Computational Science and Engineering
(Int. Master's Program)

Technische Universität München

Master's Thesis

Scalable Kernel Matrix Inversion using Hierarchical Low-Rank Approximations

Mohamed Aziz Kara borni

Computational Science and Engineering (Int. Master's Program)

Technische Universität München

Master's Thesis

Scalable Kernel Matrix Inversion using Hierarchical Low-Rank Approximations

Author:	Mohamed Aziz Kara borni
Examiner:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Assistant advisor:	M.Sc. Keerthi Gaddameedi
Submission Date:	September 14th, 2024

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

September 14th, 2024 Mohamed Aziz Kara borni

Acknowledgments

I would like to extend my heartfelt gratitude to all those who have supported me throughout the completion of this thesis. I am profoundly grateful to my advisor, Keerthi Gaddameedi, whose initial introduction to this topic ignited my interest and whose steadfast support and readiness to address every question and concern have been instrumental in bringing this research to fruition. I am also thankful to my supervisor, Univ.-Prof. Dr. Hans-Joachim Bungartz, for granting me the opportunity to explore the assigned topic. Additionally, I am deeply appreciative of all those who contributed to the development of the GOFMM software. Their dedication and innovative efforts provided the essential tools needed for this thesis, without which this work would not have been possible. Lastly, my sincere thanks go to my family and friends for their unwavering encouragement, patience, and understanding throughout this journey. Their constant support has been a cornerstone of my success and has kept me motivated through the challenges.

Abstract

Kernel methods play a critical role in modern machine learning by enabling non-linear data transformations.These methods rely heavily on the inversion of large Symmetric Positive-Definite kernel(SPD) matrices. However, traditional inversion techniques scale with cubic complexity, making them impractical for large datasets commonly encountered in real-world applications. This computational bottleneck has spurred interest in developing more efficient matrix inversion techniques.

In this thesis, we investigate the use of hierarchical low-rank approximations to mitigate the computational challenges associated with kernel matrix inversion. In particular, we focus on the Geometry-Oblivious Fast Multipole Method(GOFMM), which decomposes dense SPD kernel matrices into smaller, manageable low-rank blocks. This decomposition reduces the overall complexity of the inversion process to approximately $O(N \log N)$, while maintaining a high degree of accuracy. The GOFMM approach allows for scalable matrix inversion, making it feasible to apply kernel-based to significantly larger datasets.

We develop and implement algorithms based on GOFMM, optimizing them for parallel execution on multi-core system. Strong and weak scaling experiments are conducted to evaluate the performance of these algorithms across various setups. We test the methods on synthetic data and real-world datasets such as MNIST.

The work presented in this document demonstrates that hierarchical low-rank approximations, specifically GOFMM, offer a scalable and efficient alternative to traditional kernel matrix inversion techniques. These methods pave the way for applying kernel-based machine learning models to increasingly larger and more complex datasets, further expanding their practical utility in diverse fields such as regression, classification, and probabilistic modeling.

Contents

1 Introduction

In the field of machine learning, kernel methods such as Kernel ridge regression (KRR) and Gaussian processes (GPs) have become fundamental tools for prediction and inference. These methods rely heavily on kernel matrices, which encapsulate the relationships between data points through their similarity measures. These kernel matrices often exhibit Symmetric Positive-Definite (SPD) properties, which ensures their invertibility—a critical feature for many kernel-based algorithms such as KRR and GPs when applied to large datasets [9, 11]. The ability to invert these matrices is essential for tasks such as optimization and model training, which are at the core of machine learning models.

However, as the size of datasets grows, the associated kernel matrices increase in size, making inversion computationally prohibitive. Traditional methods for inverting SPD matrices, such as Gaussian Elimination and Cholesky Decomposition, exhibit cubic complexity $O(N^3)$, where N is the number of data points [9]. This complexity poses significant computational and memory challenges, particularly in large-scale machine learning applications [4, 34]. As machine learning models evolve to process increasingly large datasets, overcoming this computational bottleneck has become a pressing concern in the field.

To address these challenges, hierarchical low-rank approximations have emerged as a promising solution to reduce the computational burden associated with matrix inversion. These methods work by approximating the large kernel matrix through a decomposition into smaller, lower-rank components, which simplifies the inversion process and significantly reduces computational complexity [3]. Among these methods, the Geometry-Oblivious Fast Multipole Method (GOFMM) stands out for its ability to handle large datasets while preserving accuracy [39]. GOFMM efficiently compresses the dense kernel matrix and performs matrix inversion using these compressed representations, resulting in substantial computational savings.

This thesis focuses on the scalability of hierarchical low-rank approximations, particularly the GOFMM, in the context of SPD kernel matrix inversion for KRR and GPs. Our goal is to develop algorithms that reduce the complexity of matrix inversion from $O(N^3)$ to approximately $O(N \log N)$, making it feasible to apply these methods to larger datasets efficiently. Specifically, we aim to approximate the inverse of the kernel matrix K^{-1}, with the approximation satisfying the following condition:

$$\frac{||K^{-1} - \tilde{K}^{-1}||}{||K^{-1}||} \leq \epsilon, \quad 0 < \epsilon < 1,$$

where ϵ is a user-defined tolerance. The inverse calculation is performed using hierarchical methods in GOFMM, where the dense kernel matrix is first compressed and then inverted.

By leveraging these techniques, we aim to achieve efficient kernel matrix inversion that scales well with increasing dataset sizes.

The key contributions of this research are:

- **Algorithm Development**: Developing scalable algorithms for kernel matrix inversion using GOFMM and applying them to KRR and GPs.
- **Scaling Experiments**: Conducting strong and weak scaling experiments using OpenMP to evaluate the performance of these algorithms.
- **Performance Analysis**: Comparing the efficiency and accuracy of GOFMM with traditional inversion methods and analyzing the impact of these methods on the scalability of kernel-based machine learning models.

The ultimate goal of this work is to enhance the efficiency and scalability of kernel matrix inversion techniques, enabling the broader application of KRR and GPs to real-world, large-scale machine learning problems.

2 State of the art

To thoroughly understand the inversion of SPD kernel matrices using the GOFMM, it is essential to cover several key theoretical aspects. In this chapter we provide a brief review of literature on the subject. Each of these topics provides a foundation for understanding how GOFMM can be applied to kernel matrix inversion.

2.1 Kernels

In machine learning, kernel methods are used to map data into higher-dimensional spaces where linear separation is more feasible. This mapping is accomplished using a kernel function $k(x_i, x_j)$, which measures the similarity between data points x_i and x_j. The kernel function generates a kernel matrix K as follows:

$$K_{ij} = k(x_i, x_j)$$

Common kernel functions include:

- **Linear Kernel**: $k(x_i, x_j) = x_i^T x_j$
- **Polynomial Kernel**: $k(x_i, x_j) = (x_i^T x_j + c)^d$
- **Radial Basis Function (RBF) Kernel**: $k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$

The choice of kernel function affects the properties of the resulting kernel matrix and the performance of machine learning algorithms. For instance, the RBF kernel maps data into an infinite-dimensional space, which can capture more complex patterns compared to the linear kernel [34].

2.2 Symmetric Positive-Definite Matrices

For kernel methods to be effective, the kernel matrix K must be SPD. An SPD matrix has the following properties:

- **Symmetry**: The matrix K is symmetric, meaning $K = K^T$.

- **Positive-Definiteness**: The matrix K is positive-definite if for any non-zero vector $x \in \mathbb{R}^N$, the quadratic form $x^T K x$ is positive:

$$x^T K x > 0 \text{ for all } x \neq 0$$

The positive-definiteness of K ensures that all its eigenvalues are positive, which is crucial for numerical stability and the effectiveness of algorithms that rely on matrix inversion. This property allows the use of efficient numerical techniques, such as Cholesky Decomposition, for matrix inversion and other operations [9].

In the context of kernel methods, the SPD property of the kernel matrix is fundamental for ensuring that algorithms like KRR and GPs are stable and effective. For large-scale datasets, hierarchical low-rank approximations are used to handle the computational complexity of these matrices, making kernel-based methods practical for large datasets [11, 39].

2.3 Low-rank approximations

A low-rank approximation of a matrix $A \in \mathbb{R}^{m \times n}$ seeks to find a matrix $\tilde{A}$ that is close to A but has a significantly lower rank r, where $r \ll \min(m, n)$. The goal is to approximate A with a matrix of reduced dimensions that captures the most important features of the original matrix. Mathematically, this can be expressed as:

$$\tilde{A} = U_r \Sigma_r V_r^T$$

where $U_r \in \mathbb{R}^{m \times r}$, $\Sigma_r \in \mathbb{R}^{r \times r}$, and $V_r \in \mathbb{R}^{n \times r}$ are matrices such that $\tilde{A}$ approximates A with minimized reconstruction error.

Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a widely used method for low-rank approximation. Given a matrix $A \in \mathbb{R}^{m \times n}$, SVD decomposes A into three matrices:

$$A = U \Sigma V^T$$

where $U \in \mathbb{R}^{m \times m}$ is an orthogonal matrix, $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix with singular values, and $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix. The low-rank approximation is obtained by truncating the smallest singular values and corresponding vectors [9].

Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a technique that uses low-rank approximation to reduce the dimensionality of data by projecting it onto the principal components. This method approximates the covariance matrix of the data and is closely related to the SVD of the data matrix [17].

Interpolative Decomposition

Interpolative Decomposition (ID) approximates a matrix G by selecting a subset of its columns to form a basis for the remaining columns. Specifically, for a matrix $G \in \mathbb{R}^{m\times n}$ with rank k, the goal is to find k columns that capture the essential structure of G. This results in a reduced memory footprint and computational efficiency, with complexity $O(kmn)$.

Formally, there exists a matrix $G_{\text{col}} \in \mathbb{R}^{m\times k}$ and a projection matrix $P \in \mathbb{R}^{k\times n}$ such that:

$$G_{\text{col}}P = G$$

where G_{col} consists of k selected columns from G, and P typically includes an identity matrix among its columns. If the rank k is greater than s, where $k \ll n$, one can find $G_{\text{col}} \in \mathbb{R}^{m\times s}$ and $P \in \mathbb{R}^{s\times n}$ such that:

$$G_{\text{col}}P \approx G$$

The approximation error is bounded by:

$$\|G_{\text{col}}P - G\|_F \leq \frac{\sigma_{s+1}}{\sigma_s}\|G\|_F$$

where σ_{s+1} is the $(s+1)$-th singular value of G.

