between the points, the error decreases by 1/4. Exponential convergence means that the discretization error decreases much faster and cannot be expressed by a constant like 1/2 or 1/4. Despite the advantage in convergence rate, spectral methods suffer from the restriction that all points must be spaced equally. This restriction comes from the nature of the Fourier transform, which requires the points to be spaced equally. Fig. 2.5 shows the grid resulting from the spectral method. It can be seen in Fig. 2.5(a) that the domain is completely uniform and that all the points are spaced equally. It is used for applications such as studying the propagation of waves as shown in Fig. 2.5(b). Unlike the FEM and FDM, the spectral method results in the generation of a dense matrix as shown in Fig. 2.5(c) that contains the magnitudes of the differences.

2.1.3.4 The Boundary Element Method.

The Boundary Element Method (BEM) is useful in places where previously methods either cannot be used (such as exterior problems with infinite domains), lead to extremely large matrices or require higher accuracy. Unlike the previously mentioned methods, the BEM assumes that the domain



(a) Grid of the Boundary Element Method. Note that BEM assumes that the medium is homogeneous. Therefore, points are considered only on the surface.

(b) A potential application of the BEM is for studying the behaviour of radar waves bouncing off a plane.

(c) Dense matrix arising out of discretization by the Boundary Element Method.

Figure 2.6: The grid resulting from Boundary Element Method and a sample application.

is homogeneous and that points exist only on its surface. Fig. 2.6(a) shows that the domain is completely white (homogeneous) and that there are normal forces going into the domain at points on the surface. This is a rather different formulation of the problem than what can be observed in all previously mentioned methods, where points within the domain were considered for discretization. The vectors on the boundary of the domain point outwards. In BEM, the orientation of the normal vector is actually important. The orientation of the normal vector determines whether a particular term is positive or negative in the formulation. In common formulations, the direction of the normal vector is often taken from the boundary of the domain to its complement (outside). Example: Exterior problem $\rightarrow$ Inward from the boundary. Interior problem $\rightarrow$ outward from the boundary

Fig. 2.6 shows a sample application such as modeling the radar waves bouncing off an airplane in Fig. 2.6(b) and the resulting dense matrix of the interactions between the particles on the boundary in Fig. 2.6(c). The BEM is formulated by casting the partial differential equation in the domain as an integral equation on the boundary of the domain.

2.2 Common methods of solving a system of linear equations

Fig. 2.7 broadly shows the methods that have been developed for solving the system of linear equations arising from partial differential equation discretization techniques shown in Section 2.1.2.

2.2.1 Sparse matrices

The FEM and FDM shown above result in spare matrices which can be solved with various solvers such as multi-grid, nested dissection or variants of Krylov subspace method such as conjugate gradient (CG) and generalized minimal residual (GMRES) methods. The multi-grid method operates on a hierarchy of coarse to fine grids through restriction and interpolation between the coarse and fine

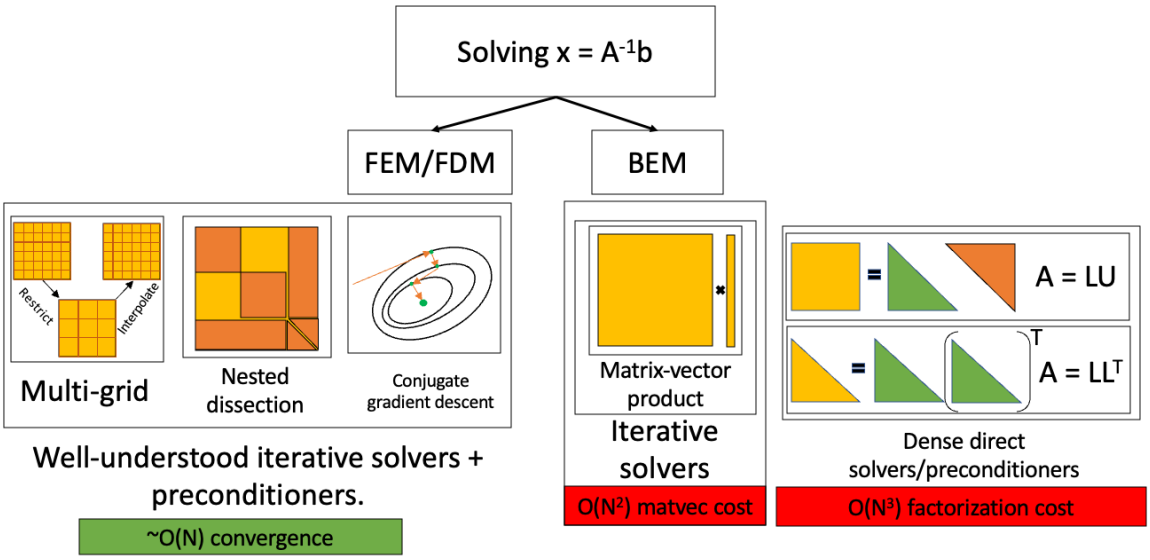


Figure 2.7: Various solvers proposed in the literature for sparse matrices arising out of FEM and FDM, and dense matrices arising out of BEM. Obtaining the solution to the system of linear equations such as $Ax = b$ is at the heart of scientific computing.

grids. Operating on coarse grids allows the problem to be solved quickly but roughly, while operating on the fine grids gives higher accuracy but takes a long time. By restricting and interpolating between the coarse and fine grids, multi-grid methods can get the best of both worlds. Multi-grid can be used for FDM and FEM, and there are also algebraic variants that do not operate directly on the grid but rather on the matrix directly. The nested dissection method [130, 5.1] reduces the fill-ins the direct factorization of the sparse matrix by pivoting the matrix. Nested dissection will recursively dissect the domain into successively smaller parts until the need for fill-ins is eliminated or drastically reduced. The conjugate gradient method is an iterative method that performs gradient descent in the Krylov subspace. It works by successive orthogonalization and correction of the target vector in order to bring it closest to the solution. The convergence of the aforementioned iterative solvers can be made faster with the use of preconditioners [125] which work on the principle of partially factorizing a matrix. Matrices arising from FDM and FEM are sparse in nature. Most such solvers and preconditioners used in sparse matrix methods take advantage of this sparse structure of the matrix. Since it so happens that nature does not throw sparse matrices with completely random structures at us, a lot of research has gone into exploiting the structure shown by various applications in order to enhance the efficiency of the solutions.

2.2.2 Dense matrices

Matrices arising from the Boundary Element Method (BEM) require the use of iterative and direct solvers for dense matrices. Such matrices take up $O(N^2)$ space for storage. Iterative solvers making use of dense matrices require $O(N^2)$ operations for every matrix-vector product, and dense direct

factorization using LU factorization or Cholesky factorization costs $O(N^3)$. Although structure is hard to find in random dense matrices, those arising out of the BEM can be approximated to reduce the time of the matrix vector product and dense direct factorization. However, research into solving systems of linear equations expressed as dense matrices has been scant compared to the vast amount of research on sparse matrices. Therefore, we make a conscious choice of focusing on this area and advance the science further in this domain.

2.3 Direct factorization and solution of dense matrices

This section serves as an introduction to the development of important factorization algorithms in dense linear algebra such as the LU and Cholesky factorization. The LU factorization [134, Section 2.6] and Cholesky factorization [130, Section 1.3] are fundamental algorithms used for the factorization of non-symmetric and symmetric positive definite dense matrices, respectively. These algorithms factorize the dense matrix into a reduced row echelon form, which is a more stable form of solving an RHS vector than computing the inverse of the matrix. These algorithms are so important for linear algebra that the HPL benchmark [117] uses the distributed LU factorization for benchmarking the world's fastest supercomputers in the Top500 list.

2.3.1 LU factorization

The LU factorization arises from the need to solve a system of linear equations such as Eq. 1.1 without making the system unstable. While the LU factorization can lead to inaccurate solutions for certain types of linear equations, it has been found to be one of the most effective techniques for solving a system of linear equations in practice.

Consider a very simple system consisting of three equations [132, Section 2.2] as shown below:

$$2x + 4y - 2z = 2 \tag{2.11}$$

$$4x + 9y - 3z = 8 \tag{2.12}$$

$$-2x - 3y + 7z = 10 \tag{2.13}$$

One of the simplest 'back-of-the-envelope' ways of solving this system of equations is to convert the above system into something that has this form:

$$\begin{aligned} 2x + 4y - 2z &= 2 \\ 1y + 1z &= 4 \\ 4z &= 8 \end{aligned}$$

This means that one can work their way backwards from $4z = 8$ and obtain the values of x, y and z after performing a series of substitutions. Since we are going backwards from the bottom-most

equation to the top, this is called the *backward substitution*. If the coefficients of the above reduced equations are put into a 3x3 coefficient matrix, it would have a row echelon form like so:

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix}$$

In order to obtain the upper triangular form from eqs. (2.11)–(2.13), it is necessary to first introduce zeros in the column below the diagonal element. This is done by introducing the multipliers l_{ij} for the elements in the strict lower triangle of the coefficient matrix. The lower triangular matrix of such multipliers is termed the *elimination matrix* E [133, Section 2.3]. Finding these multipliers requires division of the column below the diagonal with the element at the diagonal element A_{ii}. For reasons of stability, the largest element in the column of the A_{ii} is used as the *pivot* element for generating the multiplier. This requires that the rows of the system of equations be swapped in order to bring the equation with the largest element in the position of the diagonal. This procedure is termed as *partial pivoting*. If the columns of the coefficient matrix are also swapped along with the rows, it is known as *full pivoting*.

The elimination matrix E consisting of the multipliers can be written as L which multiplies the upper triangular matrix U to produce the original matrix A. Thus the matrix A is factorized into a lower triangular matrix L and upper triangular U. If pivoting has been performed, this can be further multiplied by a pivot matrix that will reshuffle the linear equations.

When performing the substitution procedure, Eq. 1.1 can now be expressed as Eq. 2.14. The step to compute x would consist of a *forward substitution* step which results in $Ux = L^{-1}b$. Since the matrix L consists of the multipliers of the upper triangle U, the operation $L^{-1}b$ can be performed without computing the inverse of the matrix L. Instead, just changing the $+l_{ij}$ to $-l_{ij}$ off the main diagonal and performing a matrix multiplication is sufficient. This can be performed using specialized BLAS procedures such as TRSM.

$$LUx = b \tag{2.14}$$

The LU factorization can be unstable if the off-diagonal elements have very large differences between them or are very close to zero. The procedure will breakdown if a non-zero pivot element cannot be found. Moreover, the use of IEEE754 standard for computing the floating point operations can introduce further errors due to limitations of the floating point format.

2.3.2 Cholesky factorization

The Cholesky factorization [130, Section 1.3] is very similar to the LU factorization shown in Section 2.3.1. An important distinction is the fact that the Cholesky factorization is exclusively applicable to symmetric positive definite (SPD) matrices. SPD matrices most often arise in nature from the

discretization of PDEs, which makes the Cholesky factorization important from a practical point of view. The pivots of an SPD matrix are all positive. This property is important because it identifies the presence of a minimum. For an SPD matrix, the elimination matrix E is a symmetric transpose of the upper triangle U. The pivots on the diagonal D can be separated out into a diagonal matrix thus allowing the expression of the matrix A as Eq. 2.15.

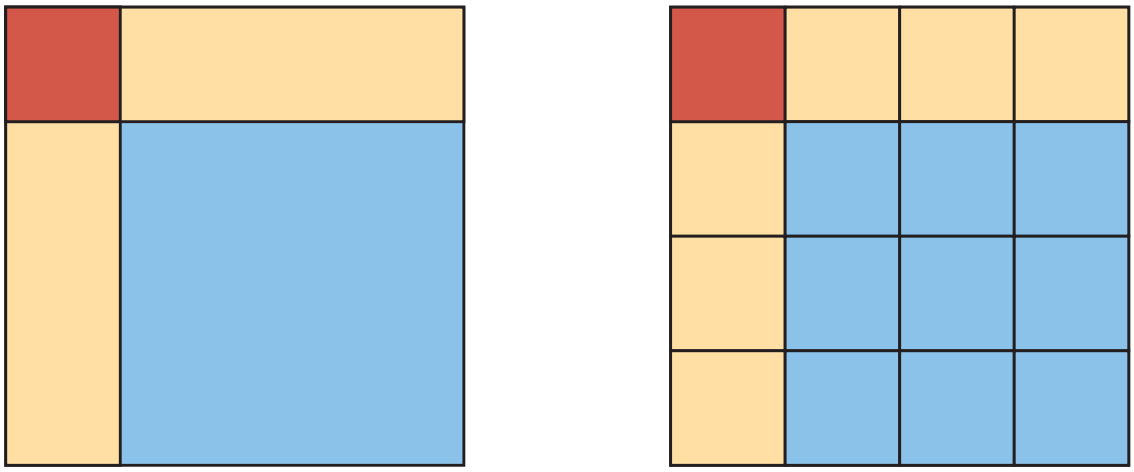
$$A = L \cdot D \cdot L^T \tag{2.15}$$

The Cholesky factorization is based on the idea that the square roots of the diagonal elements can be absorbed into the lower triangular L to form the matrix $\bar{L} = L \cdot D^{1/2}$. Therefore the lower triangular elimination matrix E can be used for expressing the original matrix A as a product of a lower triangle matrix $\bar{L}$ and its transpose as shown in Eq. 2.16. At the cost of computing the square root, we can obtain the factorization of the matrix A in about half the cost and space occupied by the LU factorization.

$$A = \bar{L} \cdot \bar{L}^T \tag{2.16}$$

2.3.3 Block LU/Cholesky factorization

The LU factorization as shown in Section 2.3.1 is computed on every element of the coefficient matrix. However, performing load and store operations on single elements can be extremely slow on modern computer architectures that make use of multi-level cache hierarchies. Early developments by the computational linear algebra community proposed the use of cache blocking so that the data being worked upon can reside in the cache of the CPU. The Basic Linear Algebra Subprograms (BLAS) were born, which were used for building the LAPACK library. LAPACK made use of *panel updates* for the purpose of using level 3 BLAS routines which can then be used as a portable software abstraction for achieving high performance. The panel update procedures can be further classified into right-looking or left-looking factorization, which differ in their IO cost and lead to differences in execution speed depending on the architecture. The left-looking variant is typically used for out-of-core execution where the entire matrix cannot fit onto the main memory of the machine and parts of it have to be streamed from the main storage [58]. Although the pivoting procedure requires the swap of the entire row of the coefficient matrix, the panel-update based LU factorization implementation will first swap the rows only of the panel under consideration and then propagate the changes to the sub-matrix as the factorization proceeds. Fig. 2.8(a) shows an illustration of the right-looking LU factorization. The red block shows that the LU factorization of that square diagonal block has been completed. The triangular updates from the lower and upper factorized portions of this diagonal block are then applied to the yellow panels that belong to the row and column, respectively. The row swapping is first performed exclusively in the yellow panel below the red diagonal block, and



(a) Right looking LU factorization with panel updates.

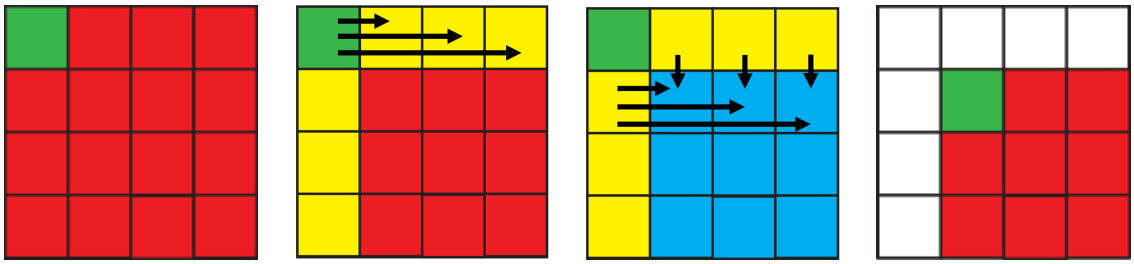
(b) Right looking block LU factorization.

Figure 2.8: Variants of the right-looking LU factorization using panel and block updates after the first diagonal block has been factorized. The red block shows that the diagonal block has been factorized with an LU factorization. The yellow shows triangle solves, and the blue blocks show trailing sub-matrix updates.

then propagated to the blue block that represents the trailing sub-matrix. The trailing sub-matrix is then updated with the schur's complement from the yellow panels.

The limitations of Moore's law have forced hardware designers to come up with clever ways of enhancing the available parallelism in their designs. An increased number of physical cores on CPUs, combined with the rise of heterogeneous computers using GPUs means that software must be modified to be more flexible with available parallelism. This increased focus on parallel execution means that panel-based factorization is no longer considered the most portable technique for achieving peak hardware utilization. A viable new alternative is the use of block-based factorization algorithms. Fig. 2.8(b) shows an illustration of the LU factorization implemented as a block algorithm. The sub-division of the matrix into a grid of blocks means that a software implementation can flexibly tune the granularity of the parallel execution in order to achieve the best possible throughput. This also allows the same algorithm to used on accelerator devices such as GPUs. Modern linear algebra libraries such as SLATE [69] or DPLASMA [31] implement some variant of the block LU factorization.

A limitation of such block algorithms is that pivoting cannot be applied directly to the row and column unlike in the panel update. This limitation can be partially overcome by propagating the pivots with block updates [7] and modifying the kernels of the factorization. The non-pivoted block LU factorization makes use of the GETRF, TRSM and GEMM kernels. However, the pivoted block LU factorization uses variants such as TSTRF, GESSM and SSSSM that help propagate the pivots along the block factorization. Although this approach accomplished the objective of changing the



(a) Factorize the top-left block shown in green using a regular LU factorization routine such as GETRF.

(b) Update the off-diagonal blocks shown in yellow using a triangular solve, typically using a routine such as TRSM.

(c) Update the trailing sub-matrix using schur's complement between the off-diagonal yellow blocks and the blue blocks in the trailing sub-matrix, typically using a routine such as GEMM.

(d) Repeat the steps shown in Fig. 2.9(a), 2.9(b), 2.9(c) for the next diagonal blocks and subsequent sub-matrix.

Figure 2.9: Steps of the block LU factorization algorithm without pivoting.

granularity of the computation, the pivoted block LU approach has been shown to lead to a loss of accuracy. However, in case of structured dense matrices arising from the BEM, the ordering of the geometry can ensure that the largest pivot element is already present in the diagonal before the factorization procedure begins. This means that pivoting is not necessary for block factorization of matrices arising from the BEM. This allows us to base the factorization of low rank matrices arising from structured dense matrices on the block LU factorization.

Fig. 2.9 shows the first 4 steps during the factorization of a block dense matrix split into 4x4 blocks. The full block LU factorization algorithm is shown in Alg. 1. The first step shown in Fig. 2.9(a) shows that the diagonal block must be factorized using an appropriate LU factorization routine such as GETRF, after which the inverse of the factorized lower and upper blocks of the diagonal block can be applied to the off-diagonal blocks as shown in Fig. 2.9(b). The third step shown in Fig. 2.9(c) leads to an update of the trailing sub-matrix. Finally, the procedure is repeated starting from the next diagonal block as shown in Fig. 2.9(d). If this algorithm is implemented using a task-based runtime system such as openMP tasks, PaRSEC or starPU, the available parallelism on the hardware can be used optimally. The principles of the LU factorization can be easily extended to the Cholesky factorization, which will require about half the computation of the LU.

2.3.4 Block forward and backward substitution

The substitution procedure for the LU factorization as shown in Section 2.3.1 can be extended to use the block LU algorithm shown in Section 2.3.3. This can be done by splitting the vector x and b from Eq. 2.9 into blocks in a similar way as the dense matrix. Alg. 2 shows the blockwise forward substitution procedure for obtaining $y = L^{-1}b$, followed by the blockwise backward substitution procedure for obtaining $x = U^{-1}y$ shown in Alg. 3.

Algorithm 1: Block LU factorization algorithm.

Input: A, NB
Result: $lu(A)$

for $i \leftarrow 1 \dots NB$ **do**
  $L_{ii} \cdot U_{ii} \leftarrow GETRF(A_{ii})$
  for $j \leftarrow i+1 \dots NB$ **do**
    $A_{ij} \leftarrow L_{ii}^{-1} \cdot A_{ij}$
    $A_{ji} \leftarrow A_{ji} \cdot U_{ii}^{-1}$
  end
  for $j \leftarrow i+1 \dots NB$ **do**
    for $k \leftarrow i+1 \dots NB$ **do**
      $A_{jk} \leftarrow A_{jk} - A_{ji} \cdot A_{ik}$
    end
  end
end

Algorithm 2: Forward substitution with factorized block LU.

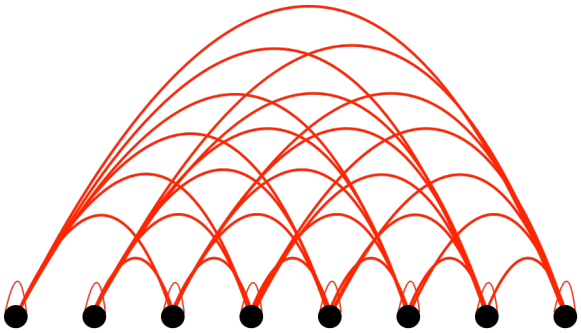
Input: L, x, b, NB
Result: y

for $i \leftarrow 1 \dots NB$ **do**
  $t_i \leftarrow 0$
  for $j \leftarrow 1 \dots NB-1$ **do**
    $t_i \leftarrow t_i + L_{ij} \cdot b_j$
  end
  $b_i \leftarrow L_{ii}^{-1}(b_i - t_i)$
end
$y \leftarrow b$

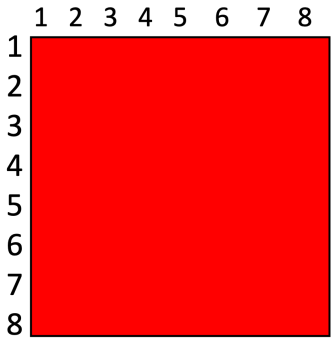
Algorithm 3: Backward substitution with factorized block LU.

Input: U, y, b, NB
Result: x

for $i \leftarrow NB \dots 1$ **do**
  $t_i \leftarrow 0$
  for $j \leftarrow NB \dots i+1$ **do**
    $t_i \leftarrow t_i + U_{ij} \cdot y_j$
  end
  $y_i \leftarrow U_{ii}^{-1}(y_i - t_i)$
end
$x \leftarrow y$



(a) A domain with 8 bodies showing connections between all bodies.



(b) The resulting dense matrix from the fully connected domain in Fig. 2.10(a).

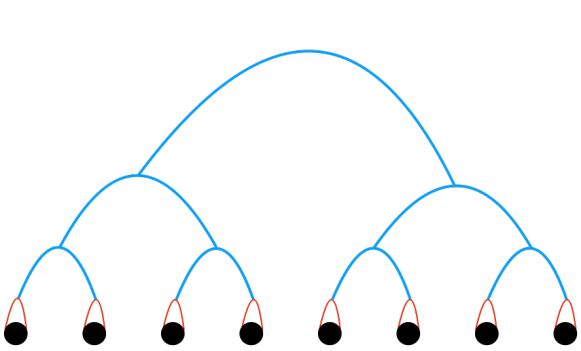
Figure 2.10: All-to-all connections between bodies in the domain leads to the formation of a dense matrix.

2.4 Low rank approximation of structured dense matrices

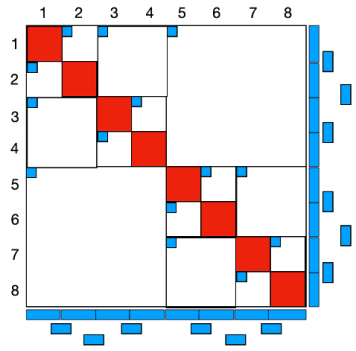
Fig. 2.10(a) shows a simple domain with 8 bodies with all-to-all connections. Since each body is connected to every other body in the domain, the edges of this graph can be expressed as the dense matrix shown in Fig. 2.10(b). This is exactly what happens in the boundary element method - the fact that the boundary integral must be applied over all the bodies in the domain results in the formation of a dense matrix. Particle distributions as shown in Fig. 2.10(a) typically contain clusters of particles in which some particles are near, and some are far. Although the interactions between the near particles must be expressed with full accuracy, the interactions between the particles that are far can be approximated. This leads to a reduction in the number of necessary interactions. The blocks of the matrix that correspond to the approximated interactions arise in the off-diagonal parts of the matrix.

The particle distributions of the Boundary Element Method as discussed in Section 2.1.3.4 typically show such near and far distribution of particles in the domain. This leads to the formation of structured dense matrices. The off-diagonal blocks in such dense matrices that correspond to far interactions in the physical domain can be compressed. Fig. 2.11 shows the approximated interactions in Fig. 2.11(a) and the corresponding approximated matrix in Fig. 2.11(b). The approximated far interactions are shown in blue and the unapproximated near interactions in red. The interactions between the upper levels can be further approximated in order to generate off-diagonal blocks that are larger in size, which leads to further savings in space and reduces the computation necessary for factorization. The matrix shown in Fig. 2.11(b) is the Hierarchically Semi-Separable matrix further elaborated in Section 2.4.3.

The low rank approximation of a dense block allows capturing the most significant row and column bases of the dense matrix. If the singular values of the dense block (typically obtained from



(a) Far interactions in the domain can be approximated in order to reduce the number of all-to-all interactions.



(b) The matrix resuling from the approximated far interactions.

Figure 2.11: Approximation of far connections in the domain can reduce the number of total interactions between particles in the domain.

the Singular Value Decomposition of the matrix) reduce very rapidly, we only need to retain the first few singular values and associated basis vectors, thus expressing the dense block with significant compression of data. The number of significant bases retained is the *numerical rank* of the matrix.

Low rank matrices are generated from the corresponding dense matrix block using a decomposition like randomized SVD (Singular Value Decomposition) [81], Interpolative Decomposition [81], and Adaptive Cross Approximation (ACA) [122] (which are more efficient than SVD). The dimensions of the low rank approximation of a dense matrix of dimension $m \times n$ using a rank of $rank$ can be represented using a tuple of 3 elements $(m, n, rank)$. A dense block $A_{m\times n}$ is represented as a product of three matrices as shown in Eq. 2.17, and represented in Fig.2.12. This form is typically created using an RSVD. ACA cannot directly create this form without orthogonalization.

$$A_{m\times n} \approx U_{m\times rank} \cdot S_{rank\times rank} \cdot V_{rank\times n} \tag{2.17}$$

Thus the total storage requirement of the matrix reduces to $m \times rank + rank \times rank + rank \times n$, a value significantly smaller than $m \times n$ memory necessary for the dense matrix, if $rank$ is much smaller than m and n.

The compressed dense matrices can be broadly clubbed under the term "Hierarchical matrices" [76]. Hierarchical matrices exploit the low rank property of the dense matrix to reduce the cost of computation from $O(N^3)$ to almost $O(N^2)$ or even $O(N)$. The complexity of the factorization is determined by the format of the hierarchical matrix and the algorithm used for the factorization. Various formats such as BLR [17], BLR^2 [21], HODLR [14], $\mathcal{H}$-matrix [73], HSS [49] and $\mathcal{H}^2$-matrix [26] have been proposed, varying by conditions of admissibility and the use of shared or nested basis.

The admissibility condition refers to the presence of dense blocks in the off-diagonal part of the hierarchical matrix. 'Weak' admissibility refers to hierarchical matrix having dense blocks only the

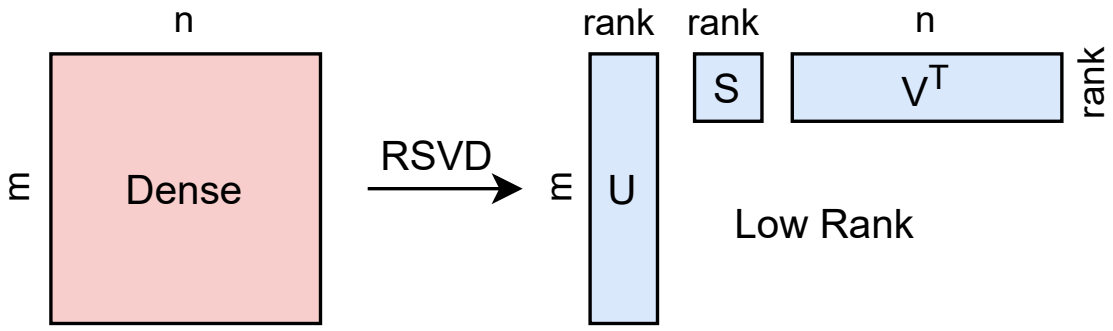


Figure 2.12: Representation of a low rank matrix using an algebraic method such as the Randomized Singular Value Decomposition (RSVD).

diagonal. Hierarchical matrices with weak admissibility are mainly used for 1D and 2D problems since the off-diagonal ranks are not as large as those in 3D problems. Examples of weakly admissible hierarchical matrices are HODLR, BLR-weak, BLR^2-weak and HSS. 'Strong' admissibility means that the hierarchical matrix has dense blocks in the off-diagonals too. Strong admissibility is useful for representing 3D problems. This is because the off-diagonal ranks are large in 3D, which can be represented as dense blocks.

Each block in the hierarchical matrix can either store its own basis or share the basis across the row and column blocks. In the latter case, the hierarchical matrix is said to be using 'shared basis'. BLR, HODLR and $\mathcal{H}$-matrix are matrix formats that do not make use of the shared basis. BLR^2, HSS and $\mathcal{H}^2$ use the shared basis. If using a multi-level structure such as the HSS matrix or $\mathcal{H}^2$-matrix, the shared basis of the upper levels can be inferred with the use of transfer matrices, which require O($rank$) storage. Such matrices are said to use the 'nested' basis.

In this section, we will introduce the most relevant hierarchical matrix formats for this work, i.e. the BLR format in Section 2.4.1, BLR^2 format in Section 2.4.2, the HSS format in Section 2.4.3 and the $\mathcal{H}^2$-matrix format in Section 2.4.4. Since we focus on applications that always result in symmetric positive definite (SPD) matrices, we show only the lower triangle of the symmetric matrix for illustrating the various types of low rank matrices and their construction.

2.4.1 Construction and notation of BLR matrices

The Block Low Rank (BLR) [17] format is the most basic format, where the off-diagonal low rank blocks are individually compressed and stored as the corresponding bases and skeleton matrices of the dense block. The BLR matrix can be obtained by first dividing the matrix into blocks, and then compressing each block individually. The blocks to compress can be determined by the admissibility condition. The biggest advantage of the BLR format compared to the formats using the shared basis is that each block can have its own basis, which can potentially lead to better accuracy for complicated problems.

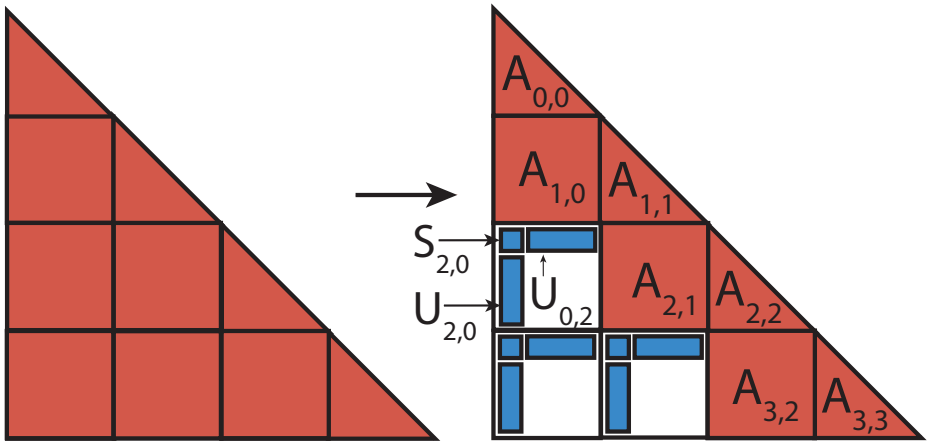


Figure 2.13: Construction of an SPD BLR matrix from a block dense matrix. This matrix is constructed from a 4x4 matrix and shows strong admissibility. A weakly admissible matrix would have dense blocks only on the diagonal.

Fig. 2.13 shows the construction of an SPD BLR matrix from a SPD dense matrix. The low rank blocks are shown in blue and the dense blocks are shown in red. Note that each low rank block has its own basis matrices and skeleton matrix. This is a contrast from the BLR^2 matrix which is closely related to the BLR matrix but with the shared basis.

2.4.2 Construction and notation of BLR^2 matrices

A symmetric positive definite (SPD) BLR^2 matrix can be constructed from a block dense matrix as shown in Fig. 2.14. A single block of this matrix at the index $(row, column)$ is denoted by $A_{row,column}$. We use a single shared bases denoted by U_{row} to denote the bases of the admissible blocks on row. For example, the block $A_{2,1}$ in Fig. 2.14 is denoted as

$$A_{2,1} \leftarrow U_2 \cdot S_{2,1} \cdot U_1^T \tag{2.18}$$

where $S_{2,1}$ denotes the skeleton block shown in blue.

