HIGH-PERFORMANCE SOFTWARE FOR QUANTUM CHEMISTRY AND HIERARCHICAL MATRICES

A Dissertation
Presented to
The Academic Faculty

By

Lucas Erlandson

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing
Computational Science and Engineering

Georgia Institute of Technology

August 2022

HIGH-PERFORMANCE SOFTWARE FOR QUANTUM CHEMISTRY AND HIERARCHICAL MATRICES

ACKNOWLEDGMENTS

I wish to express immense gratitude to my advisor Edmond Chow, who guided me with the utmost patience and expertise throughout the entirety of my Ph.D. journey. His ability to provide pertinent insight into the topics and challenges does not go unappreciated and is central to completing my studies. I consider myself fortunate to have been his Ph.D. student and for his attention.

I would also like to thank the other committee members: Yuanzhe Xi, Ruipeng Li, Tobin Isaac, and Felix Herrmann, who provided significant insight and feedback on my thesis. Similarly, I would like to thank my collaborators and mentors throughout my journey. First, I would like to thank Yousef Saad, who introduced me to numerical methods and mentored me in his lab during my undergraduate studies. Second, Yuanzhe Xi, who mentored me and helped me grow my interest and appreciation for numerical methods and continues to collaborate with me and provide guidance to this day. Third, Ruipeng Li, who mentored me during my internships at Lawrence Livermore National Lab and offered me significant Algebraic Multigrid and GPU expertise. Fourth, Ana María Estrada Gómez, who has collaborated with me, providing insight into statistics, academia, and many great conversations. Fifth, my collaborators for the SPARC project, for giving great feedback on computational chemistry. And finally, to my other collaborators: Difeng Cai, Shifan Zhao, Tianshi Xu, and Kamran Paynabar, for their time and expertise.

Next, I would like to extend my thanks to my labmates: Shikhar Shah, Hua Huang, Xin Xing, and Jordi Wolfson-Pou, for academic and personal growth. To Shikhar, for always being willing to answer and ask questions – fun or serious. To Hua, for the in-depth knowledge of high-performance computing and willingness to share it. To Xin, for the many fruitful discussions on hierarchical matrices. And to Jordi, for being there to answer questions about graduate studies.

I am grateful to the Ballroom Dance Club at Georgia Tech, and the many friends I made

within it for the enjoyment and impact on my life. Including, but not by any means limited to, Erin Wrobel, Amy Doneff, Brittney Bush, Walker Powell, Olivia Apergis, Julia Zhu, Faith Womack, for both friendship and coaching. And last, to Darye Ji, for bringing many laughs and so much joy both in and out of our practices.

Finally, I am deeply appreciative of my family's eternal love and support - my father, John, mother, Amy, sister Shae, and brothers Alex and Chris.

This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project, a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

The authors gratefully acknowledge support from the U.S. Department of Energy, Office of Science under grant DE-SC0019410 and the U.S. National Science Foundation under grants ACI-2003683 and CMMI-1839591

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Linear algebra is the underpinning of a significant portion of the computation done in the modern age. Applications relying on linear algebra include physical and chemical simulations, machine learning, artificial intelligence, optimization, partial differential equations, and many more. However, the direct use of mathematically exact linear algebra is often infeasible for the large problems of today. Numerical and iterative methods provide a way of solving the underlying problems only to the required accuracy, allowing problems that are many magnitudes larger to be solved magnitudes more quickly than if the problems were to be solved using exact linear algebra. In this dissertation, we discuss, test existing methods, and develop new high-performance numerical methods for scientific computing kernels, including matrix-multiplications, linear solves, and eigensolves, which accelerate applications including Gaussian processes and quantum chemistry simulations. Notably, we use preconditioned hierarchical matrices for the hyperparameter optimization and prediction phases of Gaussian process regression, develop a sparse triple matrix product on GPUs, and investigate 3D matrix-matrix multiplications for Chebyshev-filtered subspace iteration for Kohn-Sham density functional theory calculations.

The exploitation of the structural sparsity of many practical scientific problems can achieve a significant speedup over the dense formulations of the same problems. Even so, many problems cannot be accurately represented or approximated in a structurally sparse manner. Many of these problems, such as kernels arising from machine learning and the Electronic-Repulsion-Integral (ERI) matrices from electronic structure computations, can be accurately represented in data-sparse structures, which allows for rapid calculations. We investigate hierarchical matrices, which provide a data-sparse representation of kernel matrices. In particular, our SMASH approximation can construct and provide matrix multiplications in near-linear time, which can then be used in matrix-free methods to find the optimal hyperparameters for Gaussian processes and to do prediction asymptotically more rapidly than direct methods. To accelerate the use of hierarchical matrices further, we provide a data-driven approach (where we consider the distribution of the data points associated with a kernel matrix) that reduces a given problem's memory and computation requirements. Furthermore, we investigate the use of preconditioning in Gaussian process regression. We can use matrix-free algorithms for hyperparameter optimization and prediction phases of Gaussian process. This provides a framework for Gaussian process regression that scales to large-scale problems and is asymptotically faster than state-of-the-art methods. We provide an exploration and analysis of the conditioning and numerical issues that arise from the near-rank-deficient matrices that occur during hyperparameter optimizations.

Density Functional Theory (DFT) is a valuable method for electronic structure calculations for simulating quantum chemical systems due to its high accuracy to cost ratio. However, even with the computational power of modern computers, the $O(n^3)$ complexity of the eigensolves and other kernels mandate that new methods are developed to allow larger problems to be solved. Two promising methods for tackling these problems are using modern architectures (including state-of-the-art accelerators and multicore systems) and 3D matrix-multiplication algorithms. We investigate these methods to determine if using these methods will result in an overall speedup. Using these kernels, we provide a high-performance framework for Chebyshev-filtered subspace iteration.

GPUs are a family of accelerators that provide immense computational power but must be used correctly to achieve good efficiency. In algebraic multigrid, there arises a sparse triple matrix product, which due to the sparse (and relatively unstructured) nature, is challenging to perform efficiently on GPUs, and is typically done as two successive matrix-matrix products. However, by doing a single triple-matrix product, reducing the overhead associated with sparse matrix-matrix products on the GPU may be possible. We develop a sparse triple-matrix product that reduces the computation time required for a few classes of problems.

CHAPTER 1
INTRODUCTION

The main content of this dissertation is split into three major parts. First, hierarchical matrices, their use in various applications, and new methods developed for them are discussed. The next part discusses the use of 3D matrix-matrix products for Chebyshev-filtered subspace iteration, as well as other methods for accelerating ab-initio quantum chemistry methods. The final part discusses a sparse triple matrix product on GPUs for sparse matrix calculations.

Many scientific computing applications involve dense matrices, which arise from evaluating a function pairwise between points in a dataset. Such kernel functions often arise when considering the pairwise interactions of points, such as in n-body problems, particle physics, and Gaussian processes [1, 2, 3]. Computing the matrix corresponding with n points directly would take $O(n^2)$ time and space for a constant time kernel, which may be prohibitively expensive for large values of n. However, hierarchical matrices are a family of methods that can asymptotically reduce the computation and storage costs by approximating the interactions intelligently. The exact costs and manner depend on the type of hierarchical matrix used; some common hierarchical matrices include the $\mathcal{H}$, HODLR, HSS, $\mathcal{H}^2$ matrices [4, 5, 6, 7], each of which has its own benefits and drawbacks. In particular, this dissertation focuses on $\mathcal{H}^2$ matrices due to their ability to handle a more general set of problems compared to the other methods. The corresponding portion of this dissertation details the SMASH hierarchical matrices, and the use of hierarchical matrices in Gaussian process regression. In particular, by combining preconditioners with matrix-free methods, we build a framework for large-scale Gaussian process regression, which can handle both the hyperparameter optimization and prediction phases.

Quantum mechanics is the governing theory on the electronic and atomic scales, used in calculations including material science, chemistry, and physics [8, 9]. However, the governing Schrödinger Equation is intractable to solve for all but the smallest problems. A variety of approximations and methods have been developed to increase the tractability of the problems. One such method, Kohn-Sham density functional theory (DFT), is commonly used due to its accuracy to cost ratio, which requires the solution of eigenvalue problems. One method to further reduce the cost of the computations is the use of Chebyshev-filtered subspace iteration, which replaces many of the eigenvalue problems with the titular subspace iteration [10]. The content of this dissertation investigates Chebyshev-filtered subspace iteration, develops a high-performance framework and implementation which allows for the use of GPUs in a distributed setting, and investigates the use of 3D matrix-matrix products, which theoretically provide an improvement over traditional matrix products.

The solution of linear systems arises in many scientific computing and machine learning applications, including simulations, the finite element method, and Gaussian processes. One method of solving linear systems on discretized problems is the multigrid method, which can scale linearly with the number of discretization points [11]. The algebraic multigrid method extends the multigrid method to work outside differential equations and other applications with well-defined discretizations [12]. As part of algebraic multigrid, triple-matrix products are performed, which often involve very large, sparse matrices. Developing a GPU-supported triple matrix product can increase the performance of such algebraic multigrid solvers. In the final part of this dissertation, we implement such a triple-matrix product and demonstrate its performance with experimental results.

1.1 Outline and Contributions

In Chapter 2 we begin our discussion on hierarchical matrices. We begin with a background on hierarchical matrices, which leads into the Structured Approximation by Separation and Hierarchy (SMASH) algorithm for hierarchical matrices. We provide an overview of hierarchical matrices and SMASH to provide a foundation for later chapters.

In Chapter 3 we describe a data-driven-based approach to hierarchical matrices, building on SMASH. By using data-driven-based sampling, one can reduce the amount of computation required to achieve a given accuracy. We demonstrate the efficacy of using the data-driven-based approach, which was able to perform a computation to the same accuracy as the state-of-the-art method in 31.6% of the memory and 39.3% of the time, demonstrating its effectiveness.

In Chapter 4 we investigate the use of hierarchical matrices for high-dimensional data. We demonstrate the deficiencies of current hierarchical matrix methods, and then demonstrate the efficacy of combining proxy surface with dimensionality reduction methods. This results in hierarchical matrices which can scale better to higher dimensions, alleviating the curse of dimensionality.

In Chapter 5 we begin our investigation into Gaussian processes, in particular accelerating solves with kernel matrices arising from Gaussian processes, using preconditioned iterative methods. We provide a brief background into Gaussian process regression, including describing the scalability issues faced when considering large problems. By exploiting the block low-rank structure in the kernel matrices that arise in Gaussian process regression, one can scale Gaussian processes to much larger problems. However, the systems that arise during Gaussian process regression can be numerically low-rank and mathematically ill-conditioned, resulting in the need for insight into these issues and methods to address them. We propose two preconditioners, one based on the Nyström method for low-rank problems and one based on Factorized Sparse Approximate Inverse (FSAI), which can effectively precondition the system. We provide details and experiments that demonstrate when and why one preconditioner may be preferred.

In Chapter 6 we consider Gaussian process hyperparameter optimization and training in its entirety. The kernels used in Gaussian processes, as discussed earlier, are prime candidates for being handled effectively by hierarchical matrices. However, many details that arise during the hyperparameter optimization and prediction phases must be handled with care for the results to be effective. As such, in this chapter we include an analysis of what numerical issues may arise during Gaussian process regression, provide examples of how these problems may be handled, as well describing a framework for how matrix-free methods can be used in Gaussian processes effectively. Furthermore, through an analysis of the issues that may arise, in combination with numerical experiments, we can demonstrate classes of problems for which hierarchical matrices prove to be an extremely effective tool.

In Part II we investigate how quantum chemistry simulations can be performed more rapidly. One can either scale up or scale out the hardware used to tackle larger problems. The modern approaches for these would be to either use fatter nodes by utilizing GPUs to increase the density of calculations or extend to more nodes. Both of these approaches require algorithmic improvements to provide the speedup desired. First, we investigate the use of GPUs in a distributed memory fashion for the computationally expensive kernels and how to reduce the amount of communication between the CPU and GPU to improve speeds. Second, if the number of nodes increases, so does the communication. As such, 3D matrix-matrix multiplications appear promising to reduce the amount of communication required while still achieving high efficiency in terms of flops. However, the distributions used for such calculations may not be favorable for general-purpose codes, as they require redistributions to be compatible with existing codes. We investigate the role of 3D matrix-matrix multiplication in Chebyshev-filtered subspace iteration density functional theory codes.

In Part III we investigate the use of sparse triple-matrix products on GPUs for algebraic multigrid. A triple matrix product RAP arises within algebraic multigrid, typically evaluated as two matrix-matrix products. However, this requires multiple passes over the data and a temporary matrix. By providing a triple matrix product, we can reduce the overhead associated with such a calculation, resulting in an overall speedup compared to traditional methods.

Part I

Hierarchical Matrices

CHAPTER 2
SMASH

2.1 Introduction

Kernel matrices, where each element in the matrix is given by a kernel evaluated between two points, require $O(n^2)$ computation and storage to form and, through dense linear algebra, take $O(n^3)$ time to solve and $O(n^2)$ to perform matrix-vector products with for n points. However, depending on the kernel and point distribution, alternative representations exist which can approximate the matrix to a given precision, allowing for rapid operations to be performed. In particular, we will look at a hierarchical matrix approximation known as SMASH (Structured Approximation by Separation and Hierarchy), which provides near-linear construction and matrix-vector products in both computation and storage time.

Hierarchical matrices are a data-sparse representation that can exploit the block low-rank properties of many kernels used in data science, machine learning, and scientific computing. By data-sparse, we mean that while there may be n^2 elements in the matrix (where there are n rows and n columns), an alternative representation can well approximate the matrix with fewer than n elements. One kernel that arises in these methods is the Gaussian kernel. The Gaussian kernel (also known as the radial basis function) is defined as

$$K(x, y) = e^{\frac{-||x-y||_2^2}{2\theta^2}},$$

for two points x and y. The Gaussian kernel is used in many applications, including support vector machines [13], can be used as the normal distribution in statistics, and is commonly used in Gaussian processes [14].

Different hierarchical matrices provide various operations, typically a fast factorization or a rapid matrix-vector product, which would be asymptotically faster than the dense equivalent. There exists a variety of hierarchical matrices, including $\mathcal{H}$ matrices [4], $\mathcal{H}^2$ matrices [5], HODLR matrices [6], and HSS matrices [7], each of which has its benefits and drawbacks. In this chapter, we will focus on the SMASH approximation, which provides near-linear matrix-vector products allowing for use in matrix-free algorithms which will enable large-scale problems to be faced [15, 16].

2.2 Review of SMASH

The construction of SMASH approximations and use of SMASH approximations are provided in detail in [15] and [16]. However, we provide a brief high-level overview here. The hierarchical representation must first be formed before its use to provide matrix-vector products.

SMASH relies on two main inputs to construct the SMASH representation and matrix-vector products with the SMASH representation. First, it requires a set of n data points X in $\mathbb{R}^d$ space. As an example, these may correspond with particles for an electrostatic calculation. Additionally, SMASH requires the ability to evaluate the kernel. This results in a kernel matrix, where a kernel K can be evaluated between two points X_i and X_j yielding a scalar $K(X_i, X_j) \to \mathbb{R}$. The space in which the dataset lies is referred to as the domain.

As mentioned previously, one of the core properties of hierarchical matrices is that they exploit the block low-rank properties present in many kernels. To achieve this, the domain is split into clusters, typically done through the recursive splitting of the domain, resulting in a tree. Each node of the tree corresponds with a cluster. In the case of SMASH, we begin with a root node corresponding to the entire domain, then perform recursive adaptive splitting based on the number of data points in each cluster and the bounding boxes that the data points lie in. Intuitively, once we have a clustering, we can observe that the kernel evaluated between pairs of points in distant clusters is likely to be less sensitive than pairs of points in nearby clusters. The idea of nearby and distant clusters is defined as nearfield and farfield clusters, where an *admissibility condition* is used to determine whether two clusters are nearfield or not.

The admissibility condition is dependent on the hierarchical representation used. SMASH, by default, uses an admissibility condition defined as

$$max(diameter(X_i), diameter(X_j)) < 0.7||(center(X_i) - center(X_j))||_2, \qquad (2.1)$$

where X_i and X_j correspond with the ith and jth clusters respectively.

Ignoring the nested basis of $\mathcal{H}^2$ matrices (which will be discussed later), we can see that the block of the kernel matrix corresponding with farfield nodes may be well approximated by a low-rank representation. In contrast, the nearby blocks should remain represented as dense matrices. The low-rank representation used is an interpolative decomposition [17], which SMASH uses a strong rank-revealing QR decomposition to calculate [18]. The interpolative decomposition allows a matrix to be approximated via a few columns or rows of the matrix and a small dense matrix. When considering data points, this corresponds with selecting a few points as representative or skeleton points. Thus, we can approximate the matrix corresponding with a cluster and its farfield by selecting a few points from the cluster using strong rank-revealing QR factorization, yielding the interpolative decomposition. In SMASH, we use two different methods for constructing the basis, the interpolation-based method and the data-driven-based approach, discussed in the next chapter. By taking the union of these skeleton points with the skeleton points of the cluster's siblings, we can repeat this processing traversing up the cluster tree, which provides the hierarchical nature. $\mathcal{H}^2$ matrices achieve a lower asymptotic computation and storage costs compared to the method just described by calculating the basis associated with a given node from its children and small transfer matrices. SMASH allows $\mathcal{H}^2$ matrices to be calculated directly, rather than re-compressing a $\mathcal{H}$ matrix.

For well-separated leaf clusters X_i and X_j, the associated farfield block can be repre-

Algorithm 1 $\mathcal{H}^2$ matrix-vector product

```
1: procedure $\mathcal{H}^2$MAT-VEC($b, U, V, B, W, R$, tree)
   Output: $y = \hat{A}b$
2:     for each leaf node $i$ do
3:         $q_i = V_i^T b_i$
4:     end for
5:     for each non-leaf node $i$ from bottom to top do
6:         $q_i = \sum_{c \in \text{children of } i} W_c^T q_c$
7:     end for
8:     for each non-leaf node $i$ do
9:         $g_i = \sum_j B_{i,j} q_j, \ \forall j \in$ interaction list of $i$
10:    end for
11:    for each non-leaf node $i$ from top to bottom do
12:        $g_c = g_c + R_c g_i$
13:    end for
14:    for each leaf node $i$ do
15:        $y_i = U_i g_i + \sum_j B_{i,j} b_j, \ \forall j \in$ nearfield of $i$
16:    end for
17: end procedure
```

sented as

$$A_{\boldsymbol{i},\boldsymbol{j}} \approx U_{\boldsymbol{i}} B_{\boldsymbol{i},\boldsymbol{j}} V_{\boldsymbol{j}}^T. \tag{2.2}$$

Where $U_{\boldsymbol{i}}$ is referred to as the column basis for node i, while $V_{\boldsymbol{j}}$ is referred to as the row basis, and $B_{\boldsymbol{i},\boldsymbol{j}}$ is known as the coupling matrix. In $\mathcal{H}$ matrices, the same definition holds for the non-leaf farfield blocks. However, in $\mathcal{H}^2$ matrices, the non-leaf farfield blocks exploit a nested bases property. This allows the associated basis matrices to be calculated from the node's children basis matrices. In the case of two children $c1, c2$ and *transfer matrices* R, W, this can be defined as:

$$U_p = \begin{bmatrix} U_{c1} R_{c1} \\ U_{c2} R_{c2} \end{bmatrix}, V_p = \begin{bmatrix} V_{c1} W_{c1} \\ V_{c2} W_{c2} \end{bmatrix}.$$

From these small matrices, we can calculate the matrix-vector product of the hierarchical approximation $\hat{A}$ with an arbitrary vector b, as seen in Alg. 1. This is discussed in more detail in Section 3.

CHAPTER 3

DATA DRIVEN SMASH

It is well known that hierarchical matrix methods can provide asymptotic speedup when compared with dense linear algebra methods. However, the cost of deriving hierarchical representations can be significant, especially when the approximation rank is much larger than the actual rank. This section will focus on $\mathcal{H}^2$ matrices, which can be constructed, stored, and applied in optimal $O(n)$ time and space enabled by the nested basis property [20, 5, 21]. A simpler hierarchical structure, associated with $\mathcal{H}$ matrices [22, 23, 5, 21], does not require nested bases and has a suboptimal cost of $O(n \log n)$ in storage and matrix-vector products. Among these two types of hierarchical matrices, it is particularly difficult to create a black-box high performance implementation of $\mathcal{H}^2$ matrices.

In practice, the construction of a hierarchical matrix is much more expensive than multiplying it by a vector. As an example, the construction cost using algebraic techniques is at least quadratic, while the cost of computing a matrix-vector product associated with the resulting hierarchical format is near linear. Interpolation-based methods provide a general, yet efficient, way to construct $\mathcal{H}^2$ matrices. They can bring the construction costs down to $O(n)$ and are used to solve a wide range of problems [24, 25]. However, one issue with interpolation-based methods is that their costs have very large prefactors. This is because the low-rank factors in the resulting hierarchical matrices have much larger rank than needed for a given approximation accuracy. The other issue with interpolation-based methods is that their costs scale exponentially with respect to the number of spatial dimensions. Thus, these methods rapidly lose their efficiency in higher dimensions. The primary motivation of the *data-driven* method proposed in this section is to achieve more efficient

© 2020 IEEE. Reprinted, with permission, from Lucas Erlandson; Difeng Cai; Yuanzhe Xi; Edmond Chow , Accelerating Parallel Hierarchical Matrix-Vector Products via Data-Driven Sampling, IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2020 [16] [19], scaling while being able to handle equally as general problems as interpolation-based methods.

A commonality among hierarchical matrix implementations is that all the low-rank factors are calculated and stored during the construction, and are later used in performing matrix-vector products. This is in contrast to the fast multipole method (FMM) where the hierarchical low-rank format is generated just in time for use, and discarded after use [1]. This, on the other hand, makes FMM less efficient when a large number of matrix-vector multiplications need to be performed, for example, in the iterative solution of linear systems. In this case, the hierarchical representation has to be computed from scratch in each iteration. In order to trade off the memory and computation involved, we take advantage of the special structure of low-rank factors produced by the SMASH algorithm [26] and propose an *on-the-fly* approach when using the hierarchical format. Since most factors produced by this algorithm are submatrices of the kernel matrix, instead of storing these factors explicitly, we only store the corresponding row and column indices. This significantly reduces the memory cost and these submatrices can be rapidly assembled in parallel whenever needed.

We propose new methods to alleviate the bottlenecks that arise in $\mathcal{H}^2$ matrices and hierarchical matrices in general. In summary:

- We introduce a new data-driven sampling method, which produces lower ranks for $\mathcal{H}^2$ matrices and achieves a speedup up to 10^4 in high dimensions when compared to the interpolation-based method;
- We discuss an on-the-fly handling of the matrix-vector products, which reduces memory consumption by an order of magnitude;
- We provide parallel numerical experiments, demonstrating the effectiveness of the above contributions.

3.1 Background

Hierarchical Matrices

There exists much literature discussing hierarchical matrices and their applications. Surveys can be found in [21, 5, 27, 28]. A variety of packages are described in [29, 30, 31]. The ideas behind hierarchical matrices can be traced back to [22, 32, 1, 23], where researchers sped up the evaluation of n-body gravitational potentials [22] or Coulomb potentials [1], and the iterative solution of boundary integral equations [32, 23]. Mathematically, the computational task boils down to computing the matrix-vector product involving a dense matrix associated with certain kernel function $\mathcal{K}(x, y)$ evaluated at a set of points $X = \{x_1, \dots, x_n\}$:

$$A = [\mathcal{K}(x_i, x_j)]_{i,j=1:n}.$$

For large problems, a straightforward calculation suffers from a prohibitive $O(n^2)$ complexity in time and space. The above mentioned methods circumvent the computational bottleneck by compressing certain blocks in the original matrix and bring the cost to be near linear. The same principle has since been generalized into the algebraic framework of hierarchical matrices, in particular $\mathcal{H}$ and $\mathcal{H}^2$ matrices. The algebraic counterparts can handle a larger class of kernel functions [5, 21] and approximate A explicitly with a hierarchically low-rank matrix $\hat{A}$. For a prescribed accuracy tolerance, storing and multiplying a hierarchical matrix by a vector has near linear scaling with the matrix dimension.

The construction of a hierarchical matrix representation for a given kernel matrix involves several steps. For a general dataset X where points may not be uniformly distributed, an adaptive partitioning of the dataset is first performed, to build up the hierarchy and identify low-rank blocks for the associated kernel matrix. That is, the dataset is divided geometrically and recursively until the number of points in each resulting subset is small enough (such that performance is optimized). Meanwhile, a tree is generated to encode the hierarchical structure of the partitioning, where each node in the tree corresponds to a subset of points in the partitioning. For example, the root node corresponds to the entire set of points, and its children correspond to subsets of points after the initial partitioning. We use X_i throughout the section to denote the set of points associated with node i. Two nodes i and j are called *well-separated* if the corresponding point sets X_i and X_j are well-separated by a certain criterion (cf. [5, 26]). The included experiments consider i and j to be well-separated if the maximum diameter of X_i and X_j is less than 0.7 times the distance between the midpoints of X_i and X_j. Any submatrix associated with well-separated clusters is assumed to be numerically low-rank and can be well-approximated by a low-rank matrix. Such a submatrix is often referred as a *farfield block*. A hierarchical matrix approximation $\hat{A}$ replaces the farfield blocks in the original matrix A by low-rank approximations, i.e.,

$$A_{i,j} \approx \hat{A}_{i,j} = U_i B_{i,j} V_j^T \tag{3.1}$$

for well-separated nodes i and j, where $A_{i,j}$ and $\hat{A}_{i,j}$ denote the submatrices of A and $\hat{A}$ respectively, associated with subsets X_i and X_j. The matrices $B_{i,j}$ connecting two basis matrices U_i and V_j are called *coupling matrices*. The above structure gives rise to $\mathcal{H}$ matrices, which have $O(n \log n)$ complexity in storage and matrix-vector products. To further reduce the complexity to $O(n)$, a more complicated structure is needed, one using the nested basis property. That is, if node p is the parent of nodes i, j, k, then the corresponding basis matrices U, V are nested in the following way:

$$U_p = \begin{bmatrix} U_i R_i \\ U_j R_j \\ U_k R_k \end{bmatrix}, \quad V_p = \begin{bmatrix} V_i W_i \\ V_j W_j \\ V_k W_k \end{bmatrix},$$

where R and W matrices are called *transfer matrices* and are of size $O(1)$. This nested basis property enables one to only store transfer matrices instead of all basis matrices explicitly, as a parent node can be constructed from its children. Hierarchical matrices with such a nested basis property are called $\mathcal{H}^2$ *matrices*. Meanwhile, when two sets of points are close to each other, the corresponding matrix block is called a *nearfield block* and is not approximated. The hierarchical partitioning of the entire set of points ensures that the nearfield blocks are only associated with leaf nodes and the submatrix that consists of all the nearfield blocks is sparse (cf. [5, 21, 33]). Therefore, a reduction in cost from $O(n^2)$ to $O(n \log n)$ or $O(n)$ is achieved by storing only the nearfield blocks and the low-rank approximations for farfield blocks. All the basis matrices, transfer matrices and nearfield blocks are called the *generators* of $\mathcal{H}^2$ matrices.

To construct $\mathcal{H}^2$ representations, one needs a way to approximate farfield blocks and simultaneously maintain the nested basis property. For FMM and its variants [32, 1, 33], expansions such as Taylor expansions or spherical harmonic expansions are used due to high accuracy and low computational complexity. The so-called kernel independent fast multipole method [34, 35] derives factorizations by solving ill-posed integral equations. One limitation of these methods is that they are only valid for special kernel functions, i.e., the fundamental solutions of certain constant coefficient partial differential equations, such as the Laplace equation, low-frequency Helmholtz equations, the Stokes equation, etc. To handle general kernel functions, a common technique that allows for black-box kernel independent implementations is polynomial interpolation. Due to the efficiency and generality of interpolation, interpolation-based hierarchical matrix methods have been used for solving many types of problems [5, 24, 36, 37, 21, 26].

Interpolation-Based Construction

Interpolation was first introduced for $\mathcal{H}^2$ matrices in [5, 24] as a replacement for Taylor expansions, as Taylor expansions require evaluation of the derivatives of the desired functions which may have numerical overflow or underflow issues (cf. [33]). Conversely, interpolation only requires evaluations of the kernel function, making it ideal for constructing hierarchical matrices for arbitrary user-defined kernel functions. Compared to algebraic techniques, interpolation is able to provide explicit formulas for all low-rank factors in the hierarchical representations and hence the total computational cost is small. We review the basic idea of interpolation-based construction below.

The use of polynomial interpolation (cf. [26]) yields the following separable approximation for $\mathcal{K}(x,y)$

$$\mathcal{K}(x,y) \approx \sum_{k=1}^{r} p_k(x)\mathcal{K}(x_k,y),$$

where x_k are interpolation points and p_k are the associated Lagrange polynomials ($k = 1,\ldots,r$). The separable approximation above automatically induces a low-rank approximation of the entire farfield block for node i:

$$\begin{aligned} A_i &:= [\mathcal{K}(x,y)]_{\substack{x\in X_i \\ y\in Y_i}} \\ &\approx \left[p_1^{(i)}(x),\cdots,p_r^{(i)}(x)\right]_{x\in X_i} \left[\mathcal{K}(x_k^{(i)},y)\right]_{\substack{k=1:r \\ y\in Y_i}}, \end{aligned} \tag{3.2}$$

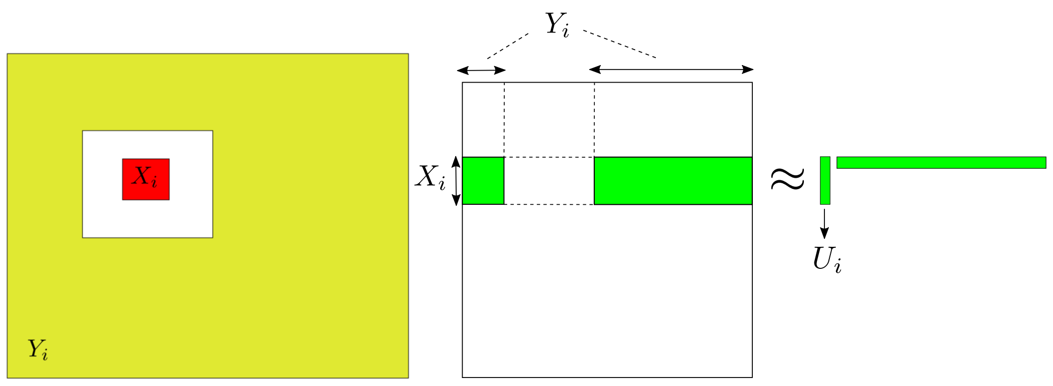
where Y_i denotes the set of *all* points that are well-separated from X_i. Thus, the column basis U_i can be chosen as

$$U_i = \left[p_1^{(i)}(x), p_2^{(i)}(x),\cdots,p_r^{(i)}(x)\right]_{x\in X_i}. \tag{3.3}$$

See Fig. 3.1 for a pictorial demonstration.

Despite its generality and computational efficiency, interpolation usually does not yield the optimal rank in the approximation. That is, the approximation rank r in (3.2) can be much larger than the optimal rank under a prescribed tolerance. This is due to the fact that interpolation does not fully exploit the information from the kernel matrix, as one can see from (3.3) that the basis U_i is independent of the kernel function $\mathcal{K}$.

A more serious limitation of interpolation is that it suffers from the curse of dimensionality. The cost of interpolation-based construction methods scales exponentially with the number of dimensions, making them a poor choice for problems involving more than a few



(a) X_i (red) and its farfield Y_i (yellow).

(b) Corresponding farfield block A_i (green) and the low-rank approximation.

Fig. 3.1: Demonstration of the farfield for node i.

dimensions. For example, in d dimensions, interpolation over a tensor grid with p points per direction yields p^d interpolation points in total, i.e., an approximation rank $r = p^d$ in (3.2). Hence, we see that interpolation-based hierarchical low-rank approximations quickly lose their efficiency in high dimensions.

3.2 Methods

In this section, we propose two novel methods for use in hierarchical matrix packages:

1. a new data-driven construction of hierarchical matrices with nested bases;

2. a memory efficient on-the-fly approach for matrix-vector products.

The data-driven approach breaks the curse of dimensionality seen by interpolation-based methods. Our experiments show that the data-driven approach yields blocks of lower rank (hence lower storage) for the same approximation error. Such a comparison can be seen in Fig. 3.2, where it is visible that the rank achieved by the data-driven method for the farfield nodes is significantly lower than the rank achieved by the interpolation based method. The on-the-fly approach further reduces the memory usage of hierarchical matrix representations by taking advantage of the special structure in coupling matrices [26]. By postponing the generation of certain matrices until they are used, the on-the-fly approach

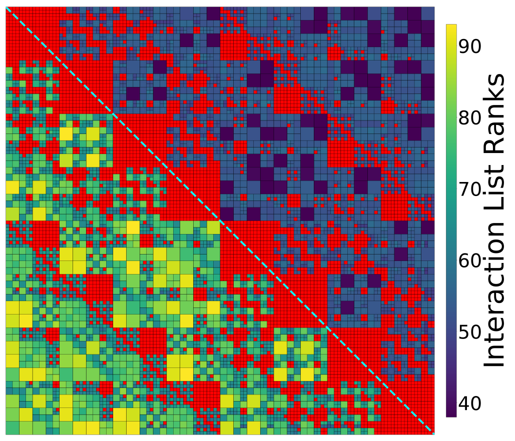


Fig. 3.2: A comparison of the rank of the bases produced by the interpolation-based (lower triangular part) method and the data-driven (upper triangular part) based method for 10,000 points randomly distributed in a cube for 1e-7 relative error for the Coulomb kernel. Red denotes nearfield interactions.

reduces memory usage, allowing larger problems to be solved.