To compute ID, one can use a rank-revealing QR decomposition. Given $G = QR$, where Q is orthonormal and R is upper triangular, we decompose G as:

$$G = \begin{bmatrix} Q_{\text{left}} & Q_{\text{right}} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

where $Q_{\text{left}} \in \mathbb{R}^{m\times s}$ and R_{11} is $s \times s$. The approximation is obtained by ignoring R_{22}:

$$G \approx Q_{\text{left}}R_{11}$$

Here, $G_{\text{col}} = Q_{\text{left}}R_{11}$ represents a subset of columns of G, and P is computed as:

$$P = I_s - R_{12}R_{11}^{-1}$$

Thus, ID provides a way to approximate a matrix by selecting key columns and using a reduced representation, with error bounds related to the singular values of G. However, rank-revealing factorizations like ID are generally less reliable than SVD [20, 23].

2.4 Hierarchical matrices

Hierarchical matrices, or H-matrices, offer a data-sparse representation of large matrices, aiming to achieve nearly linear complexity for matrix operations. For a system with N equations, achieving optimal efficiency typically requires $O(N)$ operations. However, for non-sparse matrices, this complexity can be prohibitive. H-matrices are designed to reduce

computational costs by representing matrices using fewer data points, allowing operations such as matrix-vector multiplication and factorization to be performed efficiently.

An H-matrix is structured to support efficient computations by approximating matrix operations in nearly linear time. This efficiency is achieved through methods such as two-sided compression, leading to H_2 matrices with $O(N)$ complexity for certain operations [11]. Techniques like the Fast Multipole Method (FMM) use hierarchical decompositions to approximate interactions in N-Body problems, where kernel-dependent expansions provide accurate approximations of the far-field [10].

Gramian Representation

Any SPD matrix can be described as a Gramian matrix of a set of vectors. For an SPD matrix $K \in \mathbb{R}^{N \times N}$, there exists a set of vectors $\{v_i\}$ such that:

$$K_{ij} = v_i^T v_j$$

While Cholesky decomposition provides a way to compute such vectors, the Gramian set is not unique and can be represented differently in various vector spaces [9].

Distance Metrics The following distances are derived from the Gramian representation:

- **Kernel Distance:** As discussed in Section 2.3.1, the kernel matrix K is derived from a set of Gram vectors such that $K_{ij} = \langle p_i, p_j \rangle$. The kernel distance between points p_i and p_j is computed using:

$$d_{ij} = \|p_i - p_j\|_2^2 = K_{ii} + K_{jj} - 2K_{ij}$$

- **Geometric Distance:** The geometric distance between two points x_i and x_j is defined as:

$$d_{ij} = \|x_i - x_j\|_2^2$$

  In this context, points are partitioned such that the distance d_{ij} between points within the same partition is minimized. The partitioning process continues until the number of leaf nodes reaches the predetermined maximum m. This distance metric computation requires $O(N \log N)$ work, where N is the number of points.

- **Angle Distance:** The angle distance, derived from the angle between Gram vectors, is given by:

$$d_{ij} = \sin^2\left(\frac{\angle p_i p_j}{2}\right) = \frac{1 - K_{ij}}{2}$$

  Calculating both the kernel and angle distances without sampling requires $O(N^2)$ operations.

Partitioning large matrices into nearly low-rank blocks can be challenging. The goal is to find an optimal index ordering to minimize the rank of off-diagonal blocks in the partitioned matrix. This problem involves finding subsets I_a and I_b such that:

$$\text{minimize rank}(A[I_a, I_b] - S)$$

where S is a sparse matrix, and $A[I_a, I_b]$ is a sub-block of the system matrix A. Techniques for optimal partitioning include space-filling curves, geodesic distances, and high-dimensional subspace clustering, each providing different advantages based on the application [3, 11, 1].

Physical space partitioning leverages the assumption that nearby points influence each other significantly, while distant points can be approximated efficiently. Techniques such as quad-tree splitting, geodesic distance metrics, and subspace clustering are employed to achieve effective partitioning [10, 2].

Hierarchical Decompositions The hierarchically low-rank approximation of the kernel matrix K is given by [11, 3]:

$$K = D + S + UV, \tag{2.1}$$

where D is a block-diagonal matrix with each block being an H-matrix, S is a sparse matrix, and U and V are low-rank matrices. The H-matrix K is computed to satisfy:

$$\|K - K\| \leq \epsilon \|K\|,$$

where ϵ is a user-defined tolerance with $0 < \epsilon < 1$. If S is zero in Equation 2.1, K is called a hierarchically off-diagonal low-rank approximation. Additionally, if D is also zero, the approximation is termed hierarchically semi-separable. Both the construction of K and the matrix-vector product can be performed with $O(N \log N)$ complexity.

2.5 Geometry-oblivious techniques

2.5.1 Hierarchical compression of dense SPD

In this part, we review the algorithmic approaches detailed in [39] for constructing hierarchical low-rank approximations of SPD matrices.

The compression strategy employed by the GOFMM framework, which consists of three key steps:

1. `HIERARCHICALPARTITIONING()`
2. `NEIGHBORHOODPRUNING()`
3. `SKELETONIZATION()`

Hierarchical partitioning The initial phase in building the hierarchical matrix structure entails dividing the matrix according to near and far field interactions. Leaf nodes with dimensions exceeding a specific threshold are classified as part of the far field. An approximate centroid is determined using a small sample of Gram vectors. Subsequently, a median split is performed on all nodes, beginning from the root node that includes all data points[39, 8].

Neighbor-Based Pruning Following the hierarchical partitioning, neighbor-based pruning is performed, as described in the algorithms 1, 2 and 3 taken from the document [39]. This involves constructing three lists: the neighbor list $\mathcal{N}(\alpha)$, the near interaction list Near(α), and the far interaction list Far(α). $\mathcal{N}(\alpha)$ is formed by iterating over all neighbors $j \in \mathcal{N}$ for each $i \in \mathcal{N}(\alpha)$ and including those where d_{ij} is small. The pruning process continues until either 10 iterations are completed or 80% accuracy is achieved.

Algorithm 1 Neighbor-Based Pruning[39]

```
for all i ∈ N(α) do
    for all j ∈ N(α) do
        if d_ij is small then
            N(α) = N(α) ∪ {j}
        end if
    end for
end for
```

Algorithm 2 Near(α) [39]

```
for all i ∈ N(α) do
    Near(α) = Near(α) ∪ MortonId(i)
end for
```

Finally, in the computation of $Far(\alpha)$, each leaf node β is checked to determine if $a \notin$ Near(β). If true, β is added to $Far(a)$. Otherwise, the algorithm recurses through the left and right children, merging nodes to extend the off-diagonal blocks that will be approximated.

Skeletonization

The off-diagonal blocks are approximated using interpolative decomposition. A skeleton of each off-diagonal block is constructed by selecting a subset of columns from the block. The decomposition is expressed as:

$$K_{ij} = K_{iB}P,$$

Algorithm 3 Far(a)[39]

for all $\beta \in$ leaf nodes **do**
  if $\alpha \notin \text{Near}(\beta)$ **then**
    $Far(\text{left\_child}_\alpha)$
    $Far(\text{right\_child}_\alpha)$
  else
    $Far(\alpha) = Far(\alpha) \cup \beta$
  end if
end for
$Far(\alpha) = Far(\text{left\_child}_\alpha) \cup Far(\text{right\_child}_\alpha)$
$Far(\text{left\_child}_\alpha) = Far(\text{left\_child}_\alpha) \setminus Far(\alpha)$
$Far(\text{right\_child}_\alpha) = Far(\text{right\_child}\alpha) \setminus Far(\alpha)$

where B represents a leaf node and i is the complement of B within the set. The matrix P contains the interpolation coefficients, and K_{iB} is the skeleton matrix formed from the subset of columns corresponding to B. For non-leaf nodes, the skeletons of the left and right children are recursively computed and then combined to obtain the decomposition for the parent block.

2.5.2 Factorization and hierarchical pseudo-inverse

A hierarchical approach is crucial for efficiently solving systems of linear equations. In this work, we utilize matrix factorizations and hierarchical low-rank approximations to compute approximate inversions of dense, nonsingular matrices. Consider the linear system:

$$Kx = b, \tag{2.2}$$

where K is a dense matrix, $b \in \mathbb{R}^N$ is a given vector, and x is the unknown solution vector. To facilitate computation, we partition K into block matrices:

$$K = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix}.$$

The inverse of K can be efficiently computed using the Sherman-Morrison-Woodbury (SMW) formula [11, 32]:

$$K^{-1} = \begin{bmatrix} K_{11}^{-1} + K_{11}^{-1}K_{12}S^{-1}K_{21}K_{11}^{-1} & -K_{11}^{-1}K_{12}S^{-1} \\ -S^{-1}K_{21}K_{11}^{-1} & S^{-1} \end{bmatrix},$$

where $S = K_{22} - K_{21}K_{11}^{-1}K_{12}$ is the Schur complement. In the context of hierarchical matrices ($\mathcal{H}$-matrices), this decomposition is referred to as the hierarchical inverse or $\mathcal{H}$-inverse. The SMW formula provides an exact inversion but can lead to numerical instability [15].

To mitigate instability, we use low-rank approximations, particularly for off-diagonal blocks like K_{12}, which avoids the direct inversion of these blocks. Hierarchical pseudo-inverses are employed to enable preconditioning in iterative solvers, such as preconditioned conjugate gradient methods [40, 30].

We apply this hierarchical structure in GOFMM by decomposing the matrix H into the form:

$$H = D + UV + S,$$

where D is block-diagonal, UV represents low-rank terms, and S is a sparse correction matrix. While hierarchical semi-separable (HSS) matrices do not involve sparse corrections, FMM requires corrections based on distances between particles in neighboring regions. For computational efficiency, we restrict sparse corrections to HSS without using the Schur complement.

To approximate the inverse of $H = D + UV$, we utilize the SMW formula:

$$H^{-1} = D^{-1} - D^{-1}U(I + VD^{-1}U)VD^{-1},$$

where D is a block-diagonal matrix that is easy to invert. This hierarchical framework simplifies inversion to smaller submatrices and reduces the computational complexity.

To further enhance efficiency, we follow the ULV factorization approach for hierarchical matrices [40], which allows partial pivoted LU factorizations on matrix blocks. This method ensures that we only need to compute low-rank approximations for certain blocks, minimizing computational overhead[30].

