JACOBS UNIVERSITY

GPU-parallel preconditioner for Approximated Krylov-based Kernel Ridge Regression training

by

Samundra Karki

Bachelor Thesis in Computer Science

Submission: May 17, 2021 Supervisor: Prof. Peter Zaspel

Jacobs University Bremen | Department of Computer Science and Electrical Engineering

Statutory Declaration

Family Name, Given/First Name	Karki, Samundra
Matriculation number	30002122
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

17-05-2021, [illegible]

Date, Signature

Abstract

Kernel ridge regression is a machine learning algorithm that uses the similarity between the data points to build a model for prediction of the quantitative outputs. The primary computational problem in kernel ridge regression is the solution of dense linear system. The direct solution of the dense linear system has the time complexity of O(N^3), where N is the size of the training sample. Because of this time complexity, this algorithm becomes impractical for large datasets. However, the use of iterative methods to approximate the exact solution of the linear system can reduce the time complexity of kernel ridge regression. The Krylov methods are the iterative methods that are used for approximation of the exact solution of the linear system. The time complexity of Krylov method is O(kN^2), where N is the size of training sample amd k is the number of iterations. However, the number of iterations will be very large for a poorly conditioned system. Therefore, preconditioning is required for faster convergence of Krylov method.

This work aims to build a preconditioner using *approximate Lagrange basis function*. Furthermore, this research provides the result of applying such a preconditioner on the Krylov methods. For efficiency, the Graphical Processing Unit (GPU) is used to construct the preconditioner. Furthermore, the GPU is also used to enhance performance of the matrix operations in the Krylov method.

Contents

1 Introduction

Machine learning is a branch of computer science that is primarily focused on the construction of algorithms that can learn from data to create flexible, adaptive, and meaningful models. These models are used to solve various problems in many applications. As an example, we can take a handwritten digit recognition application [1]. If the traditional algorithmic approach is used to recognize a digit drawn by a user, one would have to program every form of each digit that humans can write. However, if a large dataset of each digit is provided, then we can use a machine-learning algorithm to predict the digits drawn by the user [1]. Machine learning has several applications in various fields like health sectors [2], computer vision [3] and big data [4].

Within the field of machine learning, supervised learning [5] includes the algorithms that use the knowledge of input-output pairs of a given dataset, also known as training data, to build a model. Regression [5] is a type of supervised learning. Regression algorithms are used to predict quantitative outputs. Linear regression, basis expansion regression, kernel regression are some of the regression algorithms [5].

Kernel regression is a regression algorithm that uses a kernel function [5] to construct a model for prediction. The kernel function transforms the input to higher dimension to construct a model from the training data. However, due to overfiting [5], the prediction of a data point can have large error. Hence, the regularized [5] kernel matrix is used in kernel regression. This regularized kernel regression is known as kernel ridge regression [5]. The details about the kernel ridge regression is given in Section 2.1.

In kernel ridge regression, we need to solve a linear system. There exist two approaches to solve linear systems namely, the direct method and indirect method. Direct methods like Gaussian elimination [6], Cholskey decomposition [6] provide exact solutions of linear systems, whereas indirect methods like iterative methods [7] approximate the solution of the linear systems. The time complexity and space complexity of the direct method applied to the linear system of the kernel ridge regression is O(N^3) and O(N^2) respectively, where N is the size of the training sample [8]. The time complexity associated with direct methods makes kernel ridge regression impractical for large datasets. This thesis aims at minimizing the time taken to train the kernel ridge regression so that it can be used for large datasets.

To reduce the time complexity of solving the linear system, iterative methods can be used. The Krylov method [7] is an iterative method. In this thesis, we will use Krylov method to solve the linear system involved in the kernel ridge regression. Given an arbitrary initial guess of the solution, a Krylov method approximates the exact solution at every iteration by minimizing the residual. The residual is the actual solution subtracted from the approximated solution. The use of the Krylov method can reduce the time complexity of kernel ridge regression to O(kN^2), where k is the number of iterations and N is the size of training sample [8]. A more detailed explanation about the Krylov method is given in Section 2.2.