3.2.1 Data-Driven Hierarchical Construction

Overall Idea

The data-driven $\mathcal{H}^2$ matrix approach employs a submatrix of the kernel matrix as the basis matrix for a farfield block. For example, for a farfield block A_i as shown in Fig. 3.1, the column basis in the data-driven case is $U_i = \mathcal{K}(X_i, Y_i^*)$, where Y_i^* is a small subset ($O(1)$ in size) of Y_i. Since each Y_i contains $O(n)$ points and there are $O(n)$ nodes in total, naive sampling for each Y_i leads to at least $O(n^2)$ cost for deriving all basis matrices. Therefore, it is mandatory to sample Y_i^* hierarchically so as to lower the total cost to $O(n)$. Since the data-driven approach takes into account the kernel matrix, it enjoys an improved efficiency compared to interpolation-based methods. Particularly, the advantages of the data-driven approach are more prominent for high dimensional problems.

Nyström Sampling

The Nyström method [38] is a popular approach for deriving low-rank approximations via sampling and has been widely used in machine learning. Given point sets $S,\ T$, let $K_{S,T}$ denote a matrix with entries $\mathcal{K}(s,t)$ for $s \in S,\ t \in T$. To approximate the kernel matrix $K_{X,X}$ by a low-rank factorization, the Nyström method computes a set S that is much smaller compared to the size of X, and constructs the following approximation

$$K_{X,X} \approx K_{X,S} K_{S,S}^{+} K_{S,X},$$

where $K_{S,S}^{+}$ denotes the pseudoinverse of $K_{S,S}$. Once S is selected, $K_{X,S}$ serves as a column basis for the low-rank approximation. The original Nyström method chose S to be a subset of X associated with randomly chosen indices. The choice of S significantly affects the approximation accuracy and computational efficiency of Nyström methods. Various sampling strategies have been proposed to improve the performance of the original Nyström method [38], such as leverage score based sampling [39, 40], k-means based sampling [41], anchor net based sampling [42], etc. In this section, we adopt the anchor net based sampling in [42] due to its efficiency for high dimensional problems.

Bottom-to-Top Sweep

The key to avoiding a quadratic sampling complexity is to sample the entire dataset hierarchically. This hierarchical sampling procedure starts with a bottom-to-top sweep following the partition tree. The anchor net Nyström method [42] is used to select the sample points inside each subset. We first sample over the points associated with each leaf node and then pass the samples to the parent. Since there are $O(1)$ points in each node at the leaf level, the cost associated with each leaf node is $O(1)$. Note that a parent node has $O(1)$ children and each child passes $O(1)$ samples, so the parent of each leaf node is associated with a new set of points with $O(1)$ size. Next we perform the same operation for each parent node as in the leaf level. That is, we perform sampling over the new set of points for each parent node and pass the output to the next level. The operation is repeated until we reach the root node. Since the cost associated with each node is $O(1)$, the total cost for the bottom-to-top sweep is $O(n)$. An illustration of the samples selected at the leaf level for a 2D dataset is shown in Fig. 3.3a.

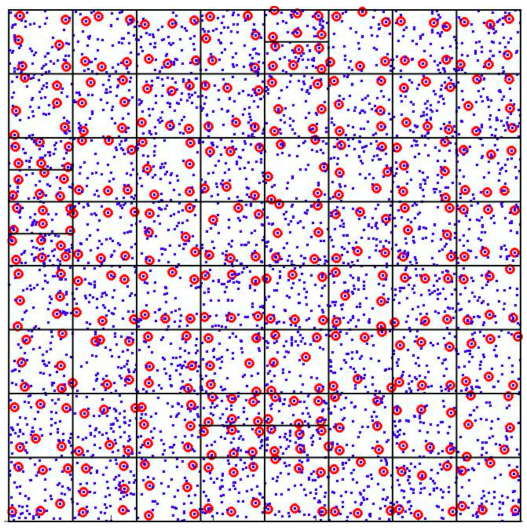
Top-to-Bottom Sweep

The top-to-bottom sweep is then performed on the samples from the farfield associated with each node. We perform sampling over each such subset and pass the output to the children nodes along the partition tree. Since computing samples at each node has $O(1)$ complexity, the total cost for this sweep is also $O(n)$. An illustration of the samples from the farfield of a block for a 2D dataset is shown in Fig. 3.3b.

Note that the sampling step is only performed on points in the original set, and is independent of the kernel function and the kernel matrix. While sampling has previously been used in hierarchical methods, to the best of our knowledge, this is the first time that sampling techniques have been used in a hierarchical way. An outline of the hierarchical sampling is shown in Alg. 2.

To summarize, the proposed data-driven method enjoys the following features:

1. allows black-box kernel independent construction of the hierarchical low-rank format;
2. provides optimal $O(n)$ complexity for the construction of nested bases, where n is the number of given points;
3. is valid for high dimensional problems (more than 3 dimensions);
4. achieves lower rank than interpolation-based methods for the same accuracy.


(a) Samples X_i^* (red circles) from all leaf nodes i.

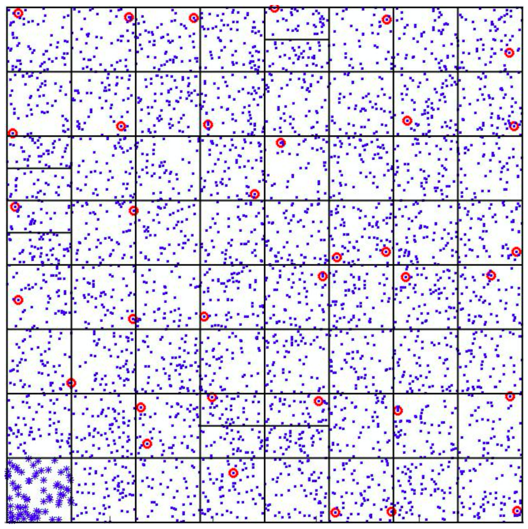

(b) Samples Y_i^* (red circles) from the farfield set Y_i for X_i (blue stars) in the bottom left corner.

Fig. 3.3: Illustration of the hierarchical sampling.

3.2.2 On-The-Fly Matrix-Vector Products

State-of-the-art methods for performing $\mathcal{H}^2$ matrix-vector products calculate the coupling matrices $B_{i,j}$ during the construction of the matrix. The $B_{i,j}$ matrices are only used to perform matrix-vector products. In the new $\mathcal{H}^2$ on-the-fly memory mode, rather than calculating the $B_{i,j}$ matrices during the construction of the matrix, they are calculated as needed in lines 9 and 15 of Alg. 3.

Existing hierarchical matrix implementations calculate and store all the generators during the construction of the matrix, which will then be (re)used later. While the memory consumption scales linearly, we observe that the majority of the memory consumption arises from the storage of the coupling matrices $B_{i,j}$. Since $B_{i,j}$ is a submatrix of the original kernel matrix, memory consumption can be significantly reduced by storing the indices instead of the whole matrix $B_{i,j}$. The use of the on-the-fly memory scheme enables problems an order of magnitude larger to be tackled compared to traditional approaches.

Algorithm 2 Data-driven hierarchical sampling

```
 1: procedure HIERARCHICAL_SAMPLE(X)
    Output: $Y_i^*$
 2:     for all $i$ do
 3:         Set $Y_i^*$ to be empty
 4:         if $i$ is a leaf node then
 5:             Set $X_i^* = X_i$, (points associated with node $i$)
 6:         else
 7:             Set $X_i^*$ to be empty
 8:         end if
 9:     end for
10:     for each node $i$ from bottom to top do
11:         Set $X_i^* = \text{Sampling}(X_i^*)$
12:         Add $X_i^*$ to $X_p^*$, the set associated with parent $p$
13:     end for
14:     for each node $i$ from top to bottom do
15:         Set $Y_i^* = \bigcup X_j^*,\ j \in$ interaction list of $i$
16:         Update $Y_i^* = \text{Sampling}(Y_i^*)$
17:         Add $Y_i^*$ to $Y_c^*$ for each child $c$ of $i$
18:     end for
19: end procedure
```

3.3 Implementation Details

In this section, we describe our shared memory parallel implementation for comparing the performance resulting from data-driven sampling vs. interpolation and on-the-fly mode vs. normal memory mode. Our description is in two major parts: the construction of the $\mathcal{H}^2$ matrix and the application of the matrix via matrix-vector products. The coarsest level of parallelism arises directly from the structure of the partition tree. During the bottom-to-top sweeps of the tree, only information from the descendants of a node is required to calculate the generators associated with that node. Thus, all nodes on the same level of the tree can be processed in parallel. Similar parallelism is found in the top-to-bottom sweeps where all nodes on a given level can be processed in parallel. Finally, certain operations require a "horizontal sweep," where there is no dependency on the ordering of the computation, and thus all nodes can be processed simultaneously.

Algorithm 3 $\mathcal{H}^2$ matrix-vector product

```
1:  procedure $\mathcal{H}^2$MAT-VEC($b, U, V, B, W, R$, tree)
    Output: $y = \hat{A}b$
2:      for each leaf node $i$ do
3:          $q_i = V_i^T b_i$
4:      end for
5:      for each non-leaf node $i$ from bottom to top do
6:          $q_i = \sum_{c \in \text{children of } i} W_c^T q_c$
7:      end for
8:      for each non-leaf node $i$ do
9:          $g_i = \sum_j B_{i,j} q_j, \ \forall j \in$ interaction list of $i$
10:     end for
11:     for each non-leaf node $i$ from top to bottom do
12:         $g_c = g_c + R_c g_i$
13:     end for
14:     for each leaf node $i$ do
15:         $y_i = U_i g_i + \sum_j B_{i,j} b_j, \ \forall j \in$ nearfield of $i$
16:     end for
17: end procedure
```

3.3.1 $\mathcal{H}^2$ Matrix Construction

Our construction phase has two parts. First, the construction of the tree and second, the construction of the matrix. The tree construction is conducted in a divide-and-conquer manner, where initially the entire set of points is considered. This set is then partitioned, where each partition can be considered independently and in parallel with others. If a given node contains more than a heuristically determined number of points, this process is recursed. During the tree construction, the parent of each node is tracked, and after the construction this information is used to determine the children associated with each node, as well as other hierarchical information such as which level each node is on. Finally, once the construction of the hierarchy information is completed, the determination of which nodes are well-separated is performed.

The determination of well-separated nodes is completed via a recursive method, which starts by considering the interaction of the root node with itself. If both nodes being considered are well-separated, they are added to each other's *interaction list*. A node's interaction list corresponds to the nodes that are in the farfield of the node, but not in the farfield of the node's parent. Otherwise, if both are leaf nodes, they are added to each other's nearfield list. If one or both have children, the process is repeated among the children.

Once the hierarchy information has been calculated, we can perform the sampling given in Alg. 2, which is independent of the kernel. Alg. 2 consists of a bottom-to-top sweep and a top-to-bottom sweep. These sweeps can be performed using the parallelization method described above, by considering all of the nodes on a level in parallel.

The construction of the basis matrices and the indices associated with coupling matrices is completed in a bottom-to-top sweep, and can be performed in parallel for all nodes at a given level. If the on-the-fly memory mode is not being utilized, the calculation of the coupling matrices is performed. This can be performed completely in parallel, by calculating the interaction between every node with the nodes in its interaction list. Note that our implementation uses a separate data structure to store the $B_{i,j}$ matrices. This is due to the fact that if the interactions between nodes are considered as a matrix, the matrix would be very sparse. Thus, our data structure consists of a sparse matrix of integers, and a sequence of dense matrices. The sparsity of the sparse matrix corresponds to the interactions between nodes, with the value of the element at (i, j) providing the linear index into a vector of dense matrices for $B_{i,j}$. Notably, this data structure is a C++ class with a matrix-free interface, and thus can be used for on-the-fly mode as well. For on-the-fly mode, rather than populating all the $B_{i,j}$ matrices, they are calculated as needed. In the symmetric case, only half of the $B_{i,j}$ matrices are required, as $B_{i,j} = (B_{j,i})^T$.

3.3.2 Matrix-Vector Product

The matrix-vector product consists of five stages, as seen in Alg. 3. First, a horizontal sweep at the leaf node is performed, during which all leaf nodes can be considered in parallel. Then, a bottom-to-top sweep is done, which can take advantage of the bottom-to-top parallelization scheme mentioned at the beginning of Section 3.3. A horizontal sweep is then performed, applying the coupling matrices associated with each node to the vector. Every application of $B_{i,j}$ can be considered in parallel. In the on-the-fly case, the matrix-vector product call to the class described above will calculate and apply $B_{i,j}$ at this point, however in the other memory modes $B_{i,j}$ is retrieved from the data structure and applied. After the horizontal sweep, a top-to-bottom sweep is performed, propagating the farfield-interactions (calculated via the interaction list) to the children. Finally, a horizontal sweep over the leaf nodes is performed, taking into account the nearfield/direct interactions.

3.3.3 Kernel Evaluation

Many of the calculations performed during the construction and application of hierarchical matrices are kernel evaluations. Thus, it is paramount to have efficient kernel evaluations. These evaluations can be accelerated by exploiting the SIMD instructions present in modern CPUs. Note that, like for direct interactions, the calculation of $B_{i,j}$ involves two clusters of points and there is an upper limit on the number of pairs of points for which the kernel evaluation will be performed. The maximum number of points per node tends to be on the order of hundreds.

3.3.4 Data-Driven Sampling

As shown in Alg. 2, data-driven sampling is performed via a bottom-to-top sweep and then a top-to-bottom sweep. In these sweeps, nodes at the same level of the tree can be processed in parallel. Note that during the sampling step, where Nyström sampling is performed by finding the points nearest to a set of lattice points, Euclidean distances between the lattice points and the considered points are calculated.

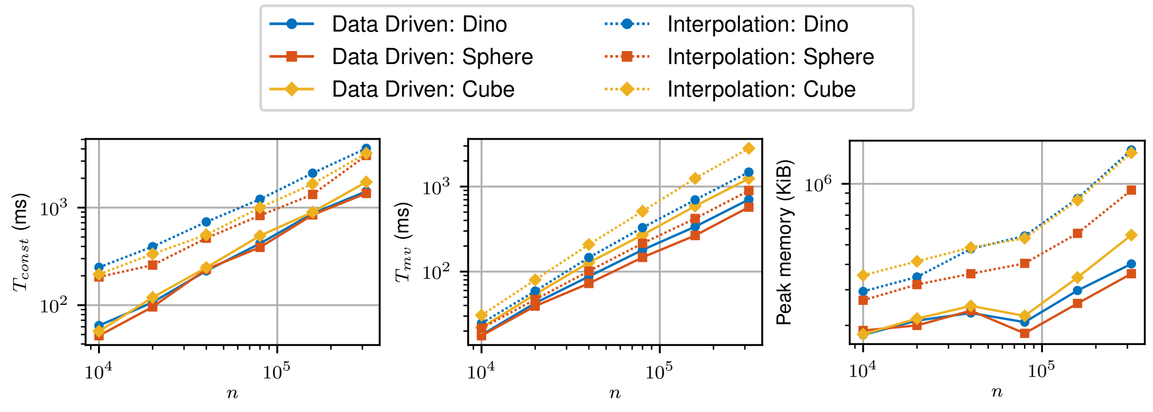
3.4 Experimental Setup

We report experimental timings for the $\mathcal{H}^2$ matrix construction and matrix-vector products. The test sets of points used of these experiments are randomly generated over the surface of a sphere (`sphere`), in the volume of a cube (`cube`), and over the surface of a dinosaur (`dino`). The dinosaur test set is a complex 3D pointcloud, which is used to demonstrate the ability for these methods to handle highly non-uniform data [26, 43]. The timings of the algorithms were measured in separate parts, T_{const}, the $\mathcal{H}^2$ matrix construction time, and T_{mv}, the time required to perform a single matrix-vector product, both in milliseconds. The construction cost only occurs once, and can be amortized over many matrix-vector products. The experiments were conducted on a single node with 128 GB of memory and two Intel Xeon E5-2680 v4 CPUs, which have a base clock speed of 2.4 GHz and 14 cores. Unless otherwise noted, experiments were performed with 14 OpenMP threads and using the Coulomb kernel $1/||x-y||_2$. The relative error is measured as $||z-\hat{z}||_2/||z||_2$, where $\hat{z}$ is composed of 12 rows sampled randomly from the $\mathcal{H}^2$ matrix-vector product, and z contains the corresponding rows in the exact matrix-vector product.

3.5 Numerical Results

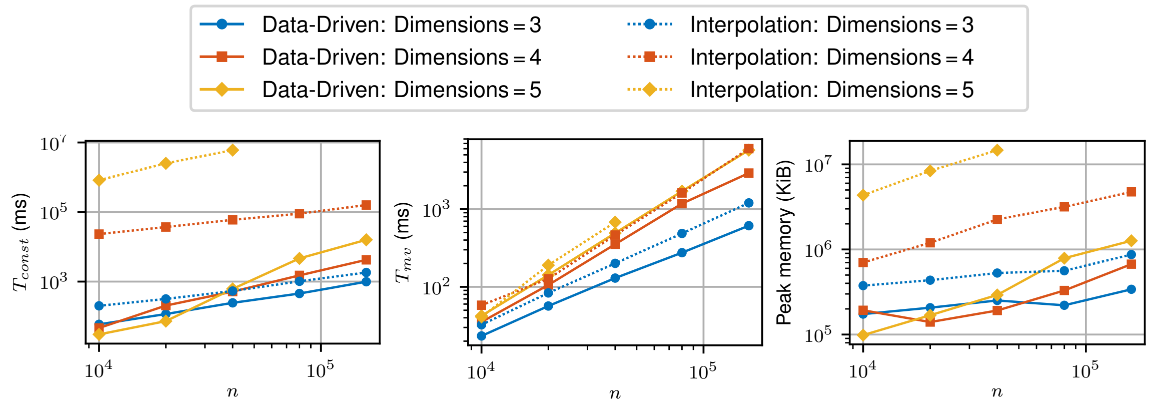
Fig. 3.4a shows that the point distribution does not have a notable impact on the construction time using on-the-fly memory mode. Fig. 3.4b shows that the asymptotic scaling remains roughly the same for the different distributions. In Fig. 3.4c, we see that the Sphere distribution requires less memory than the Cube distribution. This is due to the relative sparsity of the Sphere distribution, as the points are not uniformly distributed in the 3D domain, and there exists much empty space and fewer nearfield nodes, reducing the number of dense matrices required to be stored. The inflection point in memory usage is a result from the generally effective, but not optimally tuned, parameters of the construction method. Fig. 3.4b and Fig. 3.4c show that the data-driven method's matrix-vector products scale the same as, or better than, interpolation, and have a lower prefactor, while using less memory.

Fig. 3.5 demonstrates the scaling of the data-driven method with respect to the number of dimensions when using the on-the-fly memory mode. It is clear from Fig. 3.5a and



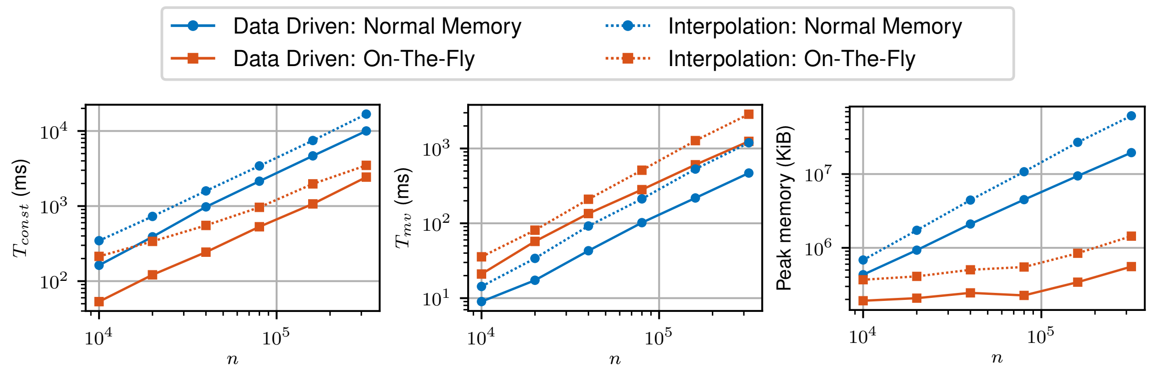
(a) Construction time (b) Matrix-vector product time (c) Peak memory usage

Fig. 3.4: Data-driven and interpolation-based methods on a variety of distributions using on-the-fly memory mode for the Coulomb kernel. The relative accuracy for all tests is around 1e-8.



(a) Construction time (b) Matrix-vector product time (c) Peak memory usage

Fig. 3.5: Data-driven and interpolation-based methods on points in increasing dimensions using the on-the-fly memory mode for the Coulomb kernel with points in the volume of a hypercube, where the relative accuracy is fixed around 1e-8.



(a) Construction time (b) Matrix-vector product time (c) Peak memory usage

Fig. 3.6: The data-driven and on-the-fly methods tested on increasing number of points for the Coulomb kernel with points in the `cube` distribution, where the relative accuracy for all tests is around 1e-8.

Fig. 3.5c that the construction and memory usage scale significantly better in the data-driven case compared to interpolation-based methods. For example, with 160,000 points, going from three to four dimensions gives a 87.05 fold increase in construction time and 5.46 fold increase in peak memory usage for the interpolation-based methods, while the data-driven method increased only 4.25, and 1.87 times, respectively. Note that due to time and memory constraints, the interpolation-based method was not tested for problems involving more than 40,000 points in five dimensions.

Fig. 3.6 details the cumulative effect of the new basis calculation via the data-driven method and the on-the-fly memory mode. We observe that the effects are cumulative, where using the data-driven method and on-the-fly memory at the same time results in the lowest memory usage and construction time. The memory scaling using on-the-fly memory is slightly better than that in the normal memory mode, as the normal memory mode scales with both the size and the number of farfield blocks while the on-the-fly memory mode scales only with the size of the blocks. As can be seen from Table 3.1, the total memory reduction is from 58.75 GiB to 543.74 MiB, for the case of 320,000 points.

Fig. 3.7 displays the scaling of on-the-fly methods with the number of OpenMP threads for 1,000,000 points. Normal memory mode was not tested, as interpolation in normal

Table 3.1: Timings and memory consumption using data driven and interpolation-based methods.

n	Basis	Memory	T_{const} (ms)	T_{mv} (ms)	Memory (KiB)
320,000	Interpolation	Normal	16789	1193	61603893
320,000	Interpolation	On-The-Fly	3488	2869	1440420
320,000	Data Driven	Normal	10011	469	19507675
320,000	Data Driven	On-The-Fly	2430	1245	556789

memory mode requires more memory for this problem size than what is available. While the scaling of the construction seen in Fig. 3.7a is sub-linear, due to the difficulty of parallelizing the upper levels of the recursive bisection, it can be seen in Fig. 3.7b that the matrix-vector products have near linear scaling in both cases. Fig. 3.7c demonstrates that the memory usage increases slightly with the number of threads, p. Each thread stores only one $B_{i,j}$ matrix at a time; thus, the concurrent memory usage is $p \cdot \text{size}(B_{i,j})$.

Fig. 3.8 shows a comparison of the data-driven and interpolation-based methods as a function of the approximation error. This demonstrates that the data-driven method with the on-the-fly memory mode, for a given relative error, requires lower construction time, memory usage, and matrix-vector time. This holds true even in the low accuracy case, where interpolation is known to be the standard choice. These results demonstrate the effectiveness of the data-driven method across a wide range of accuracy, in addition to the number of points. The performance gap becomes even larger as the accuracy increases.

Fig. 3.9 shows the generality of the new data-driven method by demonstrating the method for different kernel functions using the on-the-fly memory mode. The cubed Coulomb kernel is given by $1/||x-y||_2^3$, the exponential kernel by $exp(-||x-y||_2)$, and the Gaussian by $exp(-||x-y||_2^2/0.1)$. It can be seen that, in most cases, the plots for the different kernels are nearly indistinguishable, demonstrating the generality of the new method. With the exception of the Gaussian kernel, the scaling for the different kernels are all nearly identical.

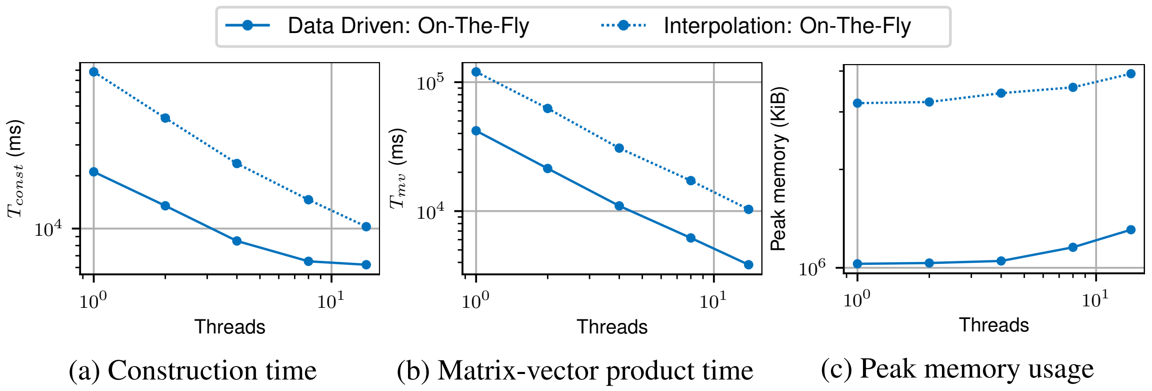


Fig. 3.7: The data-driven and interpolation-based methods vs. thread count. The on-the-fly mode was used for the Coulomb kernel with points in the `cube` distribution, where the test problem has 1,000,000 points and the relative accuracy is fixed around 1e-8.

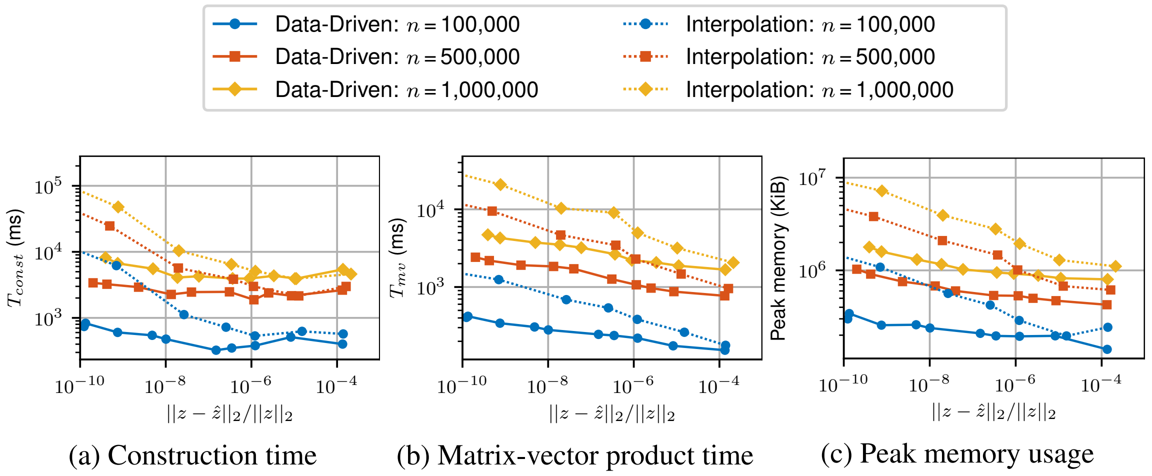
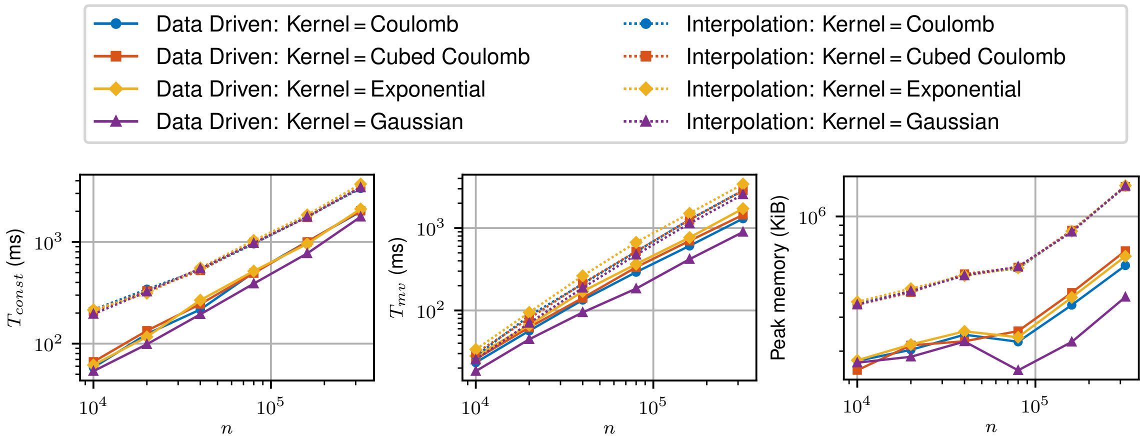


Fig. 3.8: Data-driven and interpolation-based methods using the on-the-fly memory mode as a function of accuracy for the Coulomb kernel with points in the `cube` distribution.



(a) Construction time (b) Matrix-vector product time (c) Peak memory usage

Fig. 3.9: Data-driven and interpolation-based methods for different kernel functions where the relative accuracy is fixed around 1e-8 for points in the `cube` distribution.

3.6 Discussion

3.6.1 Data-Driven Basis Construction

From Section 3.5, the benefits of the data-driven method are numerous. Compared to the interpolation-based method, the data-driven method uses much less memory, as well as reduces the time taken by the matrix-vector product and $\mathcal{H}^2$ matrix construction. The majority of the time associated with the construction of the hierarchical matrix using the data-driven method comes not from the calculation of the basis, but rather the sampling. During the matrix-vector products, the majority of the time spent is in calculating the direct or nearfield interactions. Fortunately, the hierarchical sampling is done independently of the kernel, and depends only on the points; thus, for applications where multiple kernels must be used on the same data, the cost of sampling is amortized. As seen in Fig. 3.4 and Fig. 3.9, the data-driven method is equally general as interpolation and Fig. 3.5 demonstrates that it scales significantly better with the number of dimensions. While the scaling seen is not completely independent of the number of dimensions, the scaling observed is much less severe than that seen in the interpolation-based methods.

3.6.2 On-The-Fly Memory Mode

Fig. 3.6 shows that the on-the-fly memory mode marginally increases the matrix-vector product time, but significantly decreases the $\mathcal{H}^2$ matrix construction time. This makes on-the-fly memory ideal for cases where the number of matrix-vector products for each construction is small, while the normal memory mode might be preferred in cases where many matrix-vector products are preformed for each construction.

3.7 Related Work

There exist a number of packages which, among other features, aim to extend hierarchical and FMM methods to higher dimensions. The STRUctured Matrices PACKage (STRUMPACK) [30] is a distributed memory package based on the HSS matrix format. It requires users to provide a fast matrix-vector multiplication routine in order to use randomized algorithms to perform low-rank compression. ASKIT [44] is a distributed memory package designed for performing high-dimensional kernel summations. It is based on using approximate nearest neighbor information to factorize off-diagonal blocks of kernel matrices. The Geometry-Oblivious FMM (GOFMM) distributed memory package [29] constructs an $\mathcal{H}$ matrix by sampling matrix entries without requiring any knowledge of the point coordinates or kernel functions. The main difference between the data-driven method proposed in this section and these other methods is that the sampling technique in the data-driven method does not require any evaluations or entries of the kernel $\mathcal{K}$ and is performed hierarchically in order to ensure the nested basis property for the $\mathcal{H}^2$ matrix construction.

Meanwhile, many algebraic methods have also been proposed to compress low-rank matrices. Adaptive cross approximation (ACA) [45] can provide compression algebraically using only a few entries of the matrix. However, ACA may fail for general kernel functions and complex geometries due to the heuristic nature of the method. The hybrid cross approximation improves the efficiency of ACA while achieving the convergence seen with interpolation [25]. The CUR decomposition, and the closely related interpolative decomposition, provide a decomposition of the original matrix using a subset of the rows and columns [39] [46]. While interpolative decomposition can be used efficiently in constructing nested bases once candidate bases are determined, its asymptotic complexity makes it infeasible to use to select sample points.

3.8 Conclusion

We demonstrate that bottlenecks associated with hierarchical matrices can be alleviated using our new data-driven and on-the-fly methods. We show that the data-driven method provides an equally general, but computationally more efficient way to calculate generators. Furthermore, the on-the-fly technique allows the memory savings that come with hierarchical matrices to be even more pronounced. Our implementation has near linear scaling with the number of threads for matrix-vector products with all the tested problems. Results demonstrate that both of the methods, individually and cumulatively, result in $\mathcal{H}^2$ matrices that scale linearly (as expected) with the number of points for both computation time and memory usage.

CHAPTER 4
HIGH-DIMENSIONAL HIERARCHICAL MATRICES

Hierarchical matrices are typically developed for use in low dimensions due to the curse of dimensionality seen as the number of dimensions increases. While the data-driven method seen in the previous chapter allowed the use of matrices in up to around five or six dimensions, further improvement can be obtained. This chapter discusses a few options for how hierarchical matrices can be adapted to suit high-dimensional data better and discusses results associated with each of the methods. In particular, we consider the difference between intrinsic and ambient dimensionality and investigate how PCA and a data-based partitioning can decrease the effects of dimensionality. To understand the difference between intrinsic and ambient dimensionality, consider data that lies in a plane, but the plane is located in 3D space. The data would have two intrinsic dimensions but would lie in 3D ambient dimensional space. These investigations involve the case where the intrinsic dimensionality is low, but the ambient dimensionality may be high. This is often seen in machine learning, where the belief is that the data lies on a low-dimensional manifold relative to the number of observables. Additionally, we investigate the use of alternate splitting methods during hierarchical matrix construction, and the use of alternative bases that better scale to higher dimensions. The effect of these methods is the ability for hierarchical matrices to scale to higher dimensions.