2.6 Applications

2.6.1 Kernel ridge regression

Kernel ridge regression is a regularized version of Ridge Regression that utilizes kernel methods to handle non-linear relationships between features. The objective function in KRR combines the least squares loss with an L_2 regularization term. The regularized objective function is given by:

$$J(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \lambda\|\mathbf{w}\|_2^2$$

where:

- $\mathbf{X}$ is the design matrix of size $N \times D$ (with N samples and D features),
- $\mathbf{w}$ is the weight vector of size $D \times 1$, which is obtained by minimizing the objective function:
$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{t},$$
- $\mathbf{t}$ is the target vector of size $N \times 1$,
- λ is the regularization parameter.

The formulation and solution for Ridge Regression, including the regularized objective function and the weight vector solution, are discussed in detail by Murphy [22].

Kernel trick

The kernel trick is a technique used to extend the power of linear models to non-linear problems by implicitly mapping the input data into a higher-dimensional feature space. This is achieved using a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$, which computes the inner product in the feature space without explicitly performing the transformation. The kernel matrix $\mathbf{K}$ is defined as:

$$\mathbf{K}_{ij} = K(\mathbf{x}_i, \mathbf{x}_j).$$

This allows for the computation of dot products in the higher-dimensional space efficiently [4].

In the kernel space, the objective function for ridge regression becomes:

$$J(\boldsymbol{\alpha}) = \|\mathbf{K}\boldsymbol{\alpha} - \mathbf{t}\|_2^2 + \lambda\boldsymbol{\alpha}^T\mathbf{K}\boldsymbol{\alpha},$$

where $\boldsymbol{\alpha}$ is the weight vector in the kernel space.

To find the optimal $\boldsymbol{\alpha}$, we take the derivative of $J(\boldsymbol{\alpha})$ with respect to $\boldsymbol{\alpha}$ and set it to zero:

$$\frac{\partial J}{\partial \boldsymbol{\alpha}} = 2\mathbf{K}^T(\mathbf{K}\boldsymbol{\alpha} - \mathbf{t}) + 2\lambda\mathbf{K}\boldsymbol{\alpha} = 0.$$

Solving this equation for $\boldsymbol{\alpha}$ gives:

$$(\mathbf{K} + \lambda \mathbf{I}_N)\boldsymbol{\alpha} = \mathbf{t}.$$

The solution for $\boldsymbol{\alpha}$ is:

$$\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1}\mathbf{t}. \tag{2.3}$$

This formula [38, 33] provides the weights in the kernel space. To make predictions for new data points, compute the kernel vector $\mathbf{k}_*$ between the new point and all training points, and use:

$$\mathbf{y}_* = \mathbf{k}_*^T \boldsymbol{\alpha}.$$

The ridge regression algorithm is implemented as follows :

Algorithm 4 Kernel ridge regression[33]

Input: Training set $S = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, regularization parameter $\lambda > 0$
Output: Weight vector w, dual coefficients α^*, and/or function f

1: Compute dual coefficients: $\alpha^* = (K + \lambda I)^{-1}\mathbf{y}$
2: Compute function $f(x) = \sum_{j=1}^{N} \alpha_j^* \kappa(x_j, x)$
3: Compute weight vector $\mathbf{w} = \sum_{j=1}^{N} \alpha_j^* \phi(x_j)$

2.6.2 Gaussian processes

A Gaussian process is a stochastic process that defines a distribution over functions. It is defined by a mean function $m(\mathbf{x})$ and a covariance function $k(\mathbf{x}, \mathbf{x}')$. We denote a GP as [4, 28]:

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

Here, $\mathbf{x}$ represents the input variable (often a vector in a multidimensional space), and $f(\mathbf{x})$ represents the corresponding output variable.

Before observing any data, we assume a prior distribution over functions. Typically, we assume a zero-mean prior, i.e., $m(\mathbf{x}) = 0$. The covariance function $k(\mathbf{x}, \mathbf{x}')$ captures the relationships between different input points [28].

A commonly used covariance function is the Radial Basis Function (RBF) or Gaussian kernel:

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right)$$

Here, ℓ is a hyperparameter that controls the smoothness of the function.

After observing data, we update our prior beliefs to obtain a posterior distribution over functions. Given n observed data points $\{\mathbf{X}, \mathbf{y}\}$, where $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]^\top$ and $\mathbf{y} = [y_1, y_2, \ldots, y_n]^\top$, the joint distribution of the observed outputs and the function values at the test points $\mathbf{X}_*$ is Gaussian:

$$\begin{pmatrix} \mathbf{y} \\ f(\mathbf{X}_*) \end{pmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} K(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I} & K(\mathbf{X}, \mathbf{X}_*) \\ K(\mathbf{X}_*, \mathbf{X}) & K(\mathbf{X}_*, \mathbf{X}_*) \end{pmatrix}\right)$$

Here, $K(\mathbf{X}, \mathbf{X})$ is the covariance matrix of the training points, $K(\mathbf{X}_*, \mathbf{X})$ is the covariance matrix between test and training points, $K(\mathbf{X}_*, \mathbf{X}_*)$ is the covariance matrix of the test points, and $\sigma^2 \mathbf{I}$ accounts for noise in the observations [28].

Predictive mean and variance

From the joint distribution, we can derive the posterior mean and covariance for the test points $\mathbf{X}_*$[34] :

1. **Predictive mean**:

$$\mu_* = K(\mathbf{X}_*, \mathbf{X}) \left[K(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}\right]^{-1} \mathbf{y} \tag{2.4}$$

This gives the expected values (mean predictions) of the function at the test points.

2. **Predictive variance**:

$$\mathbf{\Sigma}_* = K(\mathbf{X}_*, \mathbf{X}_*) - K(\mathbf{X}_*, \mathbf{X}) \left[K(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}\right]^{-1} K(\mathbf{X}, \mathbf{X}_*) \tag{2.5}$$

This gives the uncertainties (variances) of the predictions.

Use of kernel inverse for mean and covariance calculation

To compute the predictive mean and covariance, we use 2.4 and 2.5 so we need to invert the kernel matrix $K(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}$. One efficient way to do this is by using the GOFMM inverse of the kernel matrix.

Next, we will delve into how to compute the inverse of the kernel matrix and its role in calculating the predictive mean and covariance.

3 Scalable kernel matrix inversion

3.1 Setup

In this section, we investigate the application of the GOFMM for the inversion of kernel matrices. Specifically, we apply GOFMM techniques, as detailed in [39], to Python data structures using the Simplified Wrapper Interface Generator (*SWIG*). *SWIG* seamlessly integrates the GOFMM C++ methods into Python by generating the necessary interface and code, thereby enabling the use of the C++ implementation within Python scripts without requiring additional compilation.

The entire integrated environment is packaged within a *Docker* container. Due to the absence of *Docker* support on the Linux cluster at LRZ, the *Docker* image is converted to a *Charliecloud* image [27]. This *Charliecloud* image is then transferred to the Linux cluster, where accuracy evaluations are conducted on datasets including synthetic data [35] and the MNIST dataset [19]. For single-node tests, we compare the computed kernel inversions with those obtained using the *SciPy* library. While multi-node setups were considered, they were not tested within the scope of this thesis. Instead, For future work, multi-node configurations can be explored using C++ and compared with inversion techniques supported by C++ libraries.

To evaluate the performance of the integrated environment, we conduct both strong and weak scaling experiments using multiple cores. Runtime data is collected to assess the scalability and efficiency of the GOFMM-based inversion on the Linux cluster.

3.2 Error evaluation

Frobenius Norm

The Frobenius norm [13] is a matrix norm that measures the magnitude of a matrix's entries. It is defined as the square root of the sum of the absolute squares of its elements. Formally, for a matrix $A \in \mathbb{R}^{m \times n}$, the Frobenius norm $\|A\|_F$ is given by:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}, \tag{3.1}$$

where a_{ij} denotes the entry in the i-th row and j-th column of the matrix A.

This norm is particularly useful for comparing the accuracy of various methods for computing the inverse of kernel matrices. In the context of kernel matrix inversion, the Frobenius norm can be used to evaluate the difference between the computed inverse and the inverse obtained from other methods, such as *SciPy*'s dense matrix inversion. It provides a measure of the overall error magnitude, which helps in assessing the performance of different computational approaches [14].

Relative Squared Error (RSE)

To provide a more nuanced evaluation of the inversion accuracy, we use the Relative Squared Error (RSE)[36, 13], which normalizes the Frobenius norm of the difference between the expected and computed matrices. The RSE is defined as:

$$\text{RSE} = \frac{\|\mathbf{M}_{\text{exp}} - \mathbf{M}_{\text{the}}\|_F}{\sqrt{\sum_{i,j}(\mathbf{M}_{\text{the}})_{ij}^2}} \times 100, \tag{3.2}$$

where $\mathbf{M}_{\text{exp}}$ represents the reference matrix (i.e., the inverse computed by *SciPy*), and $\mathbf{M}_{\text{the}}$ is the matrix computed using GOFMM. The numerator, $\|\mathbf{M}_{\text{exp}} - \mathbf{M}_{\text{the}}\|_F$, measures the Frobenius norm of the error between the two matrices. The denominator, $\sqrt{\sum_{i,j}(\mathbf{M}_{\text{the}})_{ij}^2}$, normalizes this error relative to the magnitude of the computed matrix.

The RSE provides a percentage-based measure of the error relative to the magnitude of the computed matrix. This metric is particularly useful for evaluating and comparing the accuracy of different matrix inversion methods, as it allows for an assessment of how significant the error is relative to the size of the matrix being computed. In the context of this thesis, the RSE is used to evaluate the performance of the GOFMM inversion method in comparison to the other standard matrix inversion approaches.

Comparison with *SciPy* Inverse Computation

In evaluating the performance of the GOFMM for computing the inverse of dense kernel matrices, we use the *SciPy* library's **linalg.inv** function as a reference [18]. This approach allows us to benchmark the accuracy and efficiency of our GOFMM method against a well-established and widely used dense matrix inversion technique.

The *SciPy* function provides a direct and precise computation of the matrix inverse, which serves as our baseline for comparison. By contrasting the results from GOFMM with those obtained using *SciPy*, we aim to assess the effectiveness of GOFMM in approximating the inverse of kernel matrices, particularly in terms of accuracy and computational performance.