The shared basis for each column is generated by concatenating the admissible blocks in the column, denoted by $A_{+,0}$. The shared basis can then be computed by a column-pivoted QR factorization or SVD.

$$\begin{bmatrix} U_0^S & U_0^R \end{bmatrix} \leftarrow QR(A_{+,0}^T) \tag{2.19}$$

where the S and R superscripts denote the skeleton part and redundant part of the basis, respectively. In order to make it convenient to represent the ULV factorization, we permute the skeleton and redundant parts as shown in Eq. (2.20).

$$U_i = \begin{bmatrix} U_i^R & U_i^S \end{bmatrix} \tag{2.20}$$

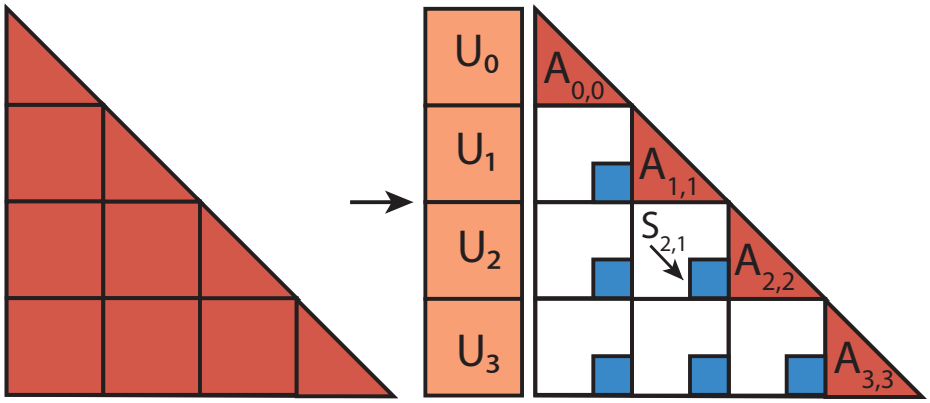


Figure 2.14: Construction of an SPD BLR^2 matrix from a block dense matrix.

Any dense block of the BLR^2 matrix in Fig. 2.14 $A_{i,j}$ can be represented as Eq. (2.21).

$$A_{i,j} = \begin{bmatrix} U_i^R & U_i^S \end{bmatrix} \cdot \begin{bmatrix} S_{i,j}^{RR} & S_{i,j}^{SR} \\ S_{i,j}^{RS} & S_{i,j}^{SS} \end{bmatrix} \cdot \begin{bmatrix} U_j^{RT} \\ U_j^{ST} \end{bmatrix} \tag{2.21}$$

Similarly, any low rank block $A_{i,j}$ can be represented as Eq. (2.22).

$$A_{i,j} = \begin{bmatrix} U_i^R & U_i^S \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 \\ 0 & S_{i,j}^{SS} \end{bmatrix} \cdot \begin{bmatrix} U_j^{RT} \\ U_j^{ST} \end{bmatrix} \tag{2.22}$$

Low rank matrix formats that have dense blocks in their off diagonals are correspond to being strongly admissible, and those that have dense blocks only on the diagonal correspond to weakly admissible.

2.4.3 Construction and notation of HSS matrices

The notion of the weakly admissibility BLR^2 matrix described above can be extended to the HSS matrix. The HSS matrix introduces multiple levels in the matrix by sharing the basis between levels. This should not be confused with the recursive hierarchical structure of the HODLR [14] matrix, which does not share the basis but instead uses recursive low rank blocks in the off-diagonals.

We introduce the notion of *level* in order to represent the blocks of the HSS matrix at various levels. At the leaf level, the dense block $A_{i,j}$ of the BLR^2 matrix can be represented as $A_{level;i,j}$ for the HSS matrix. The shared basis at the leaf level also use a similar notation and are denoted by $U_{level;i}$. Fig. 2.15 introduces the notation and construction of a 2-level HSS matrix from the BLR^2 matrix shown in Fig. 2.14. As an example, the block $A_{1;1,0}$ can be represented with the nested basis as shown in Eq. (2.23).

$$A_{1;1,0} = \begin{bmatrix} U_{2;2} & 0 \\ 0 & U_{2;3} \end{bmatrix} \cdot U_{1;1} \cdot \begin{bmatrix} 0 & 0 \\ 0 & S_{1;1,0}^{SS} \end{bmatrix} \cdot U_{1;0}^T \cdot \begin{bmatrix} U_{2;0}^T & 0 \\ 0 & U_{2;1}^T \end{bmatrix} \tag{2.23}$$

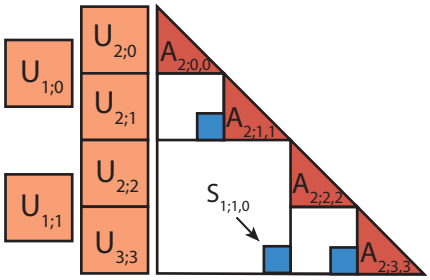


Figure 2.15: HSS construction and notation. This HSS matrix is generated from a 4x4 dense matrix.

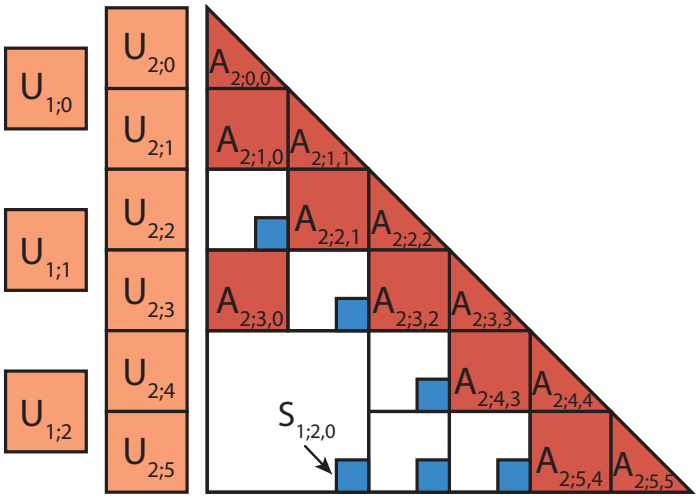


Figure 2.16: Construction from an $\mathcal{H}^2$-matrix from a 6x6 block dense matrix. Note that there are off-diagonal dense blocks in the $\mathcal{H}^2$-matrix, unlike the HSS matrix.

2.4.4 Construction and notation of $\mathcal{H}^2$-matrices

Fig. 2.16 shows an $\mathcal{H}^2$-matrix generated from a 6x6 block dense matrix. The weakly admissible HSS matrix shown in Sec. 2.4.3 can be extended to the $\mathcal{H}^2$-matrix with the use of the strong admissibility condition. This means that there will be dense blocks in the off-diagonals. Notice that the inadmissible dense blocks $A_{level;i,j}$ exist only on the leaf level. The leaf level low rank blocks are generated exactly as shown in Eq. 2.22 and the upper level low rank blocks are generated as shown in Eq. 2.23.

2.5 Matrix vector product of the block low rank matrix

The boundary element method shown in Section 2.1.3.4 can be used for obtaining solutions of the wave equation (Eq. 2.24) in order to study the behaviour of acoustic waves. The coefficient matrix of the wave equation is dense after the time t has passed sufficiently. Maxwell's equations which are solved with the Helmholtz kernel that represent a steady state also show similar solutions.

A main issue when working with the wave equation is the need to work with multiple waves at

various incident angles, which are expressed as vectors and must be multiplied with the factorized matrix. This makes it necessary to have fast routines for multiplication of a low rank matrix such as a block low rank matrix with multiple vectors. This section introduces the necessary mathematical background for Section 4.2. We introduce the block low rank matrix-vector multiplication followed by the low rank multiplication algorithm in Section 2.5.1.

$$u_{tt} = c^2 \nabla \cdot u \tag{2.24}$$

The block low rank (BLR) matrix stores the basis and skeleton blocks of every block of the full matrix as shown in Section 2.4.1. This happens exactly like it would for a block dense matrix, the only difference being that the presence of low rank blocks can be exploited to perform cheaper multiplication of basis matrices with vectors rather than having to use the full dense blocks. Since the basis vectors are of dimension $rank \times NB$, the matrix vector product with the BLR matrix can be performed in $O(N)$ time if the $rank$ is smaller than the block size NB.

2.5.1 Low Rank Multiplication

Alg. 4 shows the low rank multiplication algorithm. This is an important component in the approximation and solution of hierarchical matrices, which will be discussed later in Section Section 4.1. This algorithm involves multiplication between two 'skinny' matrices A_{V^T} and B_U and two 'small' matrices A_X and B_X of the form $A_X \times A_{V^T} \times B_U \times B_X$. Multiplication between such matrices is particularly challenging as a result of their small sizes, which makes the computation heavily memory bound. The technique that we develop in this thesis optimizes this specific step by batching a large number of independent low rank matrices together for improved performance. The low rank multiplication forms the first step of the rounded addition [25] algorithm for addition of low rank matrices, and the low rank matrix-vector multiplication.

2.6 Direct factorization of HSS matrix with HSS-ULV

This section introduces the ULV-like algorithms for low rank matrices. These algorithms build upon the foundation laid by basic factorization algorithms from Section 2.2.2. BLR^2 and HSS matrices can be factorized with the ULV factorization. The ULV factorization [49] can be thought of as a modified Cholesky factorization where the L represents the lower triangular dense blocks and the U and V represent the bases. The ULV works on the principle of nullifying the low rank off-diagonal blocks by multiplying the row and column with the shared bases. This means that there is no need to perform the triangular solve and trailing sub-matrix updates for the admissible blocks, which leads to an embarrassingly parallel factorization of successive levels of the HSS matrix. Therefore, dependencies only exist between the levels.

Algorithm 4: Low Rank matrix multiplication of two approximated matrices A and B. A_U and B_U represent the column basis of the matrix A and B, respectively. A_{V^T} and B_{V^T} represent the row basis. $rank_A$ and $rank_B$ show the rank of approximation of A and B, respectively. After multiplication G_{XY} becomes the skeleton matrix of the product.

Input: LowRank A(A_U, A_X, A_{V^T}) of $(m, k, rank_A)$, LowRank B(B_U, B_X, B_{V^T}) of $(k, n, rank_B)$

Result: G_{XY} of size $rank \times rank$

1 $C_{temp} = A_{V^T} \cdot B_U$

2 $E_{temp} = A_X \cdot C_{temp}$

3 $G_{XY} = E_{temp} \cdot B_X$

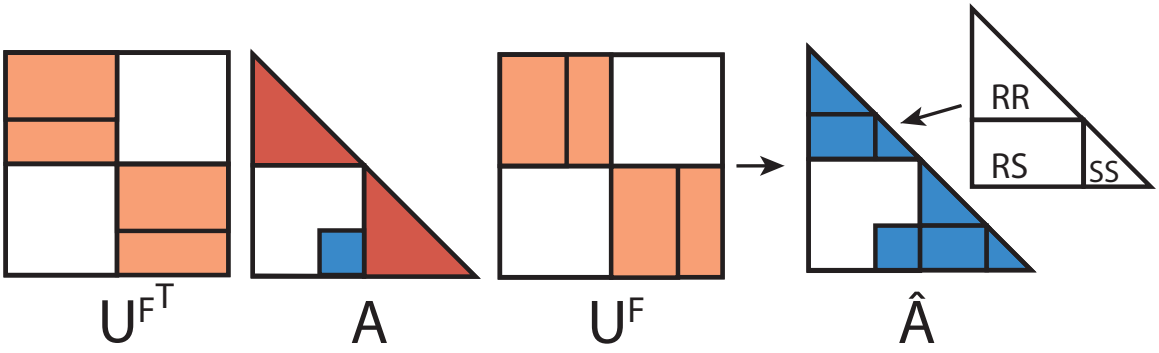


Figure 2.17: ULV factorization of a weakly admissible BLR^2 matrix.

Multiplication of the dense block shown in Eq. (2.21) with its respective row and column basis from Eq. (2.20) leads to

$$\begin{bmatrix} S_{i,j}^{RR} & S_{i,j}^{SR} \\ S_{i,j}^{RS} & S_{i,j}^{SS} \end{bmatrix} = \begin{bmatrix} {U^R}_i^T \\ {U^S}_i^T \end{bmatrix} \cdot A_{i,j} \begin{bmatrix} {U^R}_j & {U^S}_j \end{bmatrix} \tag{2.25}$$

Likewise, multiplication of the low rank block from Eq. (2.22) leads to Eq. (2.26).

$$\begin{bmatrix} 0 & 0 \\ 0 & S_{i,j}^{SS} \end{bmatrix} = \begin{bmatrix} {U^R}_i^T \\ {U^S}_i^T \end{bmatrix} \cdot A_{i,j} \begin{bmatrix} {U^R}_j & {U^S}_j \end{bmatrix} \tag{2.26}$$

As shown in Section 2.4.3, the HSS matrix is a multi-level matrix format where each level consists of a single BLR^2 matrix. We first introduce the ULV algorithm for the BLR^2 matrix in Section 2.6.1. The BLR^2-ULV can then be computed at each level of the HSS matrix in order to obtain the HSS-ULV algorithm as shown in Section 2.6.2.

2.6.1 Weak admissibility BLR^2-ULV factorization

Alg. 5 summarizes the BLR^2-ULV with weak admissibility. The block diagonal matrix U^F on line 1 is composed of the basis matrices of each row of the BLR^2 matrix as shown in Eq. (2.27). The resulting product $\hat{A}$ has dense and low rank blocks split into RR, RS and SS parts as shown in Eq. (2.25) and Eq. (2.26), respectively. This is demonstrated in Fig. 2.17 on a 2x2 BLR^2 matrix.

Algorithm 5: ULV factorization of a BLR^2 matrix with weak admissibility.

Input: A, U

/* Diagonal product. */

1 $\hat{A} \leftarrow {U^F}^T \cdot A \cdot U^F$

/* Partial factorization. */

2 $\hat{A}^{SS} \leftarrow partial\_Cholesky(\hat{A})$

/* Merge (permute) and factorize. */

3 $Cholesky(P^T \cdot \hat{A}^{SS} \cdot P)$

$$U^F = \begin{bmatrix} U_0 & 0 \\ 0 & U_1 \end{bmatrix} \tag{2.27}$$

The partial Cholesky factorization on $\hat{A}$ at Line 2 works on the RR, RS and SS blocks of each $S_{i,j}$ block. The partial factorization is only performed on the diagonals as a result of the multiplication with the complements in the preceding step.

$$L^{RR}_{i,i} {L^{RR}_{i,i}}^T \leftarrow Cholesky(\hat{A}^{RR}_{i,i}) \tag{2.28}$$

$$L^{SR}_{i,i} \leftarrow {L^{RR}_{i,i}}^{T^{-1}} \cdot \hat{A}^{SR}_{i,i} \tag{2.29}$$

$$\hat{A}^{SS}_{i,i} \leftarrow \hat{A}^{SS}_{i,i} - L^{SR}_{i,i} \cdot {L^{SR}_{i,i}}^T \tag{2.30}$$

Line 3, demonstrated by Fig. 2.18, involves permutation of the partially factorized matrix $\hat{A}^{SS}$ which brings all the SS blocks on the lower right corner. This is followed by a dense Cholesky factorization of a smaller matrix of the order of $NB \times rank$. The Cholesky factorization is then performed as follows:

$$\hat{L}^{SS} {\hat{L}^{SS}}^T \leftarrow Cholesky\left(\begin{bmatrix} \hat{A}^{SS}_{00} & 0 \\ \hat{A}^{SS}_{10} & \hat{A}^{SS}_{11} \end{bmatrix}\right) \tag{2.31}$$

Finally, the matrix A is expressed as the following factorization, where $\hat{L}$ represents a partially factorized lower diagonal matrix:

$$A = {U^F}^T \cdot \hat{L} \cdot (P \cdot \hat{L}^{SS} \cdot {\hat{L}^{SS}}^T \cdot P^T) \cdot \hat{L}^T \cdot U^F \tag{2.32}$$

The solve step for the BLR^2-ULV is shown by:

$$x = {U^F}^T \cdot \hat{L}^{T^{-1}} \cdot (P^T \cdot {\hat{L}^{SS}}^{T^{-1}} \cdot \hat{L}^{SS^{-1}} \cdot P) \cdot \hat{L}^{-1} \cdot U^F \cdot b \tag{2.33}$$

The final dense matrix in Alg. 5 can get large in size for a large rank and problem size. Therefore, even though the leaf level blocks of size $nleaf$ are factorized in $O(N)$, the overall complexity of the algorithm can reach close to $O(N^2)$. The HSS-ULV in the next section shows how the $O(N)$ time complexity can be preserved by exploiting the nested basis.

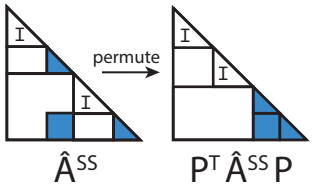


Figure 2.18: Permutation and Cholesky factorization of partially factorized BLR^2 matrix.

Algorithm 6: ULV factorization of an HSS matrix.

Input: A, U

1 **for** $l \leftarrow max\_level$ **to** 1 **do**

  `/* Diagonal product. */`

2 $\hat{A}_l \leftarrow {U^F}_l^T \cdot A_l \cdot U_l^F$ `/* Partial factorization. */`

3 $\hat{A}_l^{SS} \leftarrow partial\_Cholesky(\hat{A}_l)$ `/* Merge (permute). */`

4 $A_{l-1} \leftarrow P_l^T \cdot \hat{A}_l^{SS} \cdot P_l$

5 **end**

  `/* Final factorization. */`

6 $Cholesky(A_0)$

2.6.2 HSS-ULV factorization

Alg. 6 describes the ULV factorization of an HSS matrix. The HSS-ULV applies the BLR^2-ULV algorithm to each level of the HSS matrix. However, instead of factorizing a dense matrix in the merge step (line 3 in Alg. 5), the HSS-ULV algorithm iteratively applies the same procedure to the leftover blocks.

Fig. 2.19 illustrates the steps taken by the HSS-ULV for 2 iterations of the factorization for a 2 level HSS matrix. A factorization and permutation of the $\hat{A}_2$ matrix leads to the generation of another HSS matrix $P_2^T \cdot \hat{A}_2^{SS} \cdot P_2$, whose diagonal block has a dimension of $2 \times rank$. Another iteration of the ULV factorization of this smaller HSS matrix results in the matrix A_0 of size $2 \times rank$. Finally a dense Cholesky factorization is performed on A_0 at line 6 in Alg. 6. Unlike the merge step of the BLR^2-ULV, the final resulting dense matrix for HSS-ULV is much smaller in size. This leads to $O(N)$ time complexity for this algorithm.

The final factorized form of the matrix A after the HSS-ULV factorization is very similar to that of the BLR^2-ULV in Eq. (2.32). Each lower triangular matrix $\hat{L}$ is permuted and multiplied by a U^F corresponding to that level of the matrix. The fully factorized form of the HSS-ULV is shown in Eq. (2.34).

$$
\begin{aligned}
A = &\left(\sum_{l=max\_level}^{1} U_l^F \cdot \hat{L}_l \cdot P_l \right) \cdot \\
&\left(P_0 \cdot \hat{L}_0^{SS} \cdot \hat{L}_0^{SS^T} \cdot P_0^T \right) \cdot \\
&\left(\sum_{l=1}^{max\_level} P_l^T \cdot \hat{L}_l^T \cdot {U_l^F}^T \right)
\end{aligned}
\tag{2.34}
$$

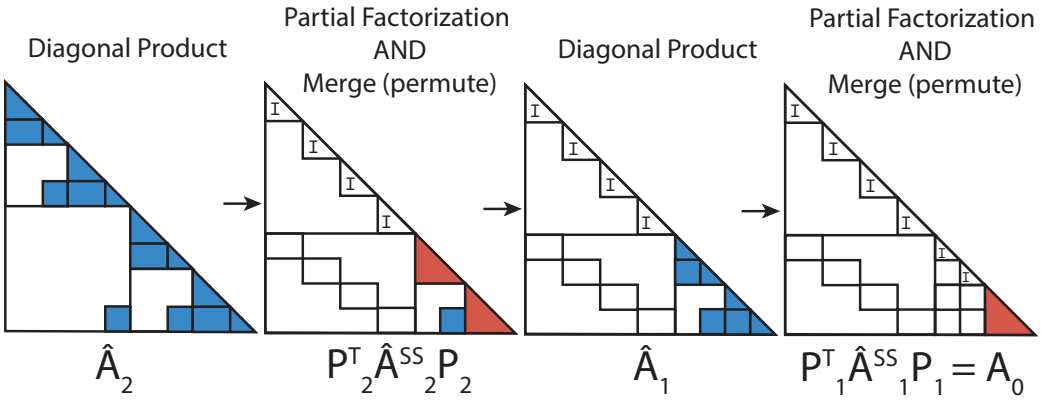


Figure 2.19: Diagonal product, factorization and permutation steps of the HSS-ULV factorization for a 2 level HSS matrix.

The solve step of the HSS-ULV is similar to that of BLR^2-ULV from Eq. (2.33). There is again a the introduction of multi-level summation terms similar to the factorization. This solve step is shown below:

$$
\begin{aligned}
x = & \left(\sum_{l=1}^{max\_level} {U_l^F}^T \cdot \hat{L}_l^{-1} \cdot P^T \right) \cdot \\
& \left(P_0^T \cdot \hat{L}_0^{SS^T{}^{-1}} \cdot \hat{L}_0^{SS^{-1}} \cdot P_0 \right) \cdot \\
& \left(\sum_{l=max\_level}^{1} P_l^T \cdot \hat{L}_l^{-1} \cdot U_l^F \cdot b \right)
\end{aligned} \tag{2.35}
$$

2.7 Direct factorization of $\mathcal{H}^2$-matrix with $\mathcal{H}^2$-ULV

2.7.1 ULV factorization of strong admissibility BLR^2-matrix

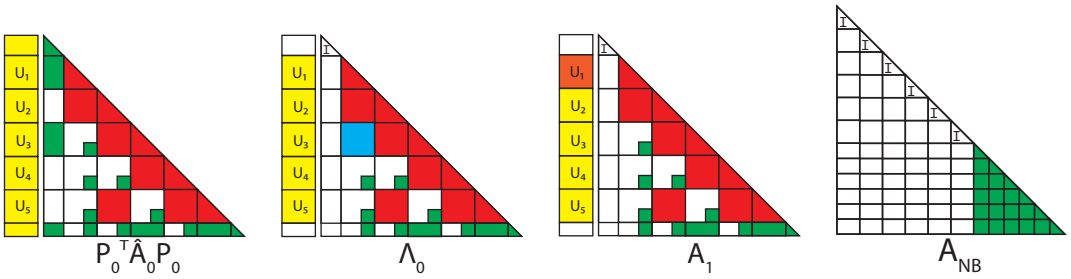
Alg. 7 shows the ULV factorization of the BLR^2-strong. The U_i^F term on line 5 is generated from the shared basis U_i by constructing an identity matrix with the U_i at the block at position (i, i). This multiplication is demonstrated for the BLR^2 matrix in Fig. 2.21. The multiplication splits the dense diagonal block into the form shown in Eq. 2.25, and the off-diagonal dense blocks into their respective redundant and skeleton parts. In order to denote intermediate stages of the matrix during the factorization, we use the subscript i with the letter denoting the matrix. The individual blocks of the matrix are indicated using a forward slash after the subscript i. So for example, the $(2, 3)$ block after the 1^{th} iteration of the partial Cholesky would be denoted by $\Lambda_{1/2,3}$

Fig. 2.20 shows the partial Cholesky, permutation and fill-in recompression stages of the BLR^2-ULV from Alg. 7. The partial Cholesky factorization shown in Fig. 2.20(b) is exactly like a block Cholesky factorization, but without the final Cholesky factorization of the trailing sub-matrix. For example, the matrix Λ_0 in Fig. 2.20(b) is obtained from $\hat{A}_0$ using the following steps:

Algorithm 7: BLR^2-strong ULV factorization. The inputs are the matrix A, the shared basis U and the number of blocks in a row NB.

Input: A, U, NB

$A_0 \leftarrow A$
$U_0 \leftarrow U$
for $i \leftarrow 0$ **to** NB **do**
  recompress_fill_ins(A_i, U_i) `/* Recompress */`
  $\hat{A}_i \leftarrow {U_i^F}^T \cdot A_i \cdot U_i^F$ `/* Multiply */`
  $\hat{A}_i \leftarrow P_i^T \cdot \hat{A}_i \cdot P_i$ `/* Permute */`
  $\Lambda_i \leftarrow$ partial_cholesky($\hat{A}_i$) `/* Partial factorize */`
  $A_{i+1} \leftarrow \Lambda_i$
end
Cholesky(A_{NB}) `/* Final factorize */`



(a) Permutation (b) Partial Factorize (c) Recompress (d) Final Factorize

Figure 2.20: Illustration of the permutation, partial factorization, fill-in recompression and the final factorization stages of BLR^2-ULV shown in Alg. 7

$$L^{RR}_{0/0,0} {L^{RR}_{0/0,0}}^T \leftarrow Cholesky(\hat{A}^{RR}_{0/0,0}) \tag{2.36}$$

$$L^{SR}_{0/0,0} \leftarrow {L^{RR}_{0/0,0}}^{T^{-1}} \cdot \hat{A}^{SR}_{0/0,0} \tag{2.37}$$

$$L^{R}_{0/1,0} \leftarrow {L^{RR}_{0/0,0}}^{T^{-1}} \cdot \hat{A}^{R}_{0/1,0} \tag{2.38}$$

$$L^{R}_{0/3,0} \leftarrow {L^{RR}_{0/0,0}}^{T^{-1}} \cdot \hat{A}^{R}_{0/3,0} \tag{2.39}$$

$$\Lambda^{SS}_{0/0,0} \leftarrow \hat{A}^{SS}_{0/0,0} - L^{SR}_{0/0,0} \cdot {L^{SR}_{0/0,0}}^T \tag{2.40}$$

$$\Lambda_{0/1,1} \leftarrow \hat{A}_{0/1,1} - L^{R}_{0/1,0} \cdot {L^{R}_{0/1,0}}^T \tag{2.41}$$

$$\Lambda_{0/2,2} \leftarrow \hat{A}^{SS}_{0/2,2} - L^{R}_{0/2,0} \cdot {L^{R}_{0/2,0}}^T \tag{2.42}$$

$$F_{0/3,1} \leftarrow -L^{R}_{0/3,0} \cdot {L^{R}_{0/1,0}}^T \tag{2.43}$$

Eq. 2.43 shows the generation of the fill-in block denoted by $F_{0/3,1}$ and shown in blue in Fig. 2.20(b). The presence of the fill-in during factorization can negatively affect the total time complexity. Therefore, the fill-in at block $(3, 1)$ is recompressed and the basis U_1 updated to reflect the change in the column basis. The recompression is possible because the fill-in is provably low-rank and hence can be incorporated as an admissible block. This procedure results in an update

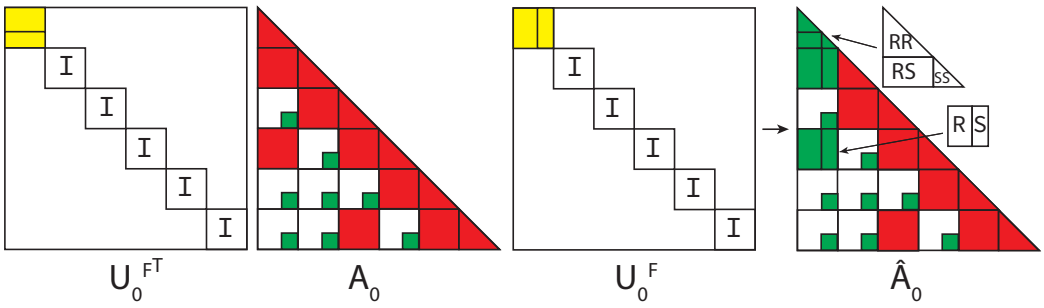


Figure 2.21: Multiplication of U_0^F with the dense parts of the BLR^2 matrix A_0.

of the U_1 basis, which is shown in orange in Fig. 2.20(c). The fill-in recompression is performed by concatenating the weighted basis for each block in the column with all the fill-ins in that column and then performing a pivoted QR factorization as shown in Eq. 2.44. We maintain a constant rank of factorization and therefore limit the resulting Q_1 to the same rank as U_1. The R_1 is discarded.

$$Q_1, R_1 \leftarrow QR \left(\begin{bmatrix} U_1 \Lambda^{SS}_{0/3,1} \\ U_1 \Lambda^{SS}_{0/4,1} \\ U_1 \Lambda^{SS}_{0/5,1} \\ F_{0/3,1} \end{bmatrix}^T \right) \tag{2.44}$$

Each skeleton block in the column is then updated by applying the inner product of the new basis with the old basis:

$$t_1 \leftarrow Q_1^T \cdot U_1 \tag{2.45}$$

$$\Lambda^{SS}_{0/3,1} \leftarrow \Lambda^{SS}_{0/3,1} \cdot t_1^T \tag{2.46}$$

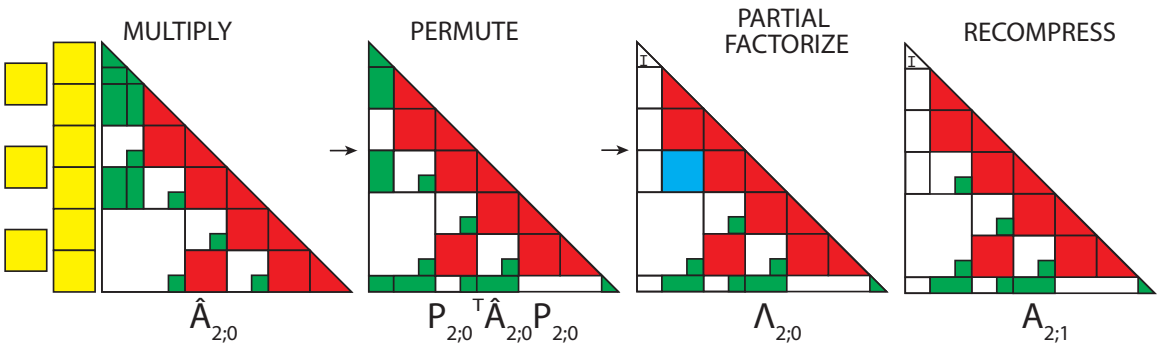
$$\Lambda^{SS}_{0/4,1} \leftarrow \Lambda^{SS}_{0/4,1} \cdot t_1^T \tag{2.47}$$

$$\Lambda^{SS}_{0/5,1} \leftarrow \Lambda^{SS}_{0/5,1} \cdot t_1^T \tag{2.48}$$

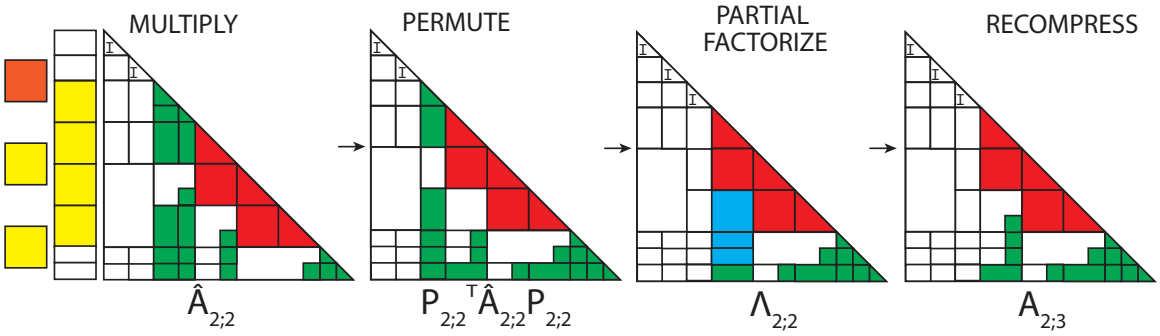
Finally, the old basis U_1 is replaced by the new basis Q_1. A similar procedure as outlined above can be applied to the rows of the matrix. The fill-in recompression is performed for the row and column of the diagonal block which is currently being factorized. Fig. 2.20(a) illustrates the permutation of the matrix at line 6 of Alg. 7. The permutation shuffles the partially factorized SS blocks to the bottom right of the matrix. After NB iterations, all the partially factorized blocks end up at the bottom right of the matrix, which results in the formation of a smaller dense matrix of dimension $rank \times NB$. This matrix is then factorized using a regular dense Cholesky factorization as shown in line 10 of Alg. 7. Fig. 2.20(d) shows the form of the matrix after completion of the for-loop in Alg. 7.

2.7.2 ULV factorization of $\mathcal{H}^2$-matrix.

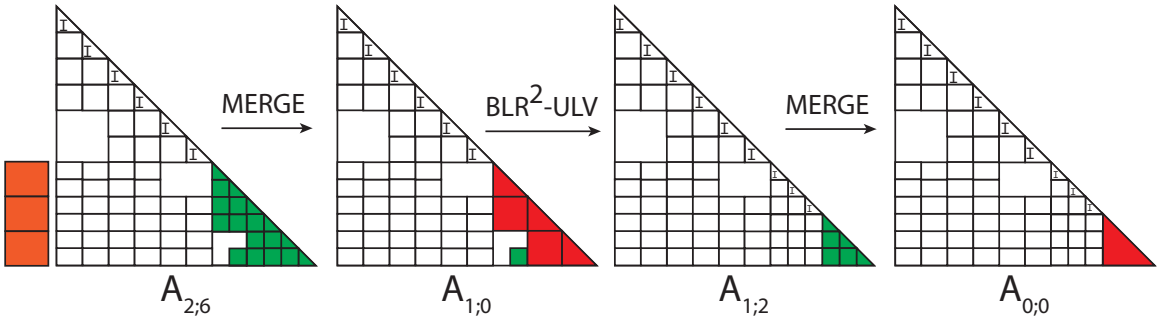
Alg. 8 shows the ULV factorization of the $\mathcal{H}^2$-matrix which are illustrated in Fig. 2.22. The $\mathcal{H}^2$-ULV factorization is very similar to the BLR^2-ULV. A recompression of the fill-in at a particular



(a) First iteration of Alg. 8 for the leaf level. Generation and re-compression of a fill-in of size $nleaf \times nleaf$ after starting out from the initial state of the matrix.