The condition number [7] of the kernel matrix in the linear system affects the number of iterations in the Krylov method. The kernel matrix can have columns that are numerically closer to linearly dependent. This causes the condition number of the kernel matrix to be very large. In this case, the number of iterations needed to solve the linear system will also be large. Therefore, preconditioning [7] should be used in the linear system involving the kernel matrix to reduce the condition number of the kernel matrix. A more detailed explanation about the condition number and preconditioning is given in Section 2.2.1.

If the preconditioner is a good approximation of the inverse of kernel matrix, then the number of iterations will be reduced significantly in the Krylov method. Therefore, this thesis aims at constructing a preconditioner in such a way that it approximates the inverse of the kernel matrix. To construct such a preconditioner, we will use the *approximate Lagrange basis function* [9]. In this method, we construct a local system for every point in the training sample such that the local system contains the point's r-nearest neighbors. Then, the following linear system is constructed from the local system

$$\hat{A}_i \hat{\alpha}^i = \hat{L}_i(\hat{X}^i), \tag{1}$$

where $\hat{X}^i$ is the local system for i^{th} point in training sample, $\hat{A}_i$ is the kernel matrix built from i^{th} local system, $\hat{\alpha}^i$ is vector of coefficients, $\hat{L}_i(\hat{X}^i)$ is the approximate Lagrange basis. The solution for the linear system, given in Eq.(1), will give an approximation of i^{th} column of the inverse of the kernel matrix. The solution of the linear system for every point in the training sample will be used in the construction of the preconditioner. In this thesis, this preconditioner is referred to as *local preconditioner*. In addition, the *tridaigonal preconditioner* is also constructed from the kernel matrix. In this thesis, these two preconditioners are used in the conjugate gradient method.

To construct the local system ($\hat{X}^i$), the r-nearest neighbors should be computed. The brute force approach to find r-nearest neighbors [5] is very inefficient for a large number of training samples. Other approaches like *KD-tree* [10] is suggested to reduce the time complexity of finding r-nearest neighbors. However, in the KD-tree approach, the search of the r-nearest neighbors cannot be parallelized efficiently. Therefore, in this thesis, *space-filling curves* [11] are used which overcomes these issues. Space-filling curve is a way of mapping multidimensional data into one-dimensional data [12]. The Hilbert space-filling curve [11], Peano space-filling curve [11], Z-order curve [11] are different types of space-filling curves. In this thesis, we will use the Z-order curve to find the r-nearest neighbors of a point for the construction of the local system.

The use of parallelism in the Krylov method can reduce the time taken to solve the linear system of kernel ridge regression significantly. For more efficiency in the Krylov method and the construction of the preconditioner, we will use Graphics Processing Units(GPUs). GPUs are many-core hardware where thousands of threads can run simultaneously. The focus of GPU is to distribute the workload on many cores for efficiency. CUDA [13] is a parallel programming model developed by Nvidia that can be used to write programs for GPUs. GPUs are used in many fields like medical image processing [14] and computer graphics [15].

This thesis is organized as follows. We discuss the background and related work in Section 2. In Section 3, we will discuss the mathematical foundations and algorithmic approach to build the local preconditioner. Furthermore, we will also discuss the construction of the *tridiagnoal preconditioner* in Section 3. In Section 4, we will discuss the details of the implementation of this thesis. Section 5 consists the results and the discussion of analysis of the results. The final section of this thesis is Section 6, which summarizes the thesis and provides a reference to the future work.

2 Background and Related work

In this section, we introduce the topics relevant to the thesis and related work. The reader is assumed to have the basic knowledge of regression algorithms like linear regression [5] and basis expansion regression [5]. The details on these topics can be found in [5].