4.1 Motivation

While the dimensionality of data encountered in scientific computing in cases such as physical simulations is often low, other data-based sciences such as machine learning or quantum chemistry face high-dimensional data regularly [13, 47]. This may arise where there are many properties associated with each data point, resulting in the data lying in $\mathbb{R}^d$ for

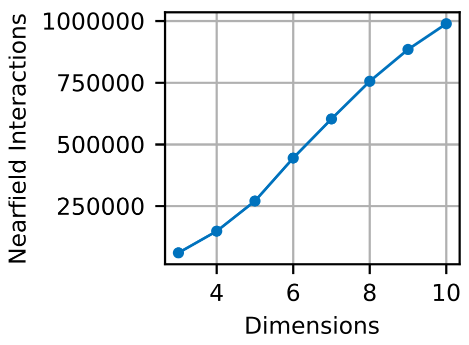
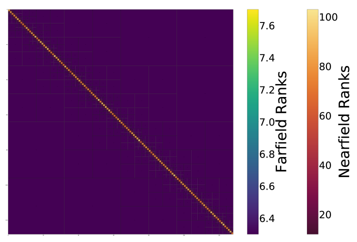


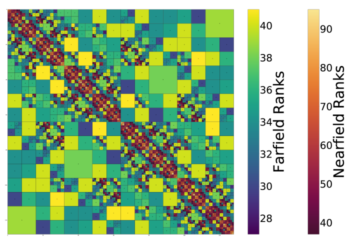
Fig. 4.1: The number of nearfield interactions when increasing the number of ambient dimensions for a 3D sphere.

$d >> 1$. Hierarchical matrices typically are not designed with this case in mind, as increasing the number of dimensions suffers from the curse of dimensionality. This is as the number of dimensions increases, the number of nearfield nodes increases. Intuitively, consider the case where the nearfield consists of the considered cluster, as well as one cluster in each direction of each dimension. This results in $2^d + 1$ nearfield blocks in d dimensions, which scales exponentially with the number of dimensions. This can be seen in Fig. 4.1, where we see that as the number of dimensions increases, the number of nearfield interactions also increases. As the nearfield interactions are calculated as dense matrices, this results in an increase in the amount of computation required, which reduces the effectiveness of hierarchical matrices.

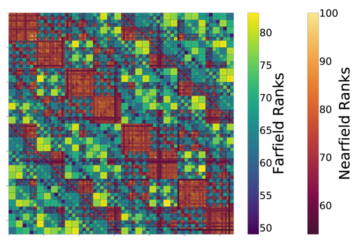
While the practical effectiveness of hierarchical matrices may be decreased in higher dimensions, applications that use high-dimensional data are often where kernel matrices can be found. Machine learning often results in the kernel matrices due to the use of the kernel trick, and their ability to be rapidly used for linear algebra. Furthermore, machine learning often requires very large matrices, and as such, when applied effectively, the asymptotic reduction seen by hierarchical matrices would significantly reduce the costs.



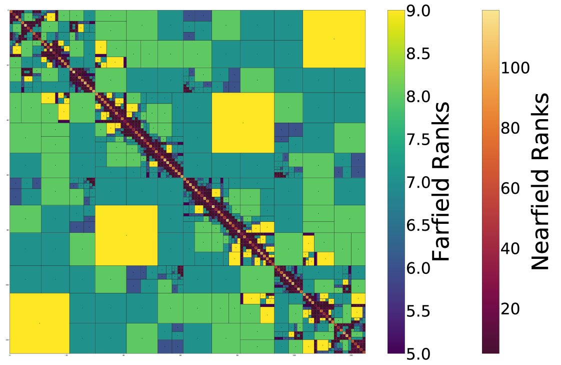
(a) Interactions corresponding with 1D random uniform points.



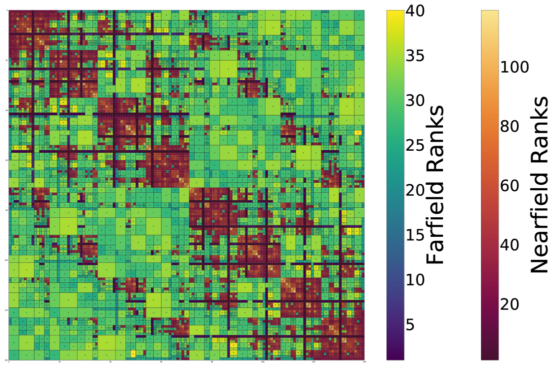
(b) Interactions corresponding with 2D random uniform points.



(c) Interactions corresponding with 3D random uniform points.



(d) Interactions corresponding with points on a 2D circle.



(e) Interactions corresponding with points on a 3D sphere.

Fig. 4.2: Comparing the interaction of uniform random points against hyperspheres. Each block corresponds with the interaction of two nodes, where the nearfield interactions are colored in the red color scheme, while the farfield interactions are colored in the green color scheme.

4.2 Methods

To develop the methods in this section, we must consider the difference between the dimension of the manifold that the data lies on compared to the ambient dimension that the data lies in. Fig. 4.2 demonstrates the difference between uniform random points and hyperspheres. In a simple case, a circle in 2D consists of a 1D manifold (intrinsic dimension) lying in 2D space. We see in Fig. 4.2, that the nearfield distribution of a circle in 2D (Fig. 4.2d) looks more similar to uniform points in 1D (Fig. 4.2a) than uniform points in 2D where the large magnitude elements are near the diagonal with no off-diagonal bands of large elements (Fig. 4.2b). Similarly, a sphere in 3D (Fig. 4.2e) looks more similar to uniform points in 2D (Fig. 4.2b), where there is a primary band around the diagonal, with a secondary pair of bands further away from the diagonal. Thus, if we instead consider their intrinsic dimension, we can reduce the effective dimensionality faced by the hierarchical representation rather than considering the ambient dimension that the points lie in. This chapter will consider two methods for performing this and utilizing an alternative sampling method. The first method considers that, rather than performing the hierarchical partitioning based on the ambient space that the data lies in, it may be more effective to partition based on the data itself. Instead of performing the hierarchical representation construction in the full ambient-dimensional space, the second method projects the data to a low-dimensional manifold. It constructs the hierarchical representation in this low-dimensional space.

Let us consider the first method, where the distribution of data points is considered during partitioning. This consists of two portions. In the first, rather than considering the spatial midpoint (or plane in the case where space is bisected, which constructs a binary tree rather than a k^d tree) as the origin for partitioning, instead uses the spatial midpoint of the bounding box encompassing the points in the node. This results in the splitting being more geometry aware and thus provides more even splittings and a better hierarchy.

In the second, after a cluster has been partitioned, rather than directly using the relative portion of the bounding box found via the splitting, the bounding box associated with the node is recalculated based on the points associated with the node. While the bounding boxes at a given level will no longer partition the total domain, the separation criteria can better determine if clusters are well-separated based upon the data itself rather than larger bounding boxes. As such, there will be fewer nearfield boxes, and thus more data can be compressed.

One downside of this method is that there is an increase in the number of farfield nodes. The data-driven-based method samples from the farfield nodes. As such, this can cause an increase in the time taken when using the data-driven-based sampling method. One alternative to the data-driven-based sampling method is using a method based on proxy points and proxy surface [48, 49]. At a high level, what the proxy surface method does, is rather than determine the representative points associated with each node based on its farfield, is to create a set of proxy surface points outside the domain associated with the node and determine the representative points based on the interaction of these points and the points of the node. Proxy surface points have the advantage that they can be calculated once for a bounding box of a given shape and size and reused at that level if the other the bounding boxes have the same shape.

We can see the impact of this combined with the data-based splitting in Fig. 4.3. We see that the time associated with the tree's construction, the most expensive portion of the construction, is significantly reduced. This figure is associated with projecting a sphere in 3D of 100k points into a higher-dimensional space, and the hierarchical clustering is performed in this higher-dimensional space. 100,000 points were used with ahead-of-time memory mode using the Coulomb kernel. This is the case of a fixed intrinsic low-dimensional manifold with a high-dimensional ambient space. We see in Fig. 4.4, that this data-based splitting results in a nearly flat number of nodes and nearfield interactions resulting in a significant decrease in the peak memory, construction cost, and matrix-vector cost. However,

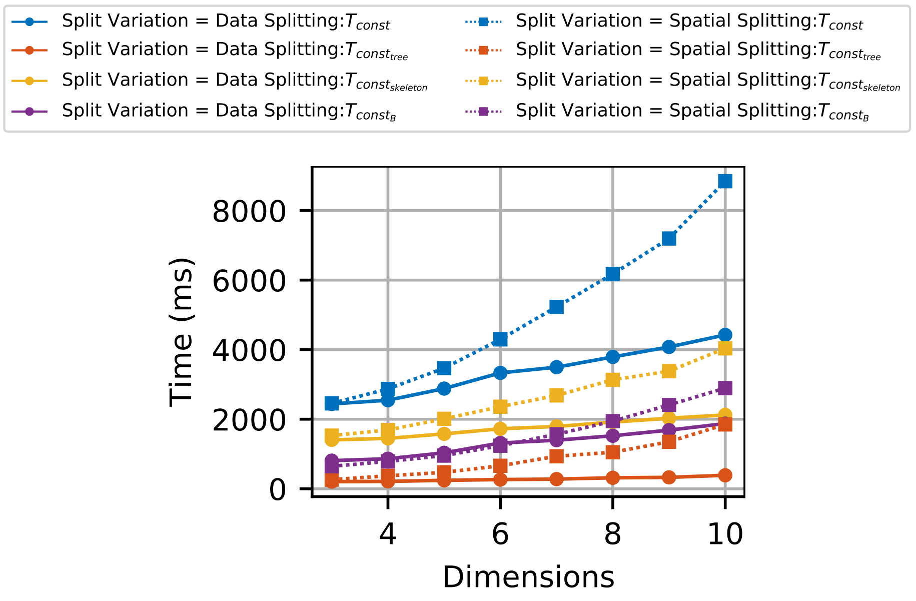
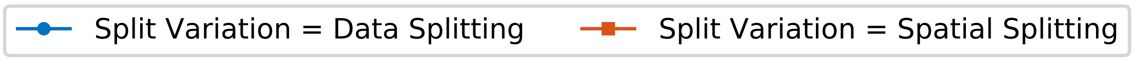
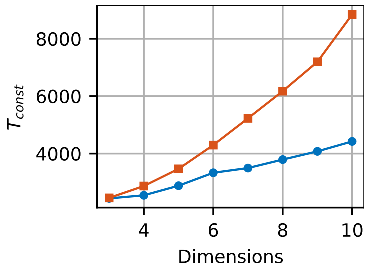
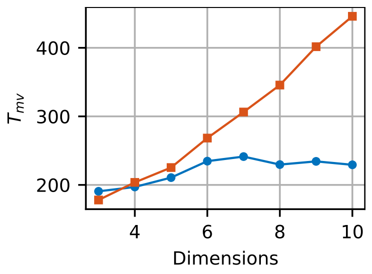


Fig. 4.3: Comparing the time taken for construction of the spatial splitting method against the data splitting with proxy surface sampling method.

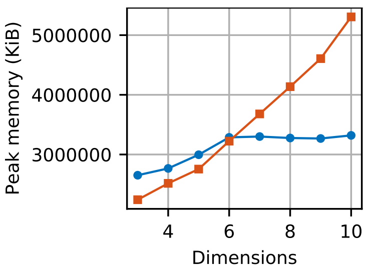




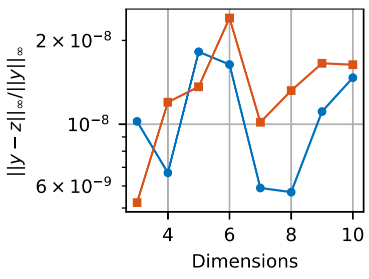
(a) The construction time (ms).



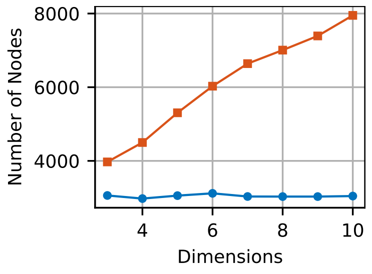
(b) The matrix-vector product time (ms).



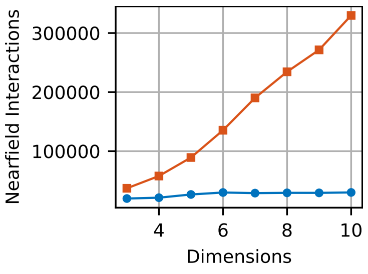
(c) The amount of memory used.



(d) Error



(e) The number of nodes in the tree.



(f) The number of nearfield interactions.

Fig. 4.4: Different metrics for the spatial splitting method and the data splitting with proxy surface sampling method.

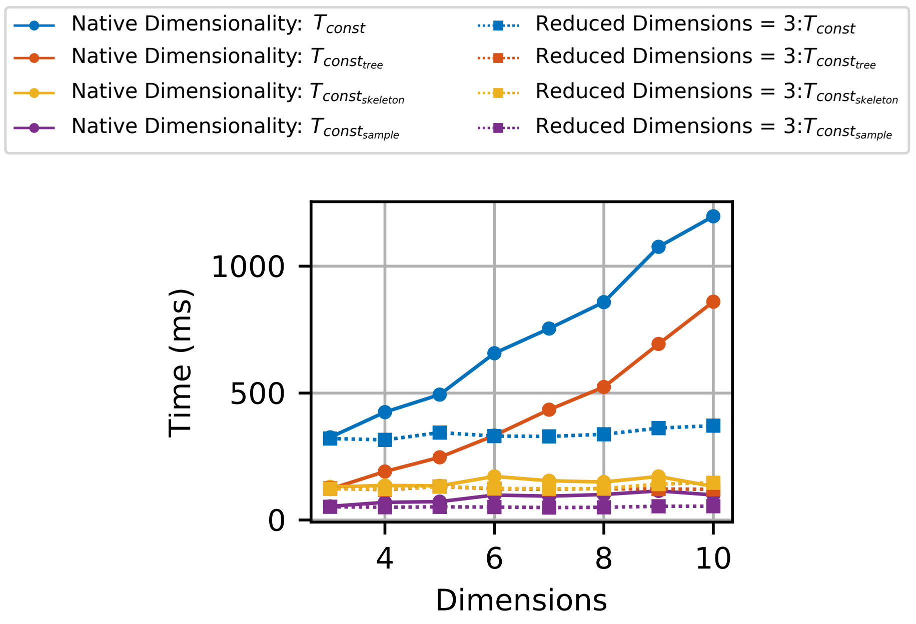
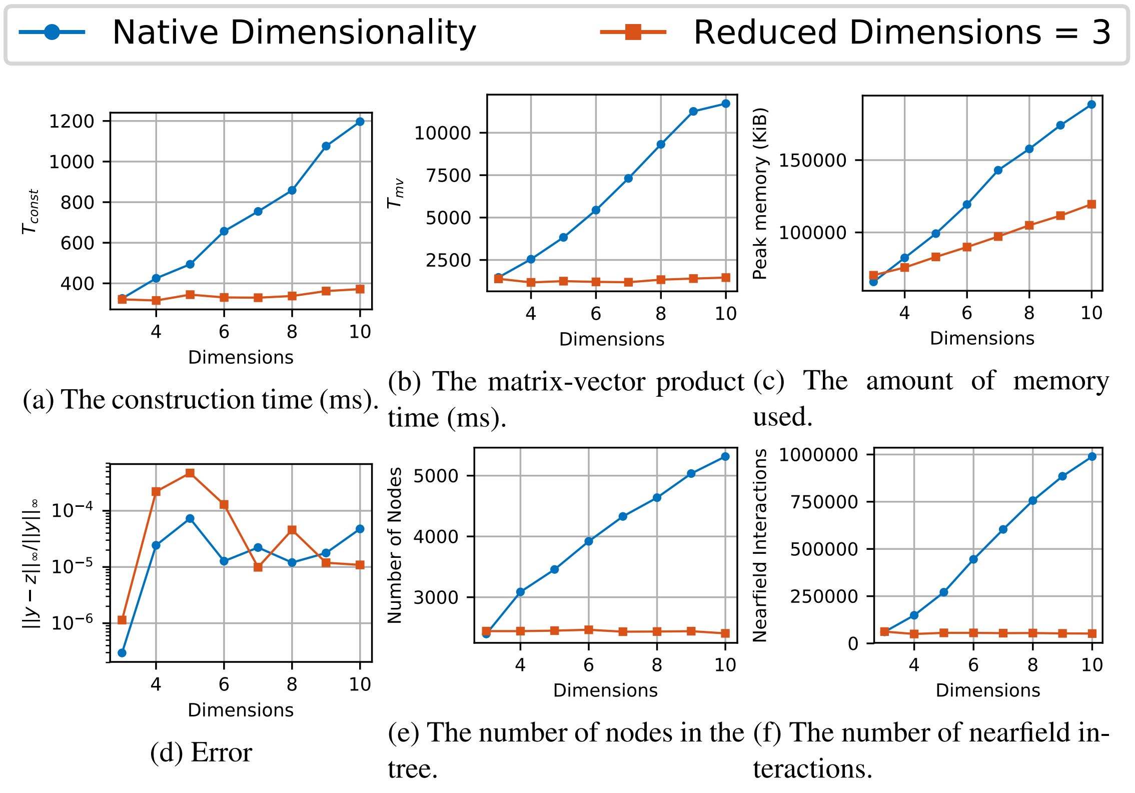


Fig. 4.5: Comparing the use of using PCA to perform hierarchical matrix operations in the intrinsic-dimensional space against using the operations in native ambient-dimensional space.



(a) The construction time (ms).

(b) The matrix-vector product time (ms).

(c) The amount of memory used.

(d) Error

(e) The number of nodes in the tree.

(f) The number of nearfield interactions.

Fig. 4.6: Comparing metrics of using the native dimensionality of the problem against using PCA to reduce the problem to lower dimensions.

no systematic increase in error is seen. Figure 4.4d demonstrates the error when comparing the SMASH matrix-vector product against the same matrix-vector product using dense matrices.

The second form of dimensionality reduction discussed is performing the hierarchical construction in low-dimensional space, rather than the ambient space. We project the data back down into 3D using principal component analysis (PCA) [50] for simplicity. PCA can be used to reduce the dimensionality of a dataset by finding a set of so-called principal components which represent the data well. PCA uses a linear mapping, while a more complex mapping such as isomap may provide a more general dimensionality reduction [51]. Once the data has been projected down into low-dimensional space, we then perform the hierarchical clustering in this low-dimensional space. Once the hierarchical clustering is performed, the resulting hierarchy tree can be used "as is" in the rest of the hierarchical matrix construction. In the data-driven case, the sampling can be performed in the lower-dimensional space to reduce the time associated with hierarchical sampling.

The impact of this dimensionality reduction can be seen in Fig. 4.5. This data corresponds with a sphere in 3D, projected up into ND space, and projected back down to 3D using PCA. 100,000 points were used with on-the-fly memory using the data-driven based sampling with the Coulomb kernel with a binary tree. Observe that the time associated with the tree construction is greatly reduced by performing the dimensionality reduction. As the calculations are performed in 3D for all cases, we see that the timing does not increase noticeably with the number of dimensions. In Fig. 4.6 we see that the number of nearfield interactions and nodes stays nearly constant with the number of dimensions. As such, the peak memory usage, construction time, and matrix-vector product time all stay nearly constant with the number of points. The increase in construction time can be attributed to the time taken for PCA to be performed.

We can see that both of these methods can perform very well for prototype problems in reducing the costs associated with doing calculations with low intrinsic-dimensional data lying in high-dimensional space.

4.3 Future Work

There exists considerably more work that can be done in this vein. One method for further reducing the costs is to use alternative methods for calculating the basis functions associated with each farfield block. In addition to the proxy-surface-based method above, sparse-grids are another alternative for selecting representative points [52, 53]. Furthermore, generalizing the dimensionality reduction method used from PCA to one such as isomap would allow the method to be used for a broader range of points and data [51].

CHAPTER 5
PRECONDITIONED GAUSSIAN PROCESSES

Gaussian process regression is a form of model-based machine learning used for its predictive power and uncertainty quantification. However, performing Gaussian process regression for large problems is intractable as the solution of a dense linear system is required, which takes $O(n^3)$ computation time and $O(n^2)$ storage complexity. This chapter describes methods for preconditioning the linear systems that arise from Gaussian process regression. We investigate a case study, provide details on what particular challenges are faced when preconditioning the linear systems that arise from Gaussian process regression, and provide matrix-free, parallel preconditioners which empirically work well on the problem.

5.1 Background on Gaussian Processes, Kernel Matrices, and Preconditioners

5.1.1 Gaussian Processes

[14] provides an introduction to Gaussian processes for Machine Learning. We will briefly review the use of Gaussian process regression for noisy observations here.

Gaussian process regression provides a method for predicting the response of a function $m(X_*)$ and its associated variance $s^2(X_*)$ given a set of observations X and their responses y (with Gaussian noise with variance σ^2) with a known kernel K with Gaussian with variance for a set of points X_*. As an example, if we have a 3D surface, we may want to be able to estimate the z coordinate given a pair of (x, y) coordinates. That is we want a way to estimate the function f which maps $f(x, y) \rightarrow z$. In this case, we have a set of measurements at locations X, with measured height $\boldsymbol{y}$. Then, the mean estimate for unknown points X_* can be calculated as $m(X_*)$, where the variance is given by $s^2(X_*)$.

We then have a joint distribution for training outputs $\boldsymbol{y}$ and test outputs $m(X_*)$ of

$$\begin{bmatrix} \boldsymbol{y} \\ m(X_*) \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X,X)+\sigma^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \tag{5.1}$$

which in turn leads to the predictive mean $m(X_*)$ and variance $s^2(X_*)$ of

$$m(X_*) = K(X_*,X)(K(X,X)+\sigma^2 I)^{-1}y \tag{5.2}$$

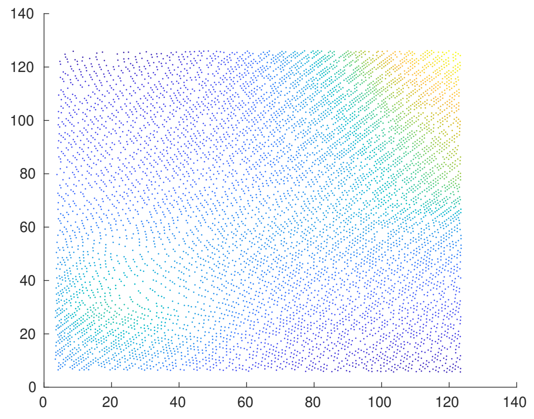
$$s(X_*)^2 = K(X_*,X_*) - K(X_*,X)((K(X,X)+\sigma^2 I)^{-1})K(X,X_*) \tag{5.3}$$

We observe that, provided the kernel function and the noise σ, computing the expected value for a given input X_* requires matrix products involving the kernel matrix and the solution of a system of equations arising from a diagonal perturbation to the kernel matrix. When solved using a Cholesky factorization, this linear solve takes $O(n^3)$ time if n is the number of observations, making this direct Gaussian process infeasible for problems greater than a few tens of thousands even on modern clusters. [54]

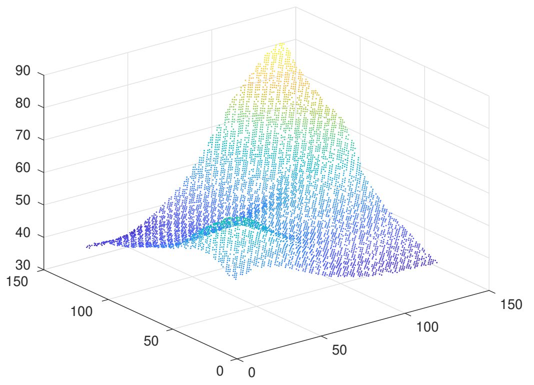
5.1.2 Preconditioners

Iterative methods provide an alternative method for solving a system of linear equations compared with direct methods such as the Cholesky-based method mentioned above. While direct methods require a known number of steps to achieve (mathematically) exact results, iterative methods start with an approximate solution which is gradually improved until it reaches the desired accuracy. Iterative methods are particularly useful in the cases where full accuracy is not required, when the system is sparse, and depending on the method used, can have great potential for parallelizability. In our case, we will use Krylov-Subspace-based methods.

Preconditioning is a critical component of iterative methods for the solution of linear systems. Without loss of generality, let us consider left preconditioning. Rather than solv-



(a) A view from above of the case study dataset.



(b) A 3D view of the case study dataset.

Fig. 5.1: A visualization of the points used in the case study.

ing the equation $Ax = y$, we instead solve the equation $MAx = My$, which in the extreme case where $M = A^{-1}$, we see that this results in $x = My$, requiring only a single matrix-vector product. The effectiveness of preconditioners depends on the underlying system of equations being solved. From a practical standpoint, an ideal preconditioner should be quick to construct, result in a system that is easy to solve and can be applied quickly. The preconditioner may be applied as an operator rather than calculated explicitly depending on the preconditioner.

5.1.3 Case Study

For demonstrative purposes, while constructing preconditioners, we will consider a case study using realistic a realistic data and scenario.The dataset used has 9635 points, where each data point consists of the z (altitude) associated with a given location on a 3D surface obtained via a laser scan. As the scan is not exact, the data is considered noisy. The dataset used can be seen in Fig. 5.1. Part of Gaussian process regression is finding the hyperparameters that best model the system. This process is known as hyperparameter optimization or training. The hyperparameters σ, θ obtained via hyperparameter optimization for this problem are $\theta \approx 11.65$ and $\sigma \approx 0.018$. σ is related to the system's noise, while θ is known as the length scale and indicates how long-range the interactions are. The kernel evaluated

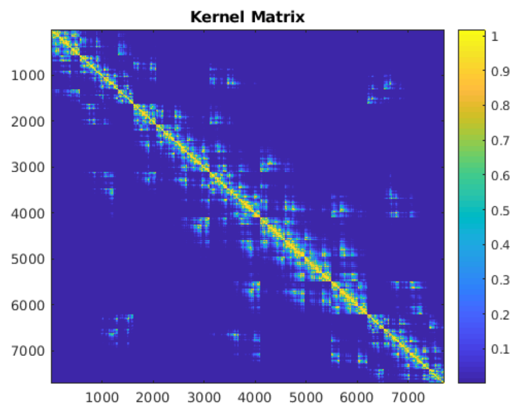


Fig. 5.2: A visualization of the kernel matrix resulting from the case study.

for this matrix can be seen in Fig. 5.2.

5.2 Preconditioning Gaussian Processes

As mentioned above, the system of equations we wish to solve takes the form $A + \sigma I$, where A is mathematically symmetric positive definite. We will be considering the use of the Gaussian kernel, which for two points x and y is given by,

$$K(x, y) = e^{-\frac{||x-y||_2^2}{\theta^2}},$$

a popular choice for Gaussian processes [14].

Notice that as θ approaches infinity, the matrix approaches a matrix of all ones, a rank-one matrix (plus a diagonal for the noise). Conversely, as θ approaches 0, the matrix approaches the identity. Thus, these cases are relatively easy to solve, but the intermediate values of θ which can be difficult to solve (ignoring the rank deficiency). This can be seen in Fig. 5.3, where the system of equations from the case study is solved for varying θ values (with a σ of 0.018). This solution is performed via the Preconditioned Conjugate Gradient [55] method, terminating when $\frac{||b-Ax||}{||b||} < 1e-6$.

We note that in practical applications, the hyperparameters (θ and σ) are rarely known ahead of time. Instead, an optimization step is performed during which the "optimal"

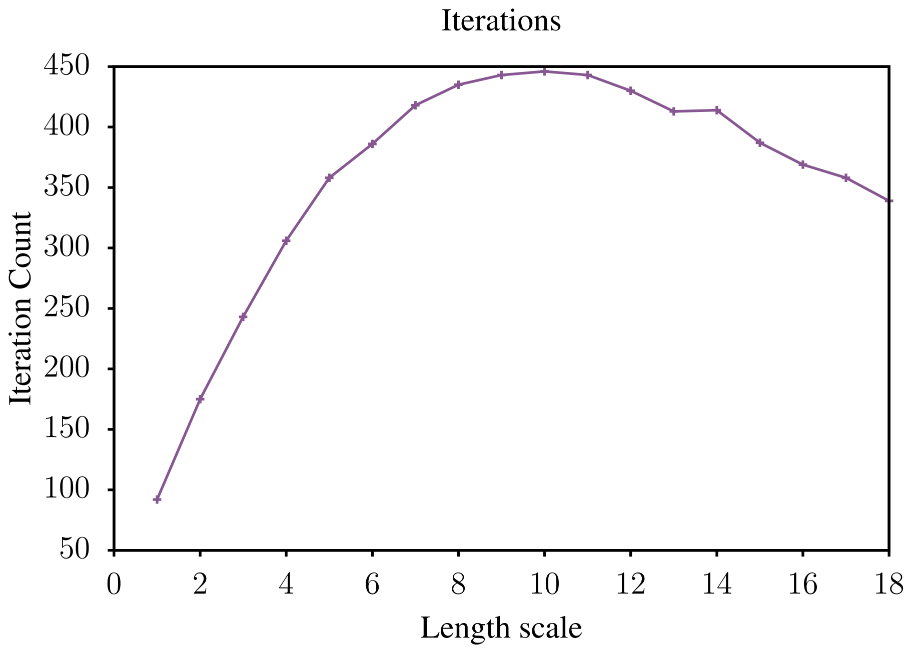


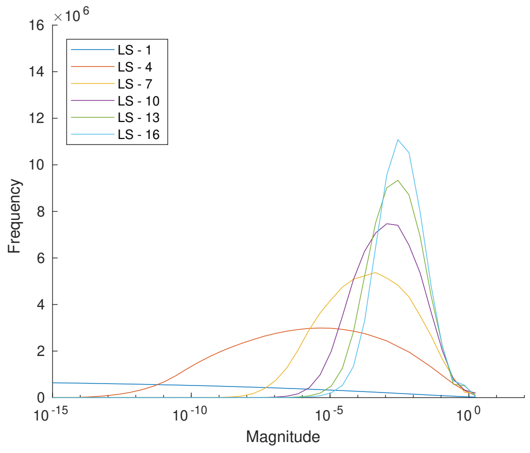
Fig. 5.3: Demonstrating the number of iterations required for a solution when varying the length scale.

parameters are determined (generally using L-BFGS-B). While this is discussed in more detail in Chapter 6, we note it here to demonstrate that an ideal preconditioner should perform well across a wide range of hyperparameters. We can use the hyperparameters within a step, as when we evaluate the log-likelihood during optimization, we are doing so for a given set of hyperparameters.

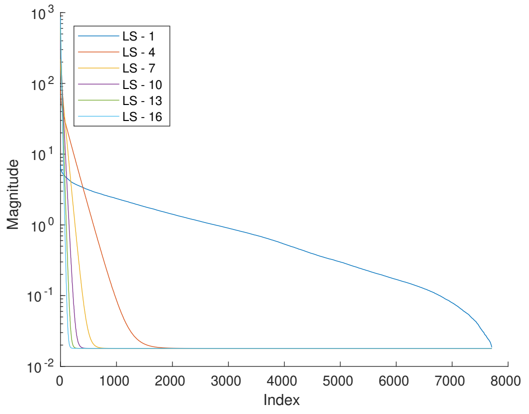
In Fig. 5.2 we can see that the matrix can be reordered such that entries with the most significant values lie close to the diagonal, while entries further from the diagonal tend to have smaller magnitudes. In this case, the matrix was reordered based upon the ordering resulting from the SMASH matrix construction.

5.2.1 FSAI

This section investigates a method based on the Factorized Sparse Approximate Inverse (FSAI) method for preconditioning [56]. Fig. 5.4a presents histograms of the magnitudes of the entries of the inverse of the kernel matrix. When the length scale is small, we can



(a) Histograms of the magnitude of the entries of the inverse of A.



(b) Spectrums of the kernel matrix for varying length scales.

Fig. 5.4: Plots of the spectrums and histograms of the values of A and A^{-1}.

observe that there are fewer entries with large magnitudes. As such, approximate inverse methods are a promising family of methods to use as preconditioners.

In particular, we use the FSAI method introduced by [56] to approximate the Cholesky factor of the inverse of the matrix A:

$$I \approx GAG^T \iff GG^T \approx A^{-1}.$$

This allows us to use such an approximation as a preconditioner. The original method constrains the approximate inverse to a given sparsity pattern, while more recent modifications have developed dynamic approaches which aim to provide adaptive sparsity patterns [57, 58, 59]. The static FSAI was previously used by [60] to precondition Gaussian sampling.