(b) Third iteration of Alg. 8 for lowermost level. Note the generation and recompression of a $nleaf \times rank$ sized fill-in. This can be clearly seen in the third matrix from the left. Note that one of the transfer matrices (colored orange) has been modified as a result of the Fig. 2.22(a).



(c) The $A_{2;6}$ on the leftmost side is merged into the $\mathcal{H}^2$ matrix in the second step after the merge step shown on line 10 of Alg. 8. This matrix is again factorized using the same algorithm as done for the leaf level. This leads to the formation a small dense matrix $A_{0;0}$ which is factorized using a regular dense Cholesky factorization shown in line 12.

Figure 2.22: States of the $\mathcal{H}^2$ matrix during the $\mathcal{H}^2$-ULV factorization at various stages during the factorization.

Algorithm 8: $\mathcal{H}^2$-ULV factorization. Matrix A and the nested basis U are the inputs.

Input: A, U

1 **for** $level \leftarrow max\_level$ **to** min_level **do**
2 $NB \leftarrow 2^{level}$ **for** $i \leftarrow 0$ **to** NB **do**
3 recompress_fill_ins($A_{level;i}$, $U_{level;i}$) `/* Recompress */`
4 update_transfer_matrices($A_{level-1;i}$, $U_{level-1;i}$) $\hat{A}_{level;i} \leftarrow {U^F_{level;i}}^T \cdot A_{level;i} \cdot U^F_{level;i}$ `/* Multiply */`
5 $\hat{A}_{level;i} \leftarrow P^T_{level;i} \cdot \hat{A}_{level;i} \cdot P_{level;i}$ `/* Permute */`
6 $\Lambda_{level;i} \leftarrow$ partial_cholesky($\hat{A}_{level;i}$) `/* Partial factorize */`
7 $A_{level;i+1} \leftarrow \Lambda_{level;i}$
8 **end**
9 $A_{level-1;0} \leftarrow$ merge($A_{level;NB}$)
10 **end**
11 Cholesky($A_{min\_level-1;0}$) `/* Final factorize */`

row requires an update of the corresponding transfer matrix at $level - 1$. Unlike the dense matrix A_{NB} shown in line 10 of Alg. 7, $\mathcal{H}^2$-ULV merges the unfactorized blocks from the previous level into the matrix A_{level}, resulting in the generation of a smaller $\mathcal{H}^2$-matrix. This $\mathcal{H}^2$-matrix has a block size that is proportional to the *rank* of the matrix rather than the leaf size. Subsequent ULV factorization of the upper levels will be cheaper than the leaf level as a result of the greatly reduced dimension of the dense blocks (assuming that the *rank* is much smaller than the leaf size).

Similar steps as shown in Eq. 2.44 and Eq. 2.45 can be used for generating the new basis after recompression of the fill-in. The new basis at the current level has to projected onto the transfer matrix one level higher as follows:

$$t_{2;1} \leftarrow Q^T_{2;1} \cdot U_{2;1} \tag{2.49}$$

$$U_{1;0} \leftarrow \begin{bmatrix} 0 & 0 \\ 0 & t_{2;1} \end{bmatrix} \cdot U_{1;0} \tag{2.50}$$

2.8 Summary of existing work on structured low-rank matrices

Fig. 2.23 shows a summary of the previous work done for the direct factorization of structured dense matrices. The contents of the graph show the various thesiss that have implemented direct factorization of structured dense matrices. The X axis shows the various programming paradigms that were used in the given publication, whereas the Y axis shows the time complexity of the publication in question. The X axis can be said to be an advancement of the implementation, and the Y axis of the algorithmic complexity.

The top row [21, 18, 146, 89, 40, 39, 42, 43, 45, 9, 116] of Fig. 2.23 uses a variant of the block LU or Cholesky algorithm shown in Section 2.3. Notably, Cao et. al. have made extensive use of the block low rank matrix from Section 2.4.1 for achieving a high degree parallelism on distributed

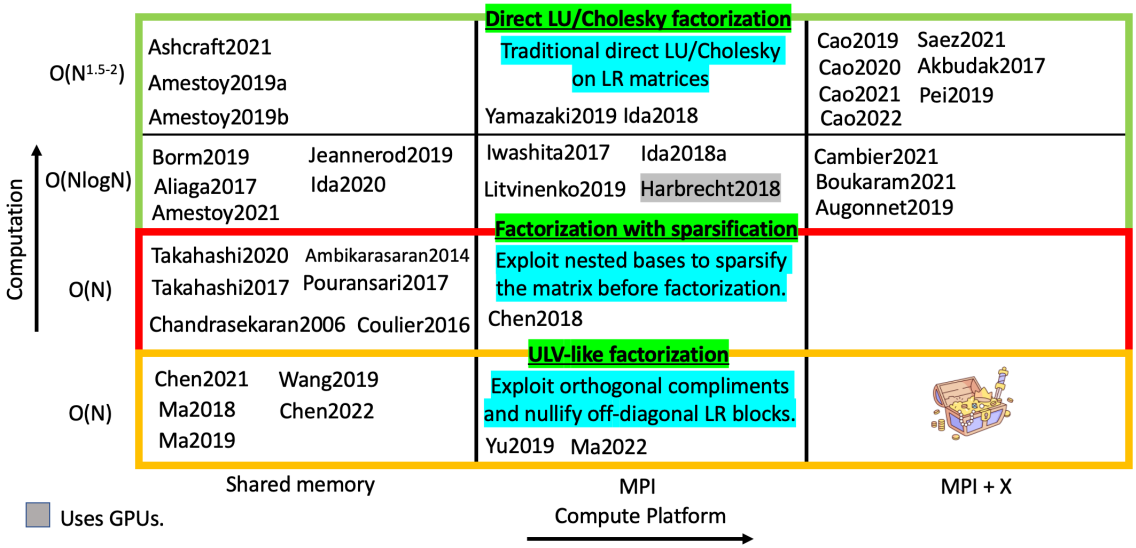


Figure 2.23: An illustrative description of the prior work done in the factorization of structured dense matrices using algorithms of varying time complexity (Y axis) and various programming paradigms (X axis).

supercomputers. The row below that [27, 29, 13, 18, 95, 91, 94, 101, 90, 102, 82, 38, 35, 152, 23, 99] makes use of the $\mathcal{H}$-matrix or HODLR format and uses a recursive version of the block Cholesky or LU factorization. Expressing large off-diagonal low rank blocks using recursive blocks reduces the time complexity of the algorithm to $O(NlogN)$. However, this significantly increases the complexity of the parallel implementation of such an algorithm as has been shown by task-based implementations by Augonnet. et. al. [23] and Börm et. al. [29] and Kriemann [99]. All these authors have had to spend considerable effort in development of methodologies for unraveling the recursive dependencies of the $\mathcal{H}$-matrix Cholesky/LU factorization and mapping them to a task-based runtime system as will be elaborated in Section 3.1.4. The parallelism inevitably suffers as a result of the overhead of the task-based runtime and the algorithms necessary for generating the directed acyclic graph from the recursive dependencies.

Further down, under the title "Factorization with sparsification" are publications that achieve $O(N)$ time complexity of factorization using sparsification [136, 135, 57, 15, 119, 54]. These publications make use of the nested basis and shared basis property HSS and $\mathcal{H}^2$-matrix and 'sparsify' the compressed matrix before obtaining a solution. Sparsification basically works by making use of the matrix-vector product of the nested basis matrix and expressing the dense, compressed matrix as a sparse matrix. Sparse matrix factorization techniques can then be combined with fill-in recompression for achieving $O(N)$ complexity. While these methods show $O(N)$ complexity, they still more computation than the ULV-like methods that are listed below the sparsification methods.

The final row of Fig. 2.23 shows the most optimal dense matrix factorization techniques using the nested basis and ULV-like factorization[53, 151, 106, 107, 108, 141]. The ULV factorization is

a Cholesky-like factorization that exploits the orthogonality of the shared basis and removes the need for performing triangular solve operations for each level of the factorization. This results in an $O(N)$ method as a result of the reduction in the number of operations that are necessary for the triangular solve and the trailing matrix update. Moreover, the fact that the transfer matrices of the HSS and $\mathcal{H}^2$ matrices are of the order of $rank$ means that non-leaf levels are much cheaper to factorize than the leaf level. Since nested basis matrices do not exhibit a recursive structure, they are also much easier to parallelize compared to HODLR or $\mathcal{H}$-matrix. Since the use of MPI+X paradigms such as asynchronous runtime systems was not used for ULV-like factorization before, Chapter 5 explores the use of asynchronous runtime systems for the ULV factorization of the HSS matrix, showing better weak scaling results than the state-of-the-art.

Chapter 3

Background of parallel numerical libraries and computer architecture

This chapter will introduce various relevant concepts for gaining an understanding of the relevant aspects of software design and computer architecture that went to designing efficient low rank approximation algorithms as described in Chapter 2. We begin by reviewing various software-based techniques for achieving the peak performance of modern CPUs in Section 3.1. We then review in Section 3.2 analytical techniques that can aid in writing software that is portable across a variety of CPU architectures yet high performance.

3.1 Software methodologies for achieving high performance.

This section dives deeper into the design of high performance software. The design decisions shown in the following sections have been shown to achieve the peak performance of the hardware. Section 3.1.1 covers cache-blocking, the most important software optimization. Ever since CPU architectures with hierarchical caches have become a standard practice in the design of CPUs, cache blocking has become an important tool for achieving peak performance. However, cache blocking hits its limitations when the data being computed is too less for the effective overlapping of communication and computation. Thus, we use batching as covered in Section 3.1.2. We conclude this section with a review of runtime systems in Section 3.1.4.

3.1.1 Cache blocking for overlapping computation and communication

Having reached the limits of Moore's law, CPU architecture designers have moved from simply improving CPU clock speed and die size to exposing multiple layers of parallelism in their designs. Innovations such as SIMD (single instruction, multiple data) architectures, Simultaneous Multithreading (SMT) and ccNUMA designs have led to potential parallelism on every level of the CPU. However, there is still a large difference in the time taken for arithmetic operations vs. time taken for fetching them from memory. For example, Fig. 3.1 shows the time to execute an FMA (fused

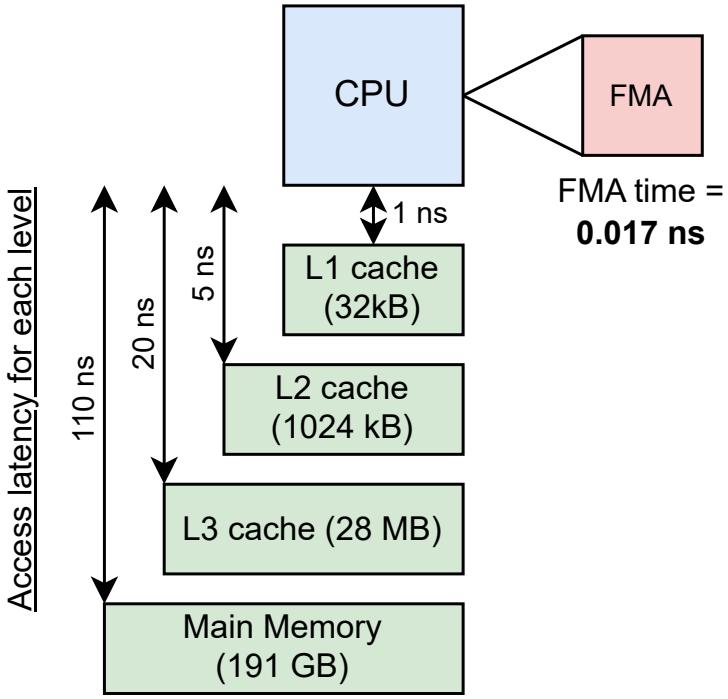
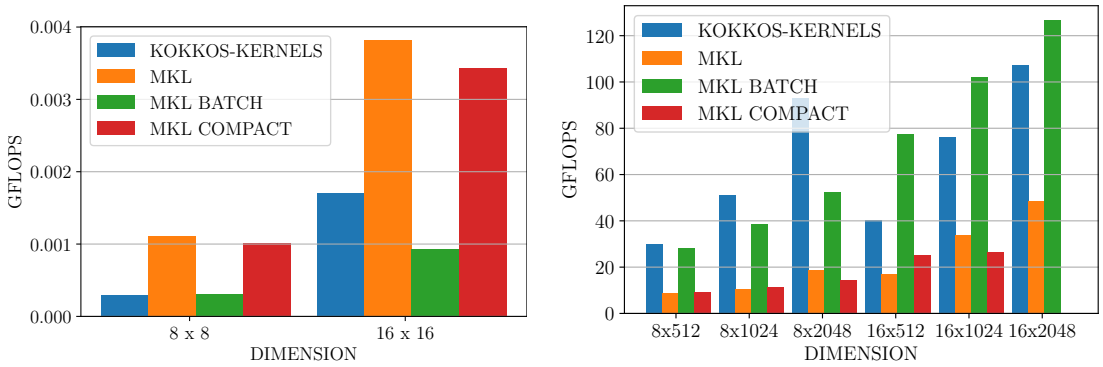


Figure 3.1: Access latency along the memory hierarchy when fetching data to execute an FMA on an Intel Xeon 6148 CPU (Skylake-X micro-architecture).

multiply-add) operation on a single core on the Intel Xeon Gold 6148 CPU vs. the time taken to fetch a single double-precision number from various layers of the cache hierarchy. Thus, the large ratio of memory vs. CPU speed (termed the 'machine balance' [112]), has led software writers to adopt innovative data locality optimizations in their designs in order to ensure that hardware spends most of its time on performing useful arithmetic calculations. The FMA time is calculated for a single core assuming an AVX512 frequency of 3.5 GHz when a single core is active. This means that a single `VMFMADD` instruction can perform a fused-multiply-add on 8 numbers at a time. Assuming that the FMA instruction has a reciprocal throughput of 0.5, this means that an FMA on a single digit is executed in 0.0625 clock cycles. Therefore, each FMA will take 0.017 ns.

The dense linear algebra community has envisioned several approaches for optimizing dense linear algebra routines using analytically modeled blocked algorithms [19, 72, 138, 153, 150, 103], auto-tuning [142], systematic derivation of algorithms [74] and recursive cache oblivious algorithms [63, 64]. Memory bound sparse matrix algorithms have similarly seen various innovations [92, 140, 114, 12] where the implementation of register blocking and new sparse matrix formats have led to increased efficiency.



(a) Small matrices

(b) Skinny matrices

Figure 3.2: Performance of batched small and skinny matrices for a batch size of 10000 using 20 physical cores using various batching techniques on a Intel Xeon Gold 6148 CPU. MKL uses non-batched MKL routines in a loop around the batch dimension. MKL BATCH performs batching with batched MKL routines without using memory interleaving across the batch dimension, and MKL COMPACT performs batching using memory interleaving as described by Dongarra et al. [60]. Kokkos-kernels [98] is a library designed from the ground up for obtaining efficiency with heterogeneous architectures and irregular matrix sizes. The final measurement for 16x2048 in Fig. 3.2(b) does not exist due to lack of memory.

3.1.2 Improved performance with batching

The tiny matrices arising out of low rank multiplication can be potentially batched together for better SIMD and bandwidth utilization, as has been done by batched matrix multiplication routines from MAGMA [80], Intel Math Kernel Library, and KokkosKernels [98]. Various approaches have been proposed for batching on both CPUs and GPUs [96, 51, 1, 2, 110]. The LIBXSMM library [83, 70] even optimizes batched matrix operations using register blocking and a JIT compiler. Alternate data layouts that interleave data across batches have been shown to outperform simple batching in some cases [60, 98]. This approach differs from the other batching approaches in that it utilizes a different permutation of the data across the batch dimension to keep the SIMD units busy. While efficient for very small matrices, the packing and unpacking quickly becomes a bottleneck as the matrix sizes increase. LibShalom [147] does away with the packing overhead for specific matrix sizes and optimizes the instruction schedule to achieve high performance of small matrix multiplications on ARM v8 CPUs. BLASFEO [65, 66] also provides batched LAPACK routines apart from matrix multiplication and achieves better performance than vendor optimized libraries on various CPUs. TSM2 and TSM2X [55, 121] are specifically built for optimizing tall-and-skinny matrix multiplication and use code generation for tuning the usage of threads and the cache hierarchy on the GPU.

Fig. 3.2 shows the performance of various batching techniques (KOKKOS-KERNELS, MKL BATCH and MKL COMPACT) vs. not batched techniques (MKL) for independent small matrices in Fig. 3.2(a) and for independent skinny matrices in Fig. 3.2(b) for a constant batch size of 10,000. Performing the same computation using batched routines shows better performance in most cases.

The batch size is fixed at 10,000 since it is found to be a sufficiently large batch size to show the benefits of batched vs. non-batched routines. Clearly, while efficient implementations for batched skinny matrices exist, small matrix operations are always under-performing. These operations are the least efficient part of many algorithms, and as such are the most promising candidates for enhancing the performance of low rank multiplication. Although Kokkos-kernels comes close to the performance of vendor-optimized batched routines, it does so for only some specific cases and we therefore use only vendor-optimized libraries in our experimental evaluation in Section 4.4.

3.1.3 Evolution of Parallel Dense Direct Factorization

Shared memory dense linear algebra libraries such as gotoBLAS [72], BLIS [139, 104], ATLAS [142], FLAME [74], MKL or SSL-2 have developed portable and efficient ways of exploiting the multi-level cache hierarchies seen in most modern CPU designs. The programming paradigms that were developed by these libraries made use of level 1, 2 and 3 BLAS routines for better reuse of data [127] residing in higher levels of cache in order to maximize the use of the floating point units of the CPU. Such software optimization has been very important for achieving optimal machine performance for the LU and Cholesky factorization algorithms shown in Section 2.2.2.

These programming paradigms then evolved into the distributed memory PBLAS and ScaLAPACK [56] routines, that made use of similar portable building blocks to achieve a fair trade-off between code portability across CPU architectures and performance. An optimal trade-off between portability and efficiency was found by PBLAS and ScaLAPACK by splitting distributed dense matrices using a block-cyclic process mapping. This allowed for considerable flexibility in the API to adapt the computation performed by each process by varying the size of the sub-matrices that reside on the process. It also allowed the user to manipulate parts of a distributed matrix by specifying sub-matrices without touching the rest of the distributed matrix. This flexible approach allowed PBLAS and ScaLAPACK to be adopted on a wide range of machines with fairly consistent efficiency, with implementations from all major supercomputer vendors. The LINPACK [59] benchmark for the Top500 list also makes use of the LU factorization and solve routines from ScaLAPACK for benchmarking the world's fastest supercomputers.

Other distributed computing libraries such as Elemental [118] took a slightly different approach than ScaLAPACK by using an elemental process distribution. This distribution relied on having successive elements on discrete processes, and using collective operations for communication. Elemental has been proven to show competitive performance on machines which rely more on a greater number of nodes than powerful CPUs such as Blue Gene [67].

However, libraries such as ScaLAPACK or Elemental were built with the assumption that both the underlying hardware and the work for each process is mostly uniform for the entire duration of the algorithm. The demise of Moore's law and Dennard's scaling has prompted a new wave of

highly parallel and heterogeneous hardware that has significantly altered the assumptions made by the previous approaches. Moreover, the effectiveness of low rank approximation for factorization of large dense matrices [79] has lead to the rise of algorithms that exhibit irregular computation for different parts of the algorithm.

Low rank approximation has been used for achieving faster factorization and multiplication algorithms for large structured dense matrices by trading off accuracy for speed [73]. Such large matrices employing low rank approximation techniques, collectively known as hierarchical matrices exist in various formats depending on the requirements of time, accuracy and implementation constraints of the target application. The BLR [17, 18], BLR^2 [21], $\mathcal{H}$-matrix [28], lattice $\mathcal{H}$-matrix [90], HODLR [16], HSS [48] and $\mathcal{H}^2$-matrix [26] are different formats of low rank matrices depending on conditions of admissibility and shared basis.

In order to address the problem for non-uniform and highly parallel hardware for dense matrix computation, libraries such as DPLASMA [30], Chameleon and SLATE [68] have made use of task-based runtime systems such as PaRSEC [32], starPU [22] and openMP tasks, respectively. These task based systems change the parallelism paradigm from fork-join to asynchronous, and allow for greater flexibility in the granularity of parallelism depending on the hardware (although SLATE still makes use of fork-join for distributed parallelism). Thus, they preserve software portability by leveraging single threaded versions of the aforementioned shared memory dense linear algebra routines, but manage the parallelism at a higher level using tasks in order to accommodate for heterogeneity in the hardware. The use of synchronous execution for dense linear algebra been shown to outperform previous fork-join based approaches [68].

The small, irregular computations arising out of calculations in low rank approximation are typically memory bound calculations, and have been addressed to some extent on shared memory with batched routines [115, 52, 60, 34, 3]. Shared memory implementations of dense matrix factorization using the block low rank format have used the starPU runtime system [5], openMP tasks [9] and have also leveraged the Chameleon library [4]. $\mathcal{H}$-matrix shared memory implementations have made used of the OmpSs [46, 13] by directly adding tasks to a runtime and with special algorithms for unraveling the hierarchical matrix structure for maximum parallelism with an asynchronous DAG [99, 29].

Distributed memory implementations of low rank approximation of large dense matrices have been implemented using both the fork-join and asynchronous paradigms. STRUMPACK [124, 71] is a well known library making use of the HSS matrix [109] for achieving $O(N)$ factorization complexity for dense [124] and optimizing the computation of sparse [123, 20] matrices. STRUMPACK uses ScaLAPACK 's block cyclic process distribution for each individual block of the HSS matrix. Recent algorithmic advances in decoupling of the off-diagonal dependencies from the trailing sub-matrix

update of a direct factorization such as LU factorization have also led to the use of pure MPI distributed $\mathcal{H}^2$-matrix routines [108].

Runtime systems have also been used for efficient distributed memory implementations of large dense matrices using low rank approximation. Use of the block low rank format can reduce the time complexity of the dense Cholesky factorization from $O(N^3)$ to $O(N^2)$. Factorization of large block low rank matrix for various applications, such as geospatial statistics and weather models, have scaled to many thousands of nodes using runtime systems [42, 120, 10, 105, 41, 43]. The $\mathcal{H}$-matrix and lattice $\mathcal{H}$-matrix formats have also been combined with a distributed task-based runtime system in order to perform a distributed dense direct factorization in $O(Nlog(N))$ time [146, 23].

3.1.4 Runtime systems

The cache blocking and batching software methodologies described in Section 3.1.1 and Section 3.1.2, respectively, have been shown to achieve good performance for a wide variety of computational kernels for shared memory execution on CPUs. However, new CPU designs making use of multiple NUMA nodes on a single CPU die such as the AMD Zen 3 architecture and the Fujitsu A64FX and the use of accelerators such as GPUs combined with a host CPU have led to further challenges for software designers to exploit the full potential of hardware.

The approaches highlighted in the previous sections all made use of fork-join parallelism for making use of multiple resources in parallel. This approach does not allow fine-grained control over the granularity and the scheduling of parallel resources, which can lead to loss of performance. Asynchronous parallelism with the use of task-based programming allows varying the granularity of each kernel within a larger computation at run time and also schedule such computation on heterogeneous hardware resources.

Fig. 3.3 shows a $3x3$ block Cholesky factorization as a directed acyclic graph (DAG). A runtime system such as PaRSEC [31] works with such a graph representation of the algorithm in order to factorize the matrix. Each node in the Directed Acyclic Graph in Fig. 3.3 is a 'task' with dependencies on some other tasks (nodes) in the graph, shown by the arrows. A task cannot begin execution unless all preceding tasks it depends on have finished their execution. Tasks that do not depend on each other can be executed in parallel. The color of the boundary of each node in the DAG corresponds to the block of the dense matrix that the node updates as a result of the computation taking place in the node.

To illustrate, consider the `GEMM` task inside the dotted black box. This task has READ dependencies on the $(2, 1)$ and $(3, 1)$ blocks, and a WRITE dependency on $(3, 2)$. The $(2, 1)$ and $(3, 1)$ blocks that the `GEMM` must read are WRITE dependencies for the two `TRSM` tasks that `GEMM` depends on. Unless both the `TRSM` tasks before the `GEMM` finish execution and hand over their blocks to `GEMM`, it cannot begin execution. Notice that the other two `SYRK` tasks within the dotted black box also have

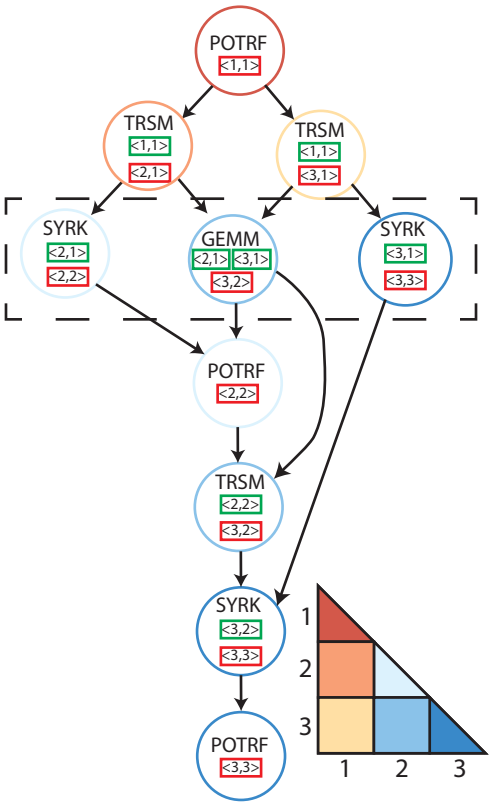


Figure 3.3: Directed Acyclic Graph (DAG) representation of the block Cholesky factorization of a 3x3 dense matrix. Each node has an associated computation and depends on certain blocks of the matrix (red or green boxes). The red boxes represent RW (Read-Write) dependencies and green boxes represent R (Read) dependencies. The nodes in the dotted blue box shows nodes that can be executed in parallel.

their respective READ dependencies $(2, 1)$ and $(3, 1)$ satisfied by the preceding `TRSM` tasks. Since they have no dependency on the `GEMM`, the two `SYRK` tasks and the `GEMM` inside the dotted black box can be executed in parallel.

Asynchronous runtime systems are available from various libraries such as openMP tasks, starPU [24], PaRSEC [31] and QUARK [148]. Shared memory and distributed memory runtime systems work in a similar manner. Each task is executed on a particular MPI rank and the data in the task is communicated to the another task executing on another MPI rank as soon as it finishes execution. This keeps other parallel tasks executing while this communication is happening, and allows for a perfect overlap between communication and computation. Libraries such as DPLASMA[31] and CHAMELEON [8] leverage the runtime system libraries and provide numerical computation routines for both shared and distributed memory execution.

PaRSEC provides multiple Domain Specific Languages (DSL) to expressing algorithms as DAGs - including Dynamic Task Discovery (DTD) and the Parameterized Task Graph (PTG). The DTD interface is similar to a distributed version of the OpenMP task programming - it submits all tasks

on every process, which allows every process to gather all necessary knowledge about the algorithm. However, this means each process discovers the entire task graph, trims the non-local tasks by removing those that do not depend on local tasks, and convert those that depends on communication to and from other processes, and finally execute only the tasks that are local to that process according to their data dependencies. The PTG interface is a custom DSL that allows for an concise parameterized description of the algorithm, and supports an event-driven execution where only local tasks are generated by each process and communications are automatically inferred from the task's dependencies. The PTG interface results in lesser runtime overhead especially when the number of tasks and processes is large as a result of not having to generate the entire task graph on every process.

Previous approaches to distributed memory execution of numerical algorithms mostly relied on various forms of fork-join parallelism. Libraries such as ScaLAPACK [56] rely on the block-cyclic process distribution and collective communication for distributed execution. Others such as Elemental [118] depend on an element-cyclic process distribution and heavy reliance on collective operations. SLATE [69], a newer alternative to ScaLAPACK and Elemental relies on fork-join parallelism for the inter-process communication and openMP tasks for asynchronous execution within the same process. SLATE is optimized for heterogenous computer architectures and the combination of fork-join parallelism for the distributed communication and asynchronous parallelism on shared memory allows it run efficiently on both CPU-only supercomputers and those using a GPU for acceleration alongwith the host. Asynchronous distributed memory libraries such as DPLASMA have been shown to outperform those using fork-join distributed parallelism for various numerical routines.

3.2 Analytical performance modeling

Performance modeling is useful for predicting the ideal or peak performance of an algorithm given architectural constraints. It can be used not only for analytically deriving the peak performance, but also for indicating which area most benefits from direct optimization.

Techniques such as the roofline model [144] are useful for gaining a measure of the memory-boundedness of an algorithm. However, these 'top-level' techniques do not dive deeper into the performance of the kernels in question. As a result, opportunities for optimization with the roofline model are difficult to identify.

3.2.1 Performance modeling of LLC misses

A simple approach for modeling the overall run time of memory bound applications is to measure the number of last-level-cache (LLC) misses [47, 87, 88, 75]. Several approaches ranging from analytical [88] and semi-empirical modeling [84] have been suggested for this purpose. More advanced approaches involving simulation have been suggested by [111, 37] using stack distances. This idea is

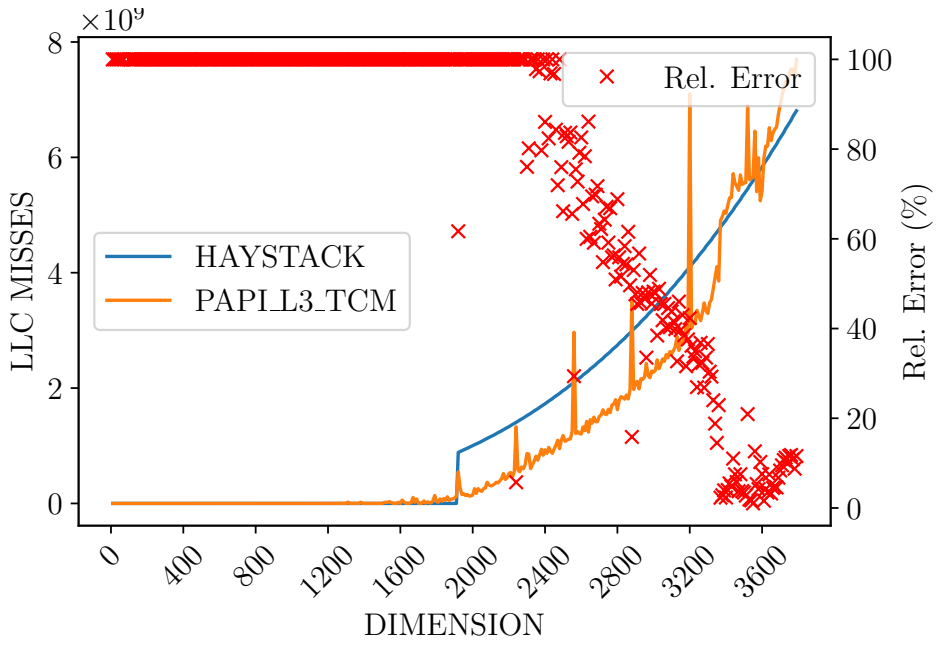


Figure 3.4: LLC misses of small matrix multiplication as predicted by haystack. The Y axis on the right shows the relative error between the calculated LLC misses by haystack vs. PAPI_L3_TCM measurements for $M = N = K$ as given in the X axis. Tests performed on a Intel Xeon Gold 6148 CPU (Skylake-X).

extended by [75], whose tool 'haystack' allows predicting the LLC misses using a faster simulation methodology than previously available. However, as Fig. 3.4 shows, haystack is not able to predict the LLC misses of smaller matrices.

3.2.2 Performance modeling of overall peak utilization rate

Huang et al. [88] propose a performance model based on the assumption that writing and reading data from caches takes zero time, and that the major cost of a memory-bound kernel is from reads from the main memory and the actual arithmetic calculation. This performance model allows us to express the utilization in GFLOPS (Giga-flops per second) of the low rank multiplication algorithm as Eq. 3.1.

$$GFLOPS = \frac{4 \times rank^3 + 2 \times rank^2 \times block\_size}{T_{compute} + T_{comm}(sec)} \times 10^{-9} \tag{3.1}$$

In Eq. 3.1, $T_{compute}$ is expressed as shown in Eq. 3.2, where τ_c is the time needed to compute a single floating point number (the inverse of the peak double precision matrix multiplication (DGEMM) utilization rate) in nanoseconds.

$$T_{compute}(sec) = \tau_c(ns) \times 10^9 \times batch\_size \times (4 \times rank^3 + 2 \times rank^2 \times block\_size) \tag{3.2}$$

BATCH SIZE	MKL	MKL BATCH	THEORY
10000	9.31	10.08	13.22
20000	9.51	10.52	13.22
40000	9.73	10.69	13.22
60000	9.78	10.76	13.22

Table 3.1: Experimental utilization in GFLOPS/sec for MKL and batched MKL compared to the theoretical peak rate for $m = n = 1024$ and $rank = 8$ using a single thread on Intel Xeon Gold 6148.

Given our assumption about the dominance of memory reads, we can assume that the total time spent in the communication T_{comm} can be expressed by Eq. 3.3. The amortized cost of fetching a single double-precision number from DRAM τ_m can be formulated as in Eq. 3.4 for a single core in nanoseconds. The bandwidth in Eq. 3.4 is the peak sustainable bandwidth for a single physical core.

$$T_{comm} = \tau_m \times 10^9 \times batch\_size \times 2 \times (rank^2 + rank \times block\_size) \tag{3.3}$$

$$\tau_m = \frac{8(Bytes)}{bandwidth\ (in\ bytes/sec)} \times \#memory\_channels \tag{3.4}$$

Plugging in theoretically achievable values of the utilization allows us to estimate the maximum achievable CPU utilization for a given problem size for a single core as shown by Table 3.1 on Intel Xeon Gold 6148. This performance model is able to show the peak achievable utilization when using a single core, however, it does not take into account the individual packing and computation kernels that exist in the algorithm. Therefore, it treats the entire algorithm like a black box, making it hard to understand where exactly effort for optimization should be directed in order to obtain the maximum possible utilization. The accuracy of this model is also questionable due to the wide variation between the experimentally analysed MKL benchmarks vs. the analytical prediction. Thus, this model is not very useful for micro-optimization of low level kernels.