The inverse of the kernel matrix is computed using *SciPy* as follows:

```
from scipy.linalg import inv
K_scipy_inv = inv(K)
```

Listing 3.1: Scipy inversion

where K_{reg} represents the kernel matrix used in KRR or Gaussian GP. The result $K_{\text{scipy\_inv}}$ is used as a reference for evaluating the accuracy of the inverses computed by GOFMM.

- **Accuracy**: We compare the inverses produced by GOFMM with $K_{\text{scipy\_inv}}$ to evaluate the relative error. This comparison ensures that the approximations provided by GOFMM are sufficiently close to the exact results from *SciPy*.
- **Efficiency**: The computational time and resource usage for GOFMM are compared against those required by *SciPy* for matrix inversion. This evaluation highlights the potential scalability benefits of GOFMM for large-scale problems.

The comparison underscores the trade-offs between approximation accuracy and computational efficiency. By benchmarking against *SciPy*, which is known for its reliable and accurate inversion methods [37], we validate the performance of GOFMM in handling large and complex kernel matrices.

3.3 Implementation

We integrate all components for the implementation in a sequential workflow. First, we retrieve or generate the required data. Next, we construct the SPD kernel using either the Kernel ridge regression or Gaussian processes packages in Python. Following this, we compute the inverse of the kernel matrix utilizing the integrated GOFMM methods, accessed within Python via *SWIG*. Finally, we evaluate the accuracy of the inversion by comparing it against the *SciPy* inversion results, and we measure the computational time for the entire process.

Python Interface Setup

Inversion process We begin by defining a class, `Inverse_calculator`, which encapsulates the inversion process using the GOFMM functions already implemented in C++ and integrated into Python via *SWIG*. The class takes as input the kernel matrix, created from the data, and computes its inverse using the GOFMM algorithms. The implementation of this class is provided in Listing 3.2.

The class parameters, such as `problem_size` and `matrix_type`, are essential for constructing a GOFMM tree, which is instrumental in compressing and processing the kernel matrix. The constructor initializes all member variables and loads the NumPy matrix into the *SWIG* interface methods. Specifically, the method `LoadDenseSpdMatrixFromConsole`

converts the NumPy matrix into an object of type `SPDMATRIX_DENSE`, which is compatible with the C++ GOFMM implementation.

For computing the inverse, we utilize the method `matinv`, as shown in Listing 3.2, which internally calls the `InverseOfDenseSpdMatrix` function integrated from the GOFMM method in C++.

```
class Inverse_calculator:
    def __init__(self, executable, problem_size, max_leaf_node_size,
        num_of_neighbors,
                 max_off_diagonal_ranks, num_rhs, user_tolerance,
                     computation_budget,
                 distance_type, matrix_type, kernel_type, spd_matrix):
        self.executable = executable
        self.problem_size = problem_size
        self.max_leaf_node_size = max_leaf_node_size
        self.num_of_neighbors = num_of_neighbors
        self.max_off_diagonal_ranks = max_off_diagonal_ranks
        self.num_rhs = num_rhs
        self.user_tolerance = user_tolerance
        self.computation_budget = computation_budget
        self.distance_type = distance_type
        self.matrix_type = matrix_type
        self.kernel_type = kernel_type
        self.spd_matrix = np.float32(spd_matrix)  # from input
        # Convert the SPD matrix to a SPDMATRIX_DENSE structure for GOFMM
        self.denseSpd = tools.LoadDenseSpdMatrixFromConsole(self.spd_matrix)
        self.matrix_length = self.problem_size * self.problem_size

    def matinv(self, lambda_inv):
        # Create GOFMM tree from the SPD matrix
        gofmmCalculator = tools.GofmmTree(self.executable, self.problem_size,
                                          self.max_leaf_node_size, self.
                                              num_of_neighbors,
                                          self.max_off_diagonal_ranks, self.
                                              num_rhs,
                                          self.user_tolerance, self.
                                              computation_budget,
                                          self.distance_type, self.matrix_type,
                                          self.kernel_type, self.denseSpd)
        # Compute the inverse using the GOFMM method
        c = gofmmCalculator.InverseOfDenseSpdMatrix(lambda_inv, self.
            matrix_length)
        print("GOFMM Inverse computation completed")

        # Reshape the result to an n x n matrix
        inv_matrix = np.resize(c, (self.problem_size, self.problem_size))
        return inv_matrix

    def compute_rse(self, matExp, matThe):
        return np.linalg.norm(matExp - matThe) / np.sqrt(np.sum(matThe ** 2)) *
            100
```

Listing 3.2: Class `Inverse_calculator`

In the final step of the inversion process, we begin by setting up the necessary parameters for the implementation. The parameters include the problem size, maximum leaf node size, number of neighbors, and various tolerances and settings required for the GOFMM algorithm. With the kernel matrix prepared, we initialize an instance of the `Inverse_calculator` class, which is designed for the SPD kernel inversion. The matrix inversion is then executed through the `matinv` method, which applies the GOFMM algorithms to obtain the inverse of the regularized kernel matrix. For comparison, the inverse is also computed using the standard *SciPy* `inv` function. The RSE is then calculated to assess the accuracy of the GOFMM-based inversion like shown in 3.3.

```

# Parameters for GOFMM
executable = "./test_gofmm"
max_leaf_node_size = int(problem_size / 2)
num_of_neighbors = 128
max_off_diagonal_ranks = int(problem_size / 2)
num_rhs = 1
user_tolerance = 1E-5
computation_budget = 0.00
distance_type = "kernel"
matrix_type = "dense"
kernel_type = "gaussian"
lambda_inv = 1.0  # regularization parameter

# Prepare inverse GOFMM calculator
kernel_matrix = K.astype("float32")
inverse_GOFMM_obj = Inverse_calculator(executable, problem_size,
    max_leaf_node_size,
num_of_neighbors, max_off_diagonal_ranks, num_rhs, user_tolerance,
    computation_budget, distance_type, matrix_type, kernel_type, K)

# INVERSE KERNEL using GOFMM
inv_gofmm = inverse_GOFMM_obj.matinv(lambda_inv)

# Compute the inverse of the regularized kernel matrix using numpy
K_reg = K + lambda_inv * np.eye(len(X_train))
K_reg_inv = inv(K_reg)

# Compute RSE of inverse
rse = inverse_GOFMM_obj.compute_rse(inv_gofmm, K_reg_inv)
```

Listing 3.3: Parameters and code to compute the inverse of the SPD Kernel

SPD Kernel Creation Using KRR and GP In this part, we outline the process of creating SPD kernels using both KRR and GP within Python. The methodologies for both

techniques share several common steps, which we summarize here.

For the KRR-based approach in the listing 3.4, we utilize the Gaussian Radial Basis Function (RBF) kernel. The `KernelRidge` class from the `sklearn.kernel_ridge` module is employed to fit the model on the training data. The kernel matrix, K, is computed using `pairwise_kernels` from `sklearn.metrics.pairwise`, allowing us to generate the SPD kernel required for subsequent inversion operations. The inverse of this kernel is then computed using both the GOFMM method and a direct *SciPy*-based approach for comparison. Finally, the learned weights are calculated using the inverted kernel matrices like discussed before in in 2.6.1.

Similarly, for the GP-based approach 3.5, we leverage the RBF kernel through the `GaussianProcessRegressor` class in `sklearn.gaussian_process`. The GP model is fitted to the training data, and kernel evaluations are computed between the test and training points. The SPD kernel matrix is then inverted, again using GOFMM and *SciPy*-based methods, to compute the predictive mean and covariance for the GP which is discussed in 2.6.2 . These inversions are crucial for deriving the predictive distribution and assessing the accuracy of the GP model.

Both approaches demonstrate the versatility of SPD kernels in machine learning tasks, whether for regression via KRR or probabilistic modeling with GP. The integration of GOFMM for kernel inversion enhances computational efficiency, particularly for large-scale problems, as discussed in subsequent sections.

```
# package for KRR Kernel creation
from sklearn.kernel_ridge import KernelRidge
from sklearn.metrics.pairwise import pairwise_kernels
# Some code here...

    # ...
    # Inverse Class creation and data retrieve
    # ...
# Preprocessing the data
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# Flattening the images
X_train = x_train.reshape((x_train.shape[0], -1))
X_test = x_test.reshape((x_test.shape[0], -1))

# Reducing dataset size for testing purposes
# Set problem size
problem_size = int(os.getenv('PROBLEM_SIZE', 2048))  # default if not set
X_train = X_train[:problem_size]
y_train = y_train[:problem_size]

# Initialize KernelRidge with Gaussian (RBF) kernel
krr = KernelRidge(kernel='rbf', gamma=0.1)

# Fit the model
```

```
krr.fit(X_train, y_train)

# Calculate the Gaussian kernel matrix using pairwise_kernels
K = pairwise_kernels(X_train, metric='rbf', gamma=0.1)
# Some code here...

    # ...
    # Inverse computation
    # ...

# Calculate the weights for KRR
weights_np = np.dot(K_reg_inv, y_train)
weights_gofmm = np.dot(inv_gofmm, y_train)

# Get the learned weights of the SKLEARN
weights = krr.dual_coef_
```

Listing 3.4: SPD Kernel creation using Kernel ridge regression

```
# package for Gaussian processes Kernel creation
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
# Some code here...

    # ...
    # Inverse Class creation and Data retrieve
    # ...
# Use a subset of the data for quicker computation
x_train, _, y_train, _ = train_test_split(x_train, y_train, train_size=
    problem_size, stratify=y_train, random_state=random_state)
x_test, _, y_test, _ = train_test_split(x_test, y_test, test_size=1024, stratify
    =y_test, random_state=random_state)

# Define kernel
kernel_standard = 1.0 * RBF(length_scale=1.0)

# Standard GP with scikit-learn
gp_standard = GaussianProcessRegressor(kernel=kernel_standard, alpha=0.1)
gp_standard.fit(x_train, y_train)
mu_star_sklearn, std_star_sklearn = gp_standard.predict(x_test, return_std=True)
# Some code here...

    # ...
    # Inverse computation
    # ...

# Compute kernel evaluations between test and training points
k_star = kernel_standard(x_train, x_test)
k_star_star = kernel_standard(x_test, x_test)

# Compute predictive mean for GP
```

```
mu_star = k_star.T @ inv_gofmm @ y_train
mu_star_np = k_star.T @ inv_spd @ y_train
sigma_star = k_star_star - (k_star.T @ inv_gofmm @ k_star)
```

Listing 3.5: SPD Kernel creation using Gaussian processes

3.4 Datasets

Synthetic Data

In evaluating scalable algorithms, especially for strong and weak scaling tasks, synthetic data offers a valuable advantage. It allows for controlled testing by isolating performance characteristics from the variability and noise inherent in real-world datasets [4]. Synthetic data is generated through algorithms or simulations to replicate the statistical properties and patterns of real-world data and is widely used in machine learning, statistical modeling, and software testing [35].