We can see in Fig. 5.5 the impact of using an FSAI-based preconditioner. In particular, the sparsity pattern is chosen such that the points within a distance of 8 of each other are included in the sparsity structure. That is to say,

$$S_{i,j} = \begin{cases} 1 & ||X_i - X_j||_2 < 8 \\ 0 & \end{cases}$$

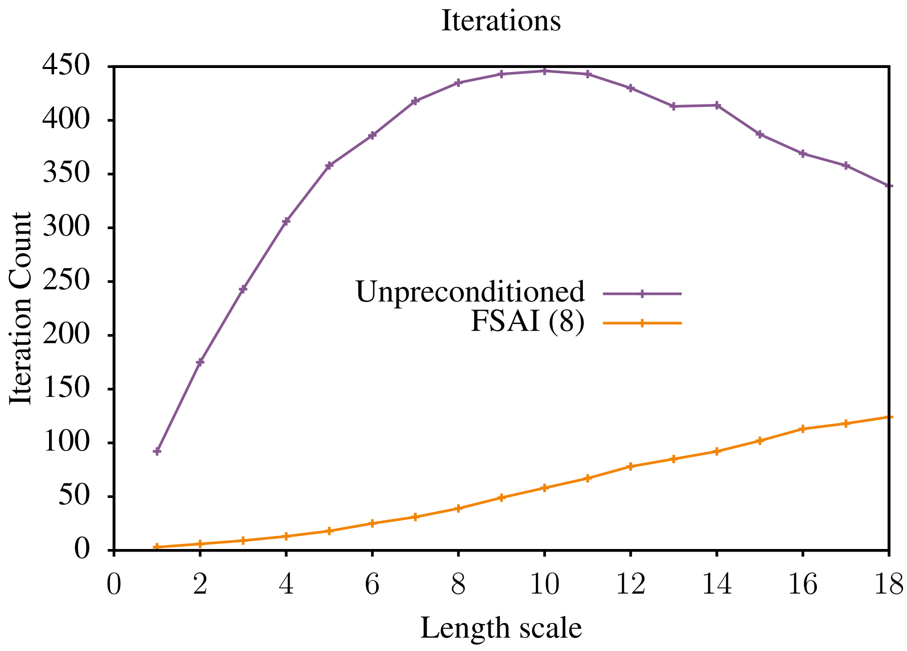


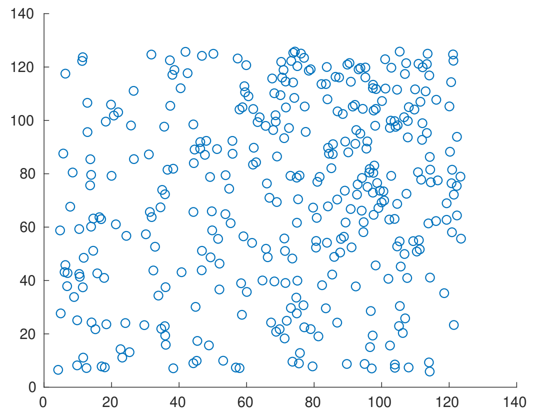
Fig. 5.5: Demonstrating the number of iterations required for a solution when varying the length scale when preconditioned with FSAI.

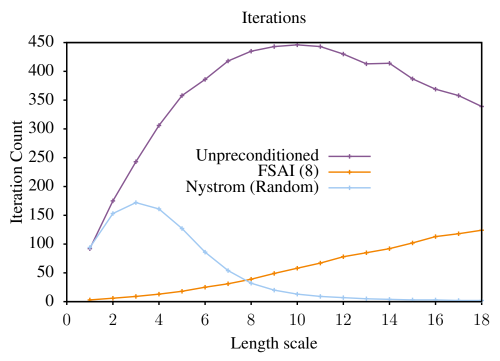
for a sparsity pattern S with points X.

We observe that using the FSAI-based preconditioner greatly reduces the number of iterations required for a large range of length scales, particularly those corresponding with small length scales. As indicated previously, this correlates with the cases where there are fewer significant entries in the inverse matrix.

5.2.2 Nyström

Recall from before that as θ increases to infinity, the matrix approaches a rank-one matrix plus a diagonal. If we look at Fig. 5.4b, we can see that the dropoff of the eigenspectrum becomes more severe as θ increases, even for values in the range of $O(1)$. This indicates that a preconditioner that approximates the matrix as a low-rank matrix plus a diagonal perturbation may be effective. The Nyström method was used by [61] to approximate $\tilde{K} \approx K(X, X)$ by selecting a small subset of $m << n$ points X_m from X. This results in


(a) Randomly sampled points from the original dataset.



(b) Comparison of Nyström-based preconditioning with random points against other methods for a variety of length scales.

Fig. 5.6: Demonstrating random point selection for Nyström-based preconditioning.

the approximation of

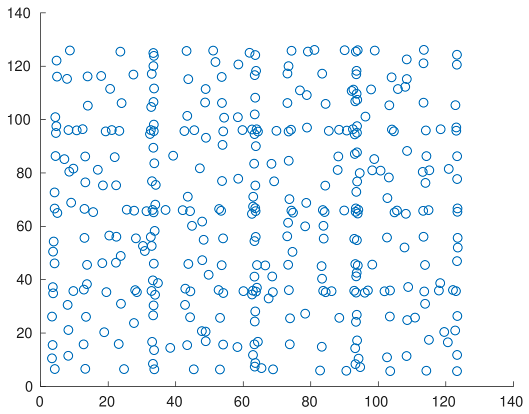
$$K(X,X) \approx \tilde{K} = K(X,X_m)K(X_m,X_m)^{-1}K(X_m,X) = \tilde{U}\tilde{\Lambda}\tilde{U}^T. \tag{5.4}$$

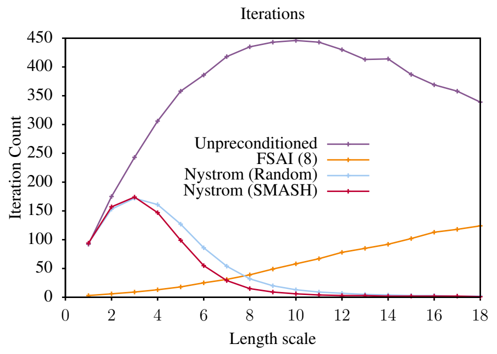
The diagonal perturbation with the Sherman-Morrison-Woodbury Identity [62] yields the following approximate solution

$$x = \frac{1}{\sigma}(b - \tilde{U}(\sigma I + \tilde{\Lambda}\tilde{U}^T\tilde{U})^{-1}\tilde{\Lambda}\tilde{U}^T b).$$

This approximation was used in [63] to accelerate solves via preconditioned conjugate gradient, with the m points being randomly selected. An example of uniformly randomly selected points from the case study can be seen in Fig. 5.6a, while the results of such a preconditioner can be seen in Fig. 5.6b. We observe that in contrast to the FSAI preconditioner, this Nyström preconditioner requires fewer iterations the larger the length scale is.

We will now discuss the impact of different point selections on the effectiveness of the preconditioner.


(a) Hierarchically sampled points from the original dataset.



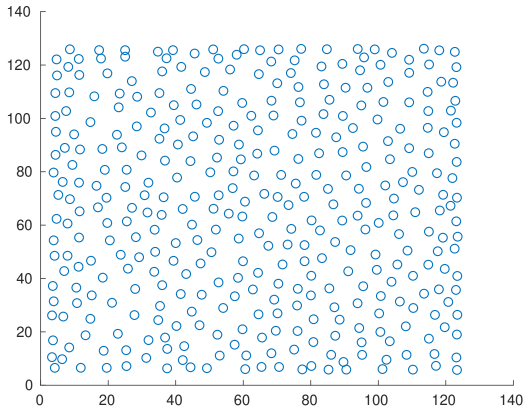
(b) Comparison of Nyström-based preconditioning with hierarchically sampled points against other methods for various length scales.

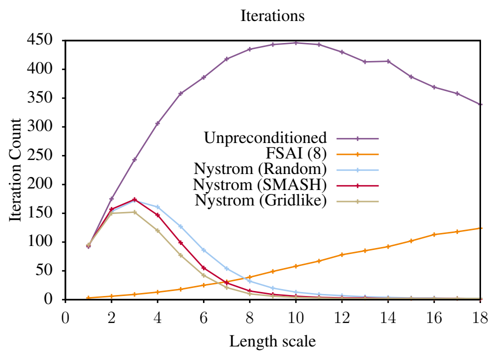
Fig. 5.7: Demonstrating hierarchical point selection for Nyström-based preconditioning.

Point Selection

While the selection of random points requires little overhead and provides very effective results, it may be possible that intentionally sampling the points could provide a better low-rank approximation – and thus a better preconditioner, without additional computation costs. As mentioned in the previous sections, SMASH is a method for constructing and applying hierarchical matrices, including the Gaussian kernel we are currently considering. During the construction of a SMASH matrix, points are selected to be part of the skeleton, which are a good set of points for the SMASH representation. As these points have already been calculated, we can use them as the sample points without any additional calculation compared to the randomly selected points. An example of the selected sample points can be seen in Fig. 5.7a, while the resulting iteration counts can be seen in Fig. 5.7b. We see that we can get lower iteration counts with no additional computation.

Furthermore, it is necessary to find points that lie near-uniform grid points during the data-driven-based SMASH construction. Intuitively, by having the selected points be uniformly distributed over the dataset, the low-rank approximation should be a good representation of the original dataset. Thus, the preconditioner would be a suitable preconditioner. We can see in Fig. 5.8b that such a point selection (seen in Fig. 5.8a) further decreases the


(a) Grid sampled points from the original dataset.



(b) Comparison of Nyström-based preconditioning with grid sampled points against other methods for various length scales.

Fig. 5.8: Demonstrating grid point selection for Nyström-based preconditioning.

number of iterations required.

CHAPTER 6
LARGE-SCALE GAUSSIAN PROCESSES VIA HIERARCHICAL MATRICES

As discussed in the Chapter 5, Gaussian processes are used for their powerful prediction abilities. However, they are computationally expensive for large problems. Within the hyperparameter optimization of Gaussian processes and prediction with Gaussian processes, the computational bottleneck is the $O(n^3)$ solution of linear systems. This chapter develops a framework for large-scale Gaussian process hyperparameter optimization and prediction using hierarchical matrices, which can achieve $O(n \log n)$ hyperparameter optimization and prediction, as demonstrated in Section 6.4.

We begin with a description of why such a matrix-free framework would be desired in Section 6.1, and the notation used for this chapter. In Section 6.2 we build up the framework for prediction (Section 6.2.1) and tuning (Section 6.2.2). Following this, we discuss the considerations that are required for a practical implementation (Section 6.3), including the numerical issues that arise from the rank deficiency (Section 6.3.1), preconditioning (Section 6.3.2), and other implementation details (Section 6.3.3).

Finally, we provide numerical experiments in Section 6.4, which demonstrate our method's accuracy, explore the impact of length scale, and demonstrate the effectiveness of this framework over numerous simulations.

6.1 Motivation and Notation

The hyperparameter optimization and prediction of Gaussian processes involve solving with the correlation matrix R in many operations. As such, it becomes infeasible to solve very large problems using direct methods, such as Cholesky decomposition. Iterative methods are a set of methods that provide an approximate solution that is iteratively improved. *Krylov subspace* methods are a class of iterative methods that can be performed in a matrix-free manner, which for solving $Ax = b$ requires only the ability to perform matrix-vector products with A. For certain problems, iterative methods can achieve asymptotic improvement, with an ideal Krylov subspace method taking $O(1)$ iterations with $O(1)$ matrix-vector products per iteration, and thus, if the matrix-vector product takes $O(n)$ time, a solution could be calculated in $O(n)$ time, rather than $O(n^3)$ via direct methods. As discussed in Chapter 5, preconditioning plays an important part in the solution of linear systems. In turn, the solution of linear systems plays an important part in prediction, especially in hyperparameter optimization. The role of hyperparameter optimization is emphasized because, during optimization, many hyperparameter combinations are tested, which may include both very smooth and very non-smooth kernels. This can result in numerical instability, which we address in Section 6.3. smashGP is a framework that provides the ability for full Gaussian process regression, hyperparameter optimization, and prediction to be performed using matrix-free methods with preconditioners designed for Gaussian processes.

The notation used in this chapter is distinct from that used in Chapter 5, as this chapter delves deeper into the statistical underpinnings of Gaussian processes without the assumption of a zero mean. In this chapter, we have observations (data points) $\tilde{Y}_i = Y(s_i) + \epsilon_i$, for $i = 1, \ldots, n$ where $\epsilon_i \sim N(0, \tau^2)$, with noiseless observations Y at locations s_i. Y follows a n-variate Gaussian distribution $Y \sim N(\mu, \Sigma)$, with constant mean μ, without loss of generality. This leads to a noiseless correlation matrix of $R = \mathcal{R}(s_i, s_j)$ for $i, j = 1 : n$, and $\Sigma = \sigma^2 R$ where σ^2 is the process's variance, and $\mathcal{R}$ is the kernel function being used – in this case the Gaussian kernel with length scale θ. $r(s)$ is the kernel evaluated between a sample point s and the training points.

Optimizing the hyperparameters of a Gaussian process means ascertaining optimal values for the hyperparameters, given the observations $\tilde{Y}$. This is often done using L-BFGS-B by minimizing the likelihood given by

$$L(\tilde{Y}) = \frac{1}{(2\pi)^{(n/2)}|\tilde{\Sigma}|^{(1/2)}} \exp\left(-\frac{1}{2}(\tilde{Y} - \mu)^\top \tilde{\Sigma}^{-1}(\tilde{Y} - \mu)\right). \tag{6.1}$$

The number of parameters is often reduced by defining $v = \sigma^2 + \tau^2$, the total variance, and $\alpha = \sigma^2/(\sigma^2 + \tau^2)$, the proportion of the variance explained by $Y(s)$. This allows us to rewrite $\tilde{\Sigma} = vR_\alpha$ with $R_\alpha = \alpha R + (1-\alpha)I$. In turn, this gives us $\hat{\mu}$ and $\hat{v}$ defined as

$$\hat{\mu} = \frac{\mathbb{1}_n^\top R_\alpha^{-1}\tilde{Y}}{\mathbb{1}_n^\top R_\alpha^{-1}\mathbb{1}_n} \qquad \hat{v} = \frac{1}{n}(\tilde{Y} - \hat{\mu}\mathbb{1}_n)^\top R_\alpha^{-1}(\tilde{Y} - \hat{\mu}\mathbb{1}_n). \tag{6.2}$$

Thus, the log-likelihood can be calculated as

$$-2\log L(\hat{\mu}, \hat{v}, \alpha, \theta; \tilde{Y}) = n\log(2\pi) + n\log\hat{v} + \log|R_\alpha| + n, \tag{6.3}$$

with derivative

$$-2\frac{\partial L(\hat{\mu}, \hat{v}, \alpha, \theta; \tilde{Y})}{\partial \phi_k} = -(\tilde{Y} - \hat{\mu}\mathbb{1}_n)^\top R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \phi_k}R_\alpha^{-1}(\tilde{Y} - \hat{\mu}\mathbb{1}_n)/\hat{v} + \text{tr}\left(R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \phi_k}\right), \tag{6.4}$$

where $\phi = (\alpha, \theta)^\top$.

We also have the noisy mean prediction given by

$$m(s) = \hat{\mu} + \hat{\alpha}r(s)^\top R_\alpha^{-1}(\tilde{Y} - \hat{\mu}\mathbb{1}_n), \tag{6.5}$$

and variance given by

$$s^2(s) = \hat{\sigma}^2(1 - \hat{\alpha}r(s)^\top R_\alpha^{-1}r(s)) + (1 - \hat{\alpha}r(s)^\top R_\alpha^{-1}\mathbb{1}_n)^2/(\mathbb{1}_n^\top(\hat{v}R_\alpha)^{-1}\mathbb{1}_n). \tag{6.6}$$

.

6.2 Matrix-Free Gaussian Process Framework

To perform Gaussian process prediction Eq. 6.2, 6.5, and 6.6 are used, while for optimization Eq. 6.3 and 6.4 must also be used. As such, a matrix-free Gaussian process framework must be able to calculate these values in a matrix-free manner. First, we build up a framework for matrix-free prediction, given a set of hyperparameters, using matrix-free linear solvers and matrix-vector products. Second, we provide a similar framework for calculating the optimal hyperparameters associated with a given dataset. This involves providing a matrix-free log-likelihood and derivative calculations, which rely upon log determinants traces, and matrix-vector products. We will first discuss the methods which can be used for matrix-free prediction. Then we will discuss the methods used in matrix-free optimization, followed by an analysis of the difficulties encountered by them.

6.2.1 Prediction

The predictive mean and the predictive variance are defined in Eq. 6.5, and 6.6, which rely on $\hat{\mu}, \hat{v}$ of Eq. 6.2. To calculate these values without forming the dense matrix of R_α requires only the ability to apply R_α as a matrix-vector product and to solve R_α with a vector/matrix. There exist many methods which can be used to solve a linear system. Two direct methods used are Gaussian elimination for general matrices and Cholesky factorization for symmetric positive definite matrices [54]. These direct methods have the advantage of (in exact math) providing the exact solution after a finite number of steps have been performed. On the other hand, iterative methods exist that begin with an initial guess and refine this guess until a certain tolerance is reached. GMRES is a Krylov subspace iterative method used for general matrices, while Preconditioned Conjugate Gradient is the typical method for solving symmetric positive definite systems [64, 65, 55]. One advantage of Krylov subspace methods is that rather than directly accessing the matrix, they work by building up a subspace via matrix-vector products, allowing for rapid iteration when matrix-vector products can be performed quickly, such as when there is a sparse representation. In particular, the Gaussian kernel results in symmetric positive definite kernel matrices. Thus, the Preconditioned Conjugate Gradient method can be used to solve the system that arises from the kernel matrix.

Calculating the matrix product of R_α with some vector x , which is needed for operations including the solve, given a matrix product of R can be done with only a few steps. We observe

$$R_a * x = (\alpha R + (1-\alpha)I)x \tag{6.7}$$

$$= \alpha Rx + (1-\alpha)x \tag{6.8}$$

$$= \alpha(Rx) + (1-\alpha)x \tag{6.9}$$

and thus, the cost of performing $R_\alpha x$ given Rx consists of scaling a vector followed by the addition of a vector - two parallel operations of $O(n)$ cost. As Preconditioned Conjugate Gradient requires only matrix-vector products to solve a linear system, we can solve $R_\alpha^{-1}y$ for a vector y via the Preconditioned Conjugate Gradient method using R in a completely matrix-free manner given α.

Algorithm 4 Pointwise prediction

```
INPUT: Solve covariance matrix $R_\alpha$, Evaluation of r(s), mean $\hat{\mu}$, training points response $\tilde{Y}$
OUTPUT: prediction $y_p$, variance $var$
  $\Psi \leftarrow R_\alpha^{-1}(\tilde{Y} - \mu\mathbb{1})$
  $y_p = \hat{\mu} + \hat{\alpha}(r(s)^\top * \Psi)$
  if Variance is desired then
    $v \leftarrow \frac{(\tilde{y}-\hat{\mu}\mathbb{1})\Psi^\top}{n}$
    $\sigma \leftarrow \alpha v$
    $tmp1 \leftarrow R_\alpha^{-1} r(s)$
    $val1 \leftarrow \alpha r(s) tmp1$
    $R1 \leftarrow R_\alpha^{-1}\mathbb{1}$
    $val2 = \frac{(1-\alpha * r(s)^\top R1)^2}{\mathbb{1}^\top R1}$
    $var = \sigma * (val1) + v * val2$
  end if
```

Thus using Alg. 4, we can perform the mean prediction, seen in Eq. 6.5, and the predictive variance in Eq. 6.6, given a set of hyperparameters θ, α.

6.2.2 Hyperparameter Optimization

The act of finding the optimal hyperparameters to be used for Gaussian process regression is known as training, tuning, or hyperparameter optimization. As with optimization problems, many methods are available to go about hyperparameter optimization. The L-BFGS-B method [66] is the typical method used for finding the hyperparameters associated with Gaussian processes [67], where it is used to find the minimum log-likelihood. To optimize via L-BFGS-B, both the function being optimized and its derivative needs to be calculated. In the case of Gaussian process regression, this is given by Eq. 6.3 and 6.4 respectively.

Similar to the prediction, the tuning of Gaussian process hyperparameters can be calculated without direct access to R using matrix-free methods. In addition to the evaluations and solutions covered in Section 6.2.1, the training also requires calculating the log-likelihood and its derivative - Eq. 6.3 and 6.4. We observe that this requires not only the solution of linear systems with R_α, it also requires calculating $\log|R_\alpha|$, $\beta^\top \frac{\partial R_\alpha}{\partial \Psi}\beta$ and $\mathrm{Tr}(R_\alpha \frac{\partial R_\alpha}{\partial \cdot})$ for $\beta = R_\alpha^{-1}(\tilde{Y} - \hat{\mu})$ and $\cdot = \alpha, \theta$. As such, we will now build the various operations required to evaluate the log-likelihood and, in turn, the hyperparameter tuning.

The first operation we will discuss is the calculation of $\log|R_\alpha|$. Recall that, for a $n \times n$ matrix A,

$$\log|A| = \log(\det(A)) \tag{6.10}$$

$$= \mathrm{Tr}(\log(A)) \tag{6.11}$$

$$= \sum_i (\log(\lambda_i)) \quad \lambda_i = \mathrm{eig}(A) \tag{6.12}$$

$$\tag{6.13}$$

And thus to calculate the log determinant of R_α, we simply have to calculate $\mathrm{Tr}(f(R_\alpha))$ where $f(x) = \log(x)$.

As R_α is symmetric positive definite, one method for achieving this is Stochastic Lanczos Quadrature (SLQ) [68]. This method utilizes the matrix-free Lanczos algorithm to provide eigenvalue estimates, which can then be used to estimate the matrix trace function of a symmetric positive definite matrix. This matrix-free trace estimation can also be used in the other operations required for the training.

The next operation we will consider is $\mathrm{Tr}(R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \alpha})$. From [69] we have that $\frac{\partial R_\alpha}{\partial \alpha} = R - I$.

Thus, we have

$$\mathrm{Tr}(R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \alpha}) = \mathrm{Tr}(R_\alpha^{-1}(R-I)) \tag{6.14}$$

$$= \mathrm{Tr}((\alpha R + (1-\alpha)I)^{-1}(R-I)). \tag{6.15}$$

Thus, we can estimate $\mathrm{Tr}(f(R))$ with $f(x) = 1/(\alpha x + (1-\alpha))(x-1)$, using matrix-vector products via SLQ.

Next, we will consider $\beta^\top(R-I)\beta$.

$$\beta^\top(R-I)\beta = \beta^\top R\beta - \beta^\top\beta \tag{6.16}$$

$$= \beta^\top(R\beta) - \beta^\top\beta \tag{6.17}$$

And thus we can calculate $\beta^\top(\frac{\partial R}{\partial \alpha})\beta$ simply using a matrix-vector product with R.

Next, let us consider $\mathrm{Tr}(R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \theta})$. From [70] we have that $\frac{\partial R_\alpha}{\partial \theta} = E/\theta^3 \circ R$ where E is the squared Euclidean distance matrix and $\circ$ is the elementwise (Hadamard) product. Thus, if we have a matrix-free approximation of $\frac{\partial R_\alpha}{\partial \theta}$, we can calculate $\mathrm{Tr}(R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \theta})$ by using a matrix-free trace estimation where the matrix-vector product $x = R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \theta} * y$ is performed in two parts: $z = \frac{\partial R_\alpha}{\partial \theta}y$, followed by $x = R_\alpha^{-1}z$, as seen in Alg. 5.

Note that, unlike the previous trace estimation, this does not involve a general matrix function, and thus a trace estimator such as Hutchinson's can be used, as seen in Alg. 6

Algorithm 5 $x = R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \theta} y$

```
$z \leftarrow \frac{\partial R_\alpha}{\partial \theta} y$
$x \leftarrow R_\alpha^{-1} z$
```

Logdet	SLQ, $f(x) = log(x); A = R_\alpha$
$\beta^\top(R - I)\beta$	$\mathcal{H}^2$approximation of R
$\text{Tr}(R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \alpha})$	SLQ, $f(x) = \frac{1}{\alpha x + (1-\alpha))(x-1)}; Ay = Ry$
$\text{Tr}(R_\alpha^{-1}\frac{\partial R_\alpha}{\partial \theta})$	Hutchinson, $Ay = Alg.\,5(y)$
$\beta^\top \frac{\partial R_\alpha}{\partial \theta}\beta$	$\mathcal{H}^2$ approximation of $\frac{\partial R_\alpha}{\partial \theta}$

Table 6.1: Overview of operations used for optimization.

[71].

Algorithm 6 Hutchinson's Estimator

```
$tr(A) \approx \frac{1}{nv}\sum_{i=1}^{nv} z_i^\top A z_i, \quad z \sim \{-1, 1\}^m$
```

Finally, we will consider the calculation of $\beta^\top \frac{\partial R_\alpha}{\partial \theta}\beta$. Once again, we use a matrix-free approximation to $\frac{\partial R_\alpha}{\partial \theta}$ to provide the $\frac{\partial R_\alpha}{\partial \theta}\beta$ matrix-vector product.

Thus, we now have all the building blocks required for calculating the log-likelihood, the derivative of the log-likelihood, and the predictive mean and predictive variance, and can thus train and provide mean and variance predictions. A summary can be seen in Table 6.1.

6.3 Practical Considerations

While in the previous section, we provided a high-level basis for the algorithms required to achieve good performance in accuracy and computation time. However, several additional considerations must be considered. In this section, we will discuss numerical issues that arise during these computations, how they can be alleviated, and implementation details to achieve efficient computations.

6.3.1 Numerical Issues

One of the underlying assumptions in Gaussian processes is that the kernel used is smooth, allowing for interpolation between points. This additionally makes the kernels used the target of many methods to exploit the near rank deficient nature of the matrix, including those based on the Nyström method [72]. However, this near rank deficiency can result in algorithmic breakdowns, as the algorithms may rely on the matrices being positive definite, which must be considered when designing any framework. Notably, the smoother the kernel is (the larger the length scale, the more points there are, and the closer the points are), the more numerically rank-deficient the kernel matrix may be.

Traditional full Gaussian processes that use direct methods partially alleviate this issue by using a nugget, a perturbation to the diagonal of the kernel matrix associated with the measurement noise. The diagonal perturbation shifts the eigenvalues by the nugget, thus decreasing the likelihood of having numerically negative eigenvalues. However, depending on the eigenvalue spectrum of the matrix, this may not be enough, and the Cholesky decomposition may fail, indicating a numerically non-positive definite matrix. In this case, variations such as the Modified Cholesky decomposition may be used [73]. Simulations, where the measurement error would be zero, may prove difficult using direct methods for Gaussian processes as a nugget may not be applied.

One advantage that the matrix-free methods described above have over traditional full Gaussian processes is that as the algorithms depend on the eigenvalue estimates, and as the eigenvalues are lower bounded (as the eigenvalues of R have a lower bound of 0, as it is positive definite, the diagonal shift will shift them upwards) we can exploit this information to correct for the numerically negative eigenvalues. Particularly, in the case of the log determinant, any eigenvalue estimate smaller than the nugget can be replaced with the nugget.

6.3.2 Preconditioning Linear Systems

As mentioned previously, it is quite common for the underlying matrices to be poorly conditioned, which may hurt the convergence of the matrix-free solutions. Preconditioning can be used to better condition the systems, allowing for quicker convergence during the solution of linear equations. Preconditioning of the kernel matrices associated with Gaussian process regression is discussed in detail in Section 5.

6.3.3 Implementation Details

By implementing the previous sections, a framework for matrix-free Gaussian processes can be developed. However, careful consideration must be given to the implementation to achieve efficient performance. This section discusses various implementation details that aim to increase the performance achieved by matrix-free Gaussian process frameworks.

Blocking

First, we will discuss the use of achieving high performance by increasing the size of the linear algebra operations being performed. This can allow for better use of the hardware, such as reducing the number of jumps, reads from memory, and increasing cache use, which can increase the percentage of theoretical FLOPs obtained.

The most obvious area where this can be implemented is during the prediction, where rather than considering each prediction point sequentially, the points are considered as a vector. This leads to a mean prediction for m points X_p with n training points X_t being performed as

$$y_{pred} = \hat{\mu}\mathbb{1}_m + \hat{\alpha}\mathcal{R}(X_p, X_t)R_\alpha^{-1}(\tilde{Y} - \hat{\mu}\mathbb{1}_n) \tag{6.18}$$

where $R_\alpha^{-1}(Y - \hat{\mu})$ occurs frequently enough in calculations where it should be stored when first calculated. Thus, prediction consists only of a $m \times n$ matrix-vector product, followed by a scaling and vector addition (this can be performed as a fused-multiply-add (FMA)) [74, 75].

Similarly, the prediction variance can be calculated as

$$var_{pred} = \hat{\sigma}^2(1 - \hat{\alpha}\mathbb{1}_m{}^\top \mathcal{R}(X_p, X_t) R_\alpha^{-1} \mathcal{R}(X_t, X_p)) + \frac{(1 - \hat{\alpha}\mathcal{R}(X_p, X_t) R_\alpha^{-1}\mathbb{1}_n).^2}{(\mathbb{1}_n^\top (\hat{v} R_\alpha)^{-1} \mathbb{1}_n)}, \quad (6.19)$$

where $.^2$ corresponds with squaring the matrix elementwise.

Algorithm 7 Block prediction.

```
INPUT: Covariance matrix $R_\alpha$, Kernel $\mathcal{R}$, Prediction points $X_p$, mean $\hat{\mu}$, training points
    $X_t$, training points response $\tilde{Y}$
OUTPUT: prediction $y_p$, variance $var$
  $\Psi \leftarrow R_\alpha^{-1}(\tilde{Y} - \hat{\mu}\mathbb{1})$
  $y_p = \hat{\mu}\mathbb{1} + \alpha(\mathcal{R}(X_p, X_t) * \Psi)$
  if Variance is desired then
    $v \leftarrow \frac{(\tilde{Y} - \hat{\mu}\mathbb{1})\Psi^\top}{n}$
    $\sigma \leftarrow \alpha v$
    $tmp1 = R_\alpha^{-1}\mathcal{R}(X_t, X_p)$
    $val1 = \alpha\mathbb{1}^\top \mathcal{R}(X_p, X_t) tmp1$
    $R1 \leftarrow R_\alpha^{-1}\mathbb{1}$
    $val2 = \frac{(1 - \alpha * \mathcal{R}(X_p, X_t) R1).^2}{\mathbb{1}^\top R1}$
    $var = \sigma * (val1) + v * val2$
  end if
```

Next, we will consider the variance prediction. First, $R_\alpha^{-1}\mathbb{1}$ and $\tilde{Y} - \hat{\mu}\mathbb{1}$ can both be calculated and stored for reuse. To calculate $\hat{\sigma}^2(1 - \alpha\mathbb{1}\mathcal{R}(X_p, X_t) R_\alpha^{-1}\mathcal{R}(X_t, X_p))$ we begin with $R_\alpha^{-1}\mathcal{R}(X_t, X_p)$. We can perform the $R_\alpha^{-1}\mathcal{R}(X_t, X_p)$ calculation in a blocked fashion. This can be performed in a single large solve via a blocked CG algorithm [55, 76], or split into blocks to be processed in parallel using such a blocked CG algorithm. Once this is calculated, the corresponding block can be right multiplied with the corresponding block of $K(X_p, X_t)$ before performing a column-wise sum, yielding the desired value. Next, we can calculate the val2 vector as written in Alg. 4. From this, the resulting variance vector can be calculated simply by combing the val1 and val2 vectors in Alg. 4.

The Hutchinson estimator provides significant parallelization, as all samples are independent of each other and can be formulated as a combination of matrix products and elementwise products [77]. If we let the number of samples be nv, then we can generate a matrix from the Rademacher Distribution (where each element is either -1 or 1), perform a matrix product with A (the matrix whose trace we are estimating), yielding B, then calculating the dot product of each column of Z with the corresponding column of B, and summing the results. This can be performed by taking the element-wise product of Z and B, then summing the result. The pseudocode for the algorithm can be seen in Alg. 8.

Algorithm 8 Blocked Hutchinson

```
$Z \leftarrow \{-1, 1\}^{m \times nv}$
$B \leftarrow AZ$
**return** $\frac{\sum_{ij}(Z \circ B)_{ij}}{nv}$
```

Similar to the Hutchinson estimator, as the Stochastic Lanczos Quadrature requires many samples, SLQ provides an excellent opportunity for performance increase from blocking. Trace function estimation via Stochastic Lanczos Quadrature relies upon a block Lanczos algorithm. We can see in Alg. 9 that the Lanczos Algorithm ((6.15) of [78]) can be adapted to have a block form while exploiting level 2 and 3 BLAS. This reduces needing $m * nv$ matrix-vector products of size n with A to requiring m matrix-matrix products of size $n \times nv$, which will provide the benefits of using level 3 BLAS-like functions.

Algorithm 9 Block Lanczos - without full reorthogonalization

```
Let $b^1 = [\beta_1^1 \beta_1^2 \dots \beta_1^{nv}] = 0$
Let $V$ be a series of $nv$ $n \times m$ matrices, where $m$ is the subspace size
Choose initial vectors $V^1$ of size $n \times nv$ with columnwise unity norm
**for** j=1,...,m **do**
    $W \leftarrow AV^j$
    $W \leftarrow W - V^{j-1} * \mathrm{diag}(b^{j-1})$
    $\alpha_v \leftarrow \mathbb{1}^\top (W \circ V^j)$
    $W^j = W^j - V^j \, \mathrm{diag}(\alpha_v^j)$
    $b^{j+1} \leftarrow \mathbb{1}^\top (W^j \circ W^j)$
    $V^{j+1} = W^j \, \mathrm{diag}(b^{j+1})^{-1}$
**end for**
```

Additional Implementation Details

As the log determinant must be calculated for a matrix that is likely to be near rank deficient, as well as having an eigenvalue floor, one can exploit these properties to develop an efficient Lanczos Based Density of States (Landos) algorithm to calculate the log determinant and trace functions [79].