3.3 The ECM performance model

The ECM (Execution-Cache-Memory) performance model [145] is an analytical technique for modeling the performance of steady-state loops. It models the ideal number of clock cycles necessary for execution of a single iteration of a loop subject to the constraints of the algorithm and machine. This level of detail allows individual modeling of the kernels of an algorithm and cycles through an optimization process until they match or are close to the predicted performance.

Alappat et al. have previously used the ECM model for optimizing sparse matrix vector multiplication on the Fujitsu A64FX [12, 11]. The ECM model has been used for optimizing conjugate

gradient based iterative solvers[62, 86], modeling the proportion of bandwidth utilized by overlapping kernels [6], and performance tuning and optimization of CFD applications [143, 145]. Witmann et al. [145] combine the ECM model with an energy consumption model to optimize both the performance and power consumption of a lattice-boltzmann CFD solver. They compare various performance modeling techniques and ultimately utilize insights gained from the ECM model for gaining the most significant speedups in their application.

The ideal number of clock cycles for executing a single iteration of a loop using a single thread on a single physical core is given by T_{ECM}. As shown in Eq. 3.5, T_{ECM} is composed of various terms, which can be described as follows:

- T_c – Clock cycles for pure compute instructions such as FMA and addition.
- T_{L1_L} – Clock cycles for loads from L1 cache into registers.
- T_{L1_S} – Clock cycles for stores from registers to L1 cache.
- T_l – Clock cycles for reads and writes between $l-1$ and level l cache.
- T_{mem} – Clock cycles for reads and writes between main memory and the last level cache (LLC).

$$T_{ECM} = max(T_c, f(T_{L1_L}, T_{L1_S}, T_2 \dots T_l, T_{mem})) \tag{3.5}$$

The main strength of the ECM model lies in the fact that it allows building an estimate of the overlap between levels of caches in a CPU. An important step in modeling the performance of any CPU using the ECM model involves first finding the function f in Eq. 3.5 in order to quantify whether the reads and writes between the caches are simultaneous or in serial. This step differs between various CPU designs and has a non-trivial impact on the predicted performance.

Since the ECM model takes into account the individual instructions that comprise a loop, various bottlenecks such as assembly code generation by the compiler, inconsistent cache usage, and lack of Out of Order execution can be quickly pointed out.

We use the methodology provided by Hofmann et al. [86] for obtaining the ECM model for a given CPU architecture. We first build the ECM model for the STREAM TRIAD kernel for each CPU. This is done by first building a model of the machine as shown in Section 3.3.1. We then build an application model specifically for STREAM TRIAD as shown in Section 3.3.2. Finally, we can obtain the equation for T_{ECM} for each CPU as shown in Section 3.3.3. Since T_{ECM} for each CPU (Table 3.2) remains constant for all steady-state loops for that CPU, the same equation can be used for our specific application of low rank multiplication.

	AMD EPYC 7502	**Fujitsu A64FX**	**Intel Xeon Gold 6148**
Vector Length (bits)	512	512	256
Instruction Set	AVX2	ARM SVE	AVX512
Microarchitecture	Zen2	ARM v8.2 SVE	Skylake-X
Cores	32	48	20
FMA (/core)	2	2	2
LOAD (/core)	2	2	2
STORE (/core)	1	1	1
Cache line size (bytes)	64	256	64
Cache write policy	write-allocate	write-allocate	write-back
Victim cache	Victim L3	-	Victim L3
L1 load (bytes/cycle)	32	64	64
L1 store (bytes/cycle)	32	64	64
L2 load (bytes/cycle)	32	64	64
L2 store (bytes/cycle)	32	32	64
L3 load (bytes/cycle)	16	-	14
L3 store (bytes/cycle)	16	-	14
L1 size	32×32 KiB	48×64 KiB	20×32 KiB
L2 size	32×512 KiB	4×8192 KiB	20×1024 KiB
L3 size	8×16 MiB	-	20×1.375 MiB
Clock freq. (Hz)	2×10^9	2×10^9	2.2×10^9

Table 3.2: Various machine parameters used for building the ECM performance model for each CPU.

3.3.1 Building the machine model

We first take into account various machine parameters as shown in Table 3.2 for our target architectures. The A64FX is configured to run on 'normal' mode, which means it runs at a frequency of 2×10^9 Hz. For the Intel Xeon CPU we disable Turbo Boost and assume the frequency that is obtained by running a simple Fused Multiply Add loop using AVX-512 instructions. We chose these architectures since they use three diverse instruction sets, AVX2, ARM SVE and AVX-512. This allows us to compare the performance of the low rank matrix multiplication depending on various capabilities provided by the Instruction Set Architecture (ISA), along with other machine parameters. The cache LOAD/STORE bandwidths from various levels of cache in Table 3.2 can be determined by formulating the STREAM usable bandwidth with empirical benchmarks and formulating and validating hypotheses about the bandwidth performance that fit the empirical measurements [86, Sec. 4].

3.3.2 Building the application model

The STREAM TRIAD [112] kernel is a simple kernel of the form $A(i) = B(i) + \alpha \times C(i)$. Assuming double precision, each execution of the kernel requires loading 16 bytes of memory ($B(i)$ and $C(i)$) and storing 8 bytes ($A(i)$) for a total of 24 bytes data transfer per iteration, in addition to a single multiply and add operation that can be done as a single operation using the fused multiply-add

instruction. Note that the actual amount of data transferred can vary according to the write policy of the caches. Every memory access is assumed to be a full cache line transfer [143].

We build an ECM application model for the STREAM TRIAD kernel on similar lines as has been done by [12, 11, 86]. The instructions that make up the STREAM TRIAD kernel, along with their latency and throughput on various architectures can be seen in Table 3.3.

	AMD EPYC 7502		**Fujitsu A64FX**		**Intel Xeon Gold 6148**	
	Latency	Reci. TPut.	Latency	Reci. TPut.	Latency	Reci. TPut.
LOAD $A(i)$, REG0	5	0.5	11	0.5	3	0.33
LOAD $B(i)$, REG1	5	0.5	11	0.5	3	0.33
FMA REG0, alpha, REG1	5	0.5	9	0.5	4	0.75
STORE REG0, $C(i)$	4	0.75	9	1	3	0.66

Table 3.3: STREAM TRIAD kernel with respective latencies and throughputs.

Each instruction shown in Table 3.3 works in units of one Vector Length (VL) corresponding to the length shown in Table 3.2. The latency and throughput can be easily obtained by running identical instructions in succession with dependencies between successive instructions for latency and without dependencies for the throughput. Alternatively, the ibench [100] tool can be used. The reciprocal throughput of the FMA instruction shown in Table 3.3 is 0.75 as opposed to 0.5 because we read the alpha variable through a memory location and not from a register. This allows for greater instruction level parallelism at the cost of 0.25 extra clock cycles for throughput.

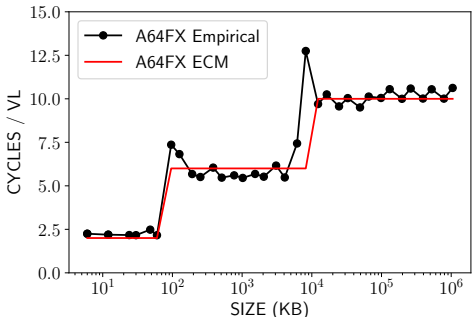
3.3.3 Building the overlap hypothesis

As shown by Hofmann et al. [86], building the overlap hypothesis is an important step in construction of the ECM model for a given CPU. Overlap hypotheses for the CPUs that we test have already been constructed for the Fujitsu A64FX [12], and also for AMD and Intel [86, 85] CPUs. In this section we use the techniques shown by the aforementioned authors to derive our own assumptions about performance using the ECM model. We build ECM models for each CPU as shown in Table 3.4. The performance assumptions from these models are validated in Fig. 3.5.

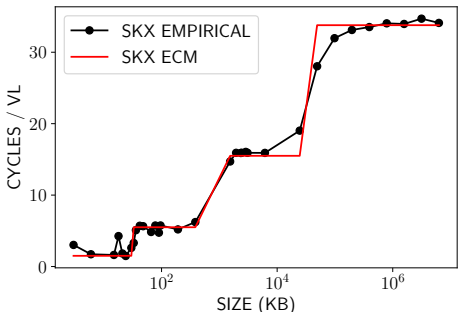
CPU	**ECM Model**
Fujitsu A64FX	$T_{ECM} = max(T_c, max(T_{L1_L} + max(T_{L1_S}, T_{L2}), T_{mem}))$
Intel Xeon Gold 6148	$T_{ECM} = max(T_c, T_{L1_L} + T_{L1_S} + T_{L2} + T_{L3} + T_{mem})$
AMD EPYC 7502	$T_{ECM} = max(T_c, T_{L1_L}, T_{L1_S}, T_{L2}, T_{L3}, T_{mem})$

Table 3.4: Derived ECM model assumptions for each CPU in our tests.

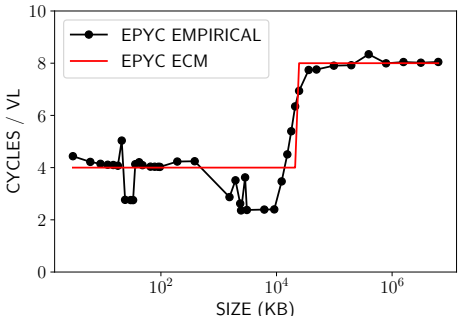
Fig. 3.5(a) shows the cycles per VL for the STREAM TRIAD after disabling the compiler's aggressive prefetching and pipelining for the Fujitsu A64FX. Each step corresponds to data being fetched from L1, L2 and main memory respectively. In practice, we observe that the compiler does a lot of aggressive prefetching into the L2 cache and exhibits behaviour as if the data were present



(a) Fujitsu A64FX



(b) Intel Xeon Gold 6148



(c) AMD EPYC 7502

Figure 3.5: Empirical vs. analytical clock cycles per vector length (VL) for each iteration of the STREAM TRIAD using a single core on each CPU. As the size of the data increases along the X axis, the number of cycles required for fetching one VL goes up. The 'steps' in each plot show how many cycles are needed when data fits into a particular level of cache. Each successive step shows that the data is being streamed from a progressively lower level of cache. The Fujitsu and Intel CPUs have a VL of 8 and the AMD CPU has a VL of 4.

in the L2 cache itself, which we use for constructing the ECM model. Fig. 3.5(c) shows that the AMD CPU goes from 4 cycles per VL to 8 after crossing the 16 MiB threshold as a result of the non-shared L3 cache present in the chiplet-based Zen2 architecture, in spite of having a total 128 MiB of L3 cache.

As per Table 4.8, even though the memory bandwidth of the AMD node is only 30% higher than the Intel node, comparing the clock cycles needed to fetch a single double precision digit from memory between Fig. 3.5(b) and Fig. 3.5(c) shows that the AMD CPU takes about half the clock cycles as the Intel CPU when the data size exceeds that of LLC as shown in Table 3.2. This can be attributed to the fact that the AMD CPU employs full memory overlap between all caches whereas the Intel CPU is completely non-overlapping, as can be seen in Table 3.4. Therefore, even though the Intel CPU employs the AVX-512 instruction set with 512-bit long SIMD , the advantage still lies with the AVX2-enabled 256-bit long AMD CPU as a result of fully overlapping communication between caches.

Thus, Fig. 3.5 shows that the Fujitsu A64FX takes the least number of clock cycles to fetch a single double precision number from main memory. This can be explained as a result of the better cache overlap design in the Fujitsu A64FX as shown in Table 3.4.

Chapter 4

Cache blocking for batched low-rank matrix multiplication

4.1 Introduction

Large dense matrices and tensors appear in applications such as Boundary integral methods, machine learning, computational finance and multivariate regression. While direct multiplication and factorization of such data sets is computationally expensive, careful use of low rank approximation techniques can drastically reduce the compute and memory cost of these methods with a controllable decrease in accuracy. Examples of such applications are the use of hierarchical matrices [77] for factorization of large dense matrices, tensor decomposition for multilinear systems [36] and FMM in Deep Learning [113].

Maximizing the computational efficiency of such methods has its unique challenges, different from both dense matrices and sparse matrices. On one hand, it shares some of the challenges of sparse matrices, where the amount of arithmetic operations (Flops) per loaded data (Bytes) decreases with the rank of the low-rank blocks. On the other hand, it is different from sparse matrices in the sense that the low-rank blocks are not completely sparse but consist of small dense matrices, providing some opportunities for data reuse and prefetch. The resulting fine-grain regularity allows these methods to more efficiently utilize SIMD operations compared to the efficacy of sparse matrices. Optimizing the throughput of these structured low-rank matrices, however, is not an area that has been investigated sufficiently in the literature. We have attempted to address this gap by manually writing our own matrix multiplication kernels for low-rank matrices that greatly outperform vendor optimized libraries in respect to this particular issue.

This work focuses on optimizing the inner kernel for this new type of problem, where the matrix is neither dense nor sparse. As such, this work runs counter to other existing work that focus more on the parallel scalability of structured low-rank matrices. Although [50, 44, 116, 124, 151] report hierarchical matrix factorization on many hundreds of nodes, the computation shows sub-optimal resource utilization mainly attributed to library routines that are not efficiently tuned for

handling the memory bound kernels of such factorization routines. We revisit this, and address the improvement of the efficiency of the low rank kernels that form a core of the computation.

In this chapter we propose a new technique for optimizing a central component of structured low-rank matrices, the low rank matrix multiplication. Our technique performs batched computation of low rank matrices and shows, through a careful utilization of the different levels of cache, more efficiency than vendor optimized math libraries such as MKL, AMD-BLIS and SSL-2 (Scientific Software Library) for a variety of block and batch sizes. Thus, when compared to vendor libraries, our method shows stronger scaling for a shared memory execution. Specifically, we make two contributions in this thesis:

1. An improved algorithm for batched computation of low rank matrix multiplication, that can achieve more than $2x$ greater throughput than vendor optimized libraries for all the CPU architectures and problem sizes tested;

2. Techniques for optimization and performance validation of low level kernels with extensive use of the ECM [86] (Execution-Cache-Memory) performance model.

The rest of this chapter is organized as follows. Section 4.2 describes the new algorithm and proposed cache blocking methodology. Section 4.3 contains a detailed analysis of each kernel within our algorithm and presents the optimization on each of our target CPUs using the ECM performance model that was explained in detail in Section 3.3. We then provide the results of our method for our target CPUs compared to various vendor-optimized libraries in Section 4.4 and finally conclude in Section 4.5.

4.2 Batching Methodology

4.2.1 Looping order of Low rank multiplication

The usual low rank multiplication is shown in Algorithm 4. If done separately, this will lead to 3 nested loops for each multiplication. Since there exist dependencies between these loops, we end up writing the temporary blocks to memory after each step. With a rewrite of the loops, we can perform the batched matrix multiplication using 6 nested loops as shown in Algorithm 9. This way the result can be accumulated into a single variable G_{XY} and written back into memory only once, at the end of the computation. We also notice that all temporary blocks used in this new algorithm are tiny matrices (size $rank \times rank$) and can fit within the vector registers when $rank$ is sufficiently small. We term the A_X and B_X as 'small matrices' and A_{V^T} and B_U as 'skinny matrices'. Since the batch dimension is typically the largest dimension in a batched multiplication of small sized matrices, performing the multiplication as specified in Algorithm 9 allows for more optimal utilization of bandwidth and therefore reduces the amount of time spent in fetching data

Algorithm 9: Batched low rank multiplication expressed as 6 nested loops.