2.1 Kernel Ridge Regression

For a given dataset, linear regression [5] assumes that the relationship between the input data and the output data is always linear. This assumption will not hold for every dataset. The basis expansion regression [5] based on the p-degree polynomial overcomes this assumption using a basis function to transform the D-dimensional input data. The growth of the transformed input data is exponential, i.e., O(D^p), where p is the degree of the polynomial. Furthermore, the models created using basis expansion regression based on the p-degree polynomial have large generalization errors. These reasons make it an unfavorable choice to provide a better model efficiently for a given dataset.

The kernel regression [5] overcomes this issue with the introduction of the kernel function. The use of the kernel function will cause the size of the transformed input to be equal to the size of the training data. Furthermore, regularization [5] can be used to prevent over-fitting. Therefore, kernel *ridge* regression [5] is one of the popular choices for regression. In this thesis, we are going to work only with kernel ridge regression.

Given the training points **X** = $\{x_1, x_2, x_3, ..., x_N\}$ $\in$R^d, Kernel matrix can be built by

$$\hat{A} = \begin{bmatrix} k(x_1,x_1) & \dots & k(x_1,x_N) \\ \dots & \dots & \dots \\ k(x_N,x_1) & \dots & k(x_N,x_N) \end{bmatrix}.$$

Here, $k(x_i, x_j)$ is a kernel function that provides information about the similarity between the training points x_i and x_j. The kernel matrix ($\hat{A}$) is a symmetric positive definite matrix. One of the widely used kernel functions is the Gaussian kernel function, which is given as

$$k(x_i, x_j) = \exp\left(-\frac{||x_i - x_j||_2^2}{\sigma^2}\right) \tag{2}$$

where, σ^2 is the variance. The regularized kernel matrix is given as $\hat{A}$ = $\hat{A}$ + λI, where λ $\in$R$_{\geq 0}$ is a shrinkage parameter [5]. Cross validation [5] can be used to estimate λ and σ^2.

Algorithm 1: Kernel ridge regression [5]

```
input  : $\{x_i, y_i\}_{i=1}^N$ training data, $x$ evaluation point, $\lambda$, $k$ positive definite kernel
output: Kernel-based ridge regression prediction $\hat{Y}(x)$
1 Build matrix $\hat{A}$.
2 Solve $(\hat{A} + \lambda I)\hat{\alpha} = y$ using preconditioned conjugate gradient method [7].
3 $\hat{Y}(x)$ = $\sum_{i=1}^N \hat{\alpha}_i\ k(x, x_i)$
4 return $\hat{Y}(x)$
```

The kernel ridge regression algorithm, given in Algorithm 1, is used to predict the output for an evaluation point. In Step 2 of Algorithm 1, for the construction of a model, the solution of the linear system should be computed. The time complexity of the kernel ridge regression algorithm is determined by this step. As described in Section 1, the direct method is a very expensive approach to compute the solution of this linear system. Therefore, we will use the Krylov method [7].

In step 3 of Algorithm 1, for prediction of the true output, we are multiplying the constructed model, i.e. $\hat{\alpha}$ with the vector that contains the similarity between the evaluation point and the points in the training sample, i.e. $\{k(x, x_i)\}_{i=1}^{N}$, where N is the number of training points. The prediction function is the linear combination of the constructed model and the kernel values of the evaluation point with the training sample.

2.2 Krylov Method

Kylov methods are iterative methods for approximating the solution of a linear system

$$Ax = b, \tag{3}$$

where A is the coefficient matrix, b is the right hand side vector and x is the vector of unknowns. The residual is given by $r_k = b - Ax_k$, where k is the index of iteration. The Krylov matrix(**K**) and Krylov subspace($\mathcal{K}$) at k^{th} iteration are given as

$$\mathbf{K}_k = \begin{bmatrix} r_0 & Ar_0 & A^2 r_0 & \dots & A^{k-1} r_0 \end{bmatrix}$$

$$\mathcal{K}_k(A, r_0) = \operatorname{span}\{r_0, Ar_0, \dots, A^{k-1} r_0\},$$

where $r_0 = b - Ax_0$. Here, x_0 is an arbitrary initial guess of the solution of the linear system. Krylov suggested that the approximation x_k to x is a linear combination of $r_0, Ar_0, .., A^{k-1}r_0$. Conjugate gradient, Minimum Residual(MINRES) , Generalized Minimum Residual(GMRES), BiConjugate Gradients are some of the methods that can be used to approximate the solution of linear system in Krylov subspace [7]. In this thesis, we will use the conjugate gradient method to approximate the solution of the linear system of the kernel ridge regression.