For testing and validating the scalability of the Kernel inversion algorithm, various synthetic data types were considered, including uniform distributions, circular distributions, Gaussian Mixture Models (GMMs), and other common techniques. Each data type provides unique benefits depending on the nature of the algorithm under test. After thorough evaluation, the Gaussian distribution was selected as the primary synthetic data model due to its broad applicability and relevance.

The Gaussian distribution, or normal distribution, is characterized by its bell-shaped curve with data points concentrated around the mean. It is defined mathematically by the probability density function:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{3.3}$$

where μ is the mean and σ^2 is the variance [26]. This distribution is chosen for its alignment with the statistical properties encountered in real-world scenarios which is shown in the figures [3.1, 3.2], ensuring that generated data reflects natural phenomena [31]. Its symmetry and adjustable variance make it ideal for testing the robustness and performance of scalable algorithms under varying conditions [26].

The Gaussian distribution's suitability extends to both strong and weak scaling tests [24]. The following Python code snippet demonstrates the generation of Gaussian-distributed data:

```
def generate_gaussian_data(size, mean=[0, 0], cov=[[1, 0.5], [0.5, 1]]):
    rng = np.random.default_rng(random_state)
    return rng.multivariate_normal(mean, cov, size)
```

Listing 3.6: Generating Gaussian Distributed Data

In addition to selecting an appropriate data distribution, the application of non-linear functions is crucial for testing the kernel inversion algorithm. Non-linear transformations, applied to synthetic data, simulate more complex relationships commonly encountered in real-world scenarios [4, 22]. These transformations increase the data's dimensionality and complexity, providing a rigorous test for the inversion process of the GOFMM (Geometric Multi-Front Matrix Multiplication) method [21].

For example, applying non-linear functions such as polynomial or sigmoid transformations to Gaussian-distributed data mimics the non-linear relationships found in practical machine learning tasks [28]. This makes the synthetic data more challenging and reflective of real-world conditions. The application of these transformations ensures that the GOFMM algorithm is robust and scalable, even with non-linearly separable data, thoroughly validating its applicability in complex scenarios [7].

```
X_train = generate_gaussian_data(data_count)
y_train = nonlinear_function(X_train[:, 0])  # Using only the first feature for
    the nonlinear function
```

Listing 3.7: Non-linear Function for Predictions Using Synthetic Data

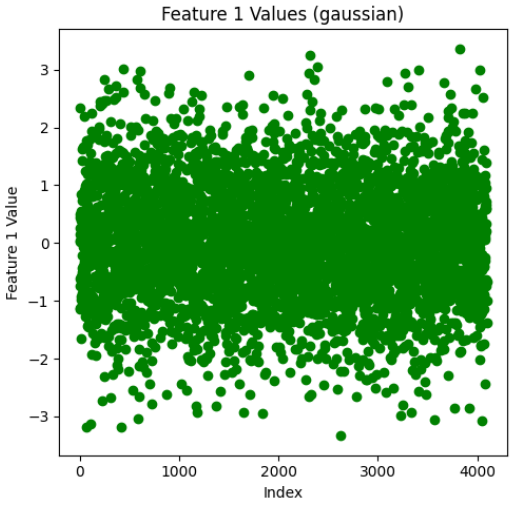

Figure 3.1: Gaussian distribution sampling of size 4096 with mean 0

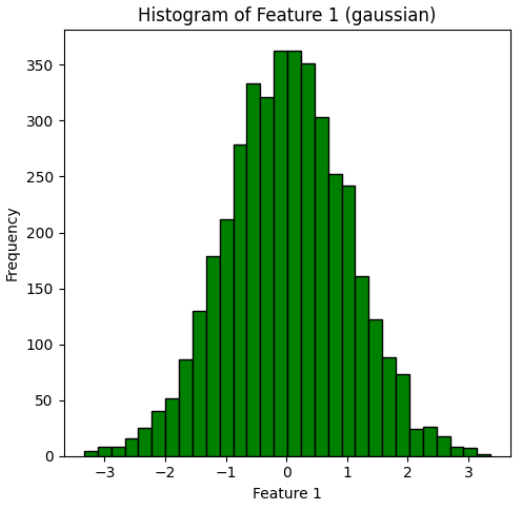

Figure 3.2: Histogram of the Gaussian distribution sampling

MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is a widely used benchmark in the field of machine learning and image recognition. It consists of

60,000 training images and 10,000 test images of handwritten digits, each 28x28 pixels in size [19]. The dataset provides a standardized, publicly available set of images that facilitate the development and evaluation of image classification algorithms. For the purposes of this thesis on scalable kernel matrix inversion using hierarchical low-rank approximations, the MNIST dataset serves as an ideal choice due to its well-defined structure and moderate size. Its simplicity allows for clear insights into the performance of kernel matrix inversion techniques under different scaling conditions. Additionally, the high dimensionality of the image data poses a meaningful challenge for testing the scalability and efficiency of the proposed methods, making it a valuable dataset for evaluating the practical aspects of the algorithm [5].

```
from tensorflow import keras
import os

# Loading the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Preprocessing the data
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# Flattening the images
X_train = x_train.reshape((x_train.shape[0], -1))
X_test = x_test.reshape((x_test.shape[0], -1))

# Reducing dataset size for testing purposes
# Set problem size
problem_size = int(os.getenv('PROBLEM_SIZE', 2048))  # default size if not set
X_train = X_train[:problem_size]
y_train = y_train[:problem_size]
```

Listing 3.8: Loading and Preprocessing the MNIST Dataset

Using both Gaussian distribution samples and the MNIST dataset in our experiments provides a robust evaluation of our algorithms. Gaussian samples offer a controlled environment with known statistical properties, allowing us to benchmark the performance under ideal conditions and test theoretical assumptions. In contrast, the MNIST dataset, representing real-world handwritten digits, introduces complexities such as noise and variability, enabling us to assess the algorithms' generalizability and practical effectiveness. This dual approach ensures that our methods are not only theoretically sound but also applicable to real-world scenarios.

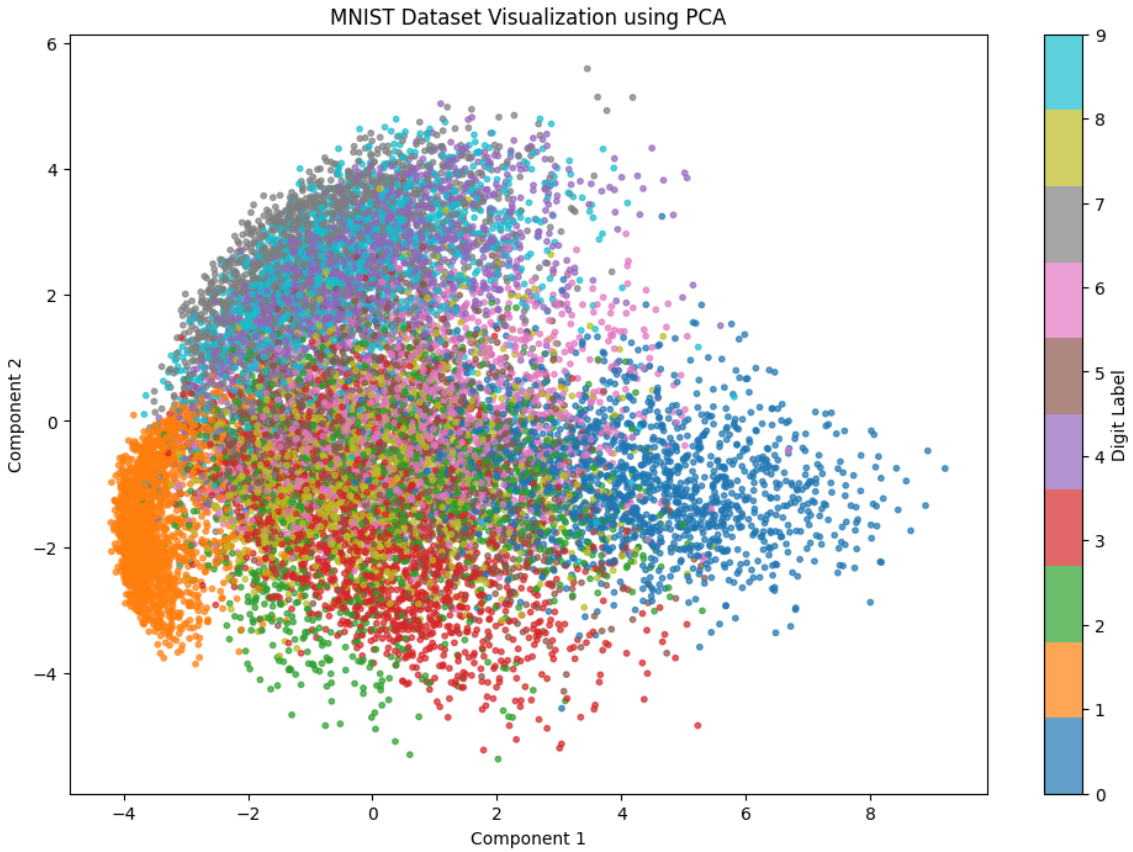

Figure 3.3: MNIST dataset with 16K samples, after reduction using PCA

4 Numerical Experiments

4.1 Accuracy analysis

The integration of operations from GOFMM is run on the CoolMUC-2 cluster with 28-way Intel Xeon E5-2690 v3 (”Haswell”) based nodes and FDR14 Infiniband interconnect. CoolMUC-2 has 812 nodes with 64GB memory per node. Therefore, the accuracy measurements and multi-core scaling experiments with OMP number of threads less than 28 are conducted on the compute node lxlogin1 of CoolMUC-2[1] . The work using OMP was done using 28 threads with same node for tolerance and accuracy measurements.

Tolerance measurements

We analyze the performance of KRR and GP Kernels inversion using GOFMM across varying tolerance levels with the goal of determining an optimal balance between computational efficiency and accuracy. We used the *SciPy* inversion as a reference. Table 4.1 presents the computation times and associated errors for different tolerance levels ($E-3$, $E-5$, and $E-7$) for matrix sizes 8192 and 4096, using 28 threads.