Input: $A_{V^T}\_batch$, B_U_batch, A_X_batch, B_X_batch
Result: G_{XY}_batch

```
1  for batch ← 0 to BATCH_SIZE do                                  /* Loop 1 */
2  |  A_{V^T} = A_{V^T}_batch(batch)
3  |  B_U = B_U_batch(batch)
4  |  A_X = A_X_batch(batch)
5  |  B_X = B_X_batch(batch)
6  |  G_XY = G_XY_batch(batch)
7  |  for m ← 0 to rank do                                         /* Loop 2 */
8  |  |  for n ← 0 to rank do                                      /* Loop 3 */
9  |  |  |  C_MN = 0 for k ← 0 to block_size do                   /* Loop 4 */
10 |  |  |  |  C_MN += A_{V^T}(m, k) × B_U(k, n)
11 |  |  |  end
12 |  |  |  for x ← 0 to rank do                                  /* Loop 5 */
13 |  |  |  |  E_XN = A_X(x, m) × C_MN for y ← 0 to rank do       /* Loop 6 */
14 |  |  |  |  |  G_XY(x, y) += E_XN × B_X(n, y)
15 |  |  |  |  end
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

from main memory for a multi-threaded implementation. This has been experimentally proven to be true in Section 4.4. Therefore, a combination of improved bandwidth utilization and accumulation of intermediate results within SIMD registers allows our algorithm to achieve superior results than vendor optimized libraries.

4.2.2 Locality optimization for low rank multiplication

BLISLAB [103] separates the implementation of the dense matrix multiplication into a portable macro kernel written in a high level language such as C, and an architecture-specific micro kernel typically written using intrinsics or assembly code. This allows libraries like BLIS to be portable across a diverse set of machines and exposes thread-level parallelism [127] in the macro kernel. This approach has been shown to attain near peak performance for a diverse set of CPU architectures as a result of enhanced data reuse.

We follow a similar approach for the low rank multiplication as shown in Algorithm 10. Assuming a three level cache hierarchy, loop 1A packs small matrices into the last level cache (L3) and loop 1B packs the skinny matrices into the L2 cache. Loop 1C then iterates over the batches of skinny matrices that are already packed in the cache. Thus loop 1 in Algorithm 9 is split into loop 1A, 1B and 1C in Algorithm 10. The data can then be streamed directly from the cache closest to the CPU (L1) when loading into registers. This can be changed to use only two cache levels in case of the A64FX CPU.

Algorithm 10: Batched low rank multiplication with a micro-kernel.

Input: $A_{V^T}\_batch$, B_U_batch, A_X_batch, B_X_batch
Result: G_{XY}_batch
`/* Pack as many small matrices in as possible in L3 cache with thread-level parallelism. */`
for $batch_{small} \leftarrow 0$ **to** $BATCH\_SIZE$ **step** B_{small} **do** `/* Loop 1A */`
  $packed\_A_X \leftarrow pack\_A_X\_matrices()$
  $packed\_B_X \leftarrow pack\_B_X\_matrices()$
  `/* Pack` B_{skinny} `skinny matrices into the L2 cache of each core. */`
  for $batch_{skinny} \leftarrow 0$ **to** $\frac{B_{small}}{B_{skinny}}$ **do** `/* Loop 1B */`
    $offset \leftarrow b_{small} \times B_{small} + b_{skinny} \times B_{skinny}$
    $packed\_B_U \leftarrow pack\_B_U()$
    $packed\_A_{V^T} \leftarrow pack\_A_{V^T}()$
    for $batch \leftarrow offset$ **to** $offset + B_{skinny}$ **do** `/* Loop 1C */`
      $G_{XY} \leftarrow G_{XY}\_batch(batch)$
      for $m_c \leftarrow 0$ **to** m **step** M_{PACK} **do** `/* Macro kernel. Loop 2 */`
        for $n_c \leftarrow 0$ **to** n **step** N_{PACK} **do** `/* Loop 3 */`
          $C_{MN} = micro\_kernel\_cmn($
            $m_c, n_c, packed\_B_U,$
            $packed\_A_{V^T}$
)
          for $x_c \leftarrow 0$ **to** x **step** X_{PACK} **do** `/* Loop 5 */`
            $E_{XN} = micro\_kernel\_exn($
              $m_c, x_c, packed\_A_X, C_{MN}$
)
            for $y_c \leftarrow 0$ **to** y **step** Y_{PACK} **do** `/* Loop 6 */`
              $G_{XY} = micro\_kernel\_gxy($
                $packed\_B_X, E_{XN},$
                n_c, y_c
)
            end
          end
        end
      end
    end
  end
end

CPU	$rank \geq VL$	X_{PACK}	Y_{PACK}	M_{PACK}	N_{PACK}
Fujitsu A64FX	YES/NO	8	8	8	8
Intel Xeon Gold 6148	YES	4	16	8	16
	NO	8	8	8	8
AMD EPYC 7502	YES/NO	4	4	4	4

Table 4.1: Values of slicing variables according to architecture and *rank*. It is experimentally found that AMD and Fujitsu micro kernels do not need modification irrespective of the *rank* whereas the Intel micro kernels perform best when the slice widths are changed if the rank is greater than the vector length (VL).

Algorithm 10 maintains portability across CPU architectures with varying cache sizes by changing the parameters B_{small} and B_{skinny} that control the number of small and skinny matrices being packed into cache, respectively. The number of small matrices B_{small} being packed into the LLC is determined using Eq. 4.1 (assuming **double** as the type of our matrices). Eq. 4.1 is obtained by dividing the number of bytes that the L3 cache can hold by the total number of bytes required for holding two $rank \times rank$ small matrices.

The L2 cache typically has enough capacity to hold multiple skinny matrices from each operand of the low rank multiplication for a variety of block and rank sizes. Fig. 4.1 shows the effect of changing the number of skinny matrices from each low rank operand packed into the L2 cache. Experimentally, we find that packing only a single skinny matrix from each low rank operand leads to the best performance when using a sufficiently large batch size of 20,000 using an entire CPU socket (20 cores). Therefore, we fix $B_{skinny} = 1$ for all future experiments.

Algorithm 10 shows how the loops shown in Algorithm 9 can be be expressed in terms of *macro kernel* loops shown by the corresponding loops 2,3,5 and 6, and three micro kernels, each for accumulating the C_{MN}, E_{XN} and G_{XY} block. Each of these blocks is of size $S_{VEC} \times S_{VEC}$ where S_{VEC} is the vector length of the CPU. Loop 4 from Algorithm 9 is absorbed into $micro\_kernel\_cmn()$ and optimized using assembly code. The macro kernel loops choose the slices of the packed blocks that must be computed by the micro kernels. The variables M_{PACK}, N_{PACK}, X_{PACK} and Y_{PACK} control the sizes of the slices that the macro kernel loops iterate over. These values are changed according to the architecture and the rank of the problem.

For the case where $rank = S_{VEC}$, an entire matrix G_{XY} of dimension $S_{VEC} \times S_{VEC}$ can be computed within the registers without having to perform expensive reads and writes of the temporary matrices C_{MN} and E_{XN}. In this case all the blocking variables in Algorithm 10, i.e. M_{PACK}, N_{PACK}, X_{PACK} and Y_{PACK} will be equal to $rank$. In these instances the computation can be performed directly within the vector registers without a single write to memory. When the $rank$ is too large to fit into the SIMD registers, we perform blocking by using register blocks of different values so that the multiplication time of the skinny matrices is minimized and yields the best performance. Table 4.1 shows the values of these variables when using a rank equal to the vector length and greater than the vector length for each CPU.

$$B_{small} = \lfloor \frac{LLC_{bytes}}{2 \times rank \times rank \times sizeof(double)} \rfloor \tag{4.1}$$

4.2.3 Packing techniques for minimization of latency

We tried out an alternate packing technique than that suggested in Section 4.2.2, where we packed the skinny matrices B_U and A_{V^T} matrices in the L3 cache and small matrices in the L2 cache. However, this showed lesser performance than packing the small matrices in the L3 cache. This is

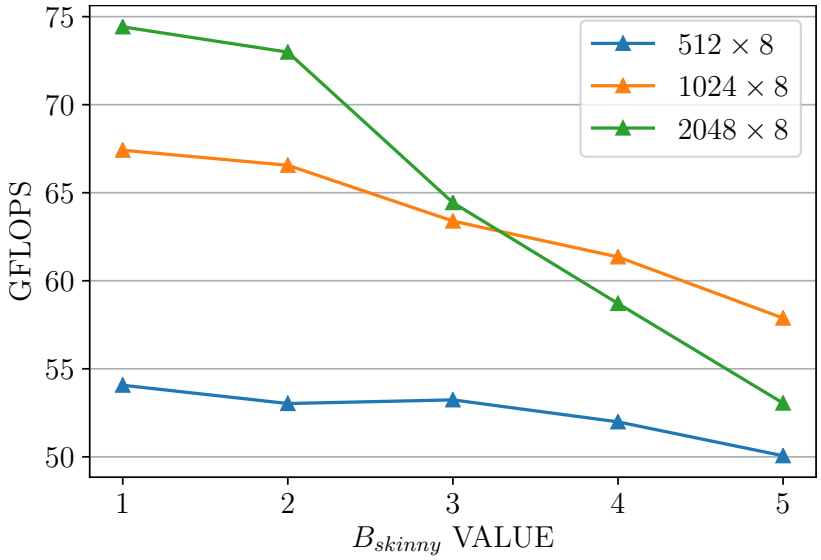


Figure 4.1: B_{skinny} is the number of skinny matrices from each low rank matrix operand being packed into the L2 cache. This experiment shows the variation in performance as B_{skinny} is varied, keeping a sufficiently large batch size constant at 20000 and using 20 physical cores of an Intel Xeon Gold 6148 CPU. It can be seen that $B_{skinny} = 1$ leads to the best performance, and this value is fixed in all future experiments.

because the time for fetching small matrices into the L3 cache is minimized when the outermost loop (loop 1A in Algorithm 10) performs this fetching since that loop has the most parallelism. Therefore, the small matrices can be packed into the cache with maximum bandwidth utilization.

Another technique is to pack the data across batches (similar to [98]). If this is done, the 6 loop structure used in Algorithm 9 cannot be used since it relies on processing each low rank matrix one after another and not when the matrices are interleaved. Moreover, the interleaved batched layout is slow for skinny matrices as shown in Fig. 3.2.

4.3 Single threaded optimization using the ECM performance model

The computation of the low rank multiplication kernels can be broadly divided into kernel operations that involve packing the 4 matrices into caches, followed by the computation in the *micro_kernel_cmn*(), *micro_kernel_exn*() and *micro_kernel_gxy*() as shown in Algorithm 10. For brevity, we refer to *micro_kernel_cmn*(), *micro_kernel_exn*() and *micro_kernel_gxy*() from Algorithm 10 as the C_{MN}, E_{XN} and G_{XY} kernels, respectively, after the intermediate products that they compute.

We use the ECM model for optimizing the performance of the C_{MN} kernel and the packing of A_{V^T} and B_U since these are the most expensive parts. Using the ECM model we have identified and addressed several bottlenecks in the default C++ code, allowing us to reach very close to the maximum machine throughput. Fig. 4.2 shows the packing order of the operands into the caches.

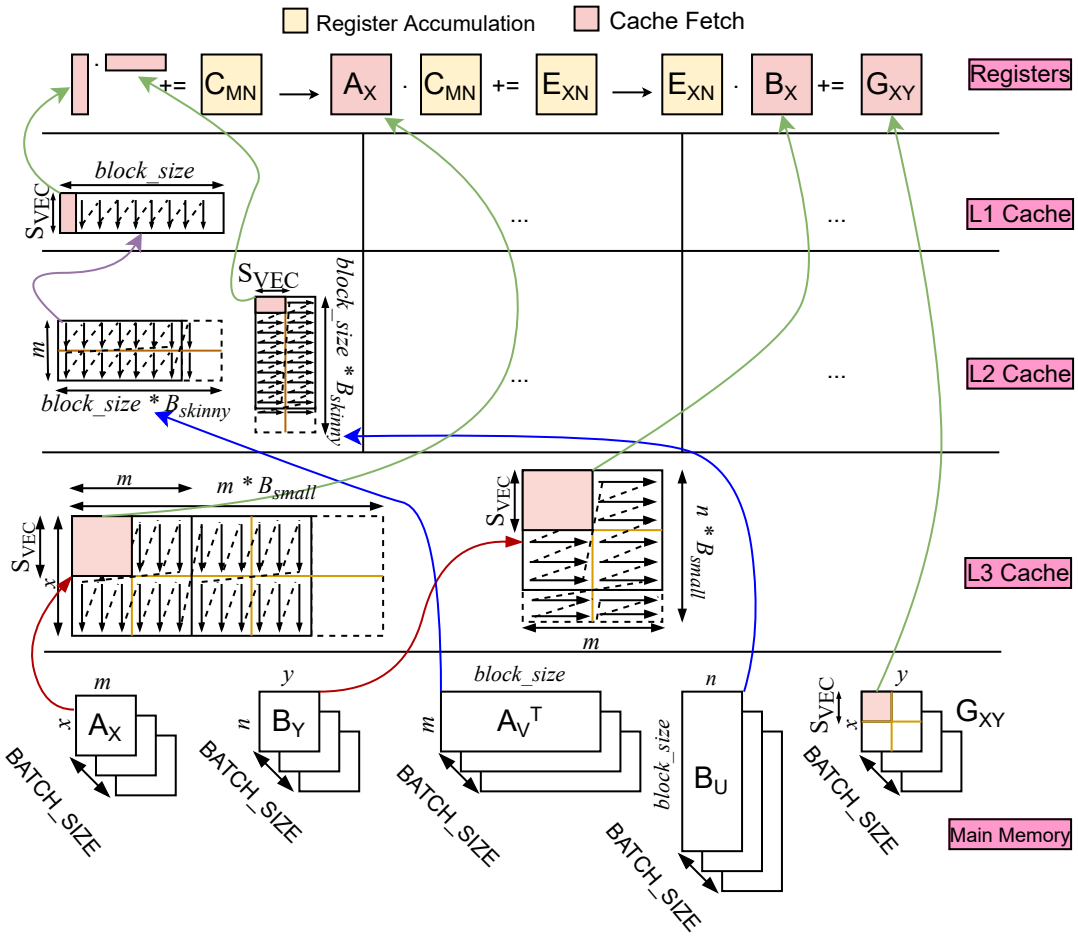


Figure 4.2: Diagram of the proposed low rank multiplication batching method.

Instruction	Description	Latency	Reci. TPut.
AMD EPYC 7502 (Zen2)			
VMOVAPD(simple load)	Standard load with immediate addressing.	5	0.5
VMOVAPD(simple store)	Standard store with immediate addressing.	4	0.75
LEA	Load effective address.	2	0.5
VGATHERQPD(gather)	Gather with stride of 8/16/32/512/1024/2048/4096 .	-	6
VBROADCASTSD(simple)	Standard broadcast with a single double in memory.	4	0.75
VFMADD231PD(simple)	FMA between 3 256 bit AVX registers.	5	0.5
Fujitsu A64FX (ARM v8.2 SVE)			
LD1D(simple)	Standard load with immediate addressing.	9	0.5
LD1D(gather, stride 8)	Gather with stride of 8.	-	2
LD1D(gather, stride 16/32/512/1024/4096)	Gather with stride of 16/32/512/1024/4096.	-	4
LD1D(gather, stride 2048)	Gather with stride of 2048.	-	16
ST1D(simple)	Standard store with immediate addressing.	9	1
LD1RD(simple)	Standard broadcast with a single double in memory.	9	0.5
FMLA(simple)	FMA between 3 512 bit SVE registers.	11	0.5
Intel Xeon Gold 6148 (Skylake-X)			
VMOVAPD(simple load)	Standard load with immediate addressing.	3	0.33
VMOVAPD(simple store)	Standard store with immediate addressing.	4	0.66
VGATHERQPD(gather)	Gather with stride of 8/16/32/512/1024/2048/4096 .	-	3
VBROADCASTSD(simple)	Standard broadcast with a single double in memory.	1	0.33
VFMADD231PD(simple)	FMA between 3 512 bit AVX registers.	5	0.5

Table 4.2: Latency and throughput of instructions depending on their operators for each CPU tested. All operations except memory-specific operations are performed on double precision floating point numbers.

The small operands are packed into the shared L3 cache whereas the skinny matrices are packed into the per-core L2 and L1 caches. The A_{V^T} is packed in column major whereas the B_U is packed in row major for facilitating loading into SIMD registers [103].

We calculate the latency and throughput of each instruction used in the kernels, and report our findings in Table 4.2. Note that we only consider bench-marking the inner kernel execution and not the full computation, therefore we ignore all instructions outside of the specified kernel. However, the full computation is considered when reporting the final results in Section 4.4.

We demonstrate the use of ECM modeling for the packing kernels when using ranks that are multiples of the vector length of the machine. Indeed, as highlighted in [86, 3.4.2] the ECM prediction is a function of the total number of full cache lines transferred, so we need to count in full cache lines even if only a part of the cache line will be used. In case of strided cases with strides greater than *rank*, extra reads are considered as a result of reading more cache lines in order to factor in the increased stride.

When reporting ECM model predictions, we report them as $T_{value} = (read) + (write)$ in order to differentiate between read and write contributions in case of caches where both values contribute to the outcome. In other cases the values show only read or only write contributions.

4.3.1 Overall comparison of analytical and empirical kernel performance

We first show analytical vs. empirical results for the computation C_{MN} and packing of B_U and A_{V^T} for all CPUs. We address the challenges faced in reaching approximate peak ECM performance for each CPU in subsequent sections. We then elaborate on the strategy used for determining T_{ECM}.

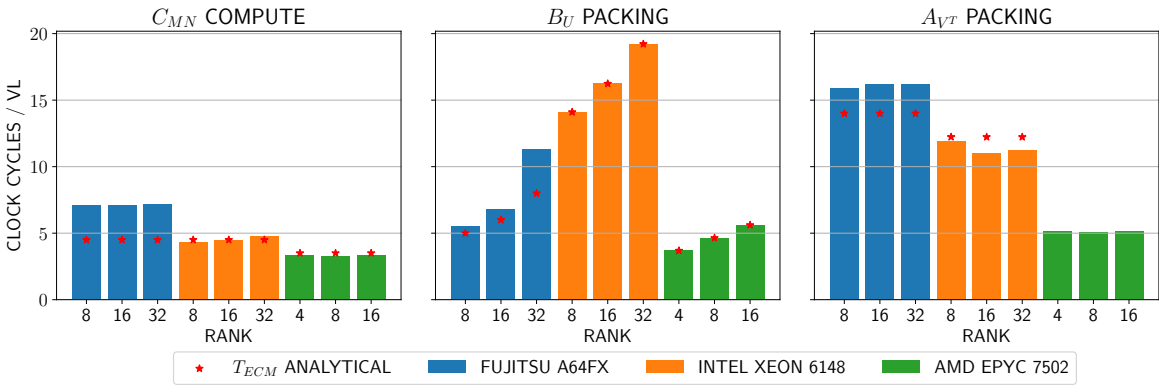


Figure 4.3: Comparison of analytical and empirical number of clock cycles for packing A_{V^T} and B_U and computing C_{MN}. All tests are performed for a block size of 1024 and batch size 10000 using a single thread of execution. The ranks are varied as shown in the X-axis. The Y-axis shows the number of clock cycles taken per vector length. The analytical measurements for the A_{V^T} packing of the AMD 7502 is not reported because the ECM model broke down for these problem sizes.

Fig. 4.3 shows the empirical performance of the three kernels vs. the analytical peak performance determined using the ECM model for each. It can be seen that our code is able to reach theoretical peak performance as determined by the ECM model for almost all cases. In subsequent experiments, we term the number of matrices in a batch as the batch size and the longest dimension of the skinny matrices as the block size. We assume that both low rank operands have equal rank and equal block size.

We report data only for block size 1024 as we do not observe a significant deviation in the results for larger block sizes, except for A_{V^T} packing using block size 2048 on the Fujitsu A64FX, which we elaborate on in Section 4.3.2.1.

4.3.2 Optimization on the Fujitsu A64FX with the ECM model

As shown in Table 4.2, the throughput of the `LD1D` instructions changes as the operands change. The reciprocal throughput of this instruction when performing a gather operation changes as the stride changes. We observe an anomaly when using a stride of 8 and 2048, where the reciprocal throughput changes to 2 and 16 respectively.

4.3.2.1 Packing A_{V^T} and B_U performance analysis

We rely on manually inserting SVE gather and load intrinsics for reducing the packing time, since we found that the Fujitsu compiler does not make full use of vectorization. The B_U matrix is packed in row major order, corresponding to the format in which it is already stored, so we can utilize `LD1D (load)` instructions for this purpose. Table 4.3 shows the ECM calculations for a variety of possible strides for B_U.

Variable	Stride 8	Stride 16	Stride 32
T_{L1_L}	0.5	0.5	0.5
T_{L1_S}	1	1	1
T_{L2}	$(\frac{64}{64}) + (\frac{64\times2}{64}) = 3$	$(\frac{64\times2}{64}) + (\frac{64\times2}{64}) = 4$	$(\frac{64\times4}{64}) + (\frac{64\times2}{64}) = 6$
T_{mem}	$(\frac{64}{64}) + (\frac{64}{32} + \frac{64}{32}) = 5$	$(\frac{(64\times2)}{64}) + (\frac{64}{32} + \frac{64}{32}) = 6$	$(\frac{(64\times4)}{64}) + (\frac{64}{32} + \frac{64}{32}) = 8$
T_{ECM}	5	6	8

Table 4.3: ECM performance breakdown for packing B_U for various strides for Fujitsu A64FX. All measurements are reported in number of clock cycles.

Block	Rank	A_V analytical	A_V empirical
2048	8	26	26.32
	16	26	26.22
	32	26	26.30

Table 4.4: Packing time per vector length for A_{V^T} for Fujitsu A64FX for block size 2048. All measurements are shown in clock cycles per VL for a batch size of 10000.

The A_{V^T} matrix is packed in column-major order, and similarly to the packing of B_U benefits from manually inserting SVE gather intrinsics. While Fig. 4.3 shows the empirical vs. analytical time taken when the block size is 1024. For most strides, the T_{ECM} can be calculated by setting $T_{L1_L} = 4$, $T_{L1_S} = 1$, $T_{L2} = (\frac{64\times8}{64}) + (\frac{64}{64} + \frac{64}{64}) = 10$ and $T_{mem} = (\frac{64\times8}{64}) + (\frac{64}{32} + \frac{64}{32}) = 12$. We observe however a performance anomaly for a stride of 2048 corresponding to the increased latency of the `LD1D` instruction shown in Table 4.2. Table 4.4 shows the empirical vs. analytical performance when the block size is 2048.

4.3.2.2 C_{MN} kernel performance analysis

The basic ARM SVE loop of the CMN kernel uses one `LD1D` instruction for loading 8 unique contiguous elements of the left operand, 8 FMA instructions for performing 8 SIMD multiplications and 8 `LD1RD` instructions for loading 8 elements of the right operand duplicated within each register for each rank-1 update that results in a matrix of size 8×8.

It can be seen that there are 8 `LD1RD` operations and 1 `LD1D` operation each taking 0.5 clock cycles, thus leading to a T_{L1_L} of 4.5. There are a total of 8 FMA operations, leading to a total T_c of 4 clock cycles. Thus $T_{ECM} = max(T_c, T_{L1_L}) = 4.5$ for a single rank-1 update.

We began by writing ARM ACLE instrinsics for the CMN kernel, but this turned out to be taking many more clock cycles than the ideal shown by the ECM model as a result of extra instructions generated in order to maintain portability between varying SVE lengths [129, 3.1]. We then enabled register length specific code generation and kept a fixed vector length of 512 using FCC compiler options, which led to more optimized code.

Table 4.2 shows that both the `LD1RD` and `FMA` instructions have a reciprocal throughput of 0.5 cycles each. Given that one rank-1 update generates an 8×8 matrix, a single `LD1RD` and `FMA` will together generate one row of this matrix, taking 1 clock cycle. 8 of these pairs will use 8 clock cycles.

Block	Rank	T_{ECM}	SVE ACLE	SVE intrinsics	SVE multi-register intrinsics
1024	8	4.5	30.11	8.59	7.12
	16	4.5	29.81	8.58	7.10
	32	4.5	29.77	8.78	7.15

Table 4.5: Comparison of successive optimizations to the CMN kernel compared to T_{ECM} for batch size of 10000 for the Fujitsu A64FX. All measurements are in terms of clock cycles per rank-1 update.

The reason why these instructions are executed likewise is because we accumulate the intermediate products within the available SIMD registers, which places an upper limit on the size of the rank-1 update that can be performed in the CMN block. This, combined with the first `LD1D` will take about 8.5 clock cycles. As far as the `LD1RD` and `FMA` instructions are concerned, exactly 64 bytes are computed for 64 bytes loaded, which leads to a 1:1 ratio between the flops and bytes. For ideal `FMA` throughput, this ratio must be at least 2:1 so that two `FMA` instructions can execute independently on the two available FMA ports of the A64FX.

In order to overcome this limitation, we utilize 8 extra registers and perform two separate rank-1 updates using alternate slices of the skinny matrices to improve the performance further. We then add these slices at the end of the of the computation in order to obtain the block in registers z0-7. This leads to the time dropping to about 7 cycles per rank-1 update as a result of improved port pressure. The technique can be described as in Fig. 4.4.

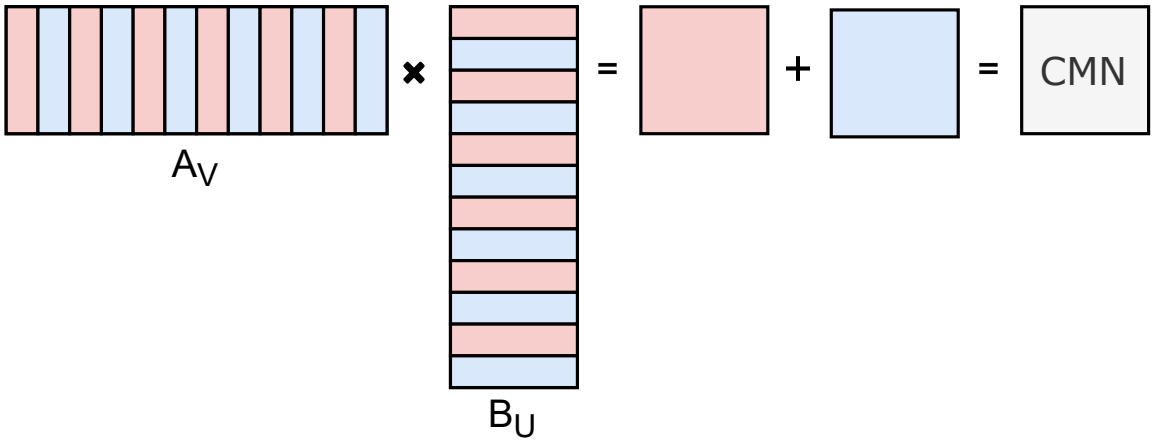


Figure 4.4: Rank-1 update for 8x8 matrix block using alternate rows and columns of the skinny matrices and then adding the blocks at the end.

The ECM model assumes that instructions run at peak throughput, however that can never be true in this case since the upper limit on the number of FMA instructions that can be executed given the constraints on blocking has been reached. Therefore, about 7 cycles per rank-1 update is the best that can be achieved given the constraints on the number of usable registers.

Table 4.5 shows the comparison of the analytical T_{ECM} compared to the performance obtained by successive optimizations. The usefulness of the ECM model can be seen in support of optimization.

Variable	Stride 8	Stride 16	Stride 32
T_{L1_L}	0.33	0.33	0.33
T_{L1_S}	0.66	0.66	0.66
T_{L2}	$(\frac{64}{64}) + (\frac{64}{64}) = 2$	$(2 \times \frac{64}{64}) + (\frac{64}{64}) = 3$	$(4 \times \frac{64}{64}) + (\frac{64}{64}) = 5$
T_{L3}	$(\frac{64}{64}) + (\frac{64}{64}) = 2$	$(2 \times \frac{64}{64}) + (\frac{64}{64}) = 3$	$(4 \times \frac{64}{64}) + (\frac{64}{64}) = 5$
T_{mem}	$(\frac{64}{14}) + (\frac{64}{14}) = 9$	$(\frac{64}{14}) + (\frac{64}{14}) = 9$	$(\frac{64}{14}) + (\frac{64}{14}) = 9$
T_{ECM}	14	16	20

Table 4.6: ECM performance breakdown for packing B_U for various strides for Intel Xeon Gold 6148 (Skylake-X).

4.3.3 Optimization on Intel Xeon Gold 6148 with the ECM model

4.3.3.1 Packing A_{V^T} and B_U performance analysis

The ECM models for packing B_U can be constructed by first noting the fact that packing one VL (i.e. 8 doubles) requires one load (VMOVAPD(load)) and one store (VMOVAPD(store)) operation. These operations have a throughput of 0.33 and 0.66, respectively. The breakdown of the analytical ECM modeling is shown in Table 4.6.

When packing A_{V^T}, we make use of gather instructions, which means there is one `VGATHERQPD` and one store (`VMOVAPD(store)`) being used per VL. It is hard to exactly model the behaviour of the gather instruction since it seems to be fetching several cache lines together without incurring the overhead for fetching each cache line individually. Our explanation is that as a result of page-based fetching for the L3 cache, we can ignore the fact that multiple cache lines are being fetched and model that as a single cache line fetch. With this assumption, we can state that $T_{L1_L} = 3$, $T_{L2_S} = 0.66$, $T_{L2} = \frac{64}{64}$, $T_{L3} = 2 \times \frac{64}{64} = 2$ and $T_{mem} = \frac{64}{14} = 4.57$.

4.3.3.2 C_{MN} kernel performance analysis

Each rank-1 update on the C_{MN} kernel requires 1 `VMOVAPD`, 8 `VBROADCASTSD`, and 8 `VFMADD231PD` instructions. Given that the data is always streamed from the L1 cache, the only cost is $T_{L1_L} = T_{ECM} = 4.66$.

4.3.4 Optimization on AMD EPYC 7502 with the ECM model

In order to keep the comparison between A64FX, AVX-512 and AVX2 fair, we use data sizes that correspond to the vector length and its multiples for building the ECM models. Therefore, in case of AVX2, the ranks used are 4, 8 and 16 which correspond to the VL, twice the VL and four times the VL for each instruction. These factors of the VL are proportional to the factors taken for A64FX and AVX512, for which the ranks are 8, 16 and 32 as shown in Section 4.3.3 and Section 4.3.2 respectively. As shown in Section 4.4.4, the target application of the low rank multiplication is the matrix vector multiplication routine for a block low rank matrix. The accuracy of the multiplication can be changed using the admissibility condition, and we do not need to change the rank of the low

Variable	Stride 4	Stride 8	Stride 16
T_{L1_L}	0.5	0.5	0.5
T_{L1_S}	0.75	0.75	0.75
T_{L2}	$\frac{32}{32} + 2 \times \frac{32}{32} = 3$	$2 \times \frac{32}{32} + 2 \times \frac{32}{32} = 4$	$4 \times \frac{32}{32} + 2 \times \frac{32}{32} = 6$
T_{L3}	$\frac{32}{32} + 2 \times \frac{32}{32} = 3$	$2 \times \frac{32}{32} + 2 \times \frac{32}{32} = 4$	$4 \times \frac{32}{32} + 2 \times \frac{32}{32} = 6$
T_{mem}	$\frac{32}{16} + \frac{32}{16} = 4$	$\frac{32}{16} + \frac{32}{16} = 4$	$\frac{32}{16} + \frac{32}{16} = 4$
T_{ECM}	4	4	6

Table 4.7: ECM performance breakdown for packing B_U for various strides for a single thread on the AMD EPYC 7502. All measurements are reported in number of clock cycles.

rank matrices in order to modify the accuracy. Therefore, we test only for multiples of the SIMD register length for each CPU.

4.3.4.1 Packing A_{V^T} and B_U performance analysis

We can derive the ECM model by using the throughput values from Table 4.2 and the machine model from Table 3.2. B_U packing takes one LOAD and one STORE operation. Since the clock cycles depend on the stride, the theoretical performance for various values of stride are shown in Table 4.7.

4.3.4.2 C_{MN} kernel performance analysis

The C_{MN} kernel in this case is very similar to that in Intel. The difference being that the VL is limited to 4 due to the AVX2 instruction set. Therefore, the rank-1 update in this case is for a 4x4 matrix block, unlike the 8x8 matrix block for the Intel. For each rank-1 update, we use 4 `VBROADCASTSD`, 4 `FMADD231PD` and 1 `VMOVAPD` instructions. This amounts to $T_{L1_L} = 3.5$. Since the data is directly streamed from the L1 cache, all other terms in the ECM equation are 0 and therefore $T_{ECM} = 3.5$.

4.4 Experimental Evaluation

We perform experiments using the nodes and compiler flags listed in Table 4.8. The hardware specification of the CPU within each node can be found in Table 3.2. Our method works for any kind of data. However, for these experiments we use randomly generated entries following a normal distribution in order to accurately evaluate all of our test matrices. Since we are only working with low rank multiplication, the data within the low rank blocks does not affect our results. All tests are performed using double precision floating point numbers.

$$GFLOPS = \frac{batch\_size \times (4 \times rank^3 + 2 \times rank^2 \times block\_size)}{time(s)} \times 10^{-9} \tag{4.2}$$

	AMD	Fujitsu	Intel
CPUs	2 x AMD EPYC 7502	1 x A64FX	2 x Intel Skylake-X 6148
Memory	2 x 256 GiB	1 x 32 GiB (HBM)	2 × 192 GiB
NUMA configuration	4/CPU	4/CPU	1 NUMA node/CPU
Compiler	g++ 7.5.0	FCC 4.5.0 tcsds-1.2.31	g++ 7.4.0
Compile options	-Wall -fopenmp -O3 -Ofast -mavx2 -funroll-loops -masm=intel	-O3 -Nfjomplib -fopenmp -Kfast,zfill -Kopenmp -Ksimd_reg_size=512	-Wall -fopenmp -O3 -Ofast -march=skylake-avx512 -masm=intel
Math library	AMD BLIS 3.0.0	Fujitsu SSL-2	Intel MKL 2020.4
Math library linking options	libblis-mt.a -lgomp -lpthread -lm -ldl	-Kopenmp -Nfjomplib -lfjlapacksve	-lmkl_intel_ilp64 -lmkl_gnu_thread -lmkl_core -lgomp -lpthread -lm -ldl
TRIAD peak (Gb/s)	195	840	150
DGEMM peak (GFLOPS)	2184	2828	2621

Table 4.8: Machine architecture and the corresponding configuration used in our experiments. The Intel node is a single node of the ABCI supercomputer and the Fujitsu node is a single node of the FUGAKU supercomputer. The AMD node is a stand-alone SMP machine.

The GFLOPS, where shown, are calculated as presented in Eq. 4.2. Bandwidth calculation depends on the cache overlapping displayed by the particular CPU and will be specified where necessary. We compare the bandwidth of each problem size with the STREAM TRIAD bandwidth for a given number of physical cores in order to demonstrate the 'ideal' usable bandwidth vs. what is actually realized.

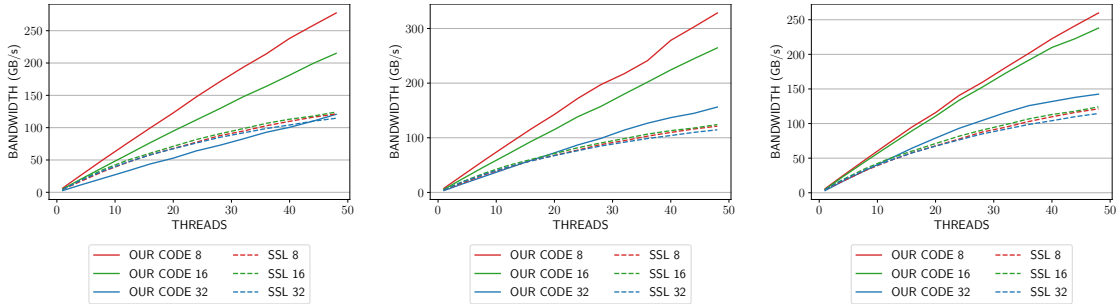
Experiments are performed using the full node available. For all cases except the A64FX we use the OpenMP configuration as `OMP_PLACES=cores` and `OMP_PROC_BIND=close`. We run all tests using `numactl` using the `--membind=all` configuration and set `--physcpubind` to bind distinct physical cores. The Fujitsu runtime in FUGAKU imposes a slightly different binding process than other machines. Spawning a single process per CMG (i.e. one process per 12 cores) is the recommended configuration for using all NUMA nodes with strictly local access. Thus, we run 4 MPI processes per node for the A64FX tests. The plots for the performance show the 90% confidence intervals of the data. We perform 100 experiments for each data point and use the `interval` function from scipy's statistics module for obtaining the confidence interval. It can be seen that there is very little variation in the data across multiple experiments.

4.4.1 Evaluation on the Fujitsu node

The utilization on the Fujitsu A64FX is shown in Fig. 4.5. It can be seen that our code outperforms Fujitsu's SSL-2 library by a wide margin except for one case using rank 32 and block size 2048

(a) Block size 512 (b) Block size 1024 (c) Block size 2048

Figure 4.5: Comparison of performance in GFLOPS vs. non-batched SSL-2 routines for Fujitsu A64FX. The batch size is kept constant at 20,000 for all the tests. The legend indicates the rank for each plot.



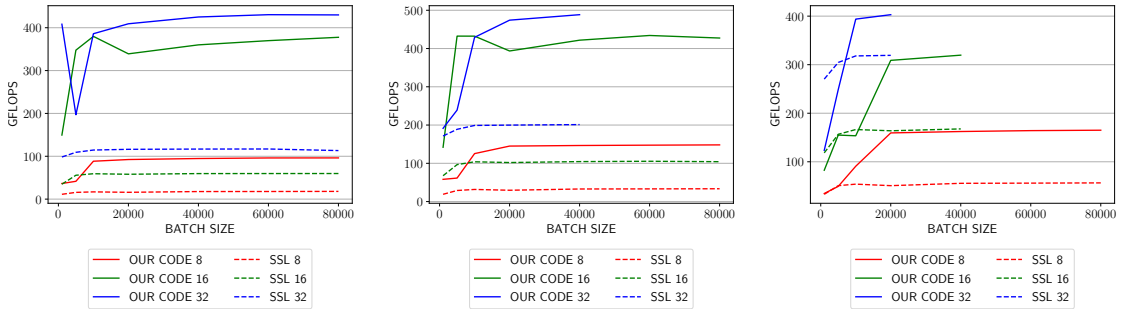
(a) Block size 512 (b) Block size 1024 (c) Block size 2048

Figure 4.6: Bandwidth utilization for varying ranks and block sizes on Fujitsu A64FX. The legend shows the rank for each plot. The bandwidth is calculated using Eq. 4.3. The peak STREAM TRIAD bandwidth for the A64FX is about 840 GB/s.

when not using all the cores in the CPU. The bandwidth utilization plots in Fig. 4.6 show that, although the GFLOPS utilization for rank 32 is consistently higher than other ranks, the bandwidth utilization is lower. This result indicates that the rank 32 case is in fact compute bound and not memory bound when using smaller ranks. Our method is able to achieve almost a 2x gain over vendor optimized BLAS libraries based on the 90% confidence intervals shown in Fig. 4.5. Since the confidence intervals are too small to be seen clearly on the picture, we can confidently say that our method outperforms vendor supplied BLAS by 2x.