We observe that, for a sufficiently smooth problem, and thus numerically rank deficient, the problem has $r << n$ numerical eigenvalues with magnitude greater than the noise floor and $n - r$ numerical eigenvalues with magnitude of the noise floor. The spectrum of such a problem can be seen in Fig. 5.4b. As the log determinant can be calculated as the sum of the log of the eigenvalues

$$\sum_i \log(\lambda_i),$$

the eigenvalues which compose the noise floor contribute the most to the log-determinant. Their contribution can be calculated as

$$\sum_{i=r}^{n} (\log(\lambda_i)) = \log(\alpha) * (n - r).$$

However, r is not known apriori.

By using Landos, we can estimate the spectral density and thus obtain an estimate of the number of eigenvalues with magnitude around α - giving us an approximation of r. Furthermore, by using Landos with the log function, we can get the contribution of the r eigenvalues to the log determinant, yielding a matrix-free approximation to the log determinant. This log determinant is likely to work well when the noise used is small and when the rank r is small.

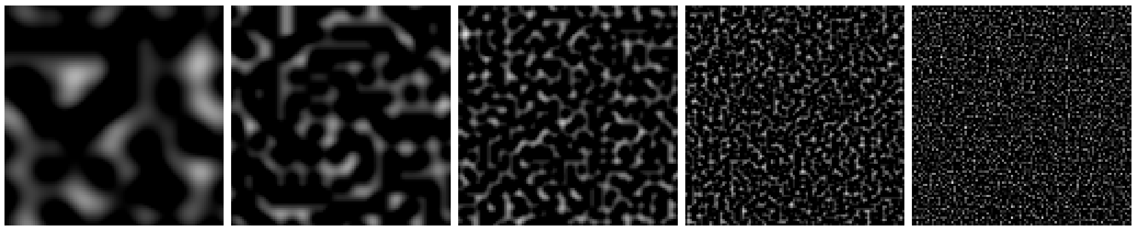
6.4 Numerical Experiments

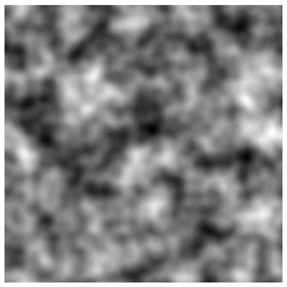
In this section, we evaluate the performance of smashGP using simulations. First, we describe the method used to generate simulation data. Next, we compare the accuracy and computation time to fit a Gaussian process for smashGP, which uses hierarchical matrix-free operations, and DiceKriging [80], which uses dense matrix decomposition and inversion. DiceKriging is presently the fastest R implementation (with a C backend) to learn a Gaussian process but still requires $\mathcal{O}(n^3)$ operations and $\mathcal{O}(n^2)$ memory. Then, we investigate the impact of the mask on smashGP and another method, SPDE. Finally, we compare the predictive accuracy of smashGP with state-of-art methods for large-scale spatial modeling with Gaussian processes.

6.4.1 Simulations

Data Generation

To allow the flexible testing of different methods on various problems to examine how the techniques can handle different types of problems, we developed generating code for simulations. This data generation allows us to examine the impact of many factors, including problem size, noise in the data, smoothness of the data, and spatial correlation of the testing data. This is achieved through Perlin noise [81], which allows for procedural generation of noise that relatively smoothly transitions over space. A highly detailed yet controlled dataset can be created by combining multiple noise levels. We utilize the `ambient` R package with `gen_perlin` to generate independent layers of a given size. In particular, for level k, we use a frequency of $5 * (2^k)$, scaling the noise by $1/max(1, k)$. A normally distributed noise is added to every location, representing measurement noise. This creates a fractal-like noise, which can be used in spatial prediction due to its spatially correlated data. An example of this can be seen in Fig. 6.1a, where Fig. 6.1b, 6.1c demonstrates the resulting surface after scaling and summing the different levels.


(a) Different layers of Perlin noise with varying frequencies


(b) Texture with three layers of Perlin noise

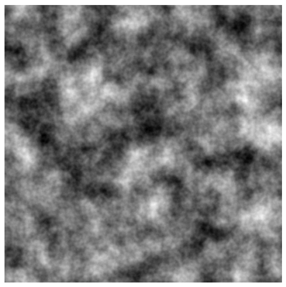
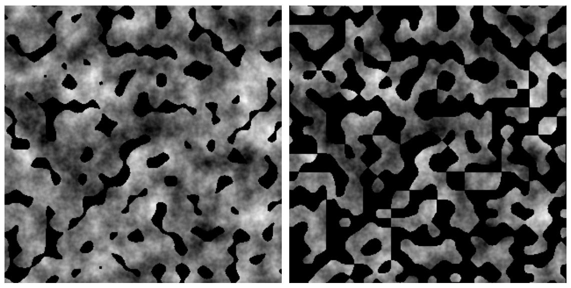

(c) Texture with five layers of Perlin noise

Fig. 6.1: Data generation using Perlin noise.


(a) Perlin test data, 20% (left), 50% (right)

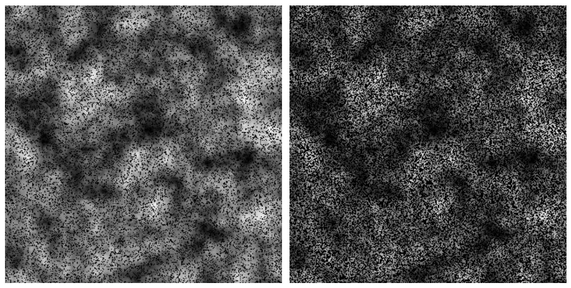

(b) Random test data, 20% (left), 50% (right)

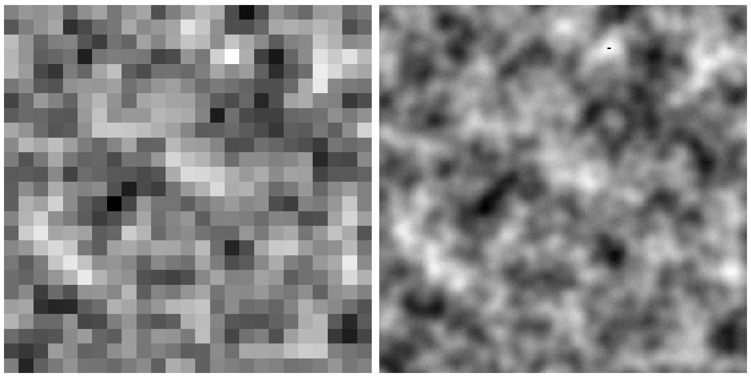
Fig. 6.2: Examples of the training and test datasets using Perlin noise. The test data appears in solid black.

To determine a mask of missing data, either a single layer of Perlin noise is used (of selectable frequency), or a layer of uniformly distributed random values is used. For a selectable parameter t of the test percentage, the locations associated with the largest $t\%$ of the domain are selected as test data. The Perlin noise mask could correspond with cloud or tree coverage, masking the elevation of the terrain. An example of these masks can be seen in Fig. 6.2. The uniform mask may correspond with how data might be sampled for machine learning tasks.

6.4.2 Comparison with Baseline

We begin with comparing smashGP with the DiceKriging R package for Gaussian process regression, demonstrating its accuracy and performance [80]. Both are configured to use an isotropic length scale and train on 80% of the test data while testing on the remaining 20%. The Gaussian kernel is used, and nugget estimation is enabled for Dice. The data used for this experiment can be seen in Fig. 6.3. One desirable property of the Perlin noise is that it is sampled from a continuous domain, and thus multiple resolutions can be used. Thus, we test for a variety of resolutions, ranging from a 25×25 grid (625 points) up to a 140×140 (19600 points) for both and up to a grid of 316×316 for smashGP (99856 points).

We see in Table 6.2 the metrics between both the methods, with the difference column being the relative difference. This includes both the direct error metrics of mean absolute error (MAE) and root mean squared error (RMSE) and the uncertainty metrics INT and CVG. Thus, we see that smashGP can achieve the same accuracy as the Dice package, which uses full precision calculations without any approximations. In Fig. 6.4 a comparison of the timings for smashGP and Dice is presented. We observe that the scaling of Dice very closely matches the trend line across the whole range, as would be expected for a fully dense and direct calculation. On the other hand, smashGP takes longer to start with, but as the problem size increases, so does the benefit of using smashGP. Above 6400 points, a small problem for hierarchical matrices, smashGP outperforms Dice. We see that the total


(a) Underlying ($25 \times 25, 140 \times 140$) data for smashGP vs Dice comparison.

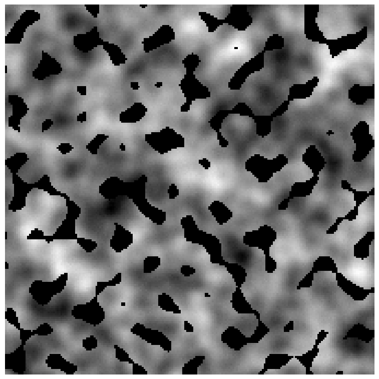

(b) Mask used for smashGP vs. Dice.

Fig. 6.3: Demonstrations of the data used to compare smashGP and Dice.

Table 6.2: Performance comparison between smashGP and DiceKriging for different problem sizes

	MAE			RMSE			INT			CVG		
Num. Points	**smashGP**	**Dice**	**Rel. Diff.**	**smashGP**	**Dice**	**Rel. Diff.**	**smashGP**	**Dice**	**Rel. Diff.**	**smashGP**	**Dice**	**Rel. Diff.**
625	0.0892	0.0886	0.0069	0.1132	0.1127	0.0042	0.5203	0.5235	-0.0060	0.9040	0.9040	0.0000
900	0.0817	0.0812	0.0058	0.1015	0.1006	0.0083	0.4947	0.4496	0.1003	0.8722	0.8944	-0.0248
1600	0.0568	0.0573	-0.0092	0.0740	0.0743	-0.0037	0.3382	0.3550	-0.0471	0.9375	0.9125	0.0274
2500	0.0331	0.0331	-0.0002	0.0433	0.0433	-0.0001	0.1930	0.1930	0.0000	0.9360	0.9360	0.0000
3600	0.0263	0.0264	-0.0002	0.0373	0.0374	-0.0036	0.1612	0.1620	-0.0055	0.9264	0.9347	-0.0089
4900	0.0231	0.0231	0.0001	0.0322	0.0321	0.0007	0.1286	0.1282	0.0026	0.9143	0.9153	-0.0011
6400	0.0186	0.0186	-0.0015	0.0251	0.0251	-0.0016	0.1105	0.1107	-0.0017	0.9383	0.9352	0.0033
8100	0.0166	0.0166	0.0003	0.0220	0.0220	0.0003	0.1027	0.1028	-0.0012	0.9420	0.9414	0.0007
10000	0.0178	0.0178	0.0009	0.0233	0.0232	0.0009	0.1006	0.1008	-0.0024	0.9260	0.9245	0.0016
14400	0.0165	0.0165	-0.0002	0.0217	0.0217	-0.0003	0.0966	0.0966	-0.0005	0.9201	0.9201	0.0000
19600	0.0143	0.0143	0.0002	0.0188	0.0188	0.0000	0.0884	0.0880	0.0053	0.9227	0.9242	-0.0017

tuning time for smashGP scales better than Dice.

6.4.3 Investigating Length scales

The next set of tests performed with the simulation data investigates the impact of the smoothness of the data and the clustering of the mask. In this case, three variables are being tested, the amount of data withheld for testing (20%, 40%), the number of levels of noise (3,5), and the clustering of the mask. We can see the example of the smooth underlying data in Fig. 6.5. We can see the resulting accuracy comparing smashGP, Dice, and SPDE, a stochastic partial differential equation method, in Fig. 6.6. We observe that as the mask becomes more uniform, the RMSE tends to decrease in all cases. This is logical,

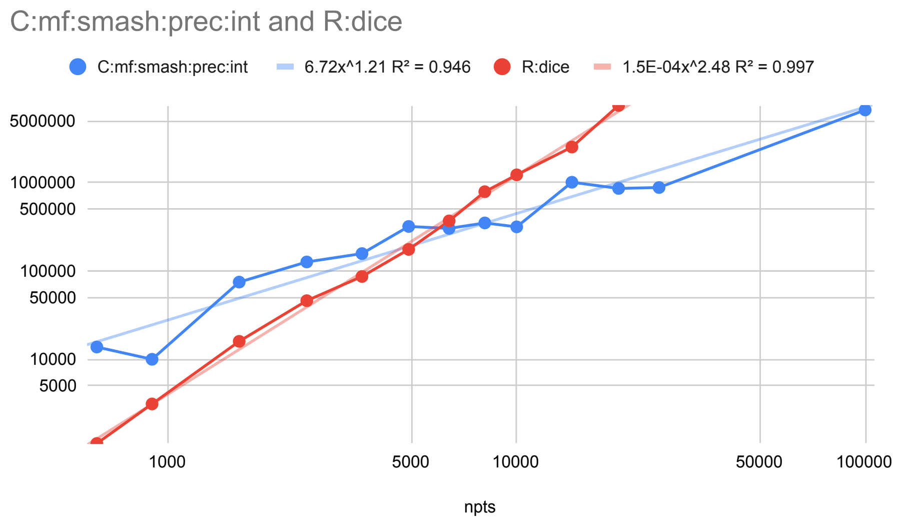
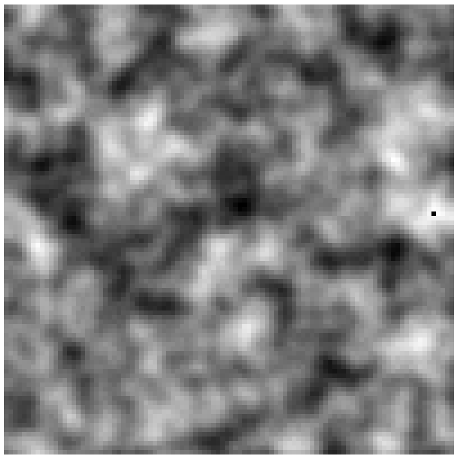


Fig. 6.4: Comparing the tuning time for smashGP and DiceKriging.

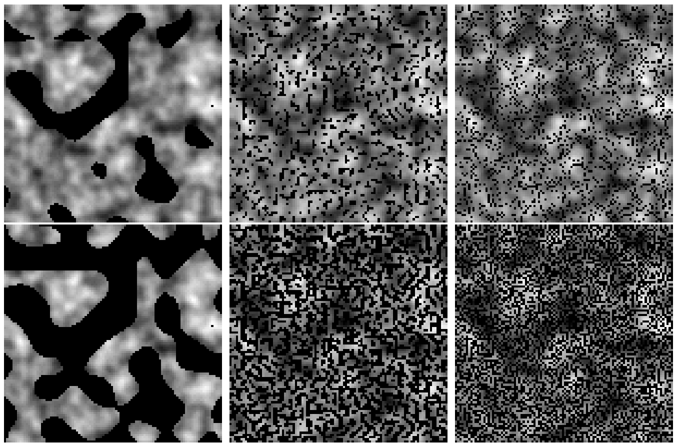
as this corresponds with having more information surrounding the missing data, and thus a better guess can be made on the missing data. Similarly, when the amount of test data is increased, the RMSE increases as less information is available in the training set. We observe that for smoother data, Dice and smashGP perform better than SPDE, while for rough data, SPDE performs better. This may be since Dice and smashGP are based on Gaussian process regression directly, they only have a single isotropic length scale to be optimized. We observe in the smooth case that the methods have a more drastic initial dropoff in RMSE, while the convergence is more gradual for the rougher data.

6.4.4 Comparison Tests

Now that we have established that smashGP closely matches the baseline Dice in metrics for a wide variety of problems, we will now compare various methods on the same dataset. In addition to the SPDE method previously seen, we also test the LatticeKriging, GPyTorch, and Partition-based methods [82, 83, 76]. These methods correspond with the Competition Case Study Paper methods [84], with LatticeKriging and SPDE being Sparse Precision methods and Partition being a Sparse Covariance method. Using the data gen-



(a) Underlying smooth surface.



(b) A variety of example test data with low (top) and high (bottom) testing percentages.

Fig. 6.5: Varying the test data for both low and high testing percentages.

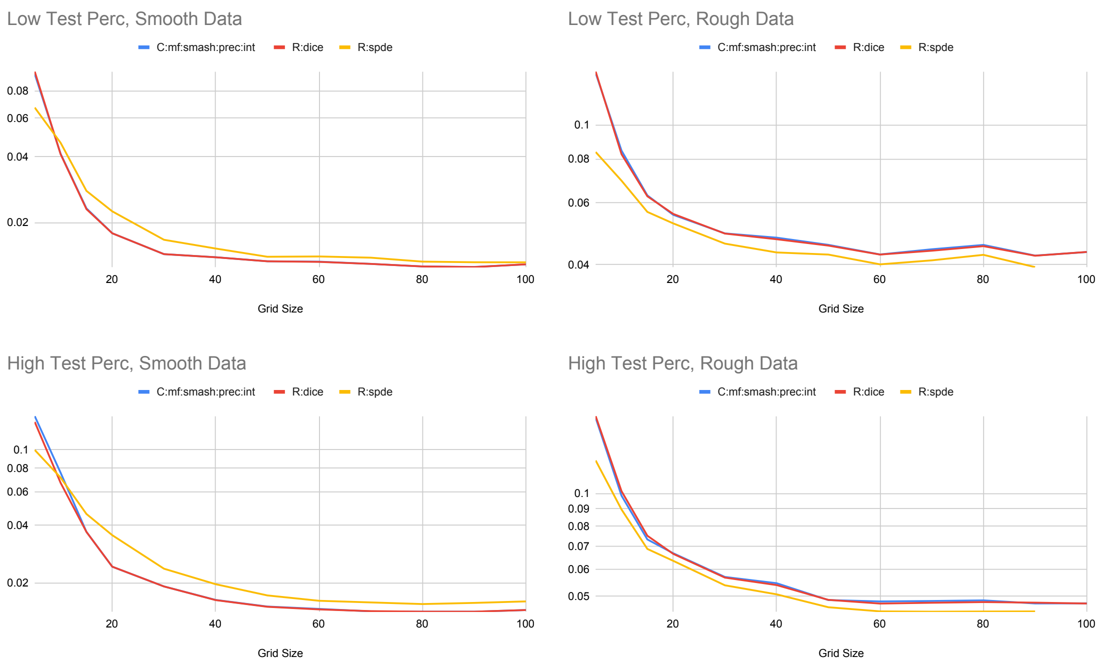


Fig. 6.6: Comparing smashGP against Dice and SPDE for differing grid sizes.

erator discussed in the previous subsection, we use a measurement noise with a standard deviation of 0.005 and 0.01, a uniform testing mask, and a Perlin-based mask, a testing percentage of 20% and 50%, as well as two different noise levels. The high smoothness case corresponds with two levels of Perlin noise, while the low smoothness case corresponds with three levels. These tests were repeated for 50 samples to ensure the statistical significance of the results.

Table 6.3 contains the averaged MAE over the tests, while Table 6.4 contains the averaged RMSE. We see that smashGP performs very well for the tests with high smoothness while still performing competitively in the low smoothness case. This likely relates to the underlying Gaussian processes used in smashGP only having a single isotropic length scale, while the other methods may make alternative statistical assumptions. On the other hand, when looking at the interval score, which is a metric for uncertainty quantification in Fig. 6.5, smashGP outperforms the other methods in nearly all the tests. Fig. 6.6 demonstrates that smashGP has the best coverage in the case of a Perlin mask while still performing competitively in the Random noise mask case. This demonstrates the effectiveness of smashGP in both cases where the data itself is fairly smooth or where uncertainty quantification is needed, which often is required to ensure the accuracy of simulations. Thus, as smashGP scales asymptotically better than native methods, it can help expand the types of problems which can be tackled with Gaussian process regression.

Table 6.3: Comparison of MAE over 50 simulation replicates. Results are in the form of mean (standard deviation).

			MAE					
Smooth.	**Noise**	**Test %**	**Test mask**	**smashGP**	**GPyTorch**	**SP**	**latticeKrig**	**SPDE**
Low	0.005	20%	Random	**0.00459(0.00005)**	0.05859(0.00178)	0.00476(0.00005)	0.00469(0.00004)	0.00503(0.00004)
			Perlin	0.02133(0.00216)	0.06945(0.00381)	0.02478(0.00155)	0.02047(0.00143)	**0.02024(0.00146)**
		50%	Random	**0.00486(0.00006)**	0.04045(0.00172)	0.00532(0.00009)	0.00503(0.00005)	0.00509(0.00003)
			Perlin	0.04014(0.00378)	0.06253(0.00858)	0.03926(0.00300)	0.03309(0.00247)	**0.03152(0.00228)**
	0.01	20%	Random	**0.00867(0.00006)**	0.05884(0.00166)	0.00921(0.00010)	0.00889(0.00007)	0.00899(0.00008)
			Perlin	0.02424(0.00147)	0.06956(0.00389)	0.02854(0.00189)	0.02380(0.00144)	**0.02330(0.00143)**
		50%	Random	**0.00894(0.00005)**	0.04096(0.00170)	0.00969(0.00013)	0.00930(0.00006)	0.00936(0.00008)
			Perlin	0.04219(0.00338)	0.06275(0.00856)	0.04238(0.00299)	0.03587(0.00251)	**0.03405(0.00224)**
High	0.005	20%	Random	**0.00416(0.00003)**	0.05640(0.00168)	0.00440(0.00004)	0.00451(0.00004)	0.00451(0.00004)
			Perlin	**0.00758(0.00039)**	0.06806(0.00444)	0.01752(0.00155)	0.01424(0.00115)	0.01092(0.00074)
		50%	Random	**0.00422(0.00002)**	0.03640(0.00130)	0.00464(0.00004)	0.00474(0.00004)	0.00469(0.00005)
			Perlin	**0.01542(0.00180)**	0.06113(0.00877)	0.03356(0.00337)	0.02698(0.00271)	0.02065(0.00190)
	0.01	20%	Random	**0.00823(0.00006)**	0.05652(0.00173)	0.00853(0.00006)	0.00866(0.00007)	0.00860(0.00038)
			Perlin	**0.01209(0.00046)**	0.06829(0.00442)	0.02087(0.00143)	0.01778(0.00115)	0.01476(0.00073)
		50%	Random	**0.00837(0.00003)**	0.03685(0.00129)	0.00877(0.00006)	0.00893(0.00006)	0.00870(0.00005)
			Perlin	**0.01927(0.00180)**	0.06124(0.00884)	0.03586(0.00322)	0.03004(0.00273)	0.02389(0.00190)

Table 6.4: Comparison of RMSE over 50 simulation replicates. Results are in the form of mean (standard deviation).

			RMSE					
Smooth.	**Noise**	**Test %**	**Test mask**	**smashGP**	**GPyTorch**	**SP**	**latticeKrig**	**SPDE**
Low	0.005	20%	Random	**0.0058 (0.0001)**	0.0732 (0.0022)	0.0060 (0.0001)	0.0059 (0.0001)	0.0063 (0.0001)
			Perlin	0.0345 (0.0042)	0.0870 (0.0049)	0.0346 (0.0024)	0.0299 (0.0023)	**0.0290 (0.0023)**
		50%	Random	**0.0061 (0.0001)**	0.0504 (0.0022)	0.0068 (0.0001)	0.0063 (0.0001)	0.0064 (0.0000)
			Perlin	0.0621 (0.0063)	0.0797 (0.0106)	0.0549 (0.0045)	0.0473 (0.0038)	**0.0445 (0.0036)**
	0.01	20%	Random	**0.0109 (0.0001)**	0.0735 (0.0021)	0.0116 (0.0001)	0.0111 (0.0001)	0.0113 (0.0001)
			Perlin	0.0359 (0.0028)	0.0871 (0.0050)	0.0384 (0.0027)	0.0331 (0.0022)	**0.0320 (0.0022)**
		50%	Random	**0.0112 (0.0001)**	0.0510 (0.0021)	0.0122 (0.0002)	0.0117 (0.0001)	0.0117 (0.0001)
			Perlin	0.0623 (0.0056)	0.0800 (0.0106)	0.0578 (0.0045)	0.0498 (0.0038)	**0.0468 (0.0034)**
High	0.005	20%	Random	**0.0052 (0.0000)**	0.0704 (0.0021)	0.0055 (0.0000)	0.0057 (0.0000)	0.0056 (0.0001)
			Perlin	**0.0109 (0.0010)**	0.0852 (0.0056)	0.0253 (0.0024)	0.0213 (0.0020)	0.0157 (0.0013)
		50%	Random	**0.0053 (0.0000)**	0.0450 (0.0016)	0.0059 (0.0000)	0.0059 (0.0000)	0.0059 (0.0001)
			Perlin	**0.0253 (0.0043)**	0.0782 (0.0111)	0.0486 (0.0054)	0.0400 (0.0044)	0.0307 (0.0033)
	0.01	20%	Random	**0.0103 (0.0001)**	0.0705 (0.0022)	0.0107 (0.0001)	0.0108 (0.0001)	0.0108 (0.0005)
			Perlin	**0.0161 (0.0010)**	0.0855 (0.0056)	0.0287 (0.0022)	0.0248 (0.0019)	0.0198 (0.0012)
		50%	Random	**0.0105 (0.0000)**	0.0455 (0.0016)	0.0110 (0.0001)	0.0112 (0.0001)	0.0109 (0.0001)
			Perlin	**0.0285 (0.0039)**	0.0783 (0.0111)	0.0507 (0.0052)	0.0428 (0.0044)	0.0339 (0.0032)

Table 6.5: Comparison of INT over 50 simulation replicates. Results are in the form of mean (standard deviation).

			INT					
Smooth.	**Noise**	**Test %**	**Test mask**	**smashGP**	**GPyTorch**	**SP**	**latticeKrig**	**SPDE**
Low	0.005	20%	Random	**0.0269 (0.00032)**	3.2006 (0.04093)	0.0365 (0.00153)	0.0276 (0.00028)	0.0311 (0.00041)
			Perlin	**0.1273 (0.01777)**	3.1967 (0.04539)	0.3004 (0.03408)	0.1545 (0.01877)	0.1709 (0.02294)
		50%	Random	**0.0286 (0.00037)**	1.6359 (0.12357)	0.0450 (0.00203)	0.0298 (0.00029)	0.0304 (0.00018)
			Perlin	**0.2565 (0.03700)**	1.9220 (0.58792)	0.5832 (0.06685)	0.2974 (0.03720)	0.2850 (0.03211)
	0.01	20%	Random	**0.0509 (0.00044)**	3.2075 (0.04051)	0.0647 (0.00190)	0.0522 (0.00047)	0.0529 (0.00059)
			Perlin	**0.1410 (0.00991)**	3.2043 (0.04511)	0.2931 (0.03551)	0.1567 (0.01522)	0.1547 (0.01696)
		50%	Random	**0.0524 (0.00035)**	1.6437 (0.12034)	0.0680 (0.00231)	0.0548 (0.00038)	0.0552 (0.00049)
			Perlin	0.2500 (0.02451)	1.9398 (0.58196)	0.5559 (0.06288)	0.2716 (0.03129)	**0.2365 (0.02291)**
High	0.005	20%	Random	**0.0244 (0.00020)**	3.1087 (0.04195)	0.0348 (0.00147)	0.0265 (0.00025)	0.0279 (0.00045)
			Perlin	**0.0444 (0.00292)**	3.1084 (0.04763)	0.2069 (0.03234)	0.1068 (0.01527)	0.0775 (0.00950)
		50%	Random	**0.0247 (0.00013)**	1.4450 (0.13101)	0.0344 (0.00175)	0.0280 (0.00023)	0.0291 (0.00051)
			Perlin	**0.0942 (0.01412)**	1.6853 (0.65200)	0.5653 (0.08719)	0.2740 (0.04968)	0.1942 (0.02950)
	0.01	20%	Random	**0.0483 (0.00034)**	3.1170 (0.04196)	0.0562 (0.00103)	0.0508 (0.00043)	0.0512 (0.00473)
			Perlin	**0.0710 (0.00337)**	3.1165 (0.04854)	0.1970 (0.02706)	0.1169 (0.01237)	0.0941 (0.01159)
		50%	Random	**0.0491 (0.00021)**	1.4598 (0.13656)	0.0605 (0.00172)	0.0525 (0.00032)	0.0512 (0.00029)
			Perlin	**0.1147 (0.01177)**	1.7090 (0.64433)	0.4958 (0.07767)	0.2525 (0.04234)	0.1744 (0.02211)

Table 6.6: Comparison of CVG over 50 simulation replicates. Results are in the form of mean (standard deviation).

			CVG					
Smooth.	**Noise**	**Test %**	**Test mask**	**smashGP**	**GPyTorch**	**SP**	**latticeKrig**	**SPDE**
Low	0.005	20%	Random	0.9508 (0.0021)	1.0000 (0.0000)	0.9972 (0.0008)	**0.9502 (0.0019)**	0.9066 (0.0050)
			Perlin	**0.9348 (0.0086)**	1.0000 (0.0000)	0.7207 (0.0158)	0.8742 (0.0127)	0.8192 (0.0149)
		50%	Random	**0.9504 (0.0019)**	1.0000 (0.0000)	0.9984 (0.0003)	0.9517 (0.0015)	0.9504 (0.0064)
			Perlin	**0.9131 (0.0125)**	1.0000 (0.0000)	0.6285 (0.0157)	0.8127 (0.0137)	0.7799 (0.0131)
	0.01	20%	Random	0.9504 (0.0021)	1.0000 (0.0000)	0.9939 (0.0012)	**0.9497 (0.0019)**	0.9418 (0.0024)
			Perlin	**0.9427 (0.0064)**	1.0000 (0.0000)	0.7676 (0.0165)	0.9064 (0.0102)	0.8888 (0.0159)
		50%	Random	**0.9504 (0.0016)**	1.0000 (0.0000)	0.9937 (0.0012)	0.9505 (0.0014)	0.9435 (0.0018)
			Perlin	**0.9341 (0.0098)**	1.0000 (0.0000)	0.6746 (0.0157)	0.8590 (0.0121)	0.8625 (0.0102)
High	0.005	20%	Random	0.9506 (0.0023)	1.0000 (0.0000)	0.9978 (0.0008)	**0.9498 (0.0018)**	0.9042 (0.0034)
			Perlin	**0.9434 (0.0064)**	1.0000 (0.0000)	0.7738 (0.0217)	0.9046 (0.0139)	0.8906 (0.0129)
		50%	Random	0.9509 (0.0015)	1.0000 (0.0000)	0.9956 (0.0015)	**0.9508 (0.0015)**	0.9038 (0.0051)
			Perlin	**0.9314 (0.0098)**	1.0000 (0.0000)	0.6164 (0.0210)	0.8088 (0.0199)	0.8056 (0.0174)
	0.01	20%	Random	0.9502 (0.0021)	1.0000 (0.0000)	0.9890 (0.0020)	**0.9499 (0.0018)**	0.9368 (0.0206)
			Perlin	**0.9422 (0.0076)**	1.0000 (0.0000)	0.8246 (0.0183)	0.9265 (0.0107)	0.9366 (0.0238)
		50%	Random	0.9509 (0.0017)	1.0000 (0.0000)	0.9925 (0.0018)	**0.9503 (0.0014)**	0.9423 (0.0016)
			Perlin	**0.9401 (0.0064)**	1.0000 (0.0000)	0.6881 (0.0207)	0.8533 (0.0175)	0.8735 (0.0212)

Part II

Parallel Quantum Chemistry

The theory of quantum mechanics can be used to understand the properties and motions of electrons via electronic structure calculations. These calculations are required in various scientific disciplines, including chemistry, chemical engineering, material science, and quantum computing, and are prohibitively expensive. As such, there exists a significant demand for efficient algorithms, methods, and implementations to allow for rapid calculations for a wide range of accuracies, system sizes, and physical systems. In this part, we investigate such computations, specifically the use of 3D matrix-matrix multiplications in a holistic Chebyshev-filtered subspace iteration engine accelerated via GPUs, and exploit new methods and hardware to advance the speed and the size of problems that can be considered.

At the core of electronic structure theory is the Schrödinger Equation, which in its time-independent form is given by

$$\hat{H}\Psi = E\Psi, \tag{6.20}$$

where $\hat{H}$ is the Hamiltonian operator, Ψ is a wave function, and E is the energy [85]. This equation describes how a wave function evolves, and it can be seen as an eigenvalue problem. Thus, it is apparent that it can only be solved analytically for very small systems and must be solved numerically for larger systems. Rather than numerically solving the equations directly, various approximations are performed to make the problem tractable.

The particular method which we consider is Chebyshev-filtered subspace iteration for self-consistent fields [10, 86]. This method is based on Kohn-Sham density functional theory, which is used for the high accuracy to cost ratio [87]. In particular, Kohn-Sham density functional theory operates by considering not the many-body wave function seen in the Schrödinger Equation but instead considers a series of single-electron orbitals. This idea depends upon the Hohenberg-Kohn theorems, which state that “the ground state properties of a many electron-system depend only on the electronic density” and that “the correct ground state density for a system is the one that minimizes the total energy through the functional E[n(r)]" [88]. Thus, we can estimate the properties of a system by finding an estimate for the electronic density.