Kernel ridge regression				**Gaussian processes**			
Size	**Tolerances**	**Time(s)**	**Errors**	**Size**	**Tolerances**	**Time(s)**	**Errors**
8192	E-3	32	4,15 E-2	8192	E-3	32.3	2,34
	E-5	34,3	5,14 E-03		E-5	35,24	2,15E-01
	E-7	55,5	2,76 E-03		E-7	52,96	1,81E-01

Table 4.1: Varying Tolerances for the same number of threads (28 Threads)

For Optimal Tolerance Selection, Table 4.1 demonstrates that adopting a tolerance level of $E-5$ yields a significant enhancement in accuracy relative to $E-3$, with only a modest increase in computational time:

- In the case of KRR, the error is reduced by nearly an order of magnitude when transitioning from $E-3$ to $E-5$, with a negligible impact on execution time.
- For GP, the error diminishes dramatically, also by several orders of magnitude, as the tolerance is tightened from $E-3$ to $E-5$, while the computation time remains virtually constant.

[1] LRZ, Overview of HPC Systems

While the use of $E - 7$ achieves marginally greater accuracy, it does so at the expense of a considerable increase in computation time, making $E - 5$ a more judicious choice for balancing precision and computational efficiency.

Problem Size/ Threads	8192	16384
1	7.38E-05	8.87E-05
2	9.48E-05	8.86E-05
4	7.28E-05	9.22E-05
6	7.27E-05	9.26E-05
8	7.27E-05	9,25E-05
16	7.39E-05	8.88E-05
28	1.68E-04	8.88E-05

Table 4.2: Errors for Gaussian processes for problem sizes: 8192, 16382

Problem Size/ Threads	8192	16384
1	9.84E-05	1.33E-04
2	9.88E-05	1.28E-04
4	9.77E-05	1.32E-04
6	7.27E-05	9.26E-05
8	9.80E-05	1.26E-04
16	9.80E-05	1.32E-04
28	9.93E-05	1.34E-04

Table 4.3: Errors for KRR for problem sizes: 8192, 16382

Error measurements

The data presented in the tables [4.2, 4.3] for GP and KRR provide insights into the error rates associated with different problem sizes and thread counts when computing the kernel inverse using the GOFMM.

For Gaussian processes in table 4.2 , the error rates for problem sizes 8192 and 16384 remain relatively stable across different thread counts, with slight variations. Notably, the error rate for 8192 threads shows a minor increase from 7.38×10^{-5} to 1.68×10^{-4} as the number of threads increases from 1 to 28. Similarly, for 16384 threads, the error rate fluctuates slightly, indicating that the inversion maintains a consistent level of accuracy across varying computational loads.

In the case of KRR in table 4.3 , the error rates exhibit a similar trend. For 8192 threads, the error rate starts at 9.84×10^{-5} and increases marginally to 9.93×10^{-5} as the number of

threads increases. For 16384 threads, the error rate shows a slight increase from 1.33×10^{-4} to 1.34×10^{-4}.

Overall, the analysis indicates that the GOFMM is a robust method for computing the Kernel inverse, providing consistent and low error rates across different problem sizes and thread counts. This stability is crucial for applications requiring high precision and efficiency, such as those involving large-scale machine learning tasks with the MNIST dataset.

4.2 Multi-core measurements

Weak scaling, or strong scaling with respect to problem size, measures how the computational performance of a parallel system changes as the problem size grows proportionally with the number of processors

Weak scaling

Weak scaling is a critical metric for assessing the performance of parallel systems, particularly as the computational demands increase. Weak scaling focuses on how well a system handles larger workloads as the problem size grows proportionally with the number of processors. In this context, the workload per processor remains constant, ensuring that the computational burden does not change as more processors are added. Mathematically, if N represents the problem size and P denotes the number of processors (or threads), the problem size per processor can be expressed as:

$$\frac{N}{P} = \text{constant} \tag{4.1}$$

For a parallel system to exhibit effective weak scaling, this ratio should remain constant as P increases. This efficiency is crucial for applications that require the processing of increasingly large datasets or more complex models, as it indicates that the system can scale effectively without introducing significant overheads or bottlenecks [29, 6]. Evaluating weak scaling is essential for pinpointing inefficiencies in both parallel algorithms and hardware configurations, helping to optimize performance [25].

In our specific experiments, we assessed weak scaling using the MNIST dataset, on a single node, employing multi-core processing, where the number of threads ranged from 1 to 16, and the problem size was scaled from 1024 to 16384 . The tables 4.4 and 4.5 show the weak scaling results for the inverse computation using the GOFMM for kernels generated from GP and KRR, respectively. The problem size is increased proportionally with the number of threads, while the efficiency and time duration for each configuration are recorded.

The decrease in efficiency observed for both GP and KRR with increasing thread counts could be due to several factors. As the number of threads grows, communication overhead often increases, leading to delays. Additionally, synchronization costs rise, which

Number Threads	1	2	4	8	16
Problem size	1024	2048	4096	8192	16384
Time duration (in secs)	2.52	4.91	12.62	45.42	209.28
Efficiency	1	0.513	0.2	0.055	0.00378

Table 4.4: Weak Scaling of GPs kernel for MNIST Dataset

Number Threads	1	2	4	8	16
Problem size	1024	2048	4096	8192	16382
Time duration (in secs)	0.79	2.56	11.81	45.7	239.16
Efficiency	1	0.31	0.067	0.0173	0.0033

Table 4.5: Weak Scaling of KRR kernel for MNIST Dataset

can result in idle times as threads wait for each other. Load imbalance may also become an issue, with some threads finishing their tasks sooner than others, thus reducing overall efficiency. Contention for shared memory resources might further impact performance as the number of threads increases.

These potential causes are speculative and will be explored in more detail in the upcoming performance analysis section, where we will examine these issues closely and consider possible optimization strategies.

Strong scaling

Strong scaling is an essential measure for evaluating the efficiency of parallel systems as the number of processors increases while the problem size remains fixed. Unlike weak scaling, which assesses performance with a proportional increase in both problem size and processors, strong scaling focuses on how well a system accelerates computation when more processors are applied to the same problem size. Mathematically, if N is the problem size and P is the number of processors (or threads), the ideal scenario is that the execution time $T(P)$ reduces by a factor proportional to $\frac{1}{P}$ as P increases:

$$T(P) = \frac{T(1)}{P} \tag{4.2}$$

For strong scaling to be effective, the system should approach this ideal linear speedup. This is critical for scenarios where reducing time to solution is paramount, such as in real-time applications or high-performance simulations [12]. Evaluating strong scaling provides insights into the parallel efficiency of algorithms and can highlight diminishing returns as more processors are added, often due to factors such as communication overhead or resource contention [6].

The results shown in Tables 4.6, 4.7, 4.8, and 4.9 illustrate the strong scaling performance of kernel inversion using the GOFMM. These experiments were conducted on kernels gen-

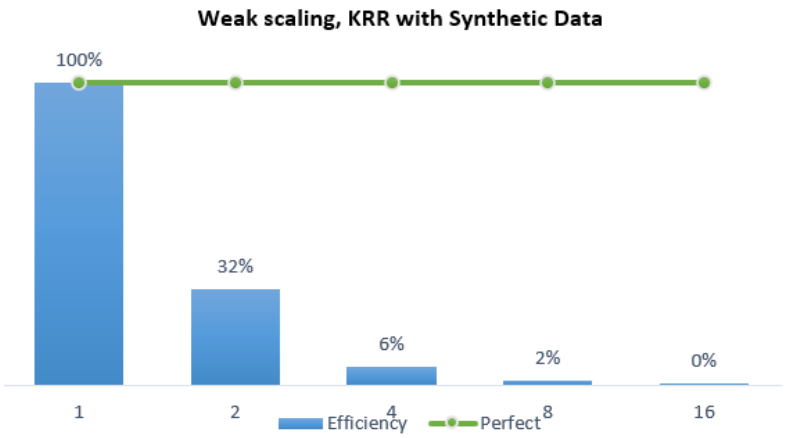

Figure 4.1: Weak scaling histogram for KRR using Gaussian distributions samples for the kernel inversion

erated from GP and KRR with varying problem sizes (8192 and 16384) and thread counts ranging from 1 to 28. As the number of threads increases, the computation time decreases, which is indicative of effective parallelization. However, the rate of speedup diminishes notably beyond 16 threads. The speedup achieved is generally higher for larger problem sizes (16384) compared to smaller ones (8192), which is expected in strong scaling scenarios.

Problem Size/Threads	**1**	**2**	**4**	**6**	**8**	**16**	**28**
8192	165.84	99.37	64.79	51.43	45.42	41.53	44.34
16384	1033.12	570.38	355.44	275.27	242.86	209.28	200.08

Table 4.6: Durations of GP kernels inversion using MNIST dataset

Problem Size/Threads	**1**	**2**	**4**	**6**	**8**	**16**	**28**
8192(in secs)	81.65	50.39	37.70	36.42	35.10	34.70	37.48
16384(in secs)	549.50	363.89	211.80	181.93	157.57	152.52	156.72

Table 4.7: Durations of GP kernels inversion using a synthetic dataset

The observed diminishing returns on speedup as the number of threads increases can be attributed to a few key factors. As more threads are added, the benefits of additional threads tend to plateau beyond a certain point, such as 16 threads, reflecting a decrease in parallel efficiency. Communication overhead also becomes more significant, which can