$$BW(GiB/s) = \frac{\text{batch\_size} \times (2 \times \text{rank}^2 + 2 \times \text{rank} \times \text{block\_size}) \times \text{sizeof}(double) \times 2^{-30}}{time(s)} \tag{4.3}$$

Lack of linear strong scaling can be seen when using rank 8 and block size 512 in Fig. 4.5, which gradually improves as the block size is increased. This effect is not observed for other ranks that show almost uniform linear scaling. The reason for this can be seen from the bandwidth utilization plot in Fig. 4.6, where the usage of the bandwidth is proportional to the GFLOPS utilization. The



(a) Block size 512 (b) Block size 1024 (c) Block size 2048

Figure 4.7: Performance for various block sizes and ranks when the number of threads is constant at 48 for varying batch sizes for Fujitsu A64FX.

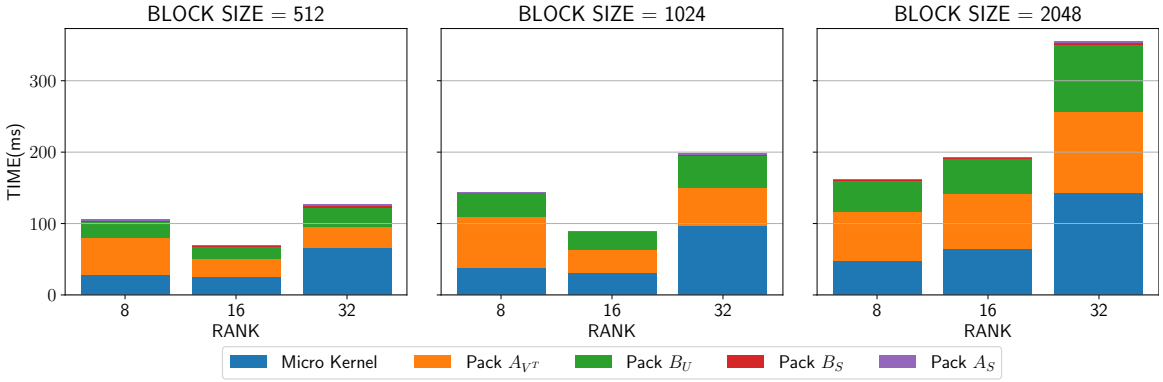


Figure 4.8: Performance breakdown for various parts of the computation for the Fujitsu node using 48 threads and a constant batch size of 20000.

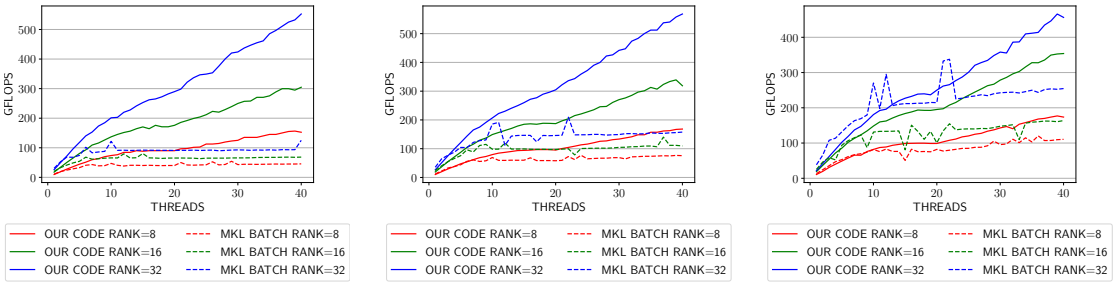
bandwidth is calculated as shown in Eq. 4.3. The bandwidth utilization is lesser than for rank 16 for the same block sizes since packing smaller skinny matrices individually into the L1 cache does not lead to optimal bandwidth utilization. Increasing the number of skinny matrices that are packed into the L1 cache during a single iteration of Loop 1 in Algorithm 9 might be a way to solve this problem.

When using rank 32 and block size 2048 in Fig. 4.6(c), it can be seen that the bandwidth utilization of SSL-2 and our code is almost the same unless all 4 NUMA nodes within the CPU are active, which happens after 36 threads are active. Fig. 4.7 shows the variation of the performance in GFLOPS as the number of threads is kept constant at 48 and batch size is varied. Our code consistently outperforms SSL-2, and there is minimal variation in the GFLOPS as the batch size is changed. Some results for the Fujitsu node could not be reported due to the limited 32 GiB HBM. However, it can be seen that each plot plateaus before we run out of memory, and therefore we can assume that performance will not degrade if there were more memory.

Fig. 4.8 shows the performance breakdown for each kernel of the computation on the A64FX when using 48 physical cores.

Batch size	Block size	Rank	SSL	OUR CODE
20000	512	96	370.62	207.55
20000	1024	96	598.93	343.912
20000	2048	96	851.74	521.93

Table 4.9: Performance of our code vs. SSL for larger rank on the Fujitsu node using 48 physical cores reported in GFLOPS. SSL shows better performance as the computation becomes more compute bound.



(a) Block size 512 (b) Block size 1024 (c) Block size 2048

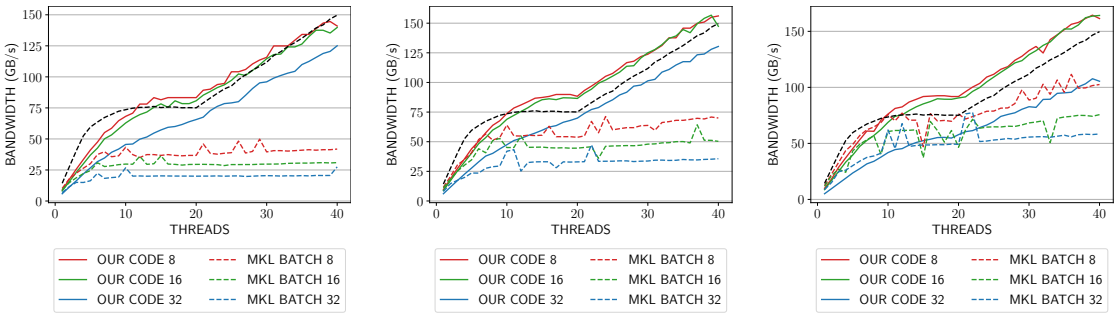
Figure 4.9: Comparison of performance in GFLOPS vs. batched MKL routines for Intel Skylake-X. The batch size is kept constant at 20,000 for all the tests.

Table 4.9 shows that as the rank increases beyond 96, we can observe SSL showing better performance than our code. This can be attributed to the fact that the algorithm becomes more compute bound. The performance actually drops below that of rank 32, which can be attributed to the fact that better bandwidth utilization as shown in Fig. 4.6 becomes harder as a lesser number of small matrices can be packed into the L1 cache. This is one of the drawbacks of relying on packing intermediate products into the SIMD registers.

4.4.2 Evaluation on the Intel node

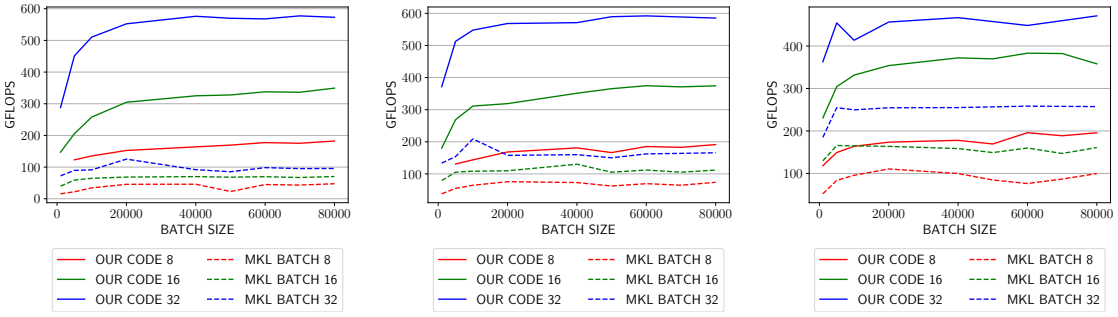
Fig. 4.9 shows the utilization in GFLOPS when the batch size is kept constant at 20,000 for a varying number of threads, block sizes, and ranks. It can be seen that our approach shows almost perfect scaling with respect to the number of threads whereas batched MKL routines stop scaling after approximately 10 physical cores have been utilized for all problem cases. We do not report findings for non-batched MKL routines since they show the least competitive performance for all problem cases.

The improved strong scaling can be attributed to the fact that our approach is able to saturate the maximum available bandwidth much better than batched MKL, as can be seen in Fig. 4.10, which shows the bandwidth utilization in GiB/second with a constant batch size of 20,000 and varying number of threads, block sizes and ranks. Comparison with batched MKL shows that our algorithm is able to saturate the available bandwidth much more optimally as a result of our unique packing strategy.



(a) Block size 512 (b) Block size 1024 (c) Block size 2048

Figure 4.10: Comparison of bandwidth utilization for varying ranks, threads and block sizes on Intel Xeon Gold 6148 (Skylake-X) for a constant batch size of 20,000. The black dashed line denotes the STREAM TRIAD bandwidth for the given number of threads.



(a) Block size 512 (b) Block size 1024 (c) Block size 2048

Figure 4.11: Performance for various block sizes and ranks when the number of threads is constant at 40 for the Intel Xeon Gold 6148 (Skylake-X).

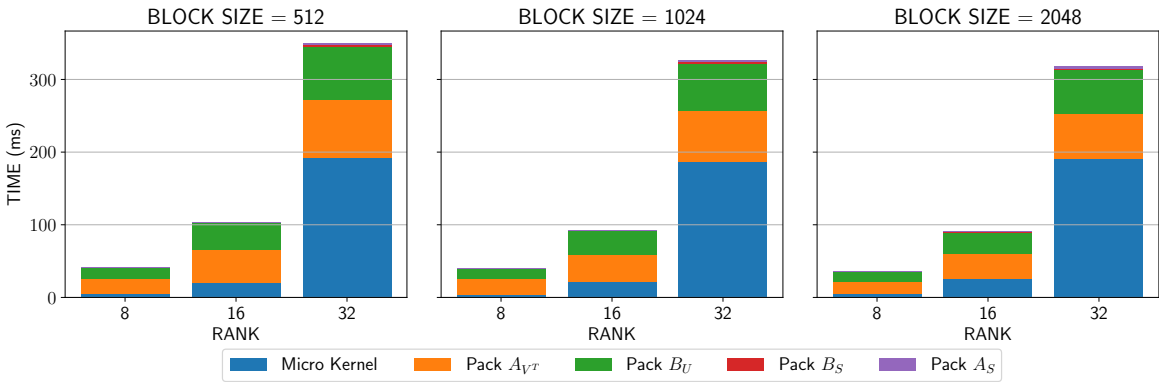


Figure 4.12: Performance breakdown for various parts of the computation for the Intel node using 40 threads and a batch size of 20000.

Batch size	Block size	Rank	MKL BATCH	OUR CODE
20000	512	128	262.281	339.849
20000	1024	128	400.602	361.89
20000	2048	128	554.721	319.161

Table 4.10: Performance of our code vs. MKL batched for larger rank on the Intel node using 40 physical cores in GFLOPS. Batched MKL shows better performance as the computation becomes more compute bound.

The bandwidth numbers shown in Fig. 4.10 are calculated as shown in Eq. 4.4. We use the term $3 \times rank^2$ in order to account for the write of the result matrix and reads of two small matrices. As the ECM model for Intel Skylake-X proves in Section 3.3.3, the reads and writes for this CPU are completely non-overlapping and the write term must be added into the bandwidth calculation.

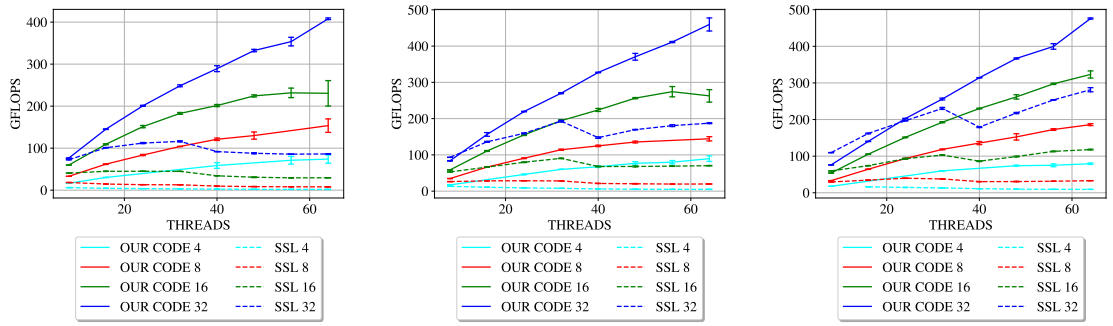
$$BW(GiB/s) = \frac{batch\_size \times (3 \times rank^2 + 2 \times rank \times block\_size) \times \text{sizeof}(double) \times 2^{-30}}{time(s)} \tag{4.4}$$

Fig. 4.11 shows the performance when changing the batch size, block size and rank and keeping the number of threads constant at 40. It can be seen that both our implementation, and batched MKL show almost constant performance irrespective of the batch size. This, combined with Fig. 4.9 shows that the scaling of the method is primarily limited by the available bandwidth, rank and block size. Additionally, increasing the batch size does not have any effect on the performance.

As the rank increases beyond 128, we can observe MKL batched showing better performance than our code. Table 4.10 shows the difference in performance as the computation becomes more compute bound than memory bound.

4.4.3 Evaluation on the AMD node

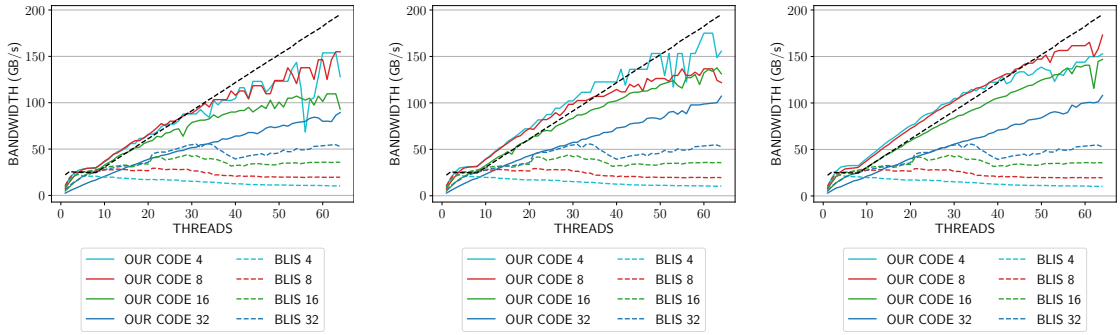
As shown in Table 3.2, the vector length of the AMD CPU is 4, whereas that of the Intel and Fujitsu CPUs is 8. Therefore, we show benchmarks for rank 4 in case of the AMD chip as well as for rank 8, 16 and 32 as shown for the others. Fig. 4.13 shows the utilization as calculated using Eq. 4.2,



(a) Block size 512 (b) Block size 1024 (c) Block size 2048

Figure 4.13: Comparison of performance in GFLOPS vs. non-batched AMD BLIS routines for AMD EPYC 7502. The batch size is kept constant at 20,000 for all the tests. The legend indicates the rank for each plot.

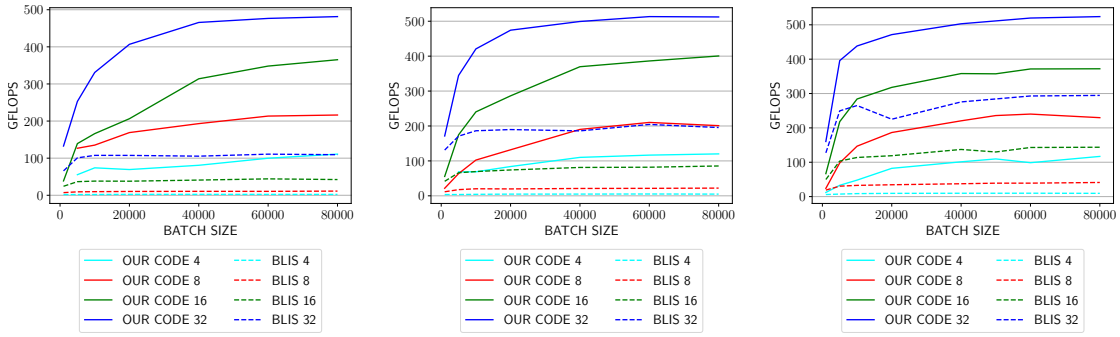
which shows that our implementation is able to outperform the vendor optimized AMD BLIS 3.0.0 BLAS implementation by a wide margin when a sufficient number of physical cores are active. The performance improves as we increase the rank, as expected, since memory bandwidth becomes less of a bottleneck as more data becomes available to keep the FMA units of the CPU busy.



(a) Block size 512 (b) Block size 1024 (c) Block size 2048

Figure 4.14: Bandwidth utilization for varying ranks and block sizes on AMD EPYC 7502. The legend shows the rank for each plot. The black dashed line shows the STREAM TRIAD bandwidth for the given number of threads.

Fig. 4.14 shows the bandwidth utilization of our code vs. AMD BLIS for a variety of problem sizes. The bandwidth is calculated as shown in Eq. 4.3. For ranks 4 and 8, the bandwidth increases in proportion to the performance for all block sizes, as shown in Fig. 4.13. However, for rank 16 and 32, we observe that the total utilized bandwidth does not saturate the available bandwidth of the system, even though the corresponding GFLOPS utilization in 4.13 shows that the utilization for ranks 16 and 32 is much higher than that for 4 and 8. This phenomena can be explained by the fact that the AMD EPYC CPU implements fully overlapping caches as pointed out in Section 3.3.3. Therefore, the computation for rank 16 and 32 is more compute bound than memory bound when a sufficient number of physical cores are active. Compared to the bandwidth behaviour of the Intel



(a) Block size 512 (b) Block size 1024 (c) Block size 2048

Figure 4.15: Performance for various block sizes and ranks when the number of threads is constant at 64 for varying batch sizes for AMD EPYC 7502.

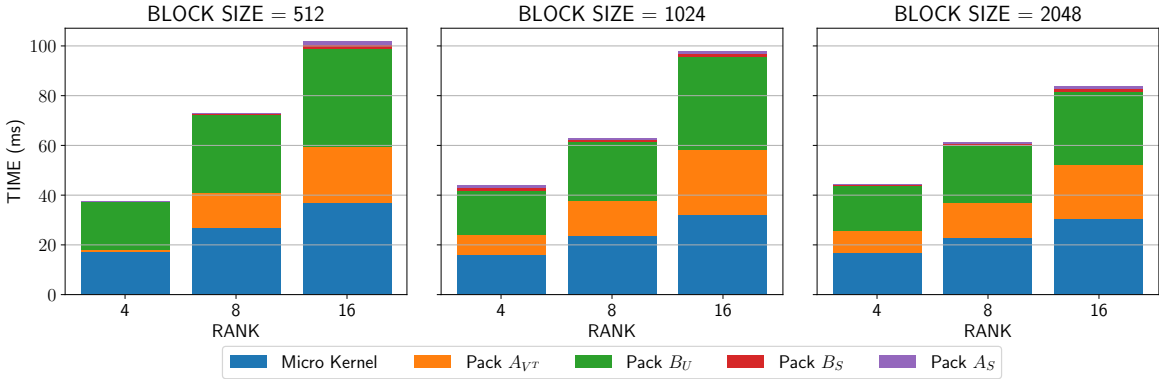


Figure 4.16: Performance breakdown for various parts of the computation for the AMD node using 64 threads and a batch size of 20000.

node in Section 4.4.2, we can see that even though the AMD chip has similar bandwidth, it is able to use the available memory bandwidth much more efficiently as a result of the overlapping design of the caches.

Fig. 4.15 shows the performance when keeping the number of threads constant at 64 (i.e. using the full node) and changing the batch sizes for variety of block sizes and ranks. While the AMD results differ from the Intel results in the fact that the batch size has a non-trivial effect on the utilization, our implementation still outperforms the AMD BLIS by a wide margin for every problem size. Fig. 4.16 shows the performance breakdown for each part of the execution for the AMD node.

Table 4.11 shows the performance of our code vs. BLIS as the problem gets more compute bound than memory bound. As shown for the Intel and Fujitsu nodes, the higher performance of AMD-BLIS can be attributed to the fact that larger sizes of small matrices leads to less optimal bandwidth utilization for our method.

Batch size	Block size	Rank	BLIS	OUR CODE
20000	512	96	197.056	237.441
20000	1024	96	373.555	283.175
20000	2048	96	559.075	348.566

Table 4.11: Performance of our code vs. BLIS for larger rank on the AMD node using 64 physical cores reported in GFLOPS. Our code can consistently beat AMD-BLIS until rank 96 is reached.

4.4.4 Evaluation of Block Low Rank matrix vector multiplication

Matrix-vector products are important computational kernels when working with the wave equation previously shown in Section 2.5. One of the fastest ways of obtaining the matrix vector product is with the use of the Fast Multipole Method (FMM) [149]. However, this will require expressing the matrix-vector product with the wave equation solution as an iterative solver, followed by multiple matrix-vector products for performing a gradient descent iteration with the vectors in the problem. The intermediate results of FMM are typically not stored, therefore FMM is able to run with a small amount of memory. However, this advantage comes with the trade-off of having to recompute all the FMM's algorithms for each different vector. This will increase the amount of computation that will be required in order to obtain the solution of multiple vectors using FMM matrix vector product.

The computation can be reduced when using the BLR matrix. Although the construction and factorization of a single BLR matrix is more expensive than generating the FMM, the BLR matrix has to be constructed and factorized only once, unlike the FMM which must be constructed from scratch with every matrix-vector product. This makes it possible to construct and factorize the BLR matrix once and use the same data structure for obtaining the solutions of hundreds of vectors. Specifically, we are looking at multiple x vectors in the equation of the form $b = A^{-1}x$, where A^{-1} is a factorized matrix. The b can then be obtained via a forward and backward solve. This can be computed in $O(N)$ time using a BLR matrix.

Fig. 4.17 shows the comparison of run time for multiplying multiple right hand sides using batched MKL vs. our implementation of batched low rank multiplication on the Intel node. A gain of about 15% can be observed. While the multiplication of just the low rank blocks and vectors shows a performance gain of about 50%, adding the intermediate vectors and multiplication of the dense blocks takes up more time, which reduces the performance gain to about 15%.

Since we show in the previous sections that our batching methodology is about twice as fast as vendor optimized libraries, we expect similar performance gains for other nodes too.

4.5 Conclusion and Future Work

In this thesis we have shown that performance of batched low rank multiplication can be improved with an alternative batching methodology based on improved data reuse and bandwidth utilization.

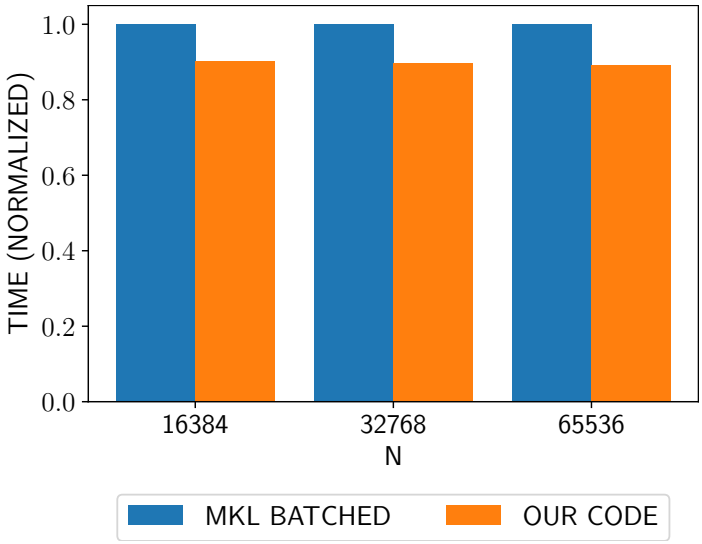


Figure 4.17: Multiplying multiple right hand sides with a weakly admissible block low rank matrix of various sizes using batched MKL vs. our code for the batched multiplication routine. The rank and number of right hand sides are kept constant at 8 for all tests.

Our results indicate better CPU utilization than vendor optimized libraries for a variety of thread counts and batch sizes on 3 major CPUs architectures, and for problem sizes critical to low rank operations. Specifically, we are able to achieve more than twice the performance of vendor optimized matrix multiplication routines when using the entire node for most problem cases. While most cases are memory bound, a larger rank of 32 actually results in a compute bound process. Thus our batching technique is able to keep the SIMD units busy enough that bandwidth is no longer the bottleneck, which is not true for vendor optimized libraries even for larger ranks. We run the same algorithm on all the CPUs, thus the cache misses generated by the data accesses would be proportional for the different libraries, thus the performance improvement comes from our optimization.

It is important to note that the constraints placed by the limited number of registers in SIMD architectures places limitations on the scalability of our method. However, hierarchical matrix factorization and vector multiplication typically involves low rank multiplication with block size up to 2048 and batch sizes not exceeding 20,000 – ranges where our approach is highly competitive with state of the art implementations, yielding significantly better results for this specific problem.

In the future we plan to use our technique for building fast factorization routines that run on distributed supercomputers. Distributed hierarchical factorization is challenging due to irregularity of communication patterns and computation. Cao et al. [44] report better load balance by using an alternate process distribution and prioritization of the critical path using the PaRSEC [33] runtime system. A more analytical approach [93, 128] leads to better understanding of the trade-off between replication and communication of data for determining the data distribution on multiple nodes, which can possibly lead to more efficient process distribution and replication methodologies for

minimization of communication overhead.

Chapter 5

HSS matrix factorization with PaRSEC

5.1 Introduction

Various scientific problems in fluid dynamics, structural mechanics, and electromagnetics are governed by partial differential equations (PDEs), which result in a sparse matrix when discretized with finite difference and finite element methods. The boundary element method (BEM) as shown in Section 2.1.3.4 has an advantage over such volume discretization methods, since it only discretizes the boundary and the number of elements can be drastically reduced. However, certain problems arising from BEM require the solution of a dense symmetric positive definite linear system, which has cubic complexity if solved directly. For very large dense matrices, the Cholesky factorization can be computed using distributed memory implementations from ScaLAPACK, DPLASMA [30], SLATE [68] or Elemental [118]. Although modern parallel computer architectures offer significant speedups due to the use of multiple threads, the cubic complexity of dense factorization remains prohibitive for large matrices.

Table 5.1 shows some of the state-of-the-art implementations of distributed dense direct factorization using various matrix formats and algorithms. Going from top to bottom, we can see libraries such as DPLASMA [30] and SLATE [68] (and SLATE's predecessor ScaLAPACK) making use of

Library	**Format**	**Algorithm**	**Compute**	**Paradigm**	**Comm.**
DPLASMA [30]	Dense	Tile Cholesky	$O(N^3)$	Asynchronous	$O(N^3)$
SLATE [68]	Dense	Panel Cholesky	$O(N^3)$	Fork-join	$O(N^3)$
LORAPO [42]	BLR	Tile Cholesky	$O(N^2)$	Asynchronous	$O(N^3)$
$\mathcal{H}$-LU [23]	$\mathcal{H}$-matrix	$\mathcal{H}$-LU	$O(Nlog(N))$	Asynchronous	$O(Nlog(N))$
STRUMPACK [124]	HSS	ULV	$O(N)$	Fork-join	$O(N^2)$
Ma et al. [108]	$\mathcal{H}^2$-matrix	Modified ULV	$O(N)$	Fork-join	$O(N)$
$\mathcal{H}$ATRIX-DTD	HSS	ULV	$O(N)$	Asynchronous	$O(N)$

Table 5.1: Comparison of dense direct factorization methods depending on the matrix format, factorization algorithm and distributed programming paradigm.

standard dense matrix formats and their associated algorithms that do not make use of low rank approximation. Although SLATE and DPLASMA use the same dense Cholesky factorization algorithm, DPLASMA makes use of asynchronous distributed execution whereas SLATE uses fork-join parallelism. The remaining libraries all make use of low rank representations of the dense matrix. More details about the low rank representation of dense matrices can be found in Section 2.4.

LORAPO [42] uses the BLR format shown in Section 2.4.1. The use of an asynchronous run time system such as PaRSEC allows LORAPO [42] to prioritize the execution of the critical path of the tile Cholesky factorization and resolve off-diagonal dependencies asynchronously. Large, adjacent low rank blocks of the BLR format can be combined to form the multi-level $\mathcal{H}$-matrix format. The tile LU (or Cholesky) factorization can then be extended to the $\mathcal{H}$-LU (or $\mathcal{H}$-Cholesky) algorithm which costs $O(Nlog(N))$. The use of an asynchronous runtime system with $\mathcal{H}$-LU has been shown to achieve good strong scaling for distributed computation since this allows for greater parallelism between the recursive blocks of the $\mathcal{H}$-matrix.

Approximation using the HSS matrix can be combined with the ULV factorization as shown in Section 2.6.2. Distributed HSS-ULV factorization has been implemented by STRUMPACK [124] using the fork-join programming paradigm as a result of relying on ScaLAPACK for the computation. STRUMPACK [124] distributes each block of the HSS matrix with a block cyclic distribution and relies on collective communication to shuffle data. Ma et al. [108] extend the ULV factorization to the $\mathcal{H}^2$-matrix, by modifying the ULV factorization to precompute the fill-ins before the factorization for the $\mathcal{H}^2$-matrix. Even though the $\mathcal{H}^2$-matrix has off-diagonal dense blocks, the method from Ma et al. [108] is able to achieve embarrassingly parallel factorization of each level by performing the factorization twice - once for precomputing the fill-ins and then for the actual factorization. Although this method is highly parallel, the fact that it factorizes twice results in a large overhead.

In this chapter, I propose $\mathcal{H}$ATRIX-DTD – an implementation of the HSS-ULV factorization with the PaRSEC runtime system. $\mathcal{H}$ATRIX-DTD makes use of the HSS-ULV algorithm that can factorize matrices arising from a variety of Green's functions with comparable accuracy to LORAPO and STRUMPACK. We choose these codes as a reference because they share certain traits with our code, besides the fact that they are the most popular libraries in this field. Similar to our code, LORAPO uses the PaRSEC runtime system, but the matrix structure is BLR. Conversely, STRUMPACK uses the HSS structure like $\mathcal{H}$ATRIX-DTD, but uses a bulk-synchronous model for parallelism instead of a runtime system. By comparing with these two references, we can isolate the effect of choosing HSS over BLR from the effect of using a runtime system over a bulk synchronous approach. The HSS-ULV factorization is computationally cheaper than the BLR-Cholesky factorization, and hence $\mathcal{H}$ATRIX-DTD can outperform LORAPO while making use of the same runtime system. The use of an asynchronous runtime system such as PaRSEC for handling the communication and dependencies between successive levels in the HSS-ULV leads to better overlap of communication and

computation, and hence $\mathcal{H}$ATRIX-DTD can outperform STRUMPACK that makes use of a similar HSS-ULV algorithm. As a result, I experimentally prove two key assumptions about the factorization algorithms and distributed memory implementation of low rank matrix formats in this thesis:

1. The use of the HSS matrix format and HSS-ULV algorithm has lower computational complexity than the BLR-tile Cholesky algorithm implemented by LORAPO [42].

2. The use of an asynchronous runtime system such as PaRSEC leads to a lower overhead of communication than the fork-join parallelism implemented by STRUMPACK [124].

I experimentally show that $\mathcal{H}$ATRIX-DTD can outperform both LORAPO and STRUMPACK [124] over a large number of nodes. $\mathcal{H}$ATRIX-DTD shows weak scaling efficiency and achieves upto 2x faster time of factorization on up to 128 nodes for a variety of Green's functions. We first introduce our notation and construction of HSS matrices, and then elaborate on the HSS-ULV algorithm. We then describe our implementation of the HSS-ULV using the PaRSEC runtime system. We then demonstrate with rigorous experimental evidence the performance of $\mathcal{H}$ATRIX-DTD against STRUMPACK [124] and LORAPO.

5.2 Distributed memory execution

In this section, we elaborate on the distributed memory implementation of $\mathcal{H}$ATRIX-DTD, and outline the differences from STRUMPACK and LORAPO. This section draws on the description of runtime systems such as PaRSEC done in Section 3.1.4. Section 5.2.6 shows the process distribution strategy followed by $\mathcal{H}$ATRIX-DTD. Finally, Section 5.2.6 shows how we map the HSS-ULV algorithm to PaRSEC.

5.2.1 Distributed Cholesky using LORAPO

LORAPO makes use of the Parameterized Task Graph (PTG) interface from PaRSEC. The use of PTG is helpful for large scale computation involving many hundreds of nodes since PTG helps reduce the runtime overhead. This because PTG uses a DSL for statically specifying the dependencies at the time of compilation. Therefore, only tasks that actually need to be executed on a node are constructed by it. Unlike DTD, this avoids the generation of the entire task graph on every node. LORAPO also prioritizes the critical path of execution with the of fake tasks that force PaRSEC to execute computation on the diagonal block before anything else. Since LORAPO is an adaptive rank code, it must communicate the extra rank parameters necessary for sending low rank matrices before any block can be communicated. Although LORAPO also provides functionality for using lower precision floating point values for representing the low rank blocks, these features are not used in our evaluation.

5.2.2 Process distribution of LORAPO

LORAPO uses a modified block cyclic distribution [43, Fig. 3]. This distribution helps reduce the communication during the critical path and achieve better load balance.

5.2.3 Distributed HSS-ULV using STRUMPACK

Fig. 5.1 shows the execution of the HSS-ULV algorithm using STRUMPACK for two leaf nodes of the HSS matrix from Fig. 2.15. The colors of the blocks of the HSS matrix in this case correspond to that of the algorithmic description of the HSS-ULV given in Section . Each block in Fig. 5.1 represents a computation of the HSS-ULV. The blocks of the HSS matrix on which the operation takes place and the processes on which the operations are executed are both listed on each task. The process distribution of the individual blocks of the HSS matrix corresponds exactly to that shown in Fig. 5.2.

Assuming that the leaf nodes of the HSS matrix are distributed on individual processors, the leaf level computation can happen completely in parallel. However, there is then the barrier synchronization which all processes must get through before being able to move to the next step. This means that the merge step (corresponding to the permutation shown in Fig. 2.18 for a single level) cannot proceed unless all leaf blocks have finished processing. Moreover, the merged blocks such as $A_{1;0,0}$ are block cyclically distributed on multiple processes. Therefore performing any operation on these blocks will involve further communication.

As this tree gets wider and deeper and the number of processes increases, the overhead of MPI synchronization and the use of ScaLAPACK for the computation causes the communication and synchronization time to dominate the total time of the algorithm. This leads to problems in the weak scaling of STRUMPACK, and will be elaborated in Section 5.3.3.2.

5.2.4 Process distribution of STRUMPACK

STRUMPACK treats the HSS matrix like a recursive tree that must be traversed in order to perform the ULV factorization. Fig. 5.2 shows the process distribution of the tree representation of the HSS matrix shown in Fig. 2.15. The tree at the top of Fig. 5.2 indicates that the process assignment begins at the leaf level with discrete processes being assigned to each leaf node. The parent nodes of these nodes are distributed across the processes that own the leaves. Nodes that exist on a single process are shown in solid colors and nodes that are distributed across multiple processes are denoted by a gradient. The processes on which the node exists is written next to the respective node.

This tree can be mapped to the blocks of the 2 level HSS matrix at the bottom of Fig. 5.2. The leaf blocks correspond to the leftmost matrix, shown with $U_{2;i}$ and $A_{2;i,j}$. The diagonal blocks $A_{2;i,i}$ and the basis matrices $U_{2;i}$ correspond to the same node $(2;i)$ of the tree, and are therefore mapped to a single process. However, the $S_{2;1,0}$ block on level 2 of the HSS matrix shows a gradient,

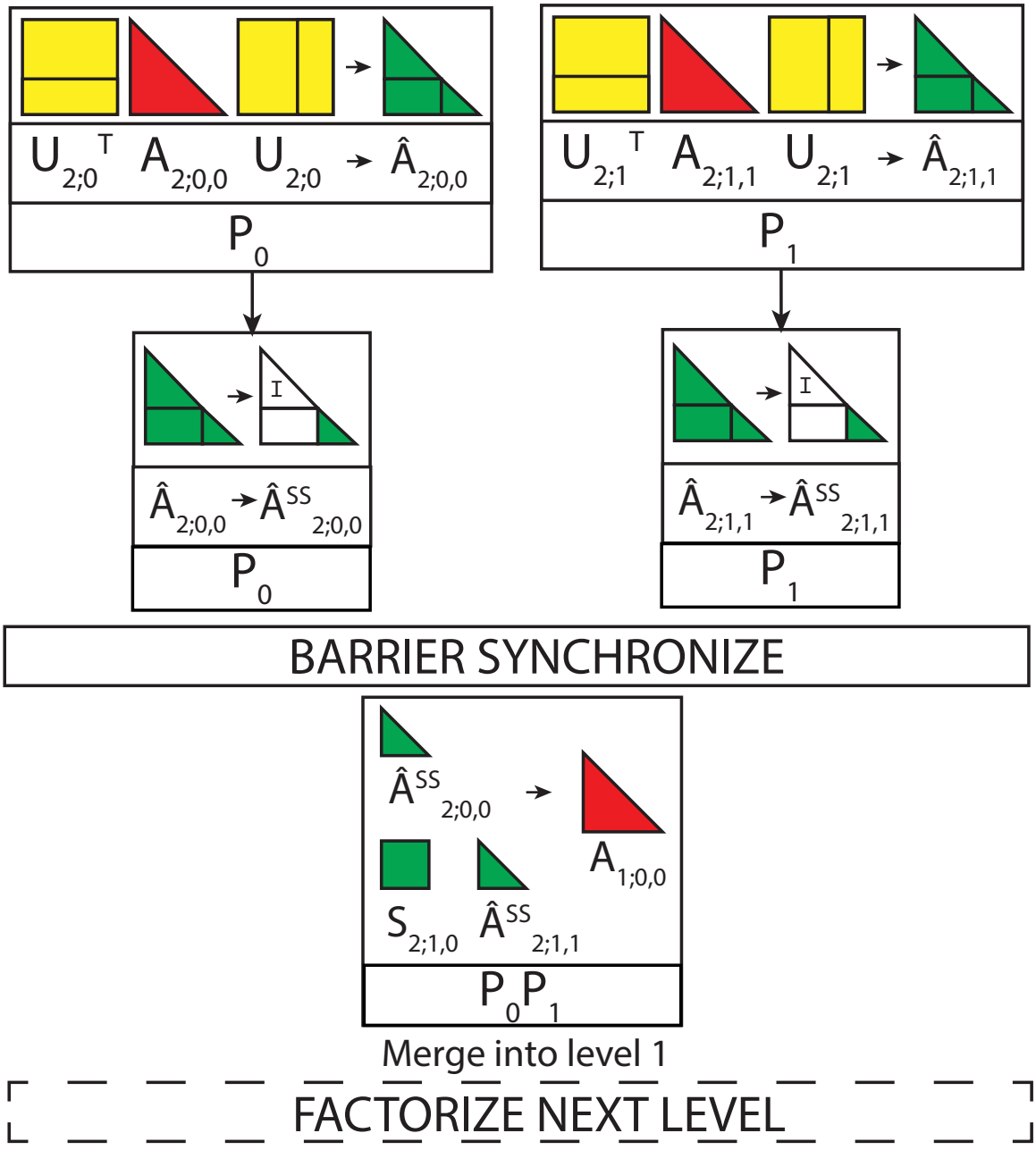


Figure 5.1: Execution of HSS-ULV within STRUMPACK. The factorize and merge steps of the first two leaf nodes of the 2 level HSS matrix in Fig. 2.19 are shown. The process numbers where the computation is executed are indicated at the bottom of each box. Multiple processes indicate that ScaLAPACK routines are used with a block cyclic distribution.

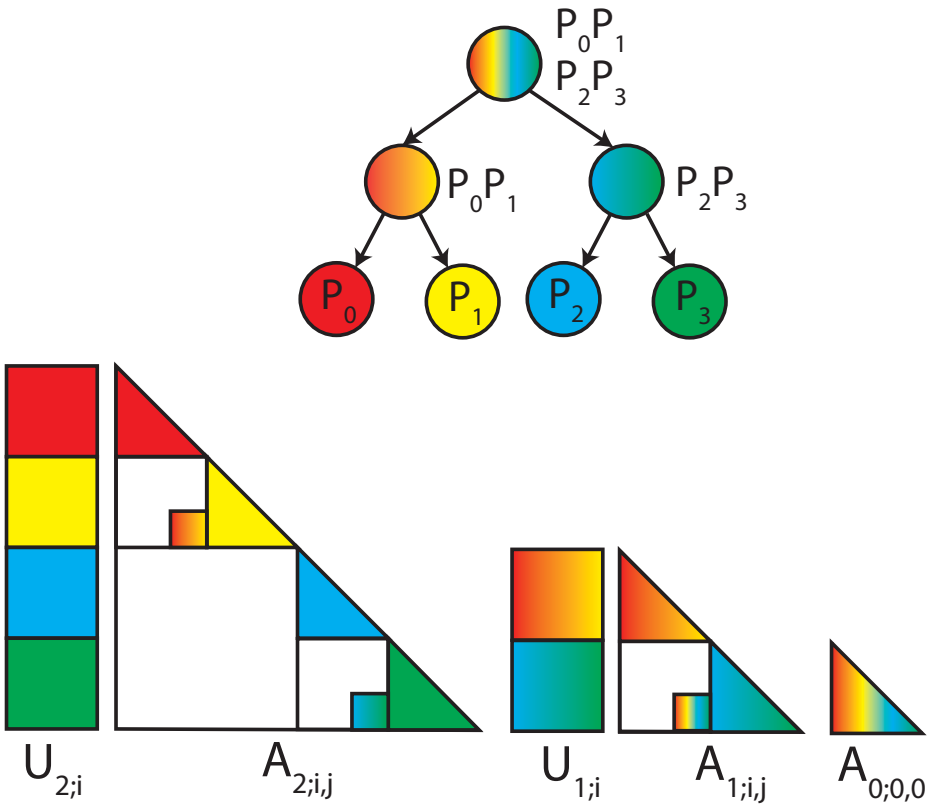


Figure 5.2: STRUMPACK's process distribution for a 2 level HSS matrix. Distinct color show discrete processes and gradients indicate that the block is shared among processes.

indicating that it is mapped to both, process 0 and 1. This is because it is present on the row of node 0 and column of node 1, and therefore is mapped to both the processes.

The matrices indicated by $U_{1;i}$ and $A_{1;i,j}$ denote the HSS matrix at level 1. These blocks are divided among multiple processors, as indicated by the gradient. Similar to the $S_{2;1,0}$ block on level 2, the $S_{1;1,0}$ block is divided among 4 processes, as indicated by the 4 color gradient. Finally, the root node is divided among 4 processes. The blocks that show a gradient are sub-divided using the block cyclic process decomposition of ScaLAPACK. This allows STRUMPACK to leverage the distributed dense linear algebra routines of ScaLAPACK and use ScaLAPACK communication routines for performing the merge step of the HSS-ULV factorization.

5.2.5 Distributed HSS-ULV using the PaRSEC runtime system.

Fig. 5.4 shows the mapping of tasks in PaRSEC for the HSS-ULV algorithm shown in Alg. 6. We denote the steps shown in Fig. 2.19 as they are expressed within the tasks of PaRSEC. The HSS-ULV algorithm operates on the dense, skeleton and bases block matrices of the HSS matrix. These blocks form the dependencies within the tasks. The "Diagonal Product" step on Line 2 of Alg. 6 results in zeroing of the off-diagonal low rank blocks, which makes the partial factorization of each dense

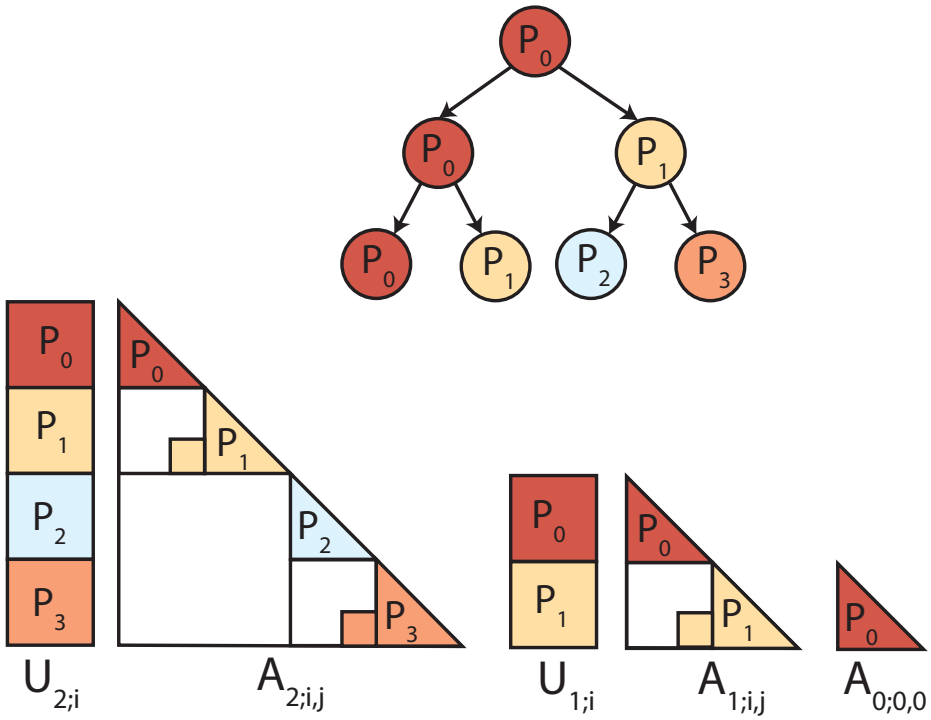


Figure 5.3: Process distribution used by $\mathcal{H}$ATRIX-DTD for a 2 level HSS matrix. Each distinct color represents a separate process. The dense, skeleton and basis blocks are distributed in a row cyclic process distribution at every level.

block independent of all the other blocks on the same level. This means that the dependencies in the HSS-ULV only come from the merge step on Line 4 of Alg. 6.

PaRSEC is able to exploit the inherently parallel factorization of each level. The "Diagonal Product" and "Partial Factorization" steps of each dense block on the same level can be executed in an embarrassingly parallel manner. The dependency between the levels exists as a result of the "Merge" step. As a result of asynchronous execution, the "Merge" step can begin right after the corresponding partial factorization for its dependencies have finished.