From this, a self-consistent field iteration can be performed based on the equations

$$[-\frac{\nabla^2}{2} + V_{\text{total}}(\rho(r), r)]\Psi_i(r) = E_i\Psi_i(r). \tag{6.21}$$

$$V_{\text{total}}(\rho(r), r) = V_{\text{ion}}(r) + V_H(\rho(r), r) + V_{XC}(\rho(r), r) \tag{6.22}$$

$$\rho(r) = \sum |\Psi_i(r)|^2. \tag{6.23}$$

By beginning with an initial density for ρ, the total potential can be calculated, V_{total}, which allows us to solve for Φ, the discretized Ψ matrix. This Φ can then be used to update ρ. This is repeated until the difference in energy between two iterations is less than some prescribed tolerance. In this part, we investigate how to effectively perform this self-consistent field iteration by exploiting modern hardware and algorithms. Notably, we have four contributions: we investigate the use of advanced matrix-matrix products, investigate acceleration using GPUs, investigate GPU accelerated distributed eigensolvers, and finally use an implementation of these to compare against a baseline implementation, demonstrating the cases where such methods are helpful.

CHAPTER 7
PARALLEL QUANTUM CHEMISTRY

Electronic structure calculations can be used to calculate the motion and properties of electrons accurately. These calculations require calculating (approximate) solutions to the Schrödinger Equation

$$\hat{H}\Psi = E\Psi, \tag{7.1}$$

where $\hat{H}$ is the Hamiltonian operator, Ψ is a wave function and E is the energy. One commonly used method is Kohn-Sham density functional theory (DFT) [89, 90]. However, at the core of Kohn-Sham Density Functional Theory is the solution to an eigenvalue problem, which becomes prohibitively expensive for large problems. Chebyshev-filtered subspace iteration is a method that reduces the cost associated with the eigensolve by instead refining a subspace via Chebyshev polynomials [10, 91, 86]. We have created a high-performance Parallel Computation Engine (libPCE) to perform Chebyshev-filtered subspace iteration in a distributed manner on GPUs. Many of the steps involved in Chebyshev-filtered subspace iteration require the multiplication of matrices, where the number of rows is orders of magnitudes larger than the number of columns. As such, traditional matrix-matrix algorithms and implementations fail to perform well, particularly in the distributed case. Our engine has been designed to provide standalone routines that can replace the computation routines currently used in such DFT codes, including the SPARC package (Simulation Package for Ab-initio Real-space Calculations) [8]. This document investigates how to best exploit modern hardware and methods for Chebyshev-filtered subspace iteration to create a high-performance implementation.

In particular, the contributions we provide are as follows: First, we provide a distributed memory GPU-based implementation to leverage GPUs' computational power and efficiency. Second, we investigate various eigensolvers available to determine which eigensolvers are helpful depending on the problem faced and the hardware available. Third, we investigate the use of advanced matrix-matrix products to accelerate various kernels within Chebyshev-filtered subspace iteration, allowing the scaling to large problems and hardware systems. Finally, we compare the cumulative effects of the GPU, eigensolver, and matrix-matrix products and demonstrate cases where they are most helpful.

7.1 Background

Theory and Modeling

Chebyshev-filtered subspace iteration for self-consistent-field calculations is primarily motivated by the ability to reduce the cost associated with solving the eigenproblem at the center of self-consistent-field calculations [86].

From [90], we have the Kohn-Sham equations, which provide a one electron approximation of the Schrödinger with the energy dependent on a functional of the charge density

$$[-\frac{\nabla^2}{2} + V_{\text{total}}(\rho(r), r)]\Psi_i(r) = E_i\Psi_i(r). \tag{7.2}$$

These equations end up being nonlinear and using the notation for total potential from [86] we have

$$V_{\text{total}}(\rho(r), r) = V_{\text{ion}}(r) + V_H(\rho(r), r) + V_{XC}(\rho(r), r) \tag{7.3}$$

which depends on the charge density from Eq. 7.2. As such, a self-consistent-field loop is used, which terminates when the change in total potential is less than some tolerance, as seen in Alg. 10.

Solving the eigenproblem in Step 2 of Alg. 10 ends up being the most computationally expensive step for large problems. As such Chebyshev-filtered subspace iteration (seen in Alg. 11) was developed by Zhou et al. to replace the eigensolve in all but the first iteration [86]. Chebyshev-filtered subspace iteration begins with an eigenbasis corresponding to

Algorithm 10 Algorithm 2.1 for Self-Consistent Iteration from [86].

```
1: Initial guess for $\rho(r)$, get $V_{\text{total}}(\rho(r), r)$.
2: Solve Eq. 7.2 for $\Psi_i(r), i = 1, 2 \dots$
3: Compute new charge density $\rho = 2 \sum_{i=1}^{n_{\text{occ}}} |\Psi_i(r)|^2$.
4: Solve for new Hartree potential $V_H$ from $\nabla^2 V_{\text{H}}(r) = -4\pi\rho(r)$.
5: Update $V_{\text{XC}}$ and $V_{ion}$; get new $\tilde{V}_{\text{total}}(\rho, r) = V_{\text{ion}}(r) + V_H(\rho, r) + V_{\text{XC}}(\rho, r)$ (often fol-
   lowed by a potential-mixing step).
6: If $||\tilde{V}_{\text{total}} - V_{\text{total}}|| < tol$, stop; Else, $V_{\text{total}} \leftarrow \tilde{V}_{\text{total}}$, goto 2.
```

the occupied states of the Hamiltonian, which is then iteratively improved via Chebyshev filtering.

The size of the matrices involved in Chebyshev-filtered subspace iteration are often highly non-square. In particular, the Φ matrix ends up being the largest matrix, with the number of rows equal to the number of finite-difference points, N_{fd}, and the number of columns equal to the number of states N_{states}, where $N_{fd} >> N_{states}$. This results in an inefficient parallelization when using traditional distributed matrix-matrix products. Due to this aspect ratio, different kernels will have different ideal distributions, as discussed next.

Algorithm 11 Algorithm 4.2 For SCF Loop with CheFSI from [86].

```
1: Initial guess for $\rho(r)$, get $V_{total}(\rho(r), r)$.
2: Solve Eq. 7.2 for $\Psi_i(r), i = 1, 2 \dots$.
3: Compute new charge density $\rho = 2 \sum_{i=1}^{n_{\text{occ}}} |\Psi_i(r)|^2$.
4: Solve for new Hartree potential $V_H$ from $\nabla^2 V_{\text{H}}(r) = -4\pi\rho(r)$.
5: Update $V_{XC}$ and $V_{\text{ion}}$; get new $\tilde{V}_{\text{total}}(\rho, r) = V_{\text{ion}}(r) + V_H(\rho, r) + V_{\text{XC}}(\rho, r)$ (often
   followed by a potential-mixing step).
6: *If* $||\tilde{V}_{total} - V_{total}|| < tol$, *stop; Else,* $V_{total} \leftarrow \tilde{V}_{total}$, *call Alg. 12 to get* $s$ *approximate
   wave functions; goto 3.*
```

We will focus on four kernels, the Hamiltonian, $H\Phi$, the two sets of matrix-matrix products (projection and subspace rotation), and the eigensolve. The Hamiltonian H matrix is of size $N_{fd} \times N_{fd}$, but is applied as an operator, the eigensolve is of size $N_{states} \times N_{states}$ which results in the same sized V matrix, while the Φ matrix is $N_{fd} \times N_{states}$. We observe that for the Φ matrix, the different kernels have different preferred distributions. In the case of the Hamiltonian operator, a band parallel (column) distribution can be used

Algorithm 12 Algorithm 4.1 For Chebyshev-Filtered Subspace Method From [86].

```
1: Get the lower bound $b_{\text{low}}$ from previous Ritz values (use the largest one).
2: Compute the upper bound $b_{\text{up}}$ of the spectrum of the current discretized Hamiltonian $H$.
3: Perform Chebyshev filtering on the previous basis $\Phi$, where $\Phi$ contains the discretized wave functions of $\Psi_i(r), i = 1, \ldots, s$ : $\Phi = \texttt{Chebyshev-filter}(\Phi, m, b_{\text{low}}, b_{\text{up}})$ (Alg. 13).
4: Orthonormalize the basis $\Phi$
5: Perform the Rayleigh-Ritz step:
   (a) Compute $\hat{H} = \Phi^T H \Phi$.
   (b) Compute the eigen-decomposition of $\hat{H}$ : $\hat{H}Q = QD$ where $Q$ contains the eigenvectors of $\hat{H}$, $D$ contains the non-increasingly ordered Ritz values of $H$ .
   (c) 'Rotate' the basis: $\Phi := \Phi Q$.
```

Algorithm 13 Algorithm 4.3 For Chebyshev-Filter From [86].

```
1: Purpose: Filter vectors in $X$ by a $m$ degree Chebyshev polynomial that dampens the interval $[a, b]$, and output the filtered vectors in $Y$.
2: $e = (b - a)/2; c = (b + a)/2;$
3: $\sigma = e/(a - c);$
4: $\sigma_1 = \sigma$
5: $Y = (HX - cX)\sigma_1/e;$
6: for $i = 2 : m$ do
7:     $\sigma_2 = 1/(2/\sigma_1 - \sigma);$
8:     $Y_{new} = 2(HY - cY)\sigma_2/e - \sigma\sigma_2 X;$
9:     $X = Y;$
10:    $Y = Y_{new};$
11:    $\sigma = \sigma_2;$
12: end for
```

effectively, as all columns can be computed independently. On the other hand, splitting the finite difference domain (row decomposition) reduces communication for the matrix-matrix product but requires communication when applying the Hamiltonian. Furthermore, as mentioned earlier, the Φ matrix is of size $N_{fd} \times N_{states}$ with $N_{fd} >> N_{states}$. As such, additional care needs to be taken to ensure that the data distributions strike a balance for both computational load and communication during operations involving it. Notably, traditional distributed matrix-matrix methods fail to maintain high efficiency in such cases. In the following sections, we provide a background on the different components that we focus on – the matrix-matrix products, the eigensolves, the Hamiltonian, and the use of GPUs to accelerate these computations.

7.1.1 3D Matrix-Matrix Products

The first component which we will discuss is three-dimensional matrix-matrix products. Three-dimensional matrix-matrix products show promise for achieving optimal communication and computation. Advanced methods such as SUMMA, 2.5D, 3D, CARMA, and COSMA achieve lower I/O during the performance of matrix-matrix products, including ones with significantly different row/column sizes, such as those seen in SCF calculations [92, 93, 94, 95, 96, 97]. However, they tend to use custom data distributions, which introduce an overhead, making them difficult to use in general-purpose linear algebra libraries. In the case where we are already looking at specific distributions in this application, this leads to the question: can we see the benefits of 3D matrix-matrix products when we can control the data distributions?

Matrix-matrix products are a core computational kernel in linear algebra, computational science, and scientific computing (in addition to machine learning and many other domains). As such, significant work has been performed to allow for rapid computations of matrix-matrix products. In a serial CPU implementation (and others), one important consideration is the use of blocking to make the best use of cache lines, and vector operations. Efficient implementations are available such as those seen in BLAS, MKL, and ATLAS, which may take into consideration the micro-architecture the code is being run on, available cache space, and many other optimizations. [98, 99, 100]

When matrix-matrix multiplications are performed in distributed memory, additional considerations must be considered. In particular, one must balance computational efficiency (such that the amount of redundant calculations compared to the serial case is bounded), communication efficiency, and memory efficiency (to allow matrix-matrix products to scale to many processors). Furthermore, in the case which is faced in the Chebyshev-filtered subspace iteration framework, the matrices involved are very non-square, where with the number of rows much greater than the number of columns. Thus, selecting a matrix-matrix product that scales to very large number of nodes and can handle very non-square matrices is paramount to achieving an efficient implementation. Naive 1D and 2D partitionings fail to be both cost-optimal and memory optimal. Cannon's algorithm is a 2D algorithm that can achieve cost optimality and memory optimality at the expense of additional messages [101]. However, it does have drawbacks, namely requiring a square grid of processors. A Scalable Universal Matrix Multiplication Algorithm (SUMMA) is another 2D algorithm based on broadcast-multiply-roll algorithm with reduced memory usage. It has been integrated into the SCALAPACK distributed linear algebra package [97] [102]. 3D matrix-matrix products were introduced to allow increased parallelism by utilizing additional memory [92]. 2.5D Matrix-matrix products provide an interpolation between the 2D and 3D matrix-matrix products, which is optimal for the case of square matrices [96]. CARMA introduced a recursive partitioning that provides asymptotic optimality for all matrices and processor grids. However, the number of processors must be a square grid [93]. COSMA, in turn, provides a partitioning based on red-blue pebbling, which aims to provide an optimal decomposition in all scenarios [95].

We will now provide a brief review of 3D matrix-matrix multiplications before reviewing the CA3DMM method for matrix-matrix multiplications. The work involved in matrix-

matrix products can be modeled as a 3D cuboid, where the faces correspond with the input and output matrices $C = AB$. In turn, the cuboid can be partitioned into further cuboids where the volume of the cuboid corresponds with the amount of computation being performed, and the surface area corresponds with the amount of communication performed. However, with 3D algorithms, there are (without loss of generalization) $p^{1/3}$ copies for each matrix, which increases the memory required to perform the matrix-matrix multiplications. The upside is that with the data present on additional cores, communication is reduced compared to traditional 2D algorithms. The 2.5D algorithm, developed in [96], interpolates between these two methods, allowing for memory usage to be scaled to the available memory usage. Thus, it can be communication optimal for all memory sizes.

As an alternative to grid-based multiplications, breadth-first and depth-first search recursive algorithms have been developed, which approach the topology hierarchically. This results in algorithms that do not require tuning and are cache-oblivious. CARMA is an approach that provides an asymptotically optimal method which is also cache and network oblivious. It proceeds by recursively splitting the current problem along the largest dimension and solving the resulting subproblem with either a breadth-first or depth-first approach, depending on memory availability. The downside of this algorithm is due to its simple, greedy nature, where it splits along the largest dimension. This results in an increase in communication volume (by a constant factor $\sqrt{3}$).

COSMA is a method that, rather than splitting the matrix and mapping them to a tree, instead calculates an optimal sequential schedule and maps the schedule domain to the matrices. This results in a method that is I/O optimal for all matrix sizes and processor counts. The algorithm begins by using the red-blue pebble game to model data reuse, then parallelizes the schedule and derives a domain decomposition.

CA3DMM is an algorithm that breaks the matrix-matrix multiplication into square subproblems, allowing Cannon's algorithm to be used. It achieves this by determining a processor grid based on decomposing the original problem via a multi-objective optimization problem, maximizing the number of processors used while minimizing the communication. The subproblems are calculated using Cannon's algorithm, which requires some duplication of the input data, followed by a reduction. From [103], the algorithm is described in Alg. 14.

Algorithm 14 CA3DMM Algorithm from [103].

```
1: Generate the process grid $p_m \times p_n \times p_k + p_r$, and let $p_l := min(p_m, p_n), s \leftarrow max(p_m, p_n)/p_l$. Processes $P_{:,:,i_k}$ form a *mn-plane group*, processors $P_{i_m,i_n,:}$ form a *k-column group*
2: Redistribute A and B matrices from input layout to CA3DMM initial layout
3: If $s > 1$ split each mn-plane group in $s$ *Cannon groups* with $p_l \times p_l$ processes in each Cannon group, use allgather to duplicate A or B block between Cannon groups. If $s = 1$, a mn-plane group is a Cannon group.
4: All $p_k \times s$ Cannon groups perform 2D Cannon algorithm in parallel.
5: All $p_m \times p_n$ k-column groups reduce-scatter the partial C blocks to get the final C block
6: Redistribute C matrices to the specified layout
```

These advanced methods demonstrate significant wall time improvement on modern supercomputer architectures compared to traditional methods such as SCALAPACK, especially in cases with very non-square matrices. The downside of these more advanced methods is that they require custom specialized data structures and distributions, which are unlikely to be compatible with pre-existing scientific computing codes which commonly utilize the block-cyclic distribution used by SCALAPACK. Thus, they may not function well as a drop-in replacement due to the increased communication time required for data redistributions. The combination of the structure of calculations in Chebyshev-filtered subspace iteration being known ahead of time, and the matrices involved having large aspect ratios, with many nodes used in the computation, make it a promising application to test such advanced algorithms in comparison to the traditional block-cyclic structures.

7.1.2 Hamiltonian

The second component that we will discuss is the Hamiltonian. The application of the Hamiltonian Hx for some vector x could be explicitly formed, then performed via sparse-matrix operations. However, when the structure is known ahead of time, it is often more efficient (particularly on GPUs) to perform the matrix-matrix products as an operator – never explicitly forming the matrix. In our case, we will consider the operation

$$(a\nabla^2 + b\,\mathrm{diag}(v) + cI)x \tag{7.4}$$

where ∇ is the Laplacian, v is the effective potential, and the cI term is used for the diagonal shift during the Chebyshev filtering for scalars a, b and b.

Laplacian

The computation associated with the discretized Laplacian, in a serial environment, can be performed as a modified stencil update, as a finite difference discretization is used. In particular, only the elements along each of the three axes are required. As such, we can apply the stencil on the CPU quickly utilizing the simple algorithm seen in Alg. 15. The source domain refers to the domain for which the current processor calculates the results, while the extended domain refers to the domain extended by the halo exchange to include all the points required for the local calculations. However, while the algorithm can be parallelized pointwise trivially, this has a very low arithmetic intensity. Each element requires reading from $O(6r)$ points, with very poor spatial locality in two dimensions.

7.1.3 Eigensolver

The third component we will discuss is the eigensolver used within Chebyshev-filtered subspace iteration. As mentioned previously, while Chebyshev-filtered subspace iteration can reduce the computation required for the eigensolve, the eigensolve within the Chebyshev-filtered subspace iteration ends up being the most expensive operation for large problems. Thus, it is imperative to fully exploit the available hardware to solve the eigenproblem. There are several packages that provide eigensolves, which we will consider SCALA-

Algorithm 15 Stencil application with $y = (a\nabla^2 + b\,\mathrm{diag}(v) + cI)x0$, where ∇^2 and c are encoded in coefs. x is in the extended domain and $v0, y$ in source domain.

```
INPUT: Source domain $[x, y, z]^{spos}$ to $[x, y, z]^{epos}$, with extended domain
  $[x, y, z]^{spos}_{ex}$ to $[x, y, z]^{epos}_{ex}$
  for $k = z^{spos}, kp = z^{spos}_{ex}; k < z^{epos}; k ++ kp ++$ do
      for $j = y^{spos}, jp = y^{spos}_{ex}; j < y^{epos}; j ++ jp ++$ do
          for $i = x^{spos}, ip = x^{spos}_{ex}; i < x^{epos}; i ++ ip ++$ do
              $res \leftarrow 0$
              for $r = 1 \ldots r$ do
                  $res_x = (x0[kp][jp][ip + r] + x0[kp][jp][ip - r]) * coefs^x[r]$
                  $res_y = (x0[kp][jp + r][ip] + x0[kp][jp - r][ip]) * coefs^y[r]$
                  $res_z = (x0[kp + r][jp][ip] + x0[kp - r][jp][ip]) * coefs^z[r]$
              end for
              $y[k][j][i] \leftarrow res + b(v0[k][j][i] * x0[kp][jp][ip])$
          end for
      end for
  end for
```

PACK, cuSOLVER, and ELPA due to their production-ready and efficient codes [104, 105, 106, 107].

SCALAPACK is the de-facto library for distributed memory linear algebra [105]. It supports a variety of operations, including eigensolves via multiple algorithms. In particular, we will look at the expert driver, which uses bisection once the matrix has been tridiagonalized. ELPA provides a distributed eigensolver and provides a two-stage tridiagonalization approach [107, 104]. This two-stage approach attempts to increase the number of Level-3 operations, reducing the bottleneck created by the memory-intensive single-stage tridiagonalization. Furthermore, a GPU implementation is provided for ELPA. Finally, cuSOLVER is the eigensolver provided by NVIDIA for their GPUs [106]. cuSOLVER provides a Jacobi-based eigensolver and a divide-and-conquer-based eigensolver for a single GPU. Despite this single GPU limitation, we find that cuSOLVER performs very well for problems of up to around 20,000 states compared with distributed CPU codes.

7.1.4 GPU

The final component we will discuss is using GPUs to accelerate the computationally expensive kernels. From Alg. 12, we see that the kernels used in Chebyshev-filtered subspace iteration can be formulated as either dense or structured linear algebra computations. As such, the use of accelerators such as Graphics Processing Units is promising.

Graphics Processing Units (GPUs) have become a common addition to many High-Performance Computing (HPC) clusters due to their ability to provide very efficient computations for computationally intensive tasks, with General Purpose GPUs considered as powerful Single Instruction Multiple Threads (SIMT) accelerators. The computational strength of GPUs results from the many (on the order of thousands) of threads available. This makes GPUs particularly well suited for dense or structured linear algebra kernels. The GPUs seen in HPC systems typically have a high-bandwidth memory separate from that used by the CPU. However, due to this separate memory pool, there exists a significant cost associated with transferring data from the CPU main memory to the GPU memory and vice versa. As such, to design particularly practical implementations, care must be taken in selecting what data is transferred and when. When such care is taken, great performance uplift can be seen – for example, the NVIDIA V100 GPU can achieve 7000 double-precision GFLOPS. In comparison the Intel Xeon 6226 Gold CPU can achieve a theoretical 1037 GFLOPS [108] [1].

The use of accelerators such as GPUs provides the potential for significant speedup compared to CPU-only codes due to their raw power. For example, over 96% of Sierra's 125 petaflops of peak performance comes from GPUs [110]. However, many high-level considerations must be taken when writing GPU codes compared to CPU codes. First, the cost of allocating memory is significantly higher on GPUs compared to CPU memory allocations. As such, one typically will want to limit the amount of dynamic memory allocation done. Second, data transfers between CPU and GPU memory tend to be expensive

[1] 32 Double-precision operations per cycle (2 AVX-512 Units) * 12 Cores * 2.4Gcycles/second [109]

and have high latency. Thus, it tends to be preferred to limit the number of transfers, doing as much computation as possible on the GPU before returning data to the CPU. Third, because of the architecture of GPUs, they perform best when threads are performing the same operations, and memory access is coalesced. Finally, GPUs have significantly more threads than CPUs, so care must be taken in selecting algorithms that allow for parallelism and high levels of thread occupancy. We have designed our computation engine with these and other considerations in mind to allow for a performance increase.

We have created an implementation of Chebyshev-filtered subspace iteration, which uses the components discussed, including advanced matrix-matrix products and distributed GPU-based calculations, to achieve high performance. We first created a distributed memory CPU code to investigate how best to create a holistic implementation of Chebyshev-filtered subspace iteration. Our implementation uses a band + domain distribution for the Φ matrix (the largest dense matrix used in calculations), allowing for large-scale parallelism. This code uses the CA3DMM 3D matrix-matrix product library [103] for distributed memory matrix-matrix products. Furthermore, it includes support for distributed nodes with GPUs, which aims to reduce the number of transfers of Φ size matrices between the CPU and GPU. We can use this implementation to observe the relative timings of the different kernels and measure changes with different eigensolvers and observe the scaling with problem sizes and the number of ranks and nodes. This platform allows us to investigate the use of CA3DMM or other 3D matrix-matrix products in SCF iterations.

7.2 Implementation

In this section, we discuss the implementation of libPCE, which is then used to provide the results in the next section. This section begins by discussing considerations for the GPU implementation and the kernels involved and going into detail on the Hamiltonian and GPU Laplacian implementation. Next, an investigation into the eigensolvers is performed to understand when each eigensolver may be preferred.

7.2.1 GPU

We begin with a discussion of practical considerations for GPU acceleration. As mentioned previously, the package developed provides computational routines which can be used in SCF-based chemistry codes, including SPARC [8]. When using SPARC for quantum molecular dynamics (MD), each MD step performs an SCF loop to determine the energies and properties involved. The GPU implementation has been designed such that the Φ matrix, which is the largest matrix involved in computations, is only transferred to and from the GPU at the beginning and end of the SCF loop – a single time per MD step. Future implementations may be able to transfer only at the initial step, updating the data in place on the GPU during force and related computations. We will consider four kernels that use the Φ matrix for the computations as follows: The matrix-matrix products (including subspace rotation and projection), the Laplacian operator, the nonlocal operator, and the eigensolve. Two GPU implementations have been developed for CA3DMM, one of which is prioritized when GPU memory usage is preferred, and the other when reducing the wall time is prioritized. In the first case, the implementation simply uses cuBLAS rather than BLAS to provide local matrix-matrix products. In the latter case, a multiple buffer approach is used, allowing the communication and computation to be overlapped. Even the memory-efficient approach results in a single-node performance increase from around 1.3 double-precision TFLOPS to 5.1 double-precision TFLOPS for multiplication of two matrices of size 10,000. This experiment was performed with two Intel Xeon 6226 Gold CPUs per node and NVIDIA V100 GPUs. In a preliminary test, we found a 1.4x speedup on a single node compared to the reference CPU implementation (SPARC).

For small problems (with an eigensolve of no more than a few thousand by a few thousand), the most expensive operation is observed to be the Chebyshev filtering. The primary kernel associated with the Chebyshev filtering is applying the Hamiltonian operator many times. This Hamiltonian is further decomposed into two parts, the nonlocal operator and the Laplacian operator. The Laplacian operator consists of a stencil that is applied to each point in the finite-difference domain. As the stencil has a radius, in the distributed case, when the data is distributed by domain, data from neighboring processors is required for a local processor to compute its edge elements. Due to its shape, the set of points from the neighboring processors is known as the halo, and the exchange of such data is called the halo exchange. To perform the Laplacian operator, first, a data redistribution corresponding with the halo exchange is performed, followed by a stencil application. The halo exchange on GPUs is parallelized using GPU streams, which allows for the packing and unpacking of data to be performed in parallel at the coarse-grain level of each neighbor and the fine-grain level of individual finite-difference points. When compared to the CPU implementation, where it was found that overlapping the computation of the local part with communication of the halo domain, we found that the GPU implementation performed better where the entire communication was performed first, followed by the computation of the whole domain. This is likely due to the overhead associated with applying the stencil to many small domains.

The stencil calculation needs to be handled with care due to its relatively low arithmetic intensity. For a stencil with radius r, $O(6r)$ doubles must be read, where the spatial locality of the data is poor. [111] presents a 3D finite-difference code on GPUs which utilizes shared memory and can allow the reuse of elements once read. However, it is insufficient for our requirements – it does not consider the halo domain, does not allow the use of different coefficients for each dimension and does not parallelize over the columns. Thus, we have developed an efficient Laplacian application that can apply the stencil and the other functions required for the Laplacian and Chebyshev filtering. In this section, we develop an efficient Laplacian application based on the 3D finite differences code mentioned above. The two main ideas in this implementation are to maximize the thread occupancy of the operation, achieving enough parallelism to saturate the threads of the GPU while minimizing the amount of reading from global data (due to the cost associated with such reads).

This is achieved by using the shared memory available on GPUs, which allows for threads in the same thread block to access memory that is shared between the threads, but with much faster access than global memory. In particular, we can exploit this by having a thread wavefront move over the domain, where each thread will store the data corresponding with its active element into shared memory, amortizing the reads. By this, we mean that all active threads will be considering a plane in space, which advances at each step. Thus, the nearby $4r$ threads will be able to read from shared memory, rather than from global memory, for that data.

In practice, as on CPU, when performing $y \leftarrow (a\nabla^2 + b\,\mathrm{diag}(v) + c)x$, the x vector is in the halo-exchange extended domain, while v and y are in the source domain, with a, b, c being scalars. Furthermore, we apply this to the many columns of Φ at once, independently and in parallel (as will be discussed). To achieve high occupancy, we want to have many threads active (up to the limit of one thread per input element), while drawing from shared memory as much as possible. Thus, to accommodate for the limited shared memory, we parallelize over 2D slices (planes) in space, where we store the actively considered elements in shared memory.

Pseudocode for a single thread can be seen in Alg. 16. Observe how the thread front progresses over the z dimension, reading from the global memory, storing it into shared memory, calculating the resulting value, then storing it back into global memory. However, while this expands upon [112] by having a different coefficient per direction, as well as handling b and v, this pseudocode does not include the changes required for a distributed memory code, where elements that need to be calculated require halo elements which themselves do not need to be calculated. We can further add parallelism over the reference implementation (SPARC), by utilizing the 3D CUDA grid, by assigning the third dimension to the columns of Φ, which allows for all columns of Φ to be calculated in parallel, further increasing the occupancy and thus performance.

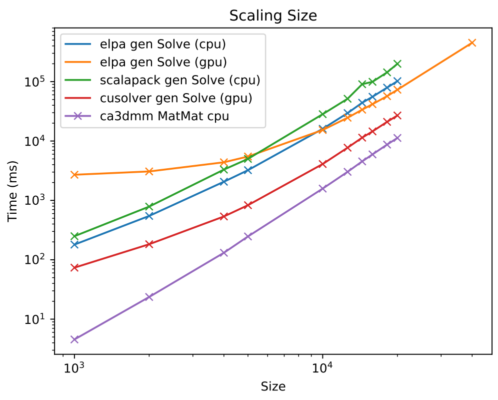
Algorithm 16 GPU Pseudocode, see Alg. 15 for more details

```
for each $z$ from $z^{spos}$ to $z^{epos}$ do
    Read $current$ element from global memory $x_{ex}$
    Store $current$ into shared memory
    $res \leftarrow coef[0] * current$
    for $r = 1; r <= radius; r++$ do
        $res+=$ the elements at distance $r$ away (scaled by the relevant coefficient), from shared memory in the $x$ and $y$ directions, but from global memory for $z$ direction
    end for
    $val \leftarrow res + b(v0[idx] * current)$
    $y[idx] \leftarrow val$
end for
.
```

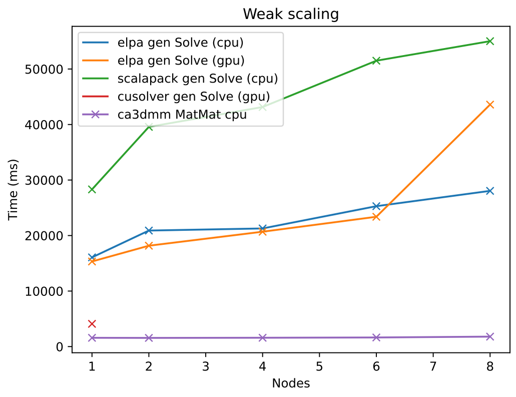
7.2.2 Eigensolver

As previously mentioned, we consider a few eigensolver packages: SCALAPACK, cuSOLVER, and ELPA. Each of these eigensolvers targets different regimes, and to effectively utilize them, it is imperative to understand in which cases they work well and which they do not. in this section, we investigate the impact of problem size and the number of nodes for various problems to investigate the tradeoffs between the eigensolvers.

In Fig. 7.1, we compare these eigensolvers for varying problem sizes and demonstrate the CA3DMM matrix-matrix product of the same size. The matrices tested are random symmetric positive definite matrices of the appropriate size. We can see that for problem sizes under 20k states, the most rapid eigensolve is cuSOLVER, for which the data starts and ends on the same GPU. In the case of the ELPA GPU solver, until the eigensolve gets to around 40k states, the overhead associated with the GPU computations makes it less preferable than the ELPA CPU solver. Notably, the ELPA GPU solver is only GPU accelerated – some computation is done on the CPU, and ELPA expects the data to start on the CPU. Thus, it requires an additional data transfer in the libPCE case where the data would otherwise be on the GPU already. We see in Fig. 7.1b that the CA3DMM matrix-matrix product has near-perfect weak scaling, while the ELPA eigensolves achieve fairly good scaling, with the CPU scaling better than the GPU. The SCALAPACK eigensolver



(a) Scaling the problem size for a single node.



(b) Weak scaling for a combination of node/problem sizes.

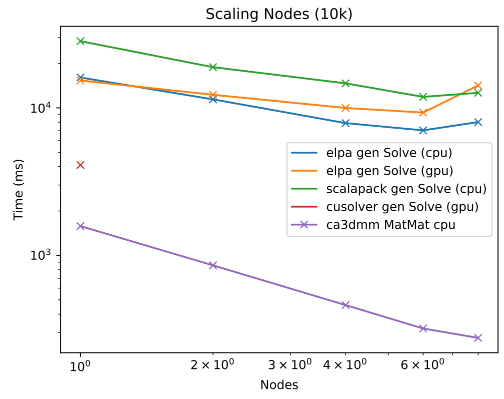
Fig. 7.1: The scaling of different methods for different size problems.

has the worst scaling.

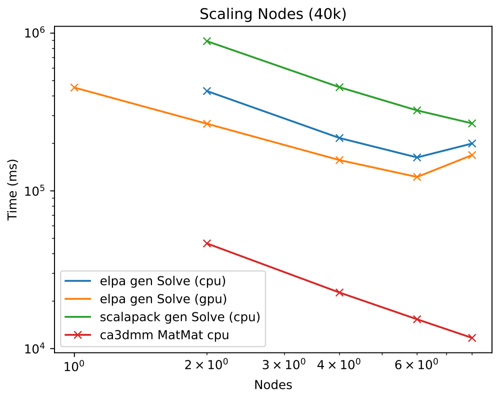
In Fig. 7.2, we see the scaling of the various methods when holding the problem size constant. In the 10k case, the cuSOLVER is the fastest, even compared to the second-fastest method with the optimal number of nodes. However, cuSOLVER on the tested GPU runs out of memory around a problem of size 20k. For the 40k problem, the problem size per GPU is large enough that ELPA GPU can outperform the CPU version. These results demonstrate that the tested eigensolvers do not sufficiently scale to many nodes. We see in Fig. 7.1b that the matrix-matrix product provided by CA3DMM can scale to many nodes well.