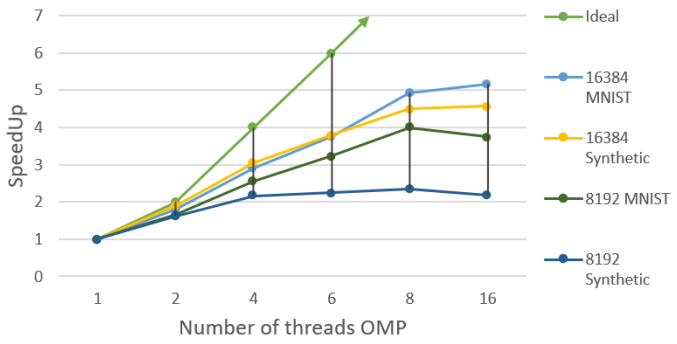

Figure 4.2: Strong scaling for GPs

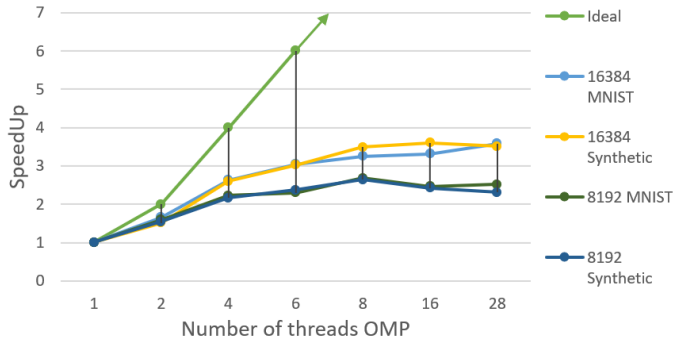

Figure 4.3: Strong scaling for KRR

Problem Size/Threads	**1**	**2**	**4**	**6**	**8**	**16**	**28**
8192(in secs)	122.38	76.74	55.04	53.35	45.70	49.62	48.71
16384(in secs)	792.15	476.84	301.72	260.21	243.71	239.16	221.17

Table 4.8: Durations of KRR kernels inversion using a MNIST dataset

Problem Size/Threads	**1**	**2**	**4**	**6**	**8**	**16**	**28**
8192(in secs)	80.58	52.34	37.20	33.98	30.50	33.20	34.89
16384(in secs)	519.32	273.87	169.83	136.85	122.72	115.60	113.74

Table 4.9: Durations of KRR kernels inversion using synthetic Data

reduce the gains from adding more threads. Additionally, competition for memory bandwidth can lead to saturation, where further increases in thread count do not yield proportional speedup. Load balancing becomes increasingly difficult, with slight imbalances causing some threads to be idle while others complete their tasks. Furthermore, while the GOFMM method effectively reduces computational complexity, it may face scalability

challenges, and the complexity of the algorithm combined with frequent thread synchronization can further impede performance.

4.3 Performance analysis

To understand the performance delays and inefficiencies observed , we perform a detailed profiling analysis. Additionally, we examine the runtime overhead introduced by the *Python-SWIG* interface compared to the native C++ implementation. This comprehensive analysis helps to pinpoint critical areas for optimization and understand the underlying causes of observed performance issues.

Profiling

In this section, we analyze the performance scaling of KRR and GP using profiling data obtained from the VTune Profiler [16]. This analysis reveals significant performance bottlenecks and inefficiencies that affect the scalability and runtime of these algorithms. In profiling tools like VTune Profiler, the metrics such as effective CPU time or overhead time appear larger than the elapsed wall clock time due to their nature of aggregation of the threads used and calculation.

The weak scaling results, detailed in Tables 4.4 and 4.5, show that as the problem size increases, the execution time grows substantially. For instance, the KRR kernel requires 0.79 seconds for a problem size of 1024 with one thread, but this time increases dramatically to 239.16 seconds for a problem size of 16384 with 16 threads. This corresponds to a severe drop in efficiency from 1 to 0.0033. A similar pattern is observed for the GP kernel. Such inefficiencies are consistent with the challenges described in the VTune Profiler User Guide, which highlights how increased problem sizes can exacerbate parallel processing issues [16].

Metric/Threads	2	4	8	16
Effective CPU utilization (%)	3.5	7	13.8	19.8
Effective CPU time (sec)	3.4	30.4	88.7	637
Lock contention (sec)	0.2	10.53	73.5	630
Overhead time (sec)	0.2	4.5	209	3707
Imbalance or serial spinning (sec)	0.21	0.7	8.4	137
Microarchitecture usage (%)	41.6	13.2	38	34

Table 4.10: CPU Utilization and metrics for KRR/MNIST.

The profiling data, as shown in Table 4.10, further elucidates these inefficiencies. Effective CPU utilization does increase with the number of threads, reaching 19.8% with 16 threads, yet this improvement is overshadowed by a disproportionate rise in effective CPU time and lock contention. Specifically, lock contention escalates from 0.2 seconds with 2 threads to 630 seconds with 16 threads, while overhead time surges from 0.2 seconds to 3707 seconds. These findings underscore significant inefficiencies in parallel resource management.

Figure 4.4 illustrates that certain functions, such as `gofmm::Compress`, `gofmm::Factorize`, and `gofmm::Solve`, are major contributors to performance degradation. `gofmm::Compress` accounts for 54.8% of the total overhead, with a lock contention of 19.5%. Similarly, `gofmm::Factorize` exhibits the same overhead and lock contention characteristics. Other functions, such as `gofmm::FindNeighbors` and `gofmm::Solve`, also contribute to overhead, albeit to a lesser degree. These functions collectively highlight critical performance bottlenecks, particularly related to kernel inversion operations.

Source Function Stack	ation ... Ov...	CPU Time: Total			
		Spin Time			Overhead Time
		Imbalance or Serial Spinning	Lock Contention	Other	
Total		2.3%	19.5%	0.1%	54.8%
_start		1.7%	19.5%	0.1%	54.8%
__libc_start_main		1.7%	19.5%	0.1%	54.8%
main		1.7%	19.5%	0.1%	54.8%
hmlp::gofmm::Compress<hmlp::gofmm::cente		0.1%	1.9%	0.0%	4.4%
hmlp::gofmm::Factorize<float, hmlp::tree::Tree		0.7%	8.1%	0.0%	21.6%
hmlp::gofmm::FindNeighbors<hmlp::gofmm::r		0.0%	0.1%	0.0%	0.2%
hmlp::gofmm::Solve<float, hmlp::tree::Tree<hm		0.8%	9.5%	0.0%	28.5%
hmlp::RunTime::init		0.0%	0.0%	0.0%	0.0%
hmlp::SPDMatrix<float>::SPDMatrix		0.0%	0.0%	0.0%	0.0%
std::vector<float, std::allocator<float>>::_M_de		0.0%	0.0%	0.0%	0.0%
clone		0.5%	0.0%	0.1%	0.0%

Figure 4.4: Top hotspots functions during runtime KRR/MNIST 8192.

Using the same reasoning for strong scaling, the table 4.8 presents the execution times for the inversion of KRR kernels using the MNIST dataset across different thread counts (1 to 28) and for two problem sizes (8192 and 16384). Upon analysis, a general reduction in execution time can be observed as the number of threads increases, but with diminishing returns beyond 8 threads.

For the problem size of 8192, the time decreases from 122.38 seconds with a single thread to 45.70 seconds with 8 threads, demonstrating reasonable scalability. However, as thread counts rise further, performance gains diminish, with execution time increasing slightly to 49.62 seconds at 16 threads, and 48.71 seconds at 28 threads. This behavior can be attributed to overhead and lock contention, as shown in the profiling results. Specifically, as thread count grows, the program suffers from increased lock contention and overhead time, which severely limits scalability. The effective CPU utilization, reported in the profiling table, indicates a rise in inefficiencies at higher thread counts, as synchronization and thread communication become significant bottlenecks.

For the larger problem size of 16384, the execution times follow a similar trend. Starting at 792.15 seconds for a single thread, the time reduces significantly to 243.71 seconds at 8 threads. However, beyond this point, the performance gain flattens. The time at 16 threads is 239.16 seconds, only marginally better than at 8 threads, and reaches 221.17 seconds with 28 threads. The profiling data suggests that this saturation in performance is likely due to a combination of increased lock contention, growing overhead, and imbalanced thread

execution. The lock contention time in the profiling table rises dramatically with more threads, leading to suboptimal performance scaling for both problem sizes.

Thus, the analysis reveals that while parallelization improves performance up to a certain threshold, scaling beyond 8 threads results in overhead and contention dominating the gains, leading to stagnation or even slight performance degradation at higher thread counts.

Figure 4.5 demonstrates the performance metrics, highlighting that despite improvements with additional threads, significant overhead and inefficiencies continue to constrain scalability. The combined insights from weak and strong scaling analyses underscore the need for optimizing parallel resource management and reducing locking overhead to enhance overall performance.

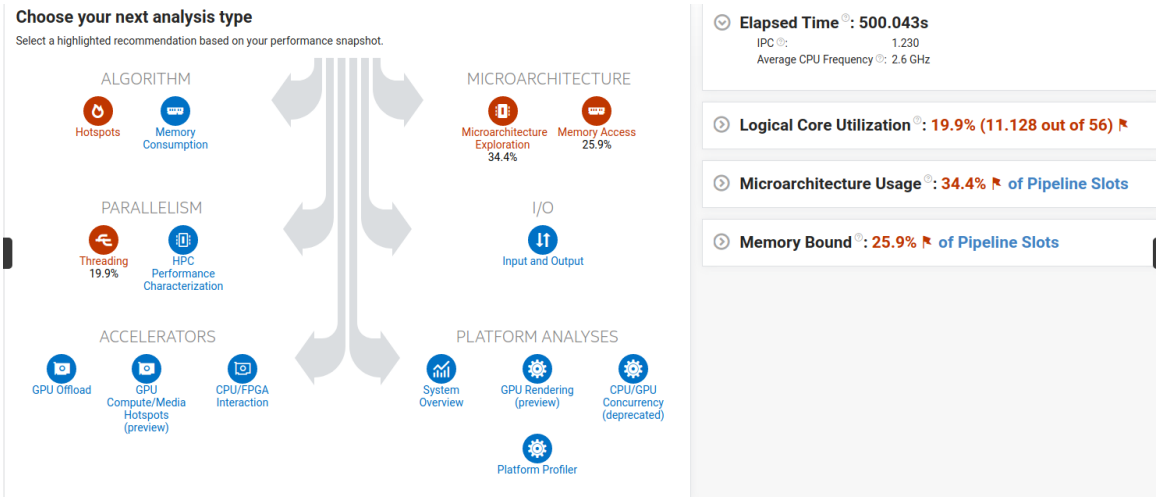

Figure 4.5: Performance metrics for 16384 datasize and 16 threads.

Python with *SWIG* Runtime Overhead

Table 4.5 presents runtime data for the Python implementation using *SWIG* to interface with C++ executables. The runtime for the Python interface is consistently higher compared to the C++ implementation shown in Table 4.11. The overhead increases with problem size, though it becomes relatively less significant at larger problem sizes, with only a 2.4% overhead at the largest problem size compared to the C++ implementation. This suggests that while the Python-SWIG interface introduces notable overhead for smaller problem sizes, the efficiency of both implementations declines as the problem size increases, reflecting inherent challenges in scaling with larger datasets.