2.2.1 Conjugate Gradient Method

The conjugate gradient method [7] approximates the exact solution of the linear system in the Krylov subspace. This method can only be applied to symmetric positive definite matrices. The kernel matrix satisfies the symmetric positive definite property. Therefore, this method can be used to reduce the time complexity of the linear system involved in kernel ridge regression.

The properties that need to hold in the case of conjugate gradient method are:

- The residual r_k at k^{th} iteration should be orthogonal to the Krylov subspace($\mathcal{K}_k$).
- The vectors d_k in Algorithm 2 should form an A-conjugate set [7].

Using these properties, an algorithm for the conjugate gradient method to approximate the exact solution of the linear system is derived [7].

The condition number [7] of a matrix A is defined as $\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}}$, where λ_{max} is the largest eigenvalue and λ_{min} is the smallest eigenvalue of matrix A. As discussed in Section 1, the condition number of a kernel matrix can be very large. If the condition number of the kernel matrix is very large, then the required number of iterations in the conjugate gradient method will be equal to the number of training data. This again leads to the time complexity of O(N^3), where N is the size of training sample. So, preconditioning [7] should be used to reduce the condition number of the kernel matrix in the linear system.

Preconditioning is a transformation applied to the linear system such that the solution can be approximated faster. To precondition a matrix, one can use the left

$$P^{-1}\hat{A}x = P^{-1}b \tag{4}$$

or right preconditioners

$$\hat{A}P^{-1}y = b, \tag{5}$$

where $y = Px$. Jacobi [7], Successive Over-Relaxation(SOR) [7], Gauss-Seidel [7] preconditioners are some of the examples of the conventional preconditioners. In this thesis, we are using two preconditioners namely, local preconditioner and tridiagonal preconditoner. The details on the construction of these preconditioners is given in Section 3.

The condition number of the preconditioned linear system's matrix should be less than unpreconditioned linear system's matrix, i.e., $\kappa(P^{-1}\hat{A}) < \kappa(\hat{A})$. The condition number of the identity matrix is 1. We know that the product of a matrix and its inverse is always an identity matrix. Therefore, if P^{-1} is very similar to A^{-1}, the condition number of the preconditioned linear systems will be close to 1. This leads to a small number of iterations in the preconditioned conjugate gradient method.

The preconditioned conjugate gradient algorithm, given in Algorithm 2, is used in this thesis for solving the linear system of kernel ridge regression. As discussed in Section 1, this algorithm has the time complexity of O(kN^2), where k is the number of iterations and N is the size of the matrix of the linear system. The time complexity of kernel ridge regression is determined by time taken to solve the linear system. The use of conjugate gradient method can reduce the time complexity of the kernel ridge regression to O(kN^2), where k is the number of iterations and N is the size of the training sample.

Algorithm 2: Preconditioned Conjugate Gradient Method [7]

```
  input  : initial guess of solution, i.e., $x_0$
  output: approximated solution after $k^{th}$ iteration, i.e., $x_k$
1 Compute $r_0 := b - Ax_0$, $z_0 = P^{-1}r_0$, and $d_0 := z_0$
2 for k = 0, 1....until convergence do
3     $\alpha_k := \langle r_k, z_k \rangle / \langle Ad_k, d_k \rangle$
4     $x_{k+1} := x_k + \alpha_k d_k$
5     $r_{k+1} := r_k - \alpha_k A d_k$
6     $z_{k+1} := P^{-1} r_{k+1}$
7     $\beta_k := \langle r_{k+1}, z_{k+1} \rangle / \