This asynchronous approach is in contrast to STRUMPACK, where each level executes after the entire previous level has finished factorization. Although the use of ScaLAPACK allows STRUMPACK to perform each level of the HSS-ULV in an embarrassingly parallel manner (assuming different compute resources), the use of fork-join parallelism for the "Merge" step means that the parent level cannot begin execution unless the child level has completely finished.

5.2.6 Process distribution of $\mathcal{H}$ATRIX-DTD.

Fig. 5.3 shows the process distribution strategy of $\mathcal{H}$ATRIX-DTD for the HSS matrix from Fig. 2.15. Unlike libraries such as ScaLAPACK and Elemental [118] which make use of block-cyclic and element-cyclic process distribution, respectively, a row-cyclic process distribution is a better fit for

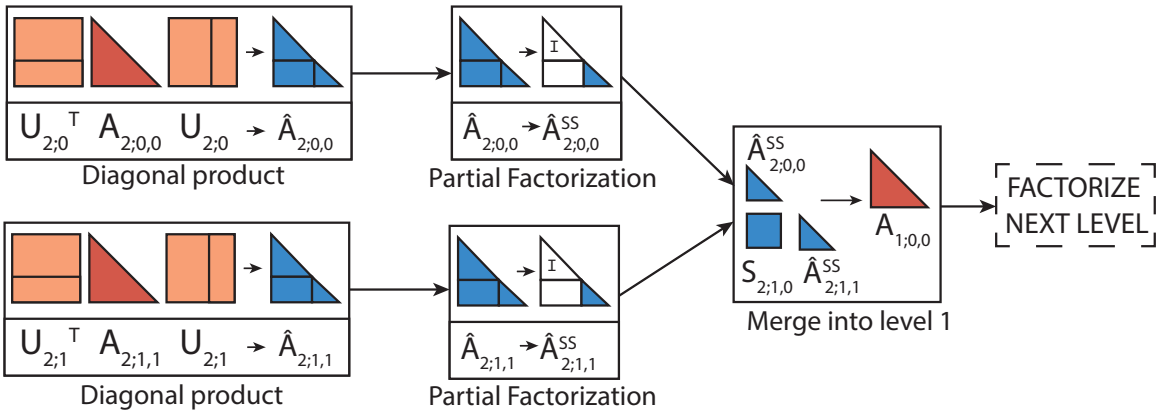


Figure 5.4: Mapping of the HSS-ULV algorithm to tasks within PaRSEC for a 2 level HSS matrix. Each block in this diagram represents some computation and its associated dependencies. These tasks represent the factorization of only the first two nodes of the leaf and its subsequent merging into the upper level. Similar steps are followed for other nodes and levels.

HSS-ULV with PaRSEC. Each block of the HSS matrix that is involved in the HSS-ULV is assigned to a single task. This keeps the number of tasks smaller, which means that the runtime system overhead is better controlled. The blocks owned by P_0 and P_1 from level 2 are merged into P_0 in level 1 as a result of the "Merge" step in Alg. 6. Merging blocks into a single process is necessary because of the need to balance the number of tasks with the time consumed by each task. Too many tasks of very little duration will be generated if we proceed with a block-cyclic distribution for the merged block $A_{1;0,0}$.

In contrast to the row cyclic distribution used in $\mathcal{H}$ATRIX-DTD, STRUMPACK [124] makes use of a block cyclic distribution for every matrix block of the HSS matrix. This is necessary since STRUMPACK relies on ScaLAPACK for computation on the matrix blocks. Such a process distribution would not be effective in $\mathcal{H}$ATRIX-DTD because it would generate too much communication between tasks on the same row (in the block cyclic distribution tasks on the same row will be placed according to the process grid on different processes).

LORAPO [43] implements a tile Cholesky algorithm on a block low rank matrix. This means that resolving the off-diagonal triangular solve and trailing sub-matrix updates is the primary bottleneck in the execution of the critical path. A modified block cyclic distribution where each tile is block-cyclically distributed on a process grid is experimentally found to be the best process distribution strategy for LORAPO. Even though $\mathcal{H}$ATRIX-DTD makes use of PaRSEC, such a data distribution in not necessary since the HSS-ULV algorithm ensures that the critical path along the diagonal can be executed in an embarrassingly parallel manner.

Kernel	Equation	Constants
Laplace 2D	$f(x,y) = -\ln(\epsilon + dist(x,y))$	$\epsilon = 10^{-9}$
Yukawa	$f(x,y) = \frac{e^{\alpha \times -(\theta + dist(x,y))}}{(\theta + dist(x,y))}$	$\alpha = 1$, $\theta = 10^{-9}$
Matern	$f(x,y) = \begin{cases} \frac{\sigma^2}{2^{\rho-1} \times \Gamma(\rho)} \times \frac{dist(x,y)}{\mu}^{\sigma} \times K\nu(\sigma, \frac{dist(x,y)}{\mu}), & \text{otherwise} \\ \sigma^2, & \text{if } dist(x,y) = 0 \end{cases}$	$\sigma = 1$, $\mu = 0.03$, $\rho = 0.5$

Table 5.2: Kernels used for evaluation and their constants.

5.3 Results

We run distributed memory tests for 3 implementations of low rank matrix factorization - $\mathcal{H}$ATRIX-DTD, STRUMPACK and LORAPO. Every implementation uses a uniform 2D grid geometry. Distributed memory tests are run on the Fugaku supercomputer at RIKEN, Japan. Each node of Fugaku has a single A64FX CPU with 48 physicals cores divided into 4 NUMA nodes of 12 cores each. Each node has 32 GB of HBM.

We implemented $\mathcal{H}$ATRIX-DTD using the DTD programming interface from PaRSEC. We make comparisons with STRUMPACK [124] and LORAPO [43] as described in Section 5.1. We compare three Green's functions from diverse applications such as electromagnetism and statistics as shown in Table 5.2 for each implementation.

5.3.1 Effect of rank on accuracy

Table 5.3 shows the impact of changing the maximum rank and leaf size on the construction and solve error for each tested kernel. The errors are calculated by first generating a normally distributed random vector b and computing the construction error as shown in Eq. (5.1), and the solve error as shown in Eq. (5.2). A_{dense} denotes the full dense matrix, and A denotes the corresponding compressed HSS matrix. The maximum rank of the HSS matrix and the size of the leaf level nodes is capped for all three libraries surveyed. We use the leaf size and maximum rank measurements of solve error from Table 5.3 to determine parameters for the weak scaling experiments in Sec. 5.3.2 in order to obtain sufficient accuracy for all kernels we benchmark.

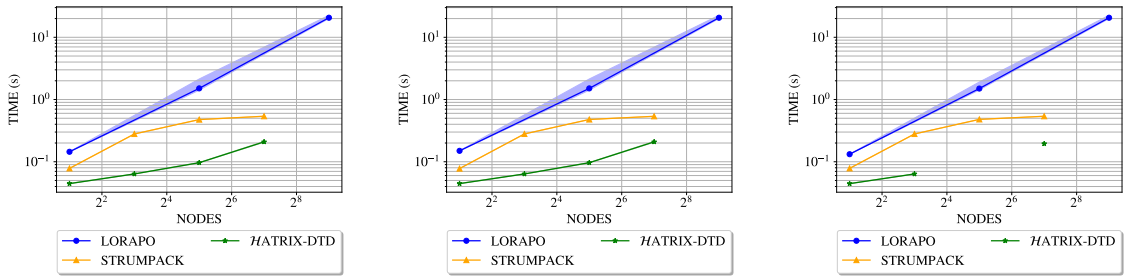
$$err_{construct} = \frac{\|A_{dense} \cdot b - A \cdot b\|}{\|A_{dense} \cdot b\|} \tag{5.1}$$

$$err_{solve} = \frac{\left\|b - A^{-1} \cdot A \cdot b\right\|}{\|b\|} \tag{5.2}$$

The construction error for all cases of $\mathcal{H}$ATRIX-DTD decreases as the rank increases, which is to be expected given that a greater rank means a greater number of basis that can be incorporated in the low rank approximation. However, the solve error seems to increase slightly as the rank increases. This slight increase can be attributed to numerical errors. Since LORAPO uses adaptive ranks, specifying the maximum rank leads to choosing ranks that are enough to satisfy the construction

	Construct Max Rank	Leaf Size	Laplace Const. Err.	Laplace Solve Err.	Yukawa Const. Err.	Yukawa Solve Err.	Matern Const. Err.	Matern Solve Err.
HATRIX	100	256	1.54e-06	4.78e-12	2.73e-08	3.04e-15	9.95e-05	3.90e-13
	200	256	2.89e-07	5.48e-12	7.63e-09	3.50e-15	2.34e-05	4.51e-12
	200	512	1.82e-07	6.06e-12	5.85e-09	3.83e-15	1.6e-05	4.89e-12
	400	512	5.51e-10	7.00e-12	5.07e-10	4.42e-15	1.0e-06	5.85e-12
LORAPO	1024	2048	1e-8	2.21e-13	1e-8	1.33e-13	1e-8	1.01e-09
	1500	2048	1e-8	2.21e-13	1e-8	1.33e-13	1e-8	1.01e-09
	1250	4096	1e-8	2.21e-13	1e-8	1.33e-13	1e-8	7.84e-10
	3000	4096	1e-8	2.21e-13	1e-8	1.33e-13	1e-8	7.84e-10
STRUMPACK	100	256	1e-8	5.76e-14	1e-8	2.13e-15	1e-8	1.50e-12
	200	256	1e-8	9.05e-11	1e-8	1.48e-14	1e-8	2.35e-09
	200	512	1e-8	3.37e-11	1e-8	1.10e-14	1e-8	4.44e-10
	400	512	1e-8	1.71e-11	1e-8	4.04e-14	1e-8	9.71e-09

Table 5.3: Impact of rank and kernel for the methods we tested for a constant problem size of 65536.



(a) Laplace 2D kernel (b) Yukawa kernel (c) Matern kernel

Figure 5.5: Weak scaling of factorization time for all the kernels shown in the Table 5.2 for varying problem sizes.

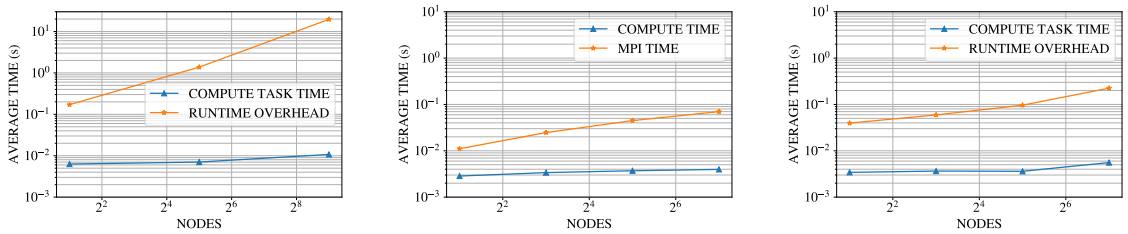
error of 10^{-8}. As a result, increasing the maximum rank from 1024 to 1500 for a leaf size of 2048 does not lead to changing solve error for any kernel. STRUMPACK allows for a similar tuning of ranks as $\mathcal{H}$ATRIX-DTD since both use the HSS matrix. The HSS-ULV algorithm used within STRUMPACK shows better solve error than $\mathcal{H}$ATRIX-DTD for a maximum rank of 100 and leaf size 256 for the laplace 2D kernel. However, the solve error decreases when going from rank 100 to 200 with leaf size 256. The reasons behind this drop in accuracy are unknown.

5.3.2 Distributed memory weak scaling

Fig. 5.5 shows weak scaling for factorization using STRUMPACK, LORAPO and $\mathcal{H}$ATRIX-DTD on upto 128 nodes of Fugaku. For $\mathcal{H}$ATRIX-DTD and STRUMPACK, the size of the matrix begins at 4096 for 2 nodes and then increases linearly with the number of nodes, until it reaches 262,144 with 128 nodes. The linear increase in problem size and number of processors is done in order to maintain constant work per process, given the $O(N)$ time complexity of the HSS-ULV. The tile Cholesky with the BLR matrix used by LORAPO as shown in Table 5.1 shows $O(N^2)$ time complexity. Therefore, we start from a problem size of 4096 with 2 nodes and increase the number of nodes by a factor of 16 for every experiment to maintain constant work per node. This means that the problem size reaches 65,536 for 512 nodes. We run each experiment in the plots 100 times. The results are plotted with a 90% confidence interval. The intervals for the STRUMPACK and $\mathcal{H}$ATRIX-DTD are barely visible since there is not much variation in the results across the runs. The LORAPO results show more variation, as can be seen with the shaded region.

The rank and leaf size are chosen from Sec. 5.3.1 in order to maintain accuracy that is better than 10^{-11} for the laplace 2D kernel, 10^{-14} for the yukawa kernel, and 10^{-9} for the matern kernel. We then experiment with combinations of rank and accuracy that provide an acceptable solve error for each problem size and kernel, and show the least time to solution in Fig. 5.5.

The results in Fig. 5.5 show that $\mathcal{H}$ATRIX-DTD exhibits better weak scalability than both STRUMPACK and LORAPO. LORAPO and $\mathcal{H}$ATRIX-DTD both make use of the PaRSEC runtime



(a) LORAPO (b) STRUMPACK (c) $\mathcal{H}$ATRIX-DTD

Figure 5.6: Performance breakdown for the 3 implementations in Fig. 5.5(b)

system, however the tile Cholesky algorithm of LORAPO involves almost $O(N^3)$ communication for the update of the trailing sub-matrix. This, coupled with the fact that the tile Cholesky is constrained by the execution of the critical path of the diagonal add to the poor weak scaling of LORAPO. Further analysis of LORAPO's weak scaling is done in Sec. 5.3.3.1. $\mathcal{H}$ATRIX-DTD and STRUMPACK both use the HSS-ULV algorithm, however $\mathcal{H}$ATRIX-DTD is faster than STRUMPACK. This is as a result of the asynchronous execution of PaRSEC, which allows $\mathcal{H}$ATRIX-DTD to begin the factorization of the parent level before the entire child level has been factorized. STRUMPACK, on the other hand, makes use of fork-join parallelism with collective communication, which requires that each level of the HSS matrix be factorized fully before the next level can begin.

5.3.3 Performance breakdown of weak scaling

In this section, we further analyse the reasons behind the weak scaling performance seen in Sec. 5.3.2. Since all the kernels show similar performance characteristics, we investigate only the Yukawa kernel in further detail.

5.3.3.1 Performance breakdown of LORAPO

Fig. 5.6(a) shows the performance breakdown for LORAPO [43] for the weak scaling graph of the Yukawa kernel shown in Fig. 5.5(b). We obtain these measurements from the PaRSEC instrumentation tools that allow for measuring the amount of time that corresponds to time spent inside the actual computational kernels and that for various runtime system management activities.

As pointed out in Section 5.1, LORAPO uses the tile Cholesky algorithm with the BLR matrix format. The "COMPUTE TASK TIME" corresponds to the average time per worker spent inside the actual computational kernels for the Cholesky factorization. The "RUNTIME OVERHEAD" corresponds to the average time per worker spent on runtime system management activities such as scheduling, memory management, submitting and executing tasks and deleting previously executed tasks. This also includes various MPI activities such as sending, receiving and polling for messages. The number of workers is the number of physical cores being used for the computation across all the nodes.

It can be seen that overhead of the runtime system far outweighs the amount of time taken for the computation. Moreoever, the growth of the overhead is proportional to the time taken for factorization in Fig. 5.5(b), whereas the growth of "COMPUTE TASK TIME" is not. This means that the poor weak scaling of LORAPO can be attributed mainly to the runtime overhead. LORAPO would have had better weak scaling if the run time overhead would grow in proportion to the problem size and number of available resources.

5.3.3.2 Performance breakdown of STRUMPACK

Fig. 5.6(b) shows the breakdown of time that STRUMPACK spends on actual computation vs. MPI for the STRUMPACK weak scaling plot in Fig. 5.5(b).

The performance statistics in Fig. 5.6(b) are obtained from the mpiP tool from LLNL (https://github.com/LLNL/mpiP). The time measurements are averaged by the total number of physical cores used by each experiment. The "MPI TIME" shows the time spent by STRUMPACK inside MPI functions such as collective communication. The "COMPUTE TIME" shows the time spent on useful computation. The "COMPUTE TIME" remains almost the same for every measurement. However, note that the time spent in MPI by each process increases as the number of nodes increases. This means that the MPI communication overhead using the fork-join paradigm leads to inefficient execution in STRUMPACK. Note the "COMPUTE TIME" graph does not show a flat profile in spite of the the HSS-ULV being embarrassingly parallel at each level of the HSS matrix. Apart from the increasing MPI time of the communication, the increasing per process compute time also contributes to the worsening of weak scaling of STRUMPACK. We show in Sec. 5.3.3.3 that our implementation in $\mathcal{H}$ATRIX-DTD can overcome these limitations.

5.3.3.3 Performance breakdown of $\mathcal{H}$ATRIX-DTD

Fig. 5.6(c) shows the performance breakdown of $\mathcal{H}$ATRIX-DTD for Fig. 5.5(b). The measurements are taken in a similar manner to LORAPO in Sec. 5.3.3.1, i.e. with use of the PaRSEC instrumentation tools. The "COMPUTE TASK TIME" and "RUNTIME OVERHEAD" have exactly the same meaning as that of LORAPO in Sec. 5.3.3.1 since both $\mathcal{H}$ATRIX-DTD and LORAPO make use of the PaRSEC runtime system.

The "COMPUTE TASK TIME" for $\mathcal{H}$ATRIX-DTD is almost completely flat. This means that almost the same amount of work is being done by each worker when the problem size is increased in proportion to the number of available resources. The final data point shows slightly higher compute time as a result of using a leaf size of 512. This means that the HSS-ULV as implemented in $\mathcal{H}$ATRIX-DTD will show perfect weak scaling in the absence of runtime overhead. The slightly poor weak scaling of $\mathcal{H}$ATRIX-DTD in Fig. 5.5 can be attributed to the runtime overhead that shows upward growth as the number of resources is increased. The runtime overhead can be explained by the fact that PaRSEC's DTD interface generates the entire task graph on every node. This leads

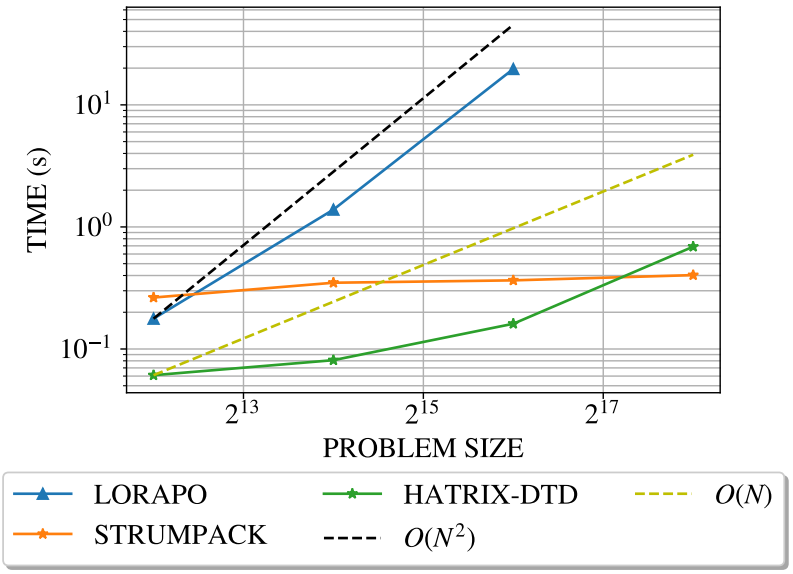


Figure 5.7: Varying problem sizes with 64 nodes on Fugaku.

to a lot of redundant work on each node, which becomes non-trivial as the number of available resources increases.

Note that the compute time per worker for $\mathcal{H}$ATRIX-DTD and compute time per thread for STRUMPACK in Sec. 5.3.3.2 are very similar. The runtime overhead of $\mathcal{H}$ATRIX-DTD appears higher than that of STRUMPACK since the MPI barrier and synchronization time is not accounted for in the STRUMPACK results. However, the $\mathcal{H}$ATRIX-DTD overhead shows the entire overhead including synchronization, communication, scheduling and other essential work by the runtime system.

5.3.4 Increasing problem size with constant resources.

Fig. 5.7 shows the time taken for factorization for varying problem sizes uses 64 nodes of Fugaku. STRUMPACK shows almost uniform time. Since we are using a large number of processes and the computation per process is not very large, the communication time dominates the computation for all cases, and what little computation needs to be done by each process is done in a short time. LORAPO shows $O(N^2)$ scaling upto a problem size of 65,536. STRUMPACK has an advantage over $\mathcal{H}$ATRIX-DTD in this case. Since the runtime overhead in $\mathcal{H}$ATRIX-DTD increases as the number of tasks increases, the performance of $\mathcal{H}$ATRIX-DTD increases as $O(N)$ even though the amount of computation is small.

5.3.5 Impact of leaf size on performance.

Fig. 5.8 shows the impact on the time for factorization for $\mathcal{H}$ATRIX-DTD, STRUMPACK and LORAPO when using a problem size of 262,144 and 128 nodes of Fugaku. The rank is kept constant at

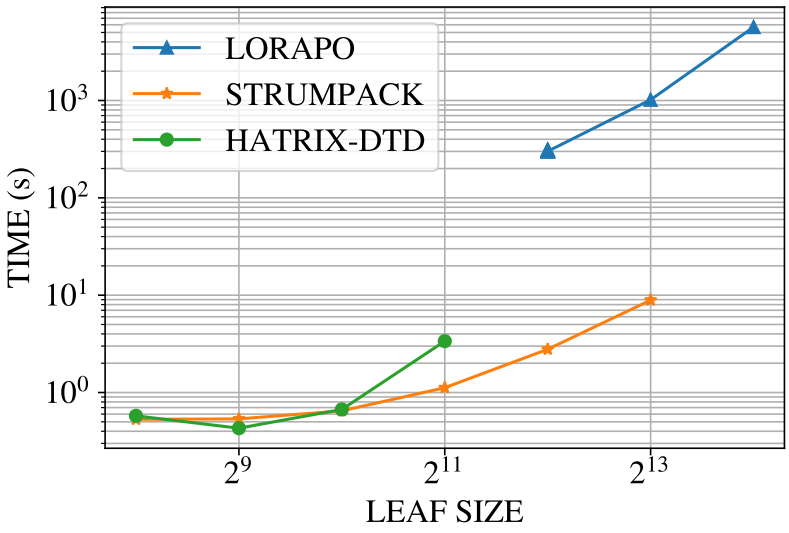


Figure 5.8: Performance impact of leaf size using 128 nodes and constant problem size of 262,144 for the Yukawa kernel.

100 for $\mathcal{H}$ATRIX-DTD and STRUMPACK and the maximum rank is half the leaf size for LORAPO. The optimal leaf size for LORAPO changes depending on the problem size. The use of low rank approximation for compressing the dense frontal matrices in the multi-frontal method is an important application of such matrices. The selection of leaf size of the HSS matrix, which correlates to the front size in the multi-frontal solver, is a crucial parameter in justifying the cost of the algorithm. Large leaf sizes can lead to very poor performance of the multi-frontal solver. The fact that $\mathcal{H}$ATRIX-DTD is faster than STRUMPACK when using small leaf sizes shows that $\mathcal{H}$ATRIX-DTD can also be used in place of STRUMPACK to factorize the dense, st5ructured fronts in multi-frontal solvers. Larger leaf sizes for $\mathcal{H}$ATRIX-DTD lead to worse performance due to reduction in the amount of available parallelism and more work to do per thread.

5.3.6 Impact of runtime system schedulers

Fig. 5.9 shows the impact of the scheduler on the weak scaling.

5.4 Conclusion

We have proposed an ULV factorization for HSS matrices, and provided an implementation, $\mathcal{H}$ATRIX-DTD, using the PaRSEC runtime system. We have showed that factorization of structured dense matrices arising from a diverse set of Green's functions for a 2D domain can be performed faster and with better weak scaling efficiency using our implementation, thanks to the use of an asynchronous runtime system and the lower computational intensity of the HSS-ULV factorization. Using $\mathcal{H}$ATRIX-DTD, we show that our algorithm has comparable or better accuracy than established

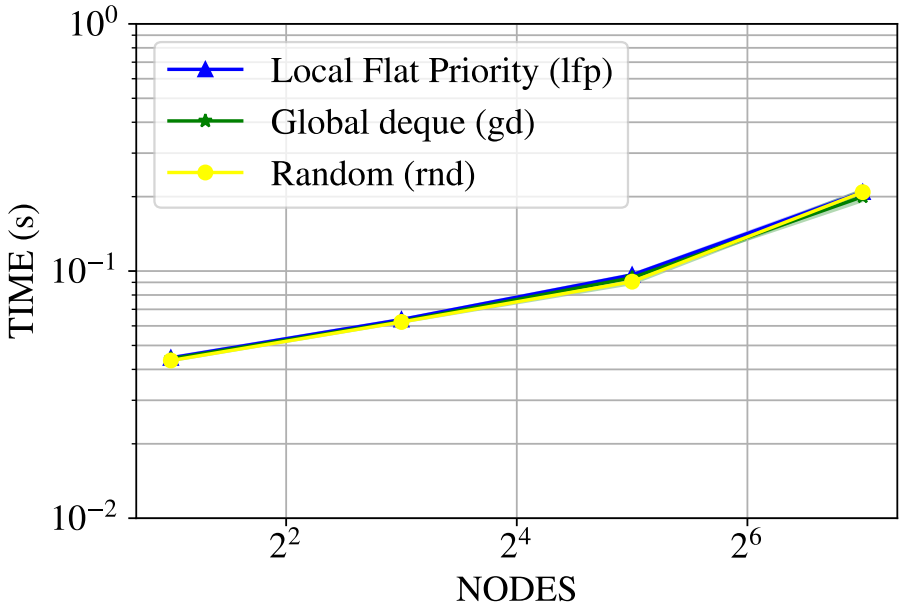


Figure 5.9: Weak scaling of $\mathcal{H}$ATRIX-DTD when using various schedulers available from PaRSEC.

state-of-the-art implementations such as STRUMPACK and LORAPO. Using performance analysis of weak scaling experiments we highlight that our implementation is indeed faster because of a combination of lesser computation and asynchronous resolution of dependencies of the multiple levels of the HSS matrix. Although the runtime overhead of PaRSEC remains a serious concern, we are able to show that $\mathcal{H}$ATRIX-DTD outperforms STRUMPACK and LORAPO for all problem sizes and choice of number of nodes. The runtime overhead of $\mathcal{H}$ATRIX-DTD can be further reduced with the help of the Parameterized Task Graph (PTG) interface from PaRSEC, which uses statically generated task graphs for reducing the number of tasks that are generated on each node. This is left as a future work for this study.

Chapter 6

Conclusion

This thesis proposes cache blocking and asynchronous parallel execution for more efficient execution of the matrix-vector product and direct factorization of low rank matrices, respectively. This thesis builds upon the work done by the dense linear algebra community and extend their innovations to the fast algorithms made possible by low rank approximation. It is shown that our methods work for a diverse set of applications and report higher performance than state-of-the-art implementations such as MKL, SSL and AMD-BLIS for the matrix-vector product, and LORAPO and STRUMPACK for the direct factorization.

The key findings of this thesis are summarized below, along with a note on possible future directions.

6.1 Summary of the main results

6.1.1 Shared memory matrix vector product

1. **Faster low rank matrix multiplication** - SIMD register accumulation and cache blocking could be used for optimizing the low rank matrix multiplication, a key kernel in the matrix vector product of the block low rank matrix. This leads to a 2x performance improvement of the low rank multiplication on the Fujitsu A64FX, Intel Xeon 6148 and AMD EPYC 7502 CPUs as shown in Section 4.4.

2. **Effective use of ECM performance modeling** - The use of the ECM performance model for optimizing the assembly micro kernels proved to be an effective strategy to point out bottlenecks in our implementation as shown in sections 4.3.2–4.3.4.

3. **Faster matrix-vector product** - The 2x performance improvement of the low rank matrix multiplication led to a 15% performance improvement in the matrix-vector product of the block low rank matrix as shown in Section 2.5.

6.1.2 Distributed memory dense direct factorization

1. **Asynchronous parallelism is faster than bulk synchronous parallelism for HSS-ULV factorization** - Comparison between the weak scaling performance of STRUMPACK, a state-of-the-art distributed HSS-ULV factorization using ScaLAPACK for the computation and our implementation $\mathcal{H}$ATRIX-DTD using the PaRSEC asynchronous runtime system shows better performance for weak scaling results as shown in Section 5.3.2. Both implementations show similar time spent in computation so we conclude that the speedup is as a result of the asynchronous parallelism.

2. **HSS-ULV is faster than block Cholesky in spite of lesser available parallelism** - LORAPO makes use of the PaRSEC runtime system for the implementation of a block Cholesky factorization on a block low rank matrix. However, in spite of having almost $O(N^3)$ parallelism in the trailing sub-matrix updates, the HSS-ULV using PaRSEC thin $\mathcal{H}$ATRIX-DTD outperforms LORAPO as shown in Section 5.3.3. HSS-ULV is an algorithm with $O(N)$ time complexity with $O(N)$ parallelism.

3. **Runtime overhead is a bottleneck for scalability of $\mathcal{H}$ATRIX-DTD** - Section 5.3.4 shows that although $\mathcal{H}$ATRIX-DTD is faster than STRUMPACK for smaller problem sizes for a constant number of nodes, STRUMPACK is eventually faster for larger problem sizes due to the runtime overhead of PaRSEC. We recognize this as a serious bottleneck and address it in the future work in Section 6.4.

6.2 Implications of this work

The low rank matrix multiplication shown in Chapter 4 is a key kernel of the block low rank matrix multiplication algorithm shown in Section 2.5. The matrix-vector multiplication with multiple right hand sides is a useful algorithm for the wave function. Use of our technique can speed up calculations making use of the wave equation. Use of our batched multiplication technique can speed up other methods such as the forward and backward substitution too. Moreover, low rank matrices are found in other approximation techniques such as tensor approximation. Therefore, such a SIMD accumulation technique can be used beyond hierarchical matrices. The portable approach of designing BLAS kernels using loops in high level code that was pioneered by the BLIS framework [139] is useful beyond basic BLAS functions. Section 4.2.1 proves that high performance can be obtained without sacrificing portability.

The dense direct factorization of HSS matrices shown in Chapter 5 is step forward for the faster factorization of matrices arising from 2D problems. Apart from its application for factorization of structured dense matrices from the boundary element method, HSS matrices are also useful for

factorization of the dense fronts from the multi-frontal method. Section 5.3.5 shows that $\mathcal{H}$ATRIX-DTD is competitive with STRUMPACK for small leaf sizes, which means that it can be used to replace STRUMPACK in multi-frontal methods too. All of this can be done without loss of accuracy as shown in Section 5.3.1.

6.3 Limitations of this work

The low rank matrix multiplication from Chapter 4 is an optimization of the matrix multiplication specifically targeted at the low rank multiplication algorithm. While Section 4.4 shows that our method is about 2x faster than vendor BLAS, the impact on the block low rank matrix multiplication shown in Section 2.5 is about 15% as a result of the overhead of addition in the multiplication process. The vector addition can be possibly optimized further. As the rank and block size keep increasing, our method loses its competitiveness and vendor BLAS can perform better. A glimpse into this can be seen in Fig. 4.5(c) where our method with a batch size of 2048 and rank 32 has almost the same performance as SSL when using all the physical cores on the CPU.

The HSS matrix factorization from Chapter 5 uses the PaRSEC runtime system for achieving asynchronous parallelism. The DTD interface from PaRSEC used for this purpose has a large run time overhead which leads to bad scalability for a large problem size and number of nodes. This can be seen in Section 5.3.4 where STRUMPACK outperforms $\mathcal{H}$ATRIX-DTD when the number of problem size is pushed beyond 2^{16} for 64 nodes. While the runtime overhead can be reduced with the use of PaRSEC's PTG interface, it will still remain a significant limitation to better scalability.

6.4 Future work and open questions

This thesis deals exclusively with 2D problems as a result of working with weak admissibility block low rank and hierarchically semi-separable (HSS) matrices. The ranks of the off-diagonals are too high for 3D problems with weakly admissible matrix formats. Therefore, we would like to consider extending our work to strongly admissible block low rank and $\mathcal{H}^2$-matrix formats in the future. This will allow for accommodating a larger range of Green's functions and geometry with controllable rank and accuracy. We would also like to explore applications related to electromagnetic fields solved by the Helmholtz kernel. This will require extension of our code to complex numbers. Since the resulting matrices are also ill-conditioned, the factorization techniques must be studied further.

Ill-conditioned matrices have been a bane of hierarchical matrices. However, many useful applications arising from electromagnetic waves or seismic waves generate ill-conditioned matrices, which makes it very important from an application point of view. The compression and factorization techniques must be adapted to such matrices, and doing so will require modifications to the ULV factorization presented in this thesis. A significant reason behind the poor accuracy of factorization

for strongly admissible approximations of ill-conditioned matrices is that the schur's complements are no longer low rank. Therefore, the use of smaller ranks in the off-diagonal does not help improve the accuracy of the factorization.

The PaRSEC runtime system has proven to be a useful tool for obtaining faster factorization. However, the runtime overhead of the DTD interface remains a serious concern for scaling to large problem sizes on a large number of nodes. The Parameterized Task Graph (PTG) and Templated Task Graph (TTG) interfaces from PaRSEC reduce the overhead by only generating the tasks that will be executed on a given node. This way, the generation of the entire task graph on every node is avoided, which happens to be the primary cause of the runtime overhead. Perfect weak scaling can possibly be attained for the results shown in Section 5.3.2 if we make use of PaRSEC's PTG interface.

Given that PTG already has data about the data flow in the task graph and the communication required for the algorithm, it might be possible to completely remove the runtime overhead by generated code that executed the algorithm. A possible way of doing this is to statically analyse the DAG and generate an MPI program with the scheduling built-in.

The process distribution is also a possible area for further innovation. Current techniques of distributing data across multiple processes are based on prior experience and rigourous experimental analysis. All algorithms in a particular library make use of the same process distribution (such as block cyclic distribution in DPLAMSA) in order to ensure uniformity and maintain an overall good load balance for all the algorithms. However, if the DAG of the computation and the communication is known in advance, it might be possible to generate an optimal schedule and process distribution that will lead to the fastest possible execution time.

Bibliography

[1] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, Tz. Kolev, I. Masliah, and S. Tomov. High-performance Tensor Contractions for GPUs. *Procedia Computer Science*, 80:108–118, January 2016. ISSN 1877-0509. doi: 10.1016/j.procs.2016.05.302.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. On the Development of Variable Size Batched Computation for Heterogeneous Parallel Architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1249–1258, Chicago, IL, USA, May 2016. IEEE. ISBN 978-1-5090-3682-0. doi: 10.1109/IPDPSW.2016.190.

[3] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Fast Batched Matrix Multiplication for Small Sizes Using Half-Precision Arithmetic on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 111–122, Rio de Janeiro, Brazil, May 2019. IEEE. doi: 10.1109/IPDPS.2019.00022.

[4] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes. Parallel Approximation of the Maximum Likelihood Estimation for the Prediction of Large-Scale Geostatistics Simulations. In *IEEE International Conference on Cluster Computing*, Belfast, UK, 2018. IEEE.