Problem Size	2048	4096	8192	16384
Runtime (s)	1.8	7.4	48	233.4

Table 4.11: Runtime of C++ executables for different problem sizes (without Python interface).

5 Conclusion and Future work

5.1 Conclusion

This thesis offers an in-depth investigation into the development and evaluation of a scalable method for kernel matrix inversion, emphasizing hierarchical low-rank approximations with a focus on the Geometry Oblivious Fast Multipole Method. The importance for this research comes from the increasing demand for efficient and scalable solutions in machine learning and scientific computing, where managing large-scale matrix operations poses significant challenges.

We established at first glance a thorough examination of the theoretical principles underlying GOFMM, followed by its practical implementation and extensive testing. This study explores critical aspects such as the method's accuracy, performance under various scaling conditions, and its comparative efficiency relative to traditional approaches. Experimental evaluations were conducted on the CoolMUC-2 cluster, providing a robust and reliable environment for rigorous testing and validation.

By addressing key metrics—accuracy, tolerance, error measurement, and scaling behavior—this work aims to help with the field of scalable computational methods. The insights gained from this study on scalable kernel matrix inversion using hierarchical low-rank approximations include:

- The application of GOFMM for computing the inverse of SPD kernels demonstrates marked performance improvements over conventional matrix inversion techniques, particularly for large-scale problems where computational efficiency and scalability are crucial.

- Accuracy, tolerance, and error assessments reveal that the GOFMM approach consistently maintains high precision across a range of tolerance levels, ensuring reliable results in practical applications.

- Performance evaluations through both weak and strong scaling experiments show that GOFMM scales effectively with increasing problem sizes and computational resources, though with some nuances in performance.

- Strong scaling tests indicate that while GOFMM exhibits substantial performance gains, it does not achieve optimal performance due to overhead bottlenecks that affect scaling efficiency.

- Weak scaling experiments highlight that the method's efficiency decreases as problem size grows, primarily due to overhead and load imbalances that arise with increasing numbers of threads.

In summary, this thesis has thoroughly examined the GOFMM approach for scalable kernel matrix inversion, offering significant insights into its efficacy and robustness. The findings underscore GOFMM's potential to address the complexities of large-scale matrix operations in machine learning and scientific computing, setting the stage for future advancements and research in this vital area.

5.2 Future work

There are several promising directions for future research that could further enhance the impact and applicability of the proposed method.

One key area for future exploration is the extension of the method to multinode environments. The current research has primarily focused on single-node implementations, demonstrating the method's effectiveness within this constrained setting. To fully exploit the potential of high-performance computing resources, it is imperative to adapt and optimize the method for distributed computing frameworks. Incorporating Message Passing Interface (MPI) will facilitate the execution of the method across multiple nodes, allowing for the handling of larger-scale problems and enhancing computational efficiency. This transition to a multinode architecture introduces challenges such as inter-node communication, data distribution, and load balancing, which will require careful consideration and optimization to maintain the method's performance and accuracy.

Additionally, future work should focus on optimizing the parallelization strategy for multinode environments. This includes addressing communication overhead, synchronization issues, and ensuring that the scalability observed in single-node tests translates effectively to a distributed setting. By refining these aspects, the method can achieve improved performance and scalability, making it more suitable for large-scale applications.

Another avenue for future research is the application of the GOFMM-based method to a wider range of kernel types and real-world datasets. While this thesis has concentrated on Gaussian processes and kernel ridge regression with synthetic data, real-world scenarios often involve more complex and varied kernels. Testing the method with different kernel functions and diverse datasets could uncover additional insights and potential benefits, enhancing the method's robustness and applicability across different domains. Furthermore, integrating advanced techniques for managing large-scale kernel matrices, such as distributed matrix operations and enhanced approximation methods, could provide additional performance improvements and extend the method's utility.

Overall, the future work will build on the foundation established in this thesis, with a focus on advancing the scalability, applicability, and efficiency of the kernel matrix inversion method in both multinode environments and practical applications. By addressing

these areas, the research can contribute to the continued development and optimization of high-performance computing techniques for complex data analysis tasks.

Bibliography

[1] Thomas W. Wright Antonio J. González. Efficient partitioning for hierarchical matrix computations. *SIAM Journal on Scientific Computing*, 2018.

[2] Joshua Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 1986.

[3] M. Bebendorf. *Hierarchical Matrices*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[4] Christopher M. Bishop. Pattern recognition and machine learning. *Springer*, pages 295–303, 2006.

[5] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3642–3649, 2012.

[6] Jack Dongarra and Dennis Sullivan. *Parallel Computing for Data Science: A Practitioner's Guide*. CRC Press, 2014.

[7] Felix N Fritsch and R E Carlson. Nonlinear regression. *Encyclopedia of Environmetrics*, pages 1455–1460, 2012.

[8] Keerthi Gaddameedi, Severin Reiz, Tobias Neckel, and Hans-Joachim Bungartz. Efficient and scalable kernel matrix approximations using hierarchical decomposition. pages 3–16, 2024.

[9] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 4th edition, 2013.

[10] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 1987.

[11] W. Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*. Springer-Verlag Berlin Heidelberg, 2015.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2011.

[13] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2002.

[14] R. A. Horn and C. R. Johnson. *Matrix Analysis*. Cambridge University Press, 2012.

[15] Thomas Huckle. Sparse approximate inverses for preconditioning of linear equations. In *Conferentie van Numeriek Wiskundigen, Woudschoten*, page 2, Zeist, The Netherlands, 1996. Citeseer.

[16] Intel Corporation. *Intel® VTune™ Profiler User Guide*, 2023.

[17] Ian T. Jolliffe. *Principal Component Analysis*. Springer, 2nd edition, 2011.

[18] Eric Jones, Travis Oliphant, Pearu Peterson, et al. Scipy: Open source scientific tools for python. *http://www. scipy. org*, 73, 2001.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324. IEEE, 1998.

[20] Luca Bortolussi Michele Gorini and Daniele Montagnini. Interpolative decomposition for large-scale matrix approximation. *SIAM Journal on Matrix Analysis and Applications*, 2012.

[21] Vanessa Minden, Anil Damle, and Lexing Ying. A fast and scalable matrix inverse using hierarchical low-rank approximations. *SIAM Journal on Scientific Computing*, 39(5):S212–S234, 2017.

[22] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, MA, 2012.

[23] M. Benzi P. Martineau. Rank-revealing qr factorization: A new approach to interpolative decomposition. *Numerical Linear Algebra with Applications*, 2016.

[24] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2016.

[25] David Pizlo and Steve Garfinkel. Evaluating weak scaling performance on modern multi-core processors. In *Proceedings of the International Conference on High Performance Computing*, pages 112–121, 2019.

[26] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007.

[27] R. Priedhorsky and T. Randles. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the 2017 ACM International Conference on Supercomputing*. Association for Computing Machinery, 2017.

[28] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[29] James Reinders and Michael O'Boyle. Understanding weak scaling and its implications. *Parallel Computing*, 40(1):14–26, 2014.

[30] Severin Maximilian Reiz. *On the Algorithmic Impact of Scientific Computing*. PhD thesis, Technische Universität München, 2024.

[31] John A. Rice. *Mathematical Statistics and Data Analysis*. Cengage Learning, 2006.

[32] Kurt S. Riedel. A sherman-morrison-woodbury identity for rank augmenting matrices with application to centering. *SIAM Journal on Matrix Analysis and Applications*, 13(2):659–662, 1992.

[33] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[34] Leslie Greengard Sivaram Ambikasaran, Daniel Foreman-Mackey. Fast direct methods for gaussian processes. *arXiv*, 2015.

[35] J. M. Smith and S. T. Smith. *Data Generation and Simulation Techniques*. Wiley, 2011.

[36] James Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*. Springer, Berlin, 2013.

[37] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[38] Max Welling. Notes on kernel ridge regression, 2019.

[39] Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious fmm for compressing dense spd matrices. *Institute for Computational Engineering and Sciences*, 2017.

[40] Chenhan D. Yu, Severin Reiz, and George Biros. Distributed $o(n)$ linear solver for dense symmetric hierarchical semi-separable matrices. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 1–8. IEEE, 2019.

Appendix

Detailed descriptions

This section includes guidelines for installing the integrated software on either a local machine or the Linux cluster at LRZ. Additionally, it offers a Python script that demonstrates how to compute the inversion of kernel matrices. The main instructions for the installation of GOFMM have been taken from this document [8].

1 Installation and compilation

You can follow the steps of installation and compilation exactly as described in the reference [8] using the GitLab repository[1] where the README explains all the steps to set up the GOFMM for both local machines or the cluster.

2 Execution for inverse kernel matrix

For execution, you can add the folder `use_cases` from the repository [2] to your local machine's docker container or cluster's container. The `use_cases` folder contains all the Python files for multiple cases: MNIST, Synthetic datasets, KRR, and GP.

For the cluster, you can run the file `run_tests.sh` after `salloc`:

```
module load charliecloud/0.25
# Define problem sizes to test
problem_sizes=(512 1024 2048 4096 8192 16384)
# Set OpenMP environment variables
export OMP_NUM_THREADS=28  # Set the number of OpenMP threads
export OMP_STACKSIZE=512M  # Set the stack size per thread
# Loop over each problem size
for size in "${problem_sizes[@]}"; do
    export PROBLEM_SIZE=$size
    echo "Testing problem size: $PROBLEM_SIZE"
    # Run the Python script with the current problem size
    ch-run --set-env=./gofmm/ch/environment -w ./gofmm -- python3
        workspace/gofmm/use_cases/mnist_inv_gauss.py
    echo "Finished testing problem size: $PROBLEM_SIZE"
done
```

[1] GOFMM Datafold
[2] GOFMM Inverse

Listing 1: run_tests.sh

For profiling, you can add:

```
# Start the Python script inside Charliecloud
ch-run --set-env=./gofmm/ch/environment -w ./gofmm -- python3 /workspace
    /gofmm/use_cases/mnist_inv_gauss.py &

# Get the PID of the running process
PID=$!

# Run VTune outside the container and attach to the Python process
vtune -collect hotspots --result-dir=./hotspots_results -target-pid $PID
```

Listing 2: Profiling with VTune

Locally in Jupyter Lab, for example:

```
./compile_swig_mpigofmm.sh
python3 mnist_inv_gauss
```

Listing 3: Local Execution