7.3 Single Node Results

We present single node results in Fig. 7.3 which compare libPCE against the reference implementation (SPARC). The problem considered is Al500, which has 900 states, corresponding with an eigensolve of size 900×900 and a finite difference domain of size 50^3. We see that the most expensive operation, the Chebyshev filtering, is reduced from 23.38 seconds to 12.56 seconds for this problem. In a larger problem, the eigensolve would take a larger portion of time, which results in the need for an efficient eigensolve, as investigated



(a) Scaling for a problem of size 10k.



(b) Scaling for a problem of size 40k.

Fig. 7.2: The scaling of different methods for a various number of nodes.

in the previous section.

7.4 Block-Cyclic Implementation

To understand how 3D matrix-matrix products compare to the traditional block-cyclic distribution, a block-cyclic implementation was created. This section describes the design, design decisions, and benefits and drawbacks of the block-cyclic implementation.

The Φ matrix for the Chebyshev filtering is stored in a band+domain "hybrid" decomposition. As such, a redistribution must be performed to be used in a block-cyclic case. While a curious overview of the hybrid distribution may appear as though it is a block (but not cyclic) distribution, as each block is a contiguous part of the matrix, it is incompatible with the block-cyclic distribution as the row sizes of each block correspond with the domain, and thus may not be the same for each block. Combined with the requirements of SCALAPACK to perform calculations, where the same processor "context" must be used for each matrix in matrix-matrix products (pdgemms), several different matrix layouts are involved.

First, a pure block-cyclic distribution of Φ is desired, which allows for rapid evaluation of, for example, $\Phi^T\Phi$, $\Phi^T H\Phi$ and ΦV. This block-cyclic distribution uses the same processor grid as the hybrid decomposition but ensures that all blocks are the same size and uses a

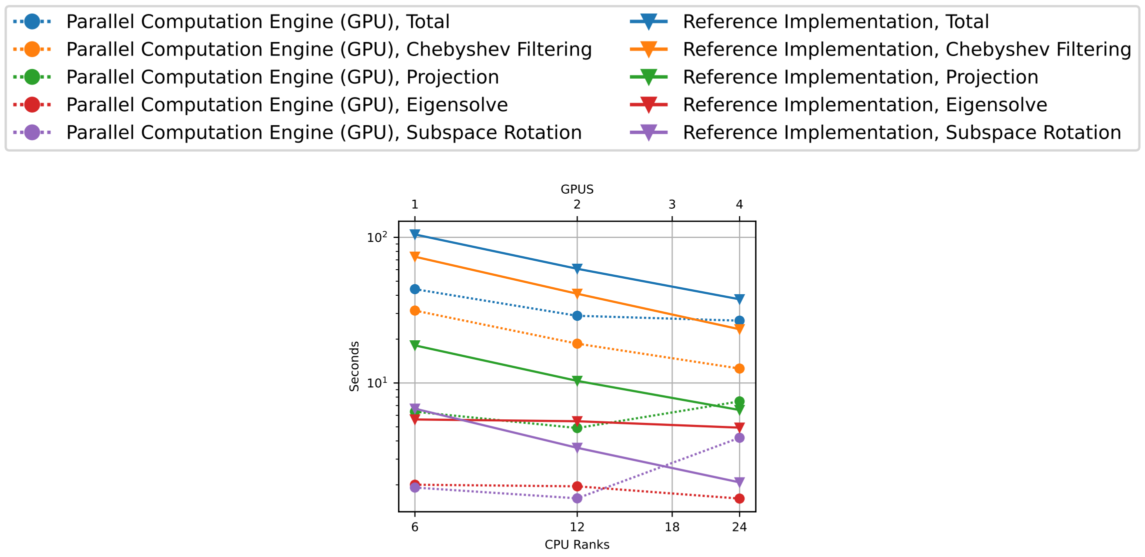


Fig. 7.3: A comparison of the GPU implementation of the Parallel Computation Engine against the reference implementation.

default block size of the minimum of 64, or $\lceil \frac{N_{fd}}{NProcs} \rceil$ in each direction. This processor grid will be called the Φ-$grid$. The results of $\Phi^T\Phi$ and $\Phi^T H\Phi$ are both square matrices, which will be used in the eigensolve, indicating that we would want a block-cyclic square grid distribution. Thus, we select the largest processor square grid of the available processors, which we will refer to as the $square$-$grid$. Similar to the previous block-cyclic distribution, the block size is selected as the minimum of 64 and $\lceil \frac{Ncols}{Nprocs} \rceil$ for the process grid.

To transfer from Φ_{hybrid}, the hybrid distribution of Φ, to Φ_Φ currently requires two steps. One to transfer Φ_{hybrid} to a block (not cyclic) distribution, which will then allow the use of SCALAPACK's $pdgemr2d$ to redistribute to a block-cyclic distribution Φ_Φ.

Unfortunately, due to the constraints required by SCALAPACK, performing $M \leftarrow \Phi^T\Phi$ and $H \leftarrow \Phi^T(H\Phi)$, has Φ on the Φ-$grid$, while we would want M, H to be on the $square$-$grid$. Thus, we must create temporary distributions M, H, which are square matrices but on the Φ-$grid$ of processors. Thus, we perform $M_\Phi \leftarrow \Phi^T\Phi$, before transferring M_Φ to the square grid, and similar for $H_\Phi \leftarrow \Phi^T\hat{H}$. Note that we can reuse the memory for M_Φ and H_Φ as they have the same distribution.

Once M, H are both distributed cyclically (on $square$-$grids$), SCALAPACK's distributed eigensolve $psdygvx$ can be used, with the resulting eigenvectors V distributed in the same fashion of M and H, on the $square\text{-}grid$. However, once again, we must again have a redistribution to perform $\Phi_\Phi \leftarrow \Phi_\Phi V$, as V is distributed as V_{square}. Thus, we transfer V to the Φ-$grid$, V_Φ, allowing us to perform $\Phi_\Phi \leftarrow \Phi_\Phi V_\Phi$. We then transfer Φ back to the hybrid grid for future iterations.

GPU

When GPU support is requested for libPCE with block-cyclic matrix-matrix products, the Φ matrix in the hybrid distribution is stored in GPU memory to allow for the rapid evaluation of the Chebyshev filtering. However, in the block-cyclic implementation, where SCALAPACK is used for the matrix-matrix products, we must ensure that the data is on the CPU for the matrix-matrix products. Due to the effectiveness of the cuSOLVER eigensolver, which solves on a single GPU but is very performant, we must ensure that the data is on the memory of a single GPU for the eigensolve portion.

Fortunately, the modifications required to (1) allow the use of SCALAPACK when the data is initially on the GPU and (2) to allow the use of cuSOLVER for rapid eigensolves are simple. First, we can transfer the Φ_{hybrid} data from the GPU to the CPU initially and in parallel without any communication. Once the data is on the CPU, the SCALAPACK matrix-matrix products proceed similarly to the CPU-only regime. The only difference arises from the need to have all the data on a single rank for the eigensolve. Fortunately, we can achieve this again without any additional redistributions compared to the CPU-only case. Rather than redistribute V from the Φ-$grid$ to $square\text{-}grid$, we instead redistributed to a $root\text{-}grid$, where only a single node has data. As the $square\text{-}grid$ is only used for the eigensolve, this is a simple drop-in replacement. As in the CPU case, after the eigensolve, we will redistribute V from the $root\text{-}grid$ to the Φ-$grid$. Finally, once $\Phi \leftarrow \Phi V$ has been performed, the Φ matrix must be returned to the CPU.

The largest drawback of this method is that, as the number of GPUs on a system tends

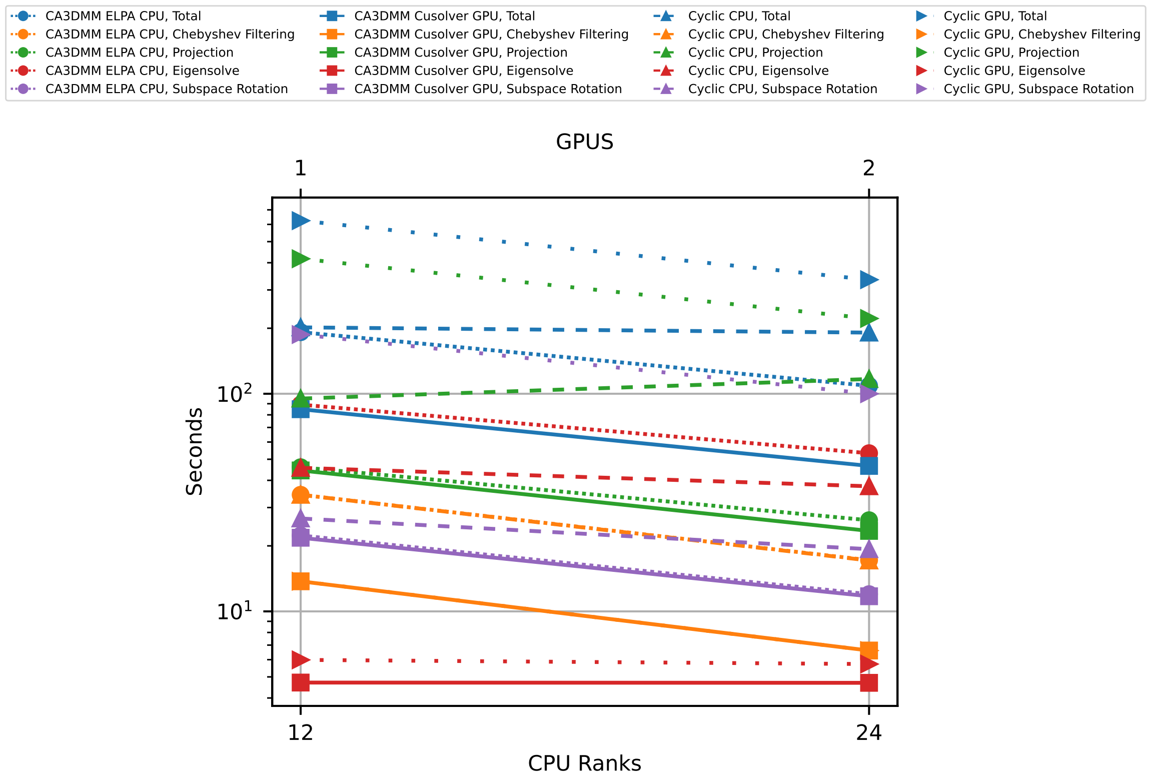


Fig. 7.4: A comparison of using block-cyclic matrix-matrix products and CA3DMM matrix-matrix products.

to be significantly less than the number of CPUs, and the matrix-matrix products are performed using SCALAPACK (CPU only), the matrix-matrix products take a considerable amount of time. A simple way to alleviate this would be to provide a distributed memory GPU matrix-matrix product that uses the block-cyclic structure.

We display results comparing libPCE with the different matrix-matrix products and eigensolve backends in Fig. 7.4. The first run uses CA3DMM on the CPU with 24 ranks and uses ELPA for the eigensolve, titled “CA3DMM ELPA CPU”. The second is CA3DMM on the GPU with two ranks (and 1 GPU per rank) using cuSOLVER on a single GPU for the eigensolve, titled “CA3DM cuSOLVER GPU”. Next are the cyclic methods. The first is entirely on the CPU using block-cyclic SCALAPACK matrix-matrix products, using SCALAPACK as the eigensolver, titled “Cyclic CPU”. The final is using block-cyclic SCALAPACK matrix-matrix products on the CPU, using cuSOLVER on the GPU for the eigensolve, title “Cyclic GPU”. Note that this results in few CPU threads being used in the matrix-matrix products.

As expected, the time taken for the eigensolve in the cyclic and CA3DMM methods remains close to the same, as they are both using cuSOLVER as the eigensolver. The cyclic GPU subspace rotation takes more time than the cyclic CPU and CA3DMM cases due to the need to redistribute the data back to hybrid form and transfer the resulting data to the GPU. We observe that the Cyclic GPU matrix-matrix products (projection and subspace rotation) take a significant amount of time. This is as the matrix-matrix products are being performed on the CPU, of which there are only two ranks. The Chebyshev filtering times between cyclic and CA3DMM remain the same, as there are no differences between them.

7.5 Multi-Node Multi-GPU Numerical Experiments

In this section, we utilize the methods built up in the previous sections to perform large-scale experiments to demonstrate the effectiveness of a multi-node multi-GPU distributed Chebyshev-filtered subspace iteration implementation on GPUs. The experiments are performed on a system with 256 aluminum atoms, with an electronic temperature of 100,000k and ionic temperature of 100,000k, and unless noted otherwise, a cell size of 30.605 with a mesh spacing of 0.75 (a 41^3 finite difference domain) and 10,000 states. Unless noted otherwise, there are 2 GPUs per node, 24 CPU processes per node, with 13 nodes – leading to an eigensolve grid of size 5^2 in the GPU case or 17^2 in the CPU case. These were performed on the Phoenix cluster at Georgia Institute of Technology, with 2 NVIDIA V100s per node and 2 Intel Xeon 6226 Gold processors per node. The numerical experiments performed include scaling the number of finite-difference points and scaling the number of states. Multiple methods are compared, including using ELPA, SCALAPACK, and cuSOLVER to perform the eigensolve, using SCALAPACK and CA3DMM for the matrix-matrix products, and using the CPU or GPU implementation. In the case of SCALAPACK matrix-matrix products, a block-cyclic distribution is used.

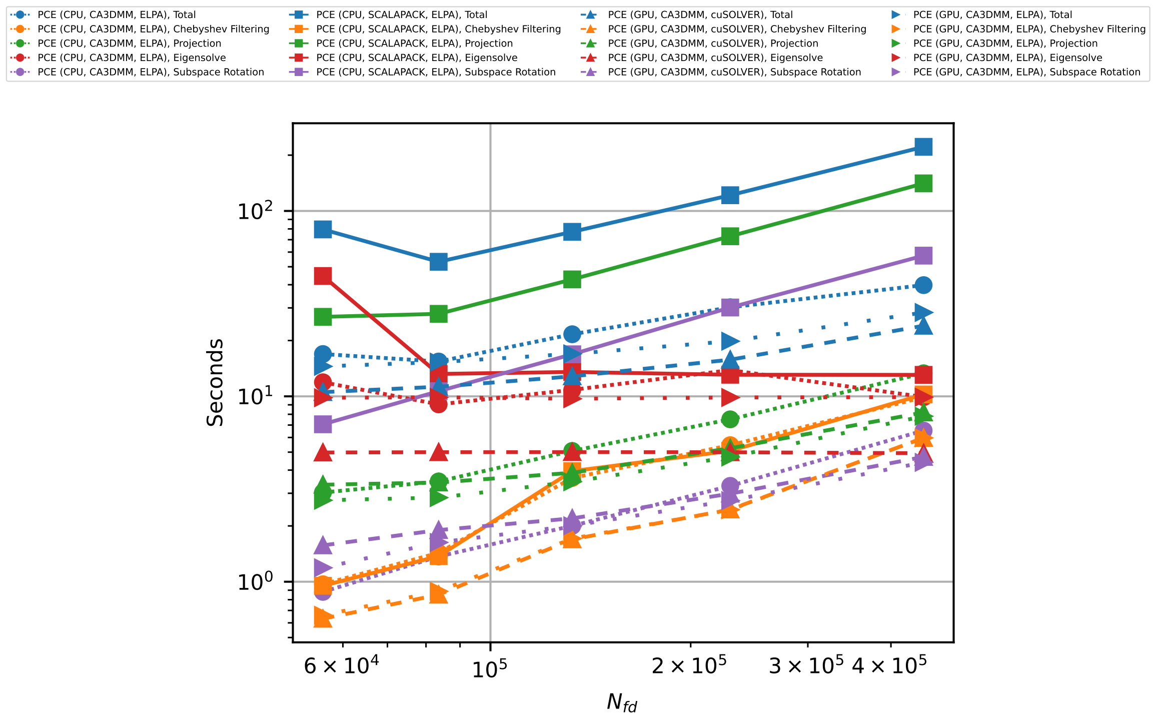
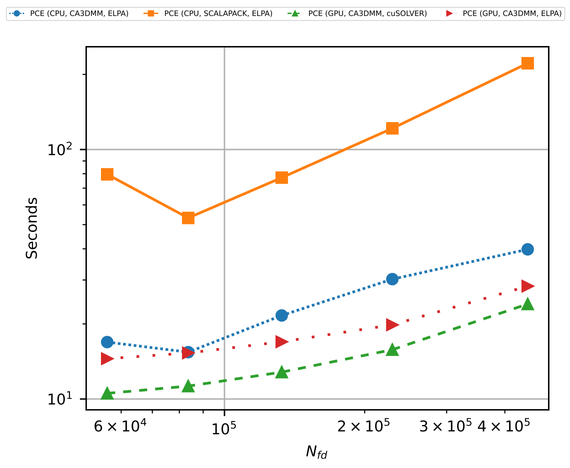


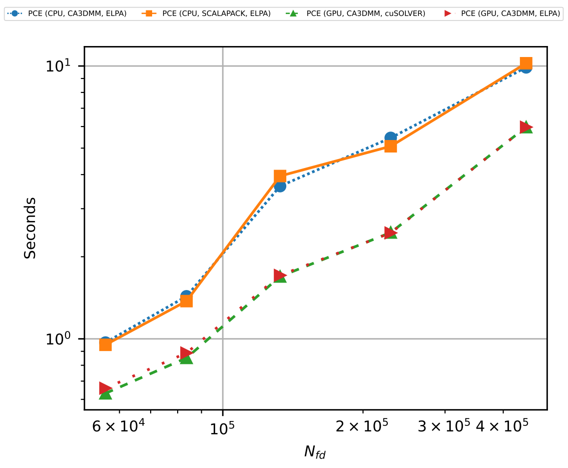
Fig. 7.5: A comparison of SCALAPACK-based and CA3DMM-based methods, varying the number of finite-difference points.

7.5.1 Scaling the Finite-Difference Domain

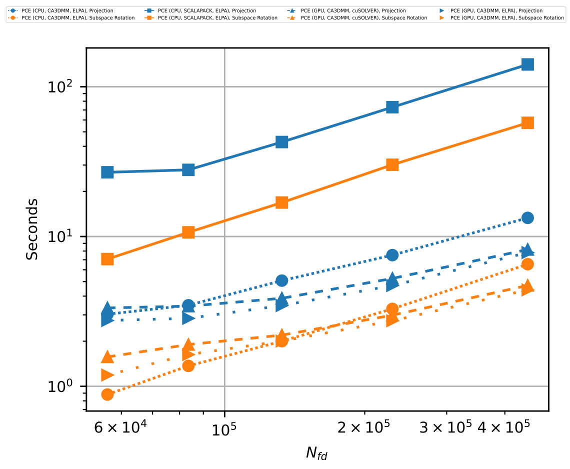
In the set of experiments shown in Fig. 7.5, we scale the number of finite-difference points. A breakdown of the experiments split by the kernel can be seen in Fig. 7.6. When considering the Φ matrix, increasing the number of finite-difference points (N_{fd}) results in changing the number of rows, increasing the so-called aspect ratio of the matrix. The number of finite-difference points ranges from $38^3(54872)$ with the most coarse mesh spacing of 0.8 to $77^3(456533)$ in the finest mesh spacing of 0.4. We observe in Fig. 7.6c that the CA3DMM matrix-matrix products on the CPU perform significantly better than the SCALAPACK-based products on the CPU. We observe that in Fig. 7.6a for the most fine case, CA3DMM with cuSOLVER performs the best, corresponding with the largest aspect ratio. As the grid becomes coarser, the benefits seen by the GPU and CA3DMM become less prominent, as the overhead associated with CPU computation increases relative to the



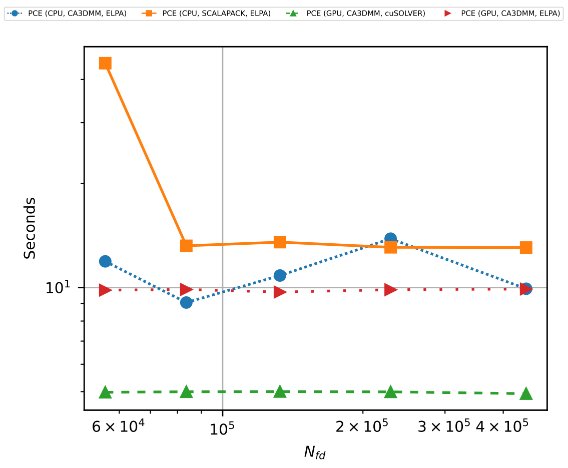
(a) The total time taken.



(b) The time taken for Chebyshev filtering.



(c) The times taken for projection and subspace rotation.



(d) The times taken for the eigensolve.

Fig. 7.6: The computation time associated with different kernels for different finite-difference grids.

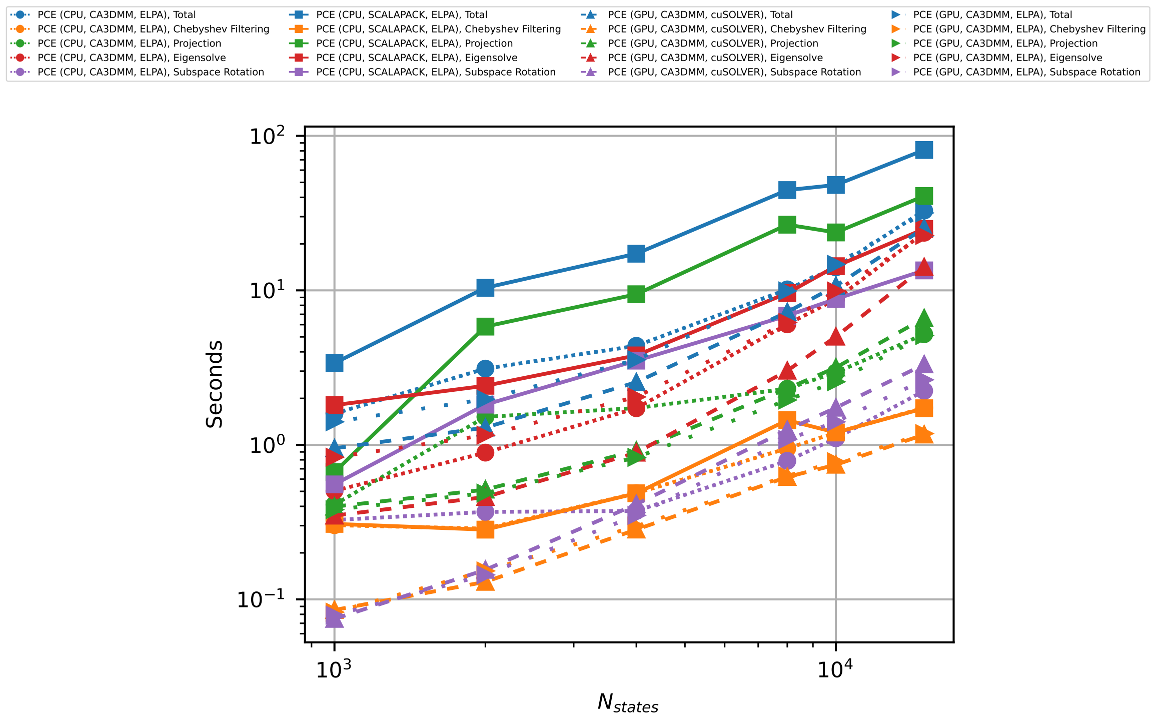


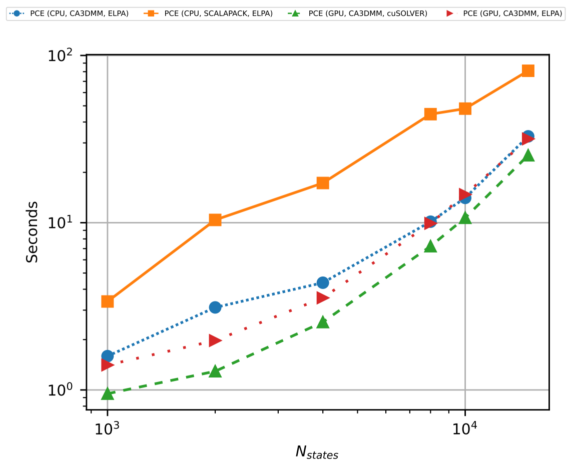
Fig. 7.7: A comparison of SCALAPACK-based and CA3DMM-based methods, varying the number of finite-difference points.

problem size, and the aspect ratio becomes closer to one.

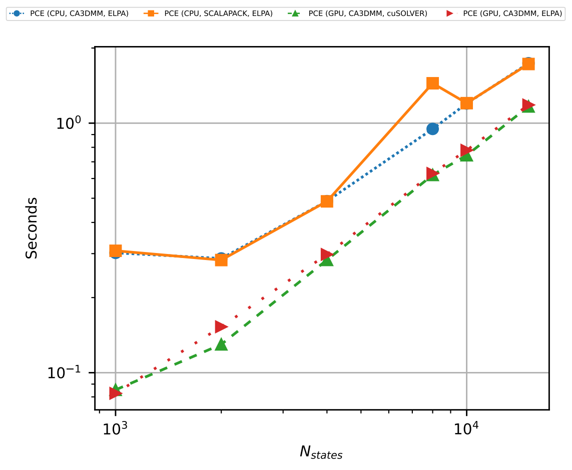
7.5.2 Scaling the Number of States

In the set of experiments shown in Fig. 7.7, we scale the number of states, which scales the number of columns of the Φ matrix. A breakdown of the experiments split by the kernel can be seen in Fig. 7.8. When considering the Φ matrix, increasing the number of states results in changing the number of columns, decreasing the so-called aspect ratio of the matrix. Notably, this increases the size of the eigensolve.

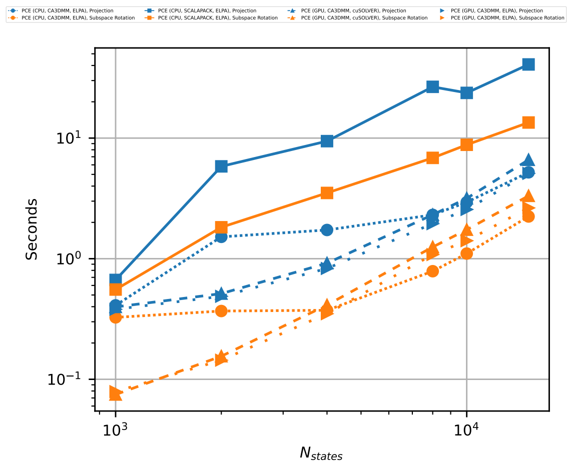
We observe in Fig. 7.8b that the choice of eigensolve does not significantly impact the Chebyshev filtering time and that the GPU methods outperform the CPU methods for all tested cases. We observe in Fig. 7.8c that on the CPU, CA3DMM outperformed SCALAPACK even with the aspect ratio closest to one. Furthermore, we observe that for the larger aspect ratios, the GPU outperforms the CPU. However, when the number of states



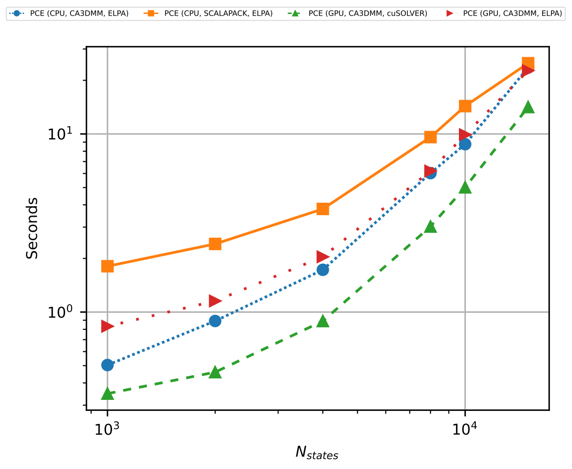
(a) The total time taken.



(b) The time taken for Chebyshev filtering.



(c) The time taken for the projection and subspace rotation.



(d) The time taken for the eigensolve.

Fig. 7.8: The computation time associated with different kernels for different number of states.

increases, we do see that CA3DMM on the CPU outperforms the GPU. In Fig. 7.8d we observe that the most rapid eigensolver is cuSOLVER on the GPU, and except for the largest problems that ELPA on the CPU outperforms ELPA on the GPU. This aligns with the eigensolver tests shown in the previous section. Cumulatively, we can see in Fig. 7.8a that across the entire range, the most rapid total time is demonstrated by libPCE on the GPU using CA3DMM and cuSOLVER.

7.6 Conclusions

These results and the discussed considerations show cases where advanced matrix-matrix products and GPUs can outperform their traditional counterparts, despite the overhead they require. In the case of advanced matrix-matrix products, as the matrices being considered become more and more non-square, CA3DMM's improvements over SCALAPACK increase. Furthermore, across all tested problems, we see that the GPU implementation of the Chebyshev filtering can outperform the CPU implementation. The benefit of the GPU for matrix-matrix products and eigensolves varies based on the problem. Still, we observe that the total time taken for the GPU codes is better than the CPU-based codes in nearly all cases, and in the cases tested where this is not the case, the timings are very close. From this, it is clear that there are benefits to using advanced matrix-matrix products and GPUs to accelerate such Chebyshev-filtered-subspace-iteration-based DFT codes with a few hundred processes and a few dozen GPUs.

7.7 Future Work

The results presented above demonstrate that there is promise in using alternative distributions, matrix-matrix products, and eigensolves to reduce the overall wall time seen during Chebyshev-filtered subspace iteration. Each of these can be investigated to varying degrees. Due to the costs associated with the data redistributions required for each computational kernel and the tradeoffs between the distributions, further investigation should be performed to determine what combination of distributions may be optimal. As part of this, a more thorough comparison of 3D matrix-matrix-based methods against traditional methods in this context should be performed. Furthermore, when considering GPUs, a significant portion of the costs seen are associated with the transfer of data from the CPU to the GPU and vice versa. Thus, an implementation that does not require the transfer of a Φ sized matrix would be able to reduce such communication costs. Finally, as we see that the current distributed eigensolvers do not scale to many nodes for the problem sizes considered, it may be promising to develop an alternative eigensolver.

As mentioned previously, there exists a tradeoff between the band (column) and domain (row) distributions, in which the band + domain distribution provides a balance between the two. However, determining the optimal number of processors and the topology to be used is a difficult task. Furthermore, the internal distribution used by the matrix-matrix products may also play a role in the overall wall time. Currently, the CA3DMM package only supports a 2D block-based matrix distribution compared to the commonly used block-cyclic distribution. It may be possible to select a common distribution for both the eigensolve and matrix products, reducing the number of redistributions.

Currently, there are many limitations with the GPU-based eigensolvers. The ELPA GPU eigensolver is only GPU accelerated and, as such, still has certain kernels performed on the CPU. This results in the ELPA code assuming that the data starts on the CPU, which, if the rest of the SCF kernel is performed on the GPU, requires a data transfer. On the other hand, the cuSOLVER eigensolver only works for a single GPU, which with the V100s hit memory limits around a problem size of 20k states. An additional option, cuSOLVERMG, allows multiple GPUs to be used but is still limited to a single node. Thus, there exists space to investigate an alternate eigensolver approach. In particular, between 20k and 100k points, the problem is too large to be solved efficiently on a single GPU but too small to be solved efficiently with the large distributed ELPA GPU code.

Finally, it may be that the ideal eigensolve process distribution may use fewer processors than available. An investigation may be done into a hybrid MPI+OpenMP parallelization scheme to allow multiple threads to work together in shared memory while taking advantage of the scaling achieved through distributed memory. Alternatively, it may be found that using all processes makes the problem size too small per process and reduces performance. For example, in Fig. 7.2 it is seen that the wall time increases when eight nodes are used compared to six nodes. Thus, using only a subset of processors or nodes may be found to increase wall time.

From this, we can see that there are many avenues that should be investigated to provide improved performance for the Chebyshev-filtered subspace iteration. This, in turn, would reduce the time associated with the self-consistent-field iteration, speeding up programs used for quantum chemistry and molecular dynamics.

Part III

Algebraic Multigrid Sparse Triple Matrix Products on GPU

CHAPTER 8
ALGEBRAIC MULTIGRID SPARSE TRIPLE MATRIX PRODUCTS ON GPU

8.1 Introduction

The iterative solvers discussed thus far in this dissertation have primarily been Krylov-Subspace-based methods, where the subspace of powers of A applied to an initial vector is continually expanded until it contains a satisfactory approximate solution x to $Ax = b$. These methods can be very general, not relying on the underlying source of data (so long as the assumptions of a given method are met). There exists another group of methods, algebraic multigrid (AMG) methods, based upon the multigrid method (MG), which aim to provide $O(n)$ solves to systems with n unknown by using multiple levels of refinement [113, 114]. Within AMG, a coarse-grid refinement is used to transfer the error between the levels of refinement, which is often performed as RAP.

This product can be performed as $R(AP)$ or $(RA)P$. However, this requires two separate matrix-multiplications, in addition to the allocation and usage of a temporary matrix. The matrices used in AMG tend to be very sparse, and sparse calculations require additional considerations to take full advantage of the computational ability offered by GPUs. In [115, 116, 117], methods are developed for performing sparse-matrix-matrix (SpGEMM) products on the CPU. On GPUs, sparse accumulators are used, as dense accumulators are infeasible due to the number of threads writing to the same accumulators. Work on such SpGEMMs has been done on GPUs [118, 119, 120]. Within this section, we develop and present an algorithm for efficiently performing the sparse triple-matrix product on GPUs, which in some instances can outperform the naive method of two independent matrix products.

8.2 Background

GPUs have recently gained popularity due to their efficiency in handling parallel kernels with high arithmetic intensity. This has led them to become accelerators for applications such as dense linear algebra, particularly in machine learning and deep neural networks. As the ecosystem has become more mature, libraries for sparse arithmetic have become available, including cuSPARSE by NVIDIA for their GPUs [121] and rocSPARSE by AMD for their GPUs [122]. However, as the space is still developing, the kernels still have much room to be optimized. In particular, we will be looking at sparse matrix-matrix and sparse matrix-matrix-matrix products.