[5] Sameh Abdulah, Hatem Ltaief, Ying Sun, Marc G. Genton, and David E. Keyes. ExaGeoStat: A High Performance Unified Software for Geostatistics on Manycore Systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2771–2784, December 2018. ISSN 1045-9219, 1558-2183, 2161-9883. doi: 10.1109/TPDS.2018.2850749.

[6] Ayesha Afzal, Georg Hager, and Gerhard Wellein. An analytic performance model for overlapping execution of memory-bound loop kernels on multicore CPUs. *arXiv:2011.00243 [cs]*, October 2020.

[7] Emmanuel Agullo, Cedric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In *9th*

ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11), Sharm El-Sheikh, Egypt, 2011.

[8] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2017.2766064.

[9] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes. Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures. In *ISC High Performance 2017, LNCS*, volume 10266, pages 22–40, Frankfurt, Germany, 2017. Springer, Cham. ISBN 978-3-319-58666-3. doi: 10.1007/978-3-319-58667-02.

[10] Noha Al-Harthi, Rabab Alomairy, Kadir Akbudak, Rui Chen, Hatem Ltaief, Hakan Bagci, and David Keyes. Solving Acoustic Boundary Integral Equations Using High Performance Tile Low-Rank LU Factorization. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, volume 12151, pages 209–229. Springer International Publishing, Cham, 2020. ISBN 978-3-030-50742-8 978-3-030-50743-5. doi: 10.1007/978-3-030-50743-5_11.

[11] Christie Alappat, Nils Meyer, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, and Tilo Wettig. ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX. *arXiv:2103.03013 [hep-lat]*, March 2021.

[12] Christie L. Alappat, Johannes Seiferth, Georg Hager, Matthias Korch, Thomas Rauber, and Gerhard Wellein. YaskSite: Stencil Optimization Techniques Applied to Explicit ODE Methods on Modern Architectures. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 174–186, February 2021. doi: 10.1109/CGO51591.2021.9370316.

[13] J. I. Aliaga, R. Carratalá-Sáez, E. S. Quintana-Ortí, and R. Kriemann. Task-Parallel LU Factorization of Hierarchical Matrices using OmpSs. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 1148–1157, Lake Buena Vista, FL, USA, 2017. IEEE. doi: 10.1109/IPDPSW.2017.124.

[14] S. Ambikasaran and E. Darve. An O(NlogN) Fast Direct Solver for Partial Hierarchically Semi-seperable Matrices. *Journal of Scientific Computing*, 57:477–501, 2013.

[15] S. Ambikasaran and E. Darve. The Inverse Fast Multipole Method. *arXiv:1407.1572v1*, 2014.

[16] S. Ambikasaran, D. Foreman-Mackey, L. Greengard, D. W. Hogg, and M. O'Neil. Fast Direct Methods for Gaussian Processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):252–265, 2016.

[17] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. Improving Multifrontal Methods by Means of Block Low-Rank Representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015.

[18] Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. Bridging the Gap Between Flat and Hierarchical Low-Rank Matrix Formats: The Multilevel Blr Format. *SIAM Journal on Scientific Computing*, 41(3):A1414–A1442, May 2019. doi: 10.1137/18M1182760.

[19] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, New York, New York, USA, October 1990. IEEE Computer Society Press. ISBN 978-0-89791-412-3.

[20] Hartwig Anzt, Erik Boman, Rob Falgout, Pieter Ghysels, Michael Heroux, Xiaoye Li, Lois Curfman McInnes, Richard Tran Mills, Sivasankaran Rajamanickam, Karl Rupp, Barry Smith, Ichitaro Yamazaki, and Ulrike Meier Yang. Preparing sparse solvers for exascale computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2166):20190053, January 2020. doi: 10.1098/rsta.2019.0053.

[21] Cleve Ashcraft, Alfredo Buttari, and Theo Mary. Block Low-Rank Matrices with Shared Bases: Potential and Limitations of the BLR$\hat{2}$ Format. *SIAM Journal on Matrix Analysis and Applications*, 42(2):990–1010, January 2021. ISSN 0895-4798, 1095-7162. doi: 10.1137/20M1386451.

[22] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, 2011. doi: 10.1002/cpe.1631.

[23] C. Augonnet, D. Goudin, M. Kuhn, X. Lacoste, R. Namyst, and P. Ramet. A Hierarchical Fast Direct Solver for Distributed Memory Machines with Manycore Nodes. Technical report, Université de Bordeaux, 2019.

[24] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par - 15th International Conference on Parallel Processing*, volume 5704 of *Lecture Notes*

in Computer Science, pages 863–874, Delft, The Netherlands, August 2009. Springer. doi: 10.1007/978-3-642-03869-3_80.

[25] M. Bebendorf and W. Hackbusch. Stabilized Rounded Addition of Hierarchical Matrices. *Numerical Linear Algebra with Applications*, 14(5):407–423, June 2007. ISSN 10705325, 10991506. doi: 10.1002/nla.525.

[26] S. Borm. $H\hat{2}$-Matrix Arithmetics in Linear Complexity. *Computing*, 77:1–28, 2006.

[27] S. Borm. Hierarchical Matrix Arithmetic with Accumulated Updates. *Computing and Visualization in Science*, pages 1–14, 2019. doi: 10.1007/s00791-019-00311-3.

[28] S. Borm, L. Grasedyck, and W. Hackbusch. Introduction to Hierarchical Matrices with Applications. *Engineering Analysis with Boundary Elements*, 27:405–422, 2003. doi: 10.1016/S0955-7997(02)00152-2.

[29] Steffen Borm, Sven Christophersen, and Ronald Kriemann. Semi-Automatic Task Graph Construction for H-Matrix Arithmetic. *arXiv:1911.07531 [cs]*, November 2019.

[30] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1432–1441, Anchorage, AK, USA, 2011. IEEE. ISBN 978-1-61284-425-1. doi: 10.1109/IPDPS.2011.299.

[31] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1432–1441, 2011. ISSN 1530-2075. doi: 10.1109/IPDPS.2011.299.

[32] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, January 2012. ISSN 01678191. doi: 10.1016/j.parco.2011.10.003.

[33] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013. ISSN 1521-9615. doi: 10.1109/MCSE.2013.98.

[34] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes. Batched QR and SVD Algorithms on GPUs with Applications in Hierarchical Matrix Compression. *Parallel Computing*, 74:19–33, 2018. doi: 10.1016/j.parco.2017.09.001.

[35] Wajih Boukaram, Stefano Zampini, George Turkiyyah, and David Keyes. H2OPUS-TLR: High Performance Tile Low Rank Symmetric Factorizations using Adaptive Randomized Approximation. *arXiv:2108.11932 [cs]*, August 2021.

[36] Michael Brazell, Na Li, Carmeliza Navasca, and Christino Tamon. Tensor and Matrix Inversions with Applications. *arXiv:1109.3830 [math]*, September 2011.

[37] Calin Ca*B*caval and David A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 150–159, San Francisco, CA, USA, June 2003. Association for Computing Machinery. ISBN 978-1-58113-733-0. doi: 10.1145/782814.782836.

[38] Léopold Cambier and Eric Darve. A task-based distributed parallel sparsified nested dissection algorithm. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–11, Geneva Switzerland, July 2021. ACM. ISBN 978-1-4503-8563-3. doi: 10.1145/3468267.3470619.

[39] Hu Cao, Yueyue Wang, Joy Chen, Dongsheng Jiang, Xiaopeng Zhang, Qi Tian, and Manning Wang. Swin-Unet: Unet-like Pure Transformer for Medical Image Segmentation, May 2021.

[40] Qinglei Cao, Yu Pei, Kadir Akbudak, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Leveraging PaRSEC Runtime Support to Tackle Challenging 3D Data-Sparse Matrix Problems. page 10, 2020.

[41] Qinglei Cao, Yu Pei, Kadir Akbudak, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Leveraging PaRSEC Runtime Support to Tackle Challenging 3D Data-Sparse Matrix Problems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 79–89, Portland, OR, USA, May 2021. IEEE. ISBN 978-1-66544-066-0. doi: 10.1109/IPDPS49936.2021.00017.

[42] Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G. Genton, David E. Keyes, Hatem Ltaief, and Ying Sun. Reshaping geostatistical modeling and prediction for extreme-scale environmental applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22, pages 1–12, Dallas, Texas, November 2022. IEEE Press.

[43] Qinglei Cao, Rabab Alomairy, Yu Pei, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. A Framework to Exploit Data Sparsity in Tile Low-Rank Cholesky Factorization. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 414–424, Lyon, France, May 2022. IEEE. doi: 10.1109/IPDPS53621.2022.00047.

[44] Quinglei Cao, Yu Pei, Thomas Herauldt, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 25–32, November 2019. doi: 10.1109/ProTools49597.2019.00009.

[45] Rocío Carratalá Sáez. *Analysis of Parallelization Strategies in the Context of Hierarchical Matrix Factorizations*. PhD thesis, Universitat Jaume I, Castelló de la Plana, March 2021.

[46] Rocío Carratalá-Sáez, Sven Christophersen, J. I. Aliaga, Vicencc Beltran, S. Börm, and E. S. Quintana-Ortí. Exploiting nested task-parallelism in the H-LU factorization. *J. Comput. Sci.*, 33:20–33, April 2019. doi: 10.1016/j.jocs.2019.02.004.

[47] Marc Casas and Greg Bronevetsky. Active Measurement of Memory Resource Consumption. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 995–1004, Phoenix, AZ, USA, May 2014. IEEE. ISBN 978-1-4799-3800-1 978-1-4799-3799-8. doi: 10.1109/IPDPS.2014.105.

[48] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. Van Der Veen, and D. White. Some Fast Algorithms for Sequentially Semiseparable Representations. *SIAM Journal on Matrix Analysis and Applications*, 27(2):341–364, 2005. doi: 10.1137/S0895479802405884.

[49] S. Chandrasekaran, M. Gu, and T. Pals. A Fast ULV Decomposition Solver for Hierarchically Semiseparable Representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3): 603–622, 2006. doi: 10.1137/S0895479803436652.

[50] Ali Charara, David Keyes, and Hatem Ltaief. Batched Tile Low-Rank GEMM on GPUs. page 12, 2018.

[51] Ali Charara, David Keyes, and Hatem Ltaief. Batched Triangular Dense Linear Algebra Kernels for Very Small Matrix Sizes on GPUs. *ACM Transactions on Mathematical Software*, 45(2):1–28, June 2019. ISSN 0098-3500, 1557-7295. doi: 10.1145/3267101.

[52] Ali Charara, David Keyes, and Hatem Ltaief. Batched Triangular Dense Linear Algebra Kernels for Very Small Matrix Sizes on GPUs. *ACM Transactions on Mathematical Software*, 45(2):1–28, June 2019. ISSN 0098-3500, 1557-7295. doi: 10.1145/3267101.

[53] Chun-Fu Chen, Rameswar Panda, and Quanfu Fan. RegionViT: Regional-to-Local Attention for Vision Transformers, March 2022.

[54] Jianyu Chen, Zaid Al-Ars, and H. Peter Hofstee. A matrix-multiply unit for posits in reconfigurable logic leveraging (open)CAPI. In *Proceedings of the Conference for Next Generation Arithmetic*, pages 1–5, Singapore Singapore, March 2018. ACM. ISBN 978-1-4503-6414-0. doi: 10.1145/3190339.3190340.

[55] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. TSM2: Optimizing Tall-and-Skinny Matrix-Matrix Multiplication on GPUs. page 11, 2019.

[56] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5(3):173–184, 1996. ISSN 1058-9244, 1875-919X. doi: 10.1155/1996/483083.

[57] P. Coulier, H. Pouransari, and E. Darve. The Inverse Fast Multipole Method: Using a Fast Approximate Direct Solver as a Preconditioner for Dense Linear Systems. *SIAM Journal on Scientific Computing*, 39(3):A761–A796, 2017. doi: 10.1137/15M1034477.

[58] J J Dongarra and S Hammarling. Key Concepts for Parallel Out-of-Core LU Factorization. 1998.

[59] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, January 1979. ISBN 978-0-89871-172-1. doi: 10.1137/1.9781611971811.

[60] Jack Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems. *Procedia Computer Science*, 108:495–504, January 2017. ISSN 1877-0509. doi: 10.1016/j.procs.2017.05.138.

[61] Jack J. Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems. In *ICCS*, 2017. doi: 10.1016/j.procs.2017.05.138.

[62] Nils-Arne Dreier and Christian Engwer. Strategies for the vectorized Block Conjugate Gradients method. *arXiv:1912.11930 [cs, math]*, December 2019.

[63] Toshio Endo. Integrating Cache Oblivious Approach with Modern Processor Architecture: The Case of Floyd-Warshall Algorithm. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2020, pages 123–130, New York, NY, USA, January 2020. Association for Computing Machinery. ISBN 978-1-4503-7236-7. doi: 10.1145/3368474.3368477.

[64] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22, January 2012. ISSN 1549-6325. doi: 10.1145/2071379.2071383.

[65] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. *ACM Transactions on Mathematical Software*, 44(4):1–30, August 2018. ISSN 0098-3500, 1557-7295. doi: 10.1145/3210754.

[66] Gianluca Frison, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. The BLAS API of BLASFEO: Optimizing performance for small matrices. *ACM Transactions on Mathematical Software*, 46(2):1–36, June 2020. ISSN 0098-3500, 1557-7295. doi: 10.1145/3378671.

[67] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, March 2005. ISSN 0018-8646, 0018-8646. doi: 10.1147/rd.492.0195.

[68] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the 2019 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, USA, 2019. ACM. doi: 10.1145/3295500.3356223.

[69] Mark Gates. Designing SLATE. January 2018.

[70] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy Of High-Performance Deep Learning Convolutions On SIMD Architectures. *arXiv:1808.05567 [cs]*, August 2018.

[71] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov. An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016. doi: 10.1137/15M1010117.

[72] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):1–25, May 2008. ISSN 0098-3500, 1557-7295. doi: 10.1145/1356052.1356053.

[73] L. Grasedyck and W. Hackbusch. Construction and Arithmetics of H-Matrices. *Computing*, 70:295–334, 2003. doi: 10.1007/s00607-003-0019-1.

[74] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001. ISSN 0098-3500. doi: 10.1145/504210.504213.

[75] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. A fast analytical model of fully associative caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 816–829, Phoenix, AZ, USA, June 2019. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314606.

[76] W. Hackbusch. A Sparse Matrix Arithmetic Based on $\Cal H$ -Matrices. Part I: Introduction to ${\Cal H}$ -Matrices. *Computing*, 62(2):89–108, April 1999. ISSN 0010-485X, 1436-5057. doi: 10.1007/s006070050015.

[77] W. Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*, volume 49 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-47324-5.

[78] W. Hackbusch and Z. P. Nowak. On the Fast Matrix Multiplication in the Boundary Element Method by Panel Clustering. *Numerische Mathematik*, 54:463–491, 1989. doi: 10.1007/BF01396324.

[79] W. Hackbusch, B. Khoromskij, and S. A. Sauter. On $H\hat{2}$-Matrices. In H. Bungartz, R. Hoppe, and C. Zenger, editors, *Lectures on Applied Mathematics*. Springer Berlin Heidelberg, 2000. doi: 10.1007/978-3-642-59709-1_2.

[80] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Towards batched linear solvers on accelerated hardware platforms. *ACM SIGPLAN Notices*, 50(8):261–262, January 2015. ISSN 0362-1340. doi: 10.1145/2858788.2688534.

[81] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. 53(2): 217–288, September 2009. ISSN 0036-1445. doi: 10.1137/090771806.

[82] Helmut Harbrecht and Peter Zaspel. A scalable H-matrix approach for the solution of boundary integral equations on multi-GPU clusters. *arXiv:1806.11558 [cs, math]*, June 2018.

[83] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 1–11, Salt Lake City, Utah, November 2016. IEEE Press. ISBN 978-1-4673-8815-3.

[84] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, SC '11, pages 1–12, Seattle, Washington, November 2011. Association for Computing Machinery. ISBN 978-1-4503-1139-7. doi: 10.1145/2063348.2063356.

[85] Johannes Hofmann and Dietmar Fey. An ECM-based energy-efficiency optimization approach for bandwidth-limited streaming kernels on recent Intel Xeon processors. *arXiv:1609.03347 [cs]*, September 2016.

[86] Johannes Hofmann, Christie L. Alappat, Georg Hager, Dietmar Fey, and Gerhard Wellein. Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors. *Supercomputing Frontiers and Innovations*, 7(2), June 2020. ISSN 23138734. doi: 10.14529/jsfi200204.

[87] Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert A. van de Geijn. Generating Families of Practical Fast Matrix Multiplication Algorithms. *arXiv:1611.01120 [cs]*, November 2016.

[88] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. Van De Geijn. Strassen's Algorithm Reloaded. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 690–701, November 2016. doi: 10.1109/SC.2016.58.

[89] A. Ida, T. Ataka, Y. Takahashi, T. Mifune, T. Iwashita, and A. Furuya. Application of Improved H-Matrices in Micromagnetic Simulations of Spin Torque Oscillator. *IEEE Transactions on Magnetics*, 54(3):7202804, 2018. doi: 10.1109/TMAG.2017.2763611.

[90] Akihiro Ida. Lattice H-Matrices on Distributed-Memory Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 389–398, Vancouver, BC, May 2018. IEEE. ISBN 978-1-5386-4368-6. doi: 10.1109/IPDPS.2018.00049.

[91] Akihiro Ida, Tadashi Ataka, and Atsushi Furuya. Lattice H-Matrices for Massively Parallel Micromagnetic Simulations of Current-Induced Domain Wall Motion. *IEEE Transactions on Magnetics*, 56(4):1–4, April 2020. ISSN 1941-0069. doi: 10.1109/TMAG.2019.2959349.

[92] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization Framework for Sparse Matrix Kernels. *The International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004. ISSN 1094-3420. doi: 10.1177/1094342004041296.

[93] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, September 2004. ISSN 0743-7315. doi: 10.1016/j.jpdc.2004.03.021.

[94] T. Iwashita, A. Ida, T. Mifune, and Y. Takahashi. Software Framework for Parallel BEM Analyses with H-matrices Using MPI and OpenMP. *Procedia Computer Science*, 108C:2200–2209, 2017. doi: 10.1016/j.procs.2017.05.263.

[95] Claude-Pierre Jeannerod, Theo Mary, Clement Pernet, and Daniel S Roche. Exploiting Fast Matrix Arithmetic in Block Low-Rank Factorizations. page 15, 2019.

[96] Lijuan Jiang, Chao Yang, and Wenjing Ma. Enabling Highly Efficient Batched Matrix Multiplications on SW26010 Many-core Processor. *ACM Transactions on Architecture and Code Optimization*, 17(1):1–23, March 2020. ISSN 1544-3566, 1544-3973. doi: 10.1145/3378176.

[97] John Katsikadelis. *The Boundary Element Method for Engineers and Scientists: Theory and Applications: Second Edition*. July 2016.

[98] Kyungjoo Kim, Timothy B. Costa, Mehmet Deveci, Andrew M. Bradley, Simon D. Hammond, Murat E. Guney, Sarah Knepper, Shane Story, and Sivasankaran Rajamanickam. Designing vector-friendly compact BLAS and LAPACK kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 1–12, Denver, Colorado, November 2017. Association for Computing Machinery. ISBN 978-1-4503-5114-0. doi: 10.1145/3126908.3126941.

[99] R. Kriemann. H-LU Factorization on Many-Core Systems. In *Computing and Visualization in Science*. Springer Berlin Heidelberg, 2014.

[100] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131, November 2018. doi: 10.1109/PMBS.2018.8641578.

[101] Alexander Litvinenko. HLIBCov: Parallel Hierarchical Matrix Approximation of Large Covariance Matrices and Likelihoods with Applications in Parameter Identification. *arXiv:1709.08625 [math, stat]*, May 2019.

[102] Alexander Litvinenko, Ying Sun, Marc G. Genton, and David Keyes. Likelihood Approximation With Hierarchical Matrices For Large Spatial Datasets, September 2018.

[103] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):12:1–12:18, August 2016. ISSN 0098-3500. doi: 10.1145/2925987.

[104] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):1–18, June 2017. ISSN 0098-3500, 1557-7295. doi: 10.1145/2925987.

[105] Hatem Ltaief, Yuxi Hong, David Keyes, Damien Gratadour, Jesse Cranney, and Laurent Gatineau. Meeting The Real-Time Challenges of Ground-Based Telescopes Using Low-Rank Matrix Computations. page 14, St. Louis, MO, USA, 2021. IEEE.

[106] Miaomiao Ma and Dan Jiao. Accuracy Controlled Direct Integral Equation Solver of Linear Complexity with Change of Basis for Large-Scale Interconnect Extraction. In *2018 IEEE/MTT-S International Microwave Symposium - IMS*, pages 197–200, June 2018. doi: 10.1109/MWSYM.2018.8439378.

[107] Miaomiao Ma and Dan Jiao. Direct Solution of General H2 -Matrices With Controlled Accuracy and Concurrent Change of Cluster Bases for Electromagnetic Analysis. *IEEE Transactions on Microwave Theory and Techniques*, 67(6):2114–2127, June 2019. ISSN 1557-9670. doi: 10.1109/TMTT.2019.2914391.

[108] Qianxiang Ma, Sameer Deshmukh, and Rio Yokota. Scalable Linear Time Dense Direct Solver for 3-D Problems without Trailing Sub-Matrix Dependencies. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1198–1209, Dallas, TX, USA, November 2022. IEEE Computer Society. ISBN 978-1-66545-444-5.

[109] P. G. Martinsson and V. Rokhlin. A Fast Direct Solver for Boundary Integral Equations in Two Dimensions. *Journal of Computational Physics*, 205:1–23, 2005. doi: 10.1016/j.jcp.2004.10.033.

[110] Ian Masliah, Ahmad Abdelfattah, A. Haidar, S. Tomov, Marc Baboulin, J. Falcou, and J. Dongarra. High-Performance Matrix-Matrix Multiplications of Very Small Matrices. In Pierre-François Dutot and Denis Trystram, editors, *Euro-Par 2016: Parallel Processing*, volume 9833, pages 659–671. Springer International Publishing, Cham, 2016. ISBN 978-3-319-43658-6 978-3-319-43659-3. doi: 10.1007/978-3-319-43659-3_48.

[111] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. ISSN 0018-8670. doi: 10.1147/sj.92.0078.

[112] John D. McCalpin. Sustainable memory bandwidth in current high performance computers. 1995.

[113] Tan M. Nguyen, Vai Suliafu, Stanley J. Osher, Long Chen, and Bao Wang. FMMformer: Efficient and Flexible Transformer via Decomposed Near-field and Far-field Attention, August 2021.

[114] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, May 2007. ISSN 1432-0622. doi: 10.1007/s00200-007-0038-9.

[115] S. Ohshima, I. Yamazaki, Akihiro Ida, and Rio Yokota. Optimization of Numerous Small Dense-Matrix–Vector Multiplications in H-Matrix Arithmetic on GPU. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, Singapore, 2019. IEEE. ISBN 978-1-72814-883-0. doi: 10.1109/MCSoC.2019.00009.

[116] Y. Pei, G. Bosilca, I. Yamazaki, A. Ida, and J. Dongarra. Evaluation of Programming Models to Address Load Imbalance on Distributed Multi-Core CPUs: A Case Study with Block Low-Rank Factorization. In *IEEE/ACM Parallel Applications Workshop, Alternatives to MPI (PAW-ATM)*, 2019.

[117] Antoine P. Petitet, R. Clint Whaley, Jack Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, December 2018.

[118] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, February 2013. ISSN 0098-3500. doi: 10.1145/2427023.2427030.

[119] H. Pouransari, P. Coulier, and E. Darve. Fast Hierarchical Solvers for Sparse Matrices Using Extended Sparsification and Low-Rank Approximation. *SIAM Journal on Scientific Computing*, 39(3):A797–A830, 2017. doi: 10.1137/15M1046939.

[120] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Extreme-Scale Task-Based Cholesky Factorization Toward Climate and Weather Prediction Applications. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '20, pages 1–11, New York, NY, USA, June 2020. Association for Computing Machinery. ISBN 978-1-4503-7993-9. doi: 10.1145/3394277.3401846.

[121] Cody Rivera, Jieyang Chen, Nan Xiong, Shuaiwen Leon Song, and Dingwen Tao. TSM2X: High-Performance Tall-and-Skinny Matrix-Matrix Multiplication on GPUs. *Journal of Parallel and Distributed Computing*, 151:70–85, May 2021. ISSN 07437315. doi: 10.1016/j.jpdc.2021.02.013.

[122] S. Rjasanow. Adaptive Cross Approximation of Dense Matrices. In *International Association for Boundary Element Methods*, UT Austin, TX, USA, May 2002.

[123] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov. A Distributed-Memory Package for Dense Hierarchically Semi-Seperable Matrix Computations Using Randomization. *ACM Transactions on Mathematical Software*, 42(4):Article 27, 2016.

[124] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense Hierarchically Semi-Separable matrix computations using randomization. *arXiv:1503.05464 [cs]*, 42(4):Article No.: 27, pages 1–35, June 2015.

[125] Youcef Saad. Communication complexity of the Gaussian elimination algorithm on multiprocessors. *Linear Algebra and its Applications*, 77:315–340, May 1986. ISSN 0024-3795. doi: 10.1016/0024-3795(86)90174-6.

[126] Abdusamad A. Salih. Finite Element Method. *Department of Aerospace Engineering Indian Institute of Space Science and Technology Thiruvananthapuram-695547, India, Z. American Journal of Physics1997*, 65(6):537–543, 2012.

[127] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1049–1059, Phoenix, AZ, USA, May 2014. IEEE. ISBN 978-1-4799-3799-8. doi: 10.1109/IPDPS.2014.110.

[128] Edgar Solomonik and James Demmel. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, Lecture Notes in Computer Science, pages 90–109, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-23397-5. doi: 10.1007/978-3-642-23397-5_10.

[129] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, March 2017. ISSN 0272-1732. doi: 10.1109/MM.2017.35.

[130] Gilbert Strang. *Introduction to Applied Mathematics*. 1986.

[131] Gilbert Strang. *Computational Science and Engineering*. Wellesley-Cambridge Press, 2007.

[132] Gilbert Strang. *Essays In Linear Algebra*. 2012.

[133] Gilbert Strang. *Differential Equations and Linear Algebra*. 2014.

[134] Gilbert Strang. *Introduction to Linear Algebra*. Fifth edition, 2016. ISBN 978-0-9802327-7-6.

[135] T. Takahashi, P. Coulier, and E. Darve. Application of the Inverse Fast Multipole Method as a Preconditioner in a 3D Helmholtz Boundary Element Method. *Journal of Computational Physics*, 341:406–428, 2017. doi: 10.1016/j.jcp.2017.04.016.

[136] Toru Takahashi, Chao Chen, and Eric Darve. Parallelization of the Inverse Fast Multipole Method with an Application to Boundary Element Method. *Computer Physics Communications*, 247:106975, February 2020. ISSN 00104655. doi: 10.1016/j.cpc.2019.106975.

[137] Lloyd N. Trefethen. *Spectral Methods in MATLAB*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics, January 2000. ISBN 978-0-89871-465-4. doi: 10.1137/1.9780898719598.

[138] Field G. Van Zee and Robert A. van de Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, June 2015. ISSN 0098-3500. doi: 10.1145/2764454.

[139] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. The BLIS Framework: Experiments in Portability. *ACM Transactions on Mathematical Software*, 42(2):12:1–12:19, June 2016. ISSN 0098-3500. doi: 10.1145/2755561.

[140] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 26–26, November 2002. doi: 10.1109/SC.2002.10025.

[141] Ruoxi Wang, Yingzhou Li, Michael W. Mahoney, and Eric Darve. Block Basis Factorization for Scalable Kernel Matrix Evaluation. *SIAM Journal on Matrix Analysis and Applications*, 40(4):1497–1526, January 2019. ISSN 0895-4798, 1095-7162. doi: 10.1137/18M1212586.

[142] R.C. Whaley and J.J. Dongarra. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 38–38, Orlando, FL, USA, November 1998. IEEE. ISBN 0-8186-8707-X. doi: 10.1109/SC.1998.10004.

[143] Karl-Robert Wichmann, Martin Kronbichler, Rainald Löhner, and Wolfgang A Wall. Practical applicability of optimizations and performance models to complex stencil-based loop kernels in CFD. *The International Journal of High Performance Computing Applications*, 33(4):602–618, July 2019. ISSN 1094-3420. doi: 10.1177/1094342018774126.

[144] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures, April 2009.

[145] Markus Wittmann, Georg Hager, Thomas Zeiser, Jan Treibig, and Gerhard Wellein. Chip-level and multi-node analysis of energy-optimized lattice-Boltzmann CFD simulations. *Concurrency and Computation: Practice and Experience*, 28(7):2295–2315, May 2016. ISSN 15320626. doi: 10.1002/cpe.3489.

[146] Ichitaro Yamazaki, Akihiro Ida, Rio Yokota, and Jack Dongarra. Distributed-memory lattice H-matrix factorization. *The International Journal of High Performance Computing Applications*, 33(5):1046–1063, September 2019. ISSN 1094-3420, 1741-2846. doi: 10.1177/1094342019861139.

[147] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. LibShalom: Optimizing Small and Irregular-Shaped Matrix Multiplications on ARMv8 Multi-Cores. page 13, 2021.

[148] Asim Yarkhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the PLASMA Numerical Library to the OpenMP Standard. *Int. J. Parallel Program.*, 45(3):612–633, June 2017. ISSN 0885-7458. doi: 10.1007/s10766-016-0441-6.

[149] R. Yokota and L. A. Barba. Treecode and Fast Multipole Method for N-Body Simulation with CUDA. In *GPU Computing Gems*, chapter 9. Morgan Kaufmann, emerald edition edition, 2011.

[150] K. Yotov, Xiaoming Li, Gang Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High-Performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, February 2005. ISSN 1558-2256. doi: 10.1109/JPROC.2004.840444.

[151] C. D. Yu, S. Reiz, and G. Biros. Distributed $O(N)$ Linear Solver for Dense Symmetric Hierarchical Semi-Separable Matrices. In *IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2019.

[152] Stefano Zampini, Wajih Boukaram, George Turkiyyah, Omar Knio, and David E Keyes. H2opus: A Distributed-Memory Multi-Gpu Software Package for Non-Local Operators. page 30, 2021.

[153] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. The BLIS Framework: Experiments in Portability. *ACM Transactions on Mathematical Software*, 42(2):12:1–12:19, June 2016. ISSN 0098-3500. doi: 10.1145/2755561.