While sparse calculations can take advantage of the sparsity of a matrix, with data structures commonly used by scientific codes, it has increased overhead compared to dense multiplications. With the Compressed Sparse Row (CSR) format, each row is stored in a compressed form where the nonzero values are stored consecutively. This allows row operations such as scaling to be performed rapidly without much overhead. However, adding or removing elements from a row may require shifting many elements. In the case of matrix-matrix products, the resulting matrix may have an irregular structure, making it difficult to perform efficient matrix-matrix products. Notably, because the cost of freeing and allocating memory on the GPU is much larger than on CPUs, efficient algorithms on GPUs should avoid reallocating memory – making common algorithms used for CPUs infeasible.

Instead, one method for GPUs is to calculate the number of nonzeros per row first, then perform the numeric product. One method for performing this is first to perform a naive row bound calculation, then use these bounds in Cohen's stochastic estimator [123], then perform a symbolic multiplication, and finally a numeric multiplication. The symbolic multiplication determines the sparsity structure of the output matrix, while the numeric multiplication determines the corresponding values. While this requires more steps than direct multiplication, these calculations are very parallel and can be performed efficiently on GPUs.

A naive lower bound on the number of nonzeros in the row i of $C = AB$ can be found by taking the maximum number of nonzeros in the rows of B corresponding with the nonzero columns of the ith row of A. Similarly, an upper bound can be found by taking the sum of the number of nonzeros.

These bounds can be used in the Stochastic estimator developed in [123] to provide a more accurate estimate of the number of nonzeros per row. This approach treats the matrix product as a network of layered bipartite graphs, where the output provides an estimate of the number of nonzeros in each row.

Once this more refined estimate is obtained, hash tables can be used to perform a symbolic multiplication yielding the exact number of nonzeros per row. A numeric multiplication can then be performed to yield the desired matrix.

8.3 Triple Matrix Product

During AMG setup, Galerkin Products are typically used to compute the coarse-grid operators. This RAP calculation can be computed via two typical matrix-matrix products, as either $(RA)P$ or $R(AP)$. However, this requires at least two different passes over the data and includes storing a temporary matrix for the initial matrix-matrix product. As such, computing this computation in a single pass may reduce the computation time and peak memory usage.

Similar to the algorithm used for the matrix-matrix products described previously, the method used is to (optionally) compute upper and lower bounds of the number of nonzeros per row, compute an estimate of the number of nonzeros per row using Cohen's row count estimator, perform symbolic multiplication, and finally perform the numeric multiplication. We will begin with a description of the bounds calculations.

8.3.1 Bounds

The naive bounds calculation is performed similarly to the bounds calculation used for the dual matrix product. In the case of triple-matrix vector product, where $D = ABC$ with $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times l}, C \in \mathbb{R}^{l\times n}$ and $D \in \mathbb{R}^{m\times n}$. The corresponding bound for row i of D can be calculated as the sum or max of the rows of BC corresponding to the nonzeros in row i, as seen in Alg. 17.

Algorithm 17 Provide a naive bound for a triple matrix product

```
1: ROWNNZ-NAIVE-ROW-TRIPLE($A$, $B$,$C$,$BoundType$)
2: $rcA \leftarrow \hat{0}$
3: for Row $i$ of $A$ do
4:     $rcBC \leftarrow \hat{0}$
5:     for Col $j$ of in $A_i$ do
6:         $nnzJ \leftarrow nnz(B, C, j, BoundType)$
7:         $rcBC_i \leftarrow reduce(nnzJ, rcBC_i, BoundType)$
8:         $rcA_i \leftarrow reduce(rcBC_i, rcA_i, BoundType)$
9:     end for
10: end for
11: $warpReduce(rcA_i, BoundType)$
```

Note that, as written, there is a significant amount of redundant calculation of the row estimates of BC (up to a duplicate of the average number of nonzeros per row of A). However, the redundant calculations can be removed by first calculating the row bounds for each row of BC, effectively memoizing the calculation.

8.3.2 Row Estimate

To obtain the estimated number of nonzeros for each row, Cohen's algorithm is used, using an additional layer corresponding with the additional matrix in the multiplication. This is a four-layer graph is used, with m, k, l, n nodes for each layer, respectively. The last layer is assigned r random variables from the exponential distribution for each node. Each node computes the element-wise minima of the vectors by the neighboring nodes in the next layer for the third, second, and first layers. The estimator of [123] is applied to the first layer to achieve the row estimates. The estimate for each row can be performed independently, with each layer being considered in sequence.

8.3.3 Symbolic Multiplication

The third step of the SpGEMMM kernel is to perform a symbolic multiplication using hash tables. This occurs by traversing all three matrices, as seen in Eq. 17. The HASH-SEARCH-INSERT$(k, v, count)$ operation attempts to insert into the hash table at key k value v, increases $count$ by one if it is a new entry, and returns -1 otherwise. Thus, we can track how many nonzeros occur in each row by tracking how many repeated inserts are to the same position. The actual value does not matter in symbolic multiplication, so we use $NULL$.

Algorithm 18 Perform symbolic triple matrix product

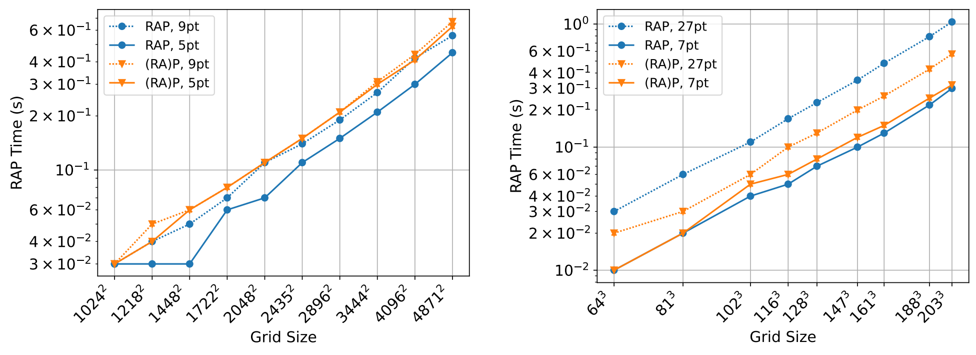
```
1:  ROWNNZ-SYMBOLIC-TRIPLE($A$, $B$,$C$,$BoundType$)
2:  for Row $i$ of $A$ do
3:      $nnz_i \leftarrow 0$
4:      for Col $j$ of in $A_i$ do
5:          for Col $k$ in $B_j$ do
6:              for Col $j$ in $C_k$ do
7:                  $pos \leftarrow$HASH-SEARCH-INSERT($j$, $NULL$,$nnz_i$)
8:                  if $-1 == pos$ then
9:                      $nnz_i \leftarrow nnz_i + 1$
10:                     $failed \leftarrow 1$
11:                 end if
12:             end for
13:         end for
14:     end for
15: end for
16: return NumNewInserts
```

8.3.4 Numeric Multiplication

The numeric multiplication is performed similarly to the symbolic multiplication. However, the value we insert is now important - it must be the numeric value from the multiplication. As such, in addition to using the column in each matrix, the corresponding value has to be tracked as well to perform the multiplication. Thus, in the call in line 7, the NULL is replaced by adding the product of the three values - one for each of A, B, C - to the current value in the hash table.

8.4 Numerical Results

Fig. 8.1, we present the performance of using the triple matrix kernel (SpGEMMM) for computing RAP. This test was done on a cluster with NVIDIA P100 GPUs. We observe that SpGEMMM performs better than using two SpGEMMs for the 5pt- and 7pt- stencils, equally well for the 9pt- stencil, and worse in the 27pt- stencil. Thus, it appears that the primary concern in the performance of RAP is the density of the matrix. A more sparse 2D kernel can see a more significant speedup than the more dense 3D kernels.



(a) 5-, 9-point 2D problems (b) 7-, 27-point 3D problems

Fig. 8.1: Computing RAP with the triple matrix product, compared with the matrix-matrix product R(AP).

From these results, we can see that a triple-matrix product on GPUs can be performed for certain stencils more efficiently than two successive matrix-matrix products. This is promising in AMG, as it can accelerate the speedup of calculations.

Part IV

Conclusion

CHAPTER 9
CONCLUSION

The modern age has seen immense growth in the amount of computational power available and the amount of data one can use in computations, and thus an increase in the size of problems one wants to solve. However, to get the most out of the new systems, new methods must be developed to take advantage of the available hardware and provide algorithmic scalability to make previously intractable problems tractable. In this dissertation, we developed two frameworks - one for parallel quantum chemistry and one for Gaussian process regression using hierarchical matrices - as well as developed a new method for sparse triple matrix products on GPUs to increase the size of problems that can be tackled and the speed at which they can be tackled. In this section, we summarize the contributions detailed in this thesis.

9.1 Hierarchical Matrices

Hierarchical matrices are a promising set of methods that can speed up many classes of computations asymptotically. In this dissertation, we expanded hierarchical matrices in a few directions. First, we developed a new method for constructing hierarchical matrices. This new method considers the locations of the points involved in the kernel matrix, which results in a speedup and reduction of memory usage. Furthermore, we developed an on-the-fly memory mode, which reduces peak memory usage considerably.

Next, we developed new methods which allow hierarchical matrices to extend into higher dimensions. These methods included using dimensionality reduction techniques to reduce the dimensionality of the problem being handled. These methods also included using alternate techniques for determining the bases associated with each node. Furthermore, a new splitting technique was developed, which increased the percentage of interactions that can be compressed. The combination of these resulted in being able to tackle higher-dimensional problems.

We then investigated the use of preconditioners for kernels arising from Gaussian process regression. This resulted in two preconditioners, a Nyström-based method and an FSAI-based method, complimentary in the classes of problems they handle well. These preconditioners significantly reduced the number of iterations required for solves involving the kernel matrices.

The final development of this dissertation for hierarchical matrices was the development of a large-scale Gaussian process regression framework. Within the framework, using matrix-free methods, combined with the other hierarchical matrix methods covered in the dissertation, an asymptotic reduction of complexity was achieved. This reduction in complexity is for both the hyperparameter optimization and the prediction phases of Gaussian process regression.

9.2 Parallel Quantum Chemistry

Quantum chemistry calculations are used in various scientific computing domains, including physics, material science, and chemistry. However, the native complexity of the problems demands efficient approximation methods. In this dissertation, we developed a distributed memory GPU implementation of Chebyshev-filtered subspace iteration for Kohn-Sham density functional theory. During the development, we investigated different matrix-matrix products and different eigensolvers. We identified a gap in the currently available software for distributed memory GPU eigensolvers when solving problems in the range of a few tens of thousands of rows and columns to around a hundred thousand rows and columns. Furthermore, the parallel computation engine (libPCE) outperformed the reference implementation, which will allow for larger problems and for them to be tackled faster.

9.3 Sparse Triple Matrix Products on GPUs

In the final part of this dissertation, we investigated sparse triple matrix products on GPUs. As sparse matrix calculations require more overhead on GPUs than CPUs, new methods must be developed to take advantage of GPUs' computational efficiency and power. By performing sparse triple matrix products in a manner that exploits the parallelism provided by GPUs, we achieved a speedup for a variety of problem classes compared to successive matrix-matrix products.

9.4 Conclusion

New methods must be developed to handle today's increasingly large and exciting problems. Through the contributions of this dissertation, we have developed new tools to tackle such large problems. These tools can be used in scientific computing to further the science, which can be done in a wide variety of applications, including machine learning, computational chemistry, and simulations.

REFERENCES

[1] L. Greengard and V. Rokhlin, “A fast algorithm for particle simulations,” *Journal of Computational Physics*, vol. 73, no. 2, pp. 325–348, Dec. 1987.

[2] S. Börm and J. Garcke, “Approximating Gaussian Processes with $H2$ Matrices,” in *European Conference on Machine Learning*, Springer, 2007, pp. 42–53.

[3] J. Barnes and P. Hut, “A hierarchical O(N log N) force-calculation algorithm,” *Nature*, vol. 324, no. 6096, p. 446, Dec. 1986.

[4] W. Hackbusch, “A sparse matrix arithmetic based on H-Matrices. Part I: Introduction to H-Matrices,” *Computing*, vol. 62, pp. 89–108, 1999.

[5] W. Hackbusch, B. Khoromskij, and S. A. Sauter, “On H2-Matrices,” in *Lectures on Applied Mathematics*, H.-J. Bungartz, R. H. W. Hoppe, and C. Zenger, Eds., Springer Berlin Heidelberg, 2000, pp. 9–29, ISBN: 978-3-642-59709-1.

[6] S. Ambikasaran, “A fast direct solver for dense linear systems,” 2013.

[7] S. Chandrasekaran, M. Gu, and W. Lyons, “A fast adaptive solver for hierarchically semiseparable representations,” *CALCOLO*, vol. 42, no. 3, pp. 171–185, Dec. 2005.

[8] Q. Xu, A. Sharma, B. Comer, H. Huang, E. Chow, A. J. Medford, J. E. Pask, and P. Suryanarayana. (May 20, 2020). “SPARC: Simulation Package for Ab-initio Real-space Calculations.” arXiv: 2005.10431 `[cond-mat, physics:physics]`.

[9] Y. Saad, J. R. Chelikowsky, and S. M. Shontz, “Numerical Methods for Electronic Structure Calculations of Materials,” *SIAM Review*, Feb. 5, 2010.

[10] Y. Zhou, J. R. Chelikowsky, and Y. Saad, “Chebyshev-filtered subspace iteration method free of sparse diagonalization for solving the Kohn–Sham equation,” *Journal of Computational Physics*, vol. 274, pp. 770–782, Oct. 1, 2014.

[11] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*. SIAM, 2000.

[12] R. D. Falgout, “An Introduction to Algebraic Multigrid,” *Computing in Science and Engineering, vol. 8, no. 6, November 1, 2006, pp. 24-33*, no. UCRL-JRNL-220851, Apr. 25, 2006.

[13] B. E. Boser, I. M. Guyon, and V. N. Vapnik, “A Training Algorithm for Optimal Margin Classifiers,” in *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, ACM Press, 1992, pp. 144–152.

[14] C. E. Rasmussen, "Gaussian Processes in Machine Learning," in *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures*, ser. Lecture Notes in Computer Science, O. Bousquet, U. von Luxburg, and G. Rätsch, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 63–71, ISBN: 978-3-540-28650-9.

[15] D. Cai, E. Chow, L. Erlandson, Y. Saad, and Y. Xi, "SMASH: Structured matrix approximation by separation and hierarchy," *Numerical Linear Algebra with Applications*, vol. 25, no. 6, e2204, 2018.

[16] L. Erlandson, D. Cai, Y. Xi, and E. Chow, "Accelerating Parallel Hierarchical Matrix-Vector Products via Data-Driven Sampling," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020.

[17] N. Halko, P. Martinsson, and J. Tropp, "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions," *SIAM Review*, vol. 53, no. 2, pp. 217–288, Jan. 1, 2011.

[18] M. Gu and S. Eisenstat, "Efficient Algorithms for Computing a Strong Rank-Revealing QR Factorization," *SIAM Journal on Scientific Computing*, vol. 17, no. 4, pp. 848–869, Jul. 1996.

[19] © 2020 IEEE. Reprinted, with permission, from Lucas Erlandson; Difeng Cai; Yuanzhe Xi; Edmond Chow , Accelerating Parallel Hierarchical Matrix-Vector Products via Data-Driven Sampling, IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2020.

[20] S. Börm and W. Hackbusch, "Data-sparse approximation by adaptive H^2-matrices," *Computing*, vol. 69, pp. 1–35, 2002.

[21] W. Hackbusch, *Hierarchical Matrices: Algorithms and Analysis*, ser. Springer Series in Computational Mathematics. Berlin Heidelberg: Springer-Verlag, 2015, ISBN: 978-3-662-47323-8.

[22] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[23] W. Hackbusch and Z. P. Nowak, "On the fast matrix multiplication in the boundary element method by panel clustering," *Numerische Mathematik*, vol. 54, no. 4, pp. 463–491, 1989.

[24] W. Hackbusch and S. Börm, "H2-matrix approximation of integral operators by interpolation," *Applied Numerical Mathematics*, 19th Dundee Biennial Conference on Numerical Analysis, vol. 43, no. 1, pp. 129–143, Oct. 2002.

[25] S. Börm and L. Grasedyck, “Hybrid cross approximation of integral operators,” *Numerische Mathematik*, vol. 101, no. 2, pp. 221–249, Aug. 2005.

[26] D. Cai, E. Chow, L. Erlandson, Y. Saad, and Y. Xi, “SMASH: Structured matrix approximation by separation and hierarchy,” *Numerical Linear Algebra with Applications*, vol. 25, no. 6, e2204, 2018.

[27] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, “Fast algorithms for hierarchically semiseparable matrices,” *Numerical Linear Algebra with Applications*, vol. 17, no. 6, pp. 953–976, Dec. 2010.

[28] S. Börm, L. Grasedyck, and W. Hackbusch, “Introduction to hierarchical matrices with applications,” *Engineering Analysis with Boundary Elements*, Large Scale Problems Using BEM, vol. 27, no. 5, pp. 405–422, May 2003.

[29] C. D. Yu, S. Reiz, and G. Biros, “Distributed-memory Hierarchical Compression of Dense SPD Matrices,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC ’18, Piscataway, NJ, USA: IEEE Press, 2018, 15:1–15:15.

[30] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, “A Distributed-Memory Package for Dense Hierarchically Semi-Separable Matrix Computations Using Randomization,” *ACM Trans. Math. Softw.*, vol. 42, no. 4, 27:1–27:35, Jun. 2016.

[31] S. Börm, *Efficient Numerical Methods for Non-Local Operators: H^2-Matrix Compression, Algorithms and Analysis*, ser. EMS Tracts in Mathematics. EMS, 2010, vol. 14.

[32] V. Rokhlin, “Rapid solution of integral equations of classical potential theory,” *Journal of Computational Physics*, vol. 60, no. 2, pp. 187–207, 1985.

[33] D. Cai and J. Xia, “A stable and efficient matrix version of the fast multipole method,” *preprint*,

[34] G. Biros, L. Ying, and D. Zorin, “A fast solver for the Stokes equations with distributed forces in complex geometries,” *Journal of Computational Physics*, vol. 193, no. 1, pp. 317–348, Jan. 2004.

[35] Lexing Ying, G. Biros, D. Zorin, and H. Langston, “A New Parallel Kernel-Independent Fast Multipole Method,” in *SC ’03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, Nov. 2003, pp. 14–14.

[36] W. Chai and D. J., “An $\mathcal{H}^2$-Matrix-Based Integral-Equation Solver of Reduced Complexity and Controlled Accuracy for Solving Electrodynamic Problems,” *IEEE Transactions on Antennas and Propagation*, vol. 57, no. 10, pp. 3147–3159, 2009.

[37] W. Fong and E. Darve, "The black-box fast multipole method," *Journal of Computational Physics*, vol. 228, no. 23, pp. 8712–8725, 2009.

[38] C. K. Williams and M. Seeger, "Using the Nyström method to speed up kernel machines," in *Advances in neural information processing systems*, 2001, pp. 682–688.

[39] M. W. Mahoney and P. Drineas, "CUR matrix decompositions for improved data analysis," *Proceedings of the National Academy of Sciences*, vol. 106, no. 3, pp. 697–702, Jan. 2009.

[40] A. Gittens and M. W. Mahoney, "Revisiting the Nyström method for improved large-scale machine learning," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 3977–4041, 2016.

[41] K. Zhang, I. W. Tsang, and J. T. Kwok, "Improved Nyström low-rank approximation and error analysis," in *Proceedings of the 25th International Conference on Machine Learning*, ACM, 2008, pp. 1232–1239.

[42] D. Cai, J. G. Nagy, and Y. Xi, "A Matrix Free Nyström Method via Anchor Net," *preprint*,

[43] P. Glira, N. Pfeifer, C. Briese, and C. Ressl, "A Correspondence Framework for ALS Strip Adjustments based on Variants of the ICP Algorithm," *Photogrammetrie - Fernerkundung - Geoinformation*, vol. 2015, Aug. 2015.

[44] W. March, B. Xiao, C. Yu, and G. Biros, "ASKIT: An Efficient, Parallel Library for High-Dimensional Kernel Summations," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, S720–S749, Jan. 2016.

[45] M. Bebendorf, "Approximation of boundary element matrices," *Numerische Mathematik*, vol. 86, no. 4, pp. 565–589, Oct. 2000.

[46] N. Halko, P. Martinsson, and J. Tropp, "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions," *SIAM Review*, vol. 53, no. 2, pp. 217–288, Jan. 2011.

[47] X. Xing, H. Huang, and E. Chow, "A linear scaling hierarchical block low-rank representation of the electron repulsion integral tensor," *The Journal of Chemical Physics*, vol. 153, no. 8, p. 084 119, Aug. 28, 2020.

[48] X. Xing, "The proxy point method for rank-structured matrices," Nov. 6, 2019.

[49] X. Xing and E. Chow, "Interpolative Decomposition via Proxy Points for Kernel Matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 41, no. 1, pp. 221–243, Jan. 2020.

[50] K. Pearson, "LIII. On lines and planes of closest fit to systems of points in space," Nov. 1, 1901.

[51] J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A Global Geometric Framework for Nonlinear Dimensionality Reduction," *Science*, vol. 290, no. 5500, pp. 2319–2323, Dec. 22, 2000.

[52] J. Garcke, "Sparse Grids in a Nutshell," in *Sparse Grids and Applications*, ser. Lecture Notes in Computational Science and Engineering, J. Garcke and M. Griebel, Eds., vol. 88, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 57–80, ISBN: 978-3-642-31702-6 978-3-642-31703-3.

[53] H.-J. Bungartz and M. Griebel, "Sparse grids," *Acta numerica*, vol. 13, pp. 147–269, 2004.

[54] G. H. Golub and C. F. V. Loan, *Matrix Computations*. JHU Press, 2013, 781 pp., ISBN: 978-1-4214-0794-4. Google Books: X5YfsuCWpxMC.

[55] J. A. Meijerink and H. A. van der Vorst, "An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric \$M\$-Matrix," *Mathematics of Computation*, vol. 31, no. 137, pp. 148–162, 1977. JSTOR: 2005786.

[56] L. Kolotilina and A. Yeremin, "Factorized Sparse Approximate Inverse Preconditionings I. Theory," *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 1, pp. 45–58, Jan. 1, 1993.

[57] C. Janna and M. Ferronato, "Adaptive Pattern Research for Block FSAI Preconditioning," *SIAM Journal on Scientific Computing*, vol. 33, no. 6, pp. 3357–3380, Jan. 1, 2011.

[58] M. Bernaschi, M. Carrozzo, A. Franceschini, and C. Janna, "A Dynamic Pattern Factored Sparse Approximate Inverse Preconditioner on Graphics Processing Units," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C139–C160, Jan. 1, 2019.

[59] C. Janna, M. Ferronato, F. Sartoretto, and G. Gambolati, "FSAIPACK: A Software Package for High-Performance Factored Sparse Approximate Inverse Preconditioning," *ACM Transactions on Mathematical Software*, vol. 41, no. 2, pp. 1–26, Feb. 4, 2015.

[60] E. Chow and Y. Saad, "Preconditioned Krylov Subspace Methods for Sampling Multivariate Gaussian Distributions," *SIAM Journal on Scientific Computing*, vol. 36, no. 2, A588–A608, Jan. 1, 2014.

[61] C. K. I. Williams and M. Seeger, "Using the Nyström Method to Speed Up Kernel Machines," in *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp, Eds., MIT Press, 2001, pp. 682–688.

[62] M. A. Woodbury, *Inverting Modified Matrices*. Statistical Research Group, 1950.

[63] K. Cutajar, M. Osborne, J. Cunningham, and M. Filippone, "Preconditioning Kernel Matrices," in *International Conference on Machine Learning*, Jun. 11, 2016, pp. 2529–2538.

[64] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[65] M. R. Hestenes and E. Stiefel, *Methods of Conjugate Gradients for Solving Linear Systems*, 1. NBS Washington, DC, 1952, vol. 49.

[66] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, "A Limited Memory Algorithm for Bound Constrained Optimization," *SIAM Journal on Scientific Computing*, vol. 16, no. 5, pp. 1190–1208, Sep. 1, 1995.

[67] O. Roustant, D. Ginsbourger, and Y. Deville, "DiceKriging, DiceOptim: Two R Packages for the Analysis of Computer Experiments by Kriging-Based Metamodeling and Optimization," *Journal of Statistical Software*, vol. 51, no. 1, pp. 1–55, Oct. 22, 2012.

[68] S. Ubaru, J. Chen, and Y. Saad, "Fast Estimation of tr(f(A)) via Stochastic Lanczos Quadrature," *SIAM Journal on Matrix Analysis and Applications*, vol. 38, no. 4, pp. 1075–1099, 2017.

[69] O. Roustant, D. Ginsbourger, and Y. Deville, "DiceKriging, DiceOptim: Two R Packages for the Analysis of Computer Experiments by Kriging-Based Metamodeling and Optimization," *Journal of Statistical Software*, vol. 51, no. 1, pp. 1–55, Oct. 22, 2012.

[70] J.-S. Park and J. Baek, "Efficient computation of maximum likelihood estimators in a spatial linear model with power exponential covariogram," *Computers & Geosciences*, vol. 27, no. 1, pp. 1–7, Feb. 1, 2001.

[71] M. Hutchinson, “A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines,” *Communication in Statistics- Simulation and Computation*, vol. 18, pp. 1059–1076, Jan. 1, 1989.

[72] E. J. Nystrom, “Uber Die Praktische Auflosung von Integralgleichungen mit Anwendungen auf Randwertaufgaben,” *Acta Mathematica*, vol. 54, pp. 185–204, none Jan. 1930.

[73] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*. SIAM, 2019.

[74] R. K. Montoye, E. Hokenek, and S. L. Runyon, “Design of the IBM RISC System/6000 floating-point execution unit,” *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59–70, Jan. 1990.

[75] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2008*, pp. 1–70, Aug. 2008.

[76] K. Wang, G. Pleiss, J. Gardner, S. Tyree, K. Q. Weinberger, and A. G. Wilson, “Exact Gaussian Processes on a Million Data Points,” in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019.

[77] H. Avron and S. Toledo, “Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix,” *Journal of the ACM (JACM)*, vol. 58, no. 2, pp. 1–34, 2011.

[78] Y. Saad, *Iterative Methods for Sparse Linear Systems*, ser. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 1, 2003, 537 pp., ISBN: 978-0-89871-534-7.

[79] L. Lin, Y. Saad, and C. Yang. (Aug. 25, 2013). “Approximating spectral densities of large matrices.” arXiv: 1308.5467 [math].

[80] O. Roustant, D. Ginsbourger, and Y. Deville, “Dicekriging, diceoptim: Two r packages for the analysis of computer experiments by kriging-based metamodeling and optimization,” *Journal of Statistical Software, Articles*, vol. 51, no. 1, pp. 1–55, 2012.

[81] K. Perlin, “An image synthesizer,” *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 287–296, Jul. 1, 1985.

[82] M. J. Heaton, W. F. Christensen, and M. A. Terres, “Nonstationary gaussian process models using spatial hierarchical clustering from finite differences,” *Technometrics*, vol. 59, no. 1, pp. 93–101, 2017.

[83] D. Nychka, S. Bandyopadhyay, D. Hammerling, F. Lindgren, and S. Sain, "A Multiresolution Gaussian Process Model for the Analysis of Large Spatial Datasets," *Journal of Computational and Graphical Statistics*, vol. 24, no. 2, pp. 579–599, Apr. 3, 2015.

[84] M. J. Heaton, A. Datta, A. O. Finley, R. Furrer, J. Guinness, R. Guhaniyogi, F. Gerber, R. B. Gramacy, D. Hammerling, M. Katzfuss, F. Lindgren, D. W. Nychka, F. Sun, and A. Zammit-Mangion, "A Case Study Competition Among Methods for Analyzing Large Spatial Data," *Journal of Agricultural, Biological and Environmental Statistics*, vol. 24, no. 3, pp. 398–425, Sep. 1, 2019.

[85] E. Schrödinger, "An Undulatory Theory of the Mechanics of Atoms and Molecules," *Physical Review*, vol. 28, no. 6, pp. 1049–1070, Dec. 1, 1926.

[86] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky, "Self-consistent-field calculations using Chebyshev-filtered subspace iteration," *Journal of Computational Physics*, vol. 219, no. 1, pp. 172–184, Nov. 20, 2006.

[87] Q. Xu, A. Sharma, B. Comer, H. Huang, E. Chow, A. J. Medford, J. E. Pask, and P. Suryanarayana, "SPARC: Simulation Package for Ab-initio Real-space Calculations," *SoftwareX*, vol. 15, p. 100 709, Jul. 1, 2021.

[88] D. Sherrill, "Introduction to Density Functional Theory."

[89] P. Hohenberg and W. Kohn, "Inhomogeneous Electron Gas," *Physical Review*, vol. 136, B864–B871, 3B Nov. 9, 1964.

[90] W. Kohn and L. J. Sham, "Self-Consistent Equations Including Exchange and Correlation Effects," *Physical Review*, vol. 140, A1133–A1138, 4A Nov. 15, 1965.

[91] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky, "Parallel self-consistent-field calculations via Chebyshev-filtered subspace acceleration," *Physical Review E*, vol. 74, no. 6, p. 066 704, Dec. 28, 2006.

[92] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, Sep. 1995.

[93] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication," *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, Oct. 1, 2012.

[94] H. Gupta and P. Sadayappan, "Communication-efficient matrix multiplication on hypercubes," *Parallel Computing*, vol. 22, no. 1, pp. 75–99, Jan. 1, 1996.

[95] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler. (Dec. 13, 2019). "Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication." arXiv: 1908.09606 [cs].

[96] E. Solomonik and J. Demmel, "Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds., vol. 6853, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 90–109, ISBN: 978-3-642-23396-8 978-3-642-23397-5.

[97] R. A. Van De Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, Apr. 1997.

[98] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel Math Kernel Library," in, May 15, 2014, pp. 167–188, ISBN: 978-3-319-06485-7.

[99] R. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Nov. 1998, pp. 38–38.

[100] L. S. Blackford, "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, p. 17,

[101] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University, USA, 1969, 228 pp.

[102] J. Choi, D. W. Walker, J. J. Dongarra, and R. Pozo, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," Oak Ridge National Lab., TN (United States), CONF-9210148-1, Sep. 1, 1992.

[103] H. Huang. "Huanghua1994/CA3DMM: Communication-Avoiding 3D Matrix Multiplication," GitHub.

[104] T. Auckenthaler, V. Blum, H. .-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. R. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations," *Parallel Computing*, 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10), vol. 37, no. 12, pp. 783–794, Dec. 1, 2011.

[105] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997, ISBN: 0-89871-397-8 (paperback).

[106] NVIDIA. “cuSOLVER.”

[107] V. W.-z. Yu, J. Moussa, P. Kůs, A. Marek, P. Messmer, M. Yoon, H. Lederer, and V. Blum, “GPU-acceleration of the ELPA2 distributed eigensolver for dense symmetric and hermitian eigenproblems,” *Computer Physics Communications*, vol. 262, p. 107 808, May 1, 2021.

[108] NVIDIA. “NVIDIA V100,” NVIDIA.

[109] Intel. “Intel® Xeon® Gold 6226 Processor Product Specifications.”

[110] Lawrence Livermore National Lab. “Sierra — High Performance Computing.”

[111] P. Micikevicius, “3D finite difference computation on GPUs using CUDA,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2, New York, NY, USA: Association for Computing Machinery, Mar. 8, 2009, pp. 79–84, ISBN: 978-1-60558-517-8.

[112] “3D finite difference computation on GPUs using CUDA — Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units.”

[113] B. A, “Algebraic multigrid (AMG) for sparse matrix eqations,” *Sparsity and its Applications*, pp. 257–284, 1984.

[114] J. W. Ruge and K. Stüben, “4. Algebraic Multigrid,” in *Multigrid Methods*, ser. Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics, Jan. 1987, pp. 73–130, ISBN: 978-1-61197-188-0.

[115] J. R. Gilbert, C. Moler, and R. Schreiber, “Sparse Matrices in MATLAB: Design and Implementation,” *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, Jan. 1, 1992.

[116] F. G. Gustavson, “Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition,” *ACM Transactions on Mathematical Software*, vol. 4, no. 3, pp. 250–269, Sep. 1, 1978.

[117] R. E. Bank and C. C. Douglas, “Sparse matrix multiplication package (SMMP),” *Advances in Computational Mathematics*, vol. 1, no. 1, pp. 127–137, Feb. 1993.

[118] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann, “GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging,” *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C54–C71, Jan. 2015.

[119] S. Dalton, L. Olson, and N. Bell, “Optimizing Sparse Matrix&#x2014;Matrix Multiplication for the GPU,” *ACM Transactions on Mathematical Software*, vol. 41, no. 4, 25:1–25:20, Oct. 26, 2015.

[120] M. Deveci, C. Trott, and S. Rajamanickam, “Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures,” *Parallel Computing*, vol. 78, pp. 33–46, Oct. 1, 2018.

[121] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, “Cusparse library,” in *GPU Technology Conference*, 2010.

[122] *rocSPARSE*, ROCm Software Platform, Apr. 12, 2022.

[123] E. Cohen, “Structure Prediction and Computation of Sparse Matrix Products,” *Journal of Combinatorial Optimization*, vol. 2, no. 4, pp. 307–332, Dec. 1, 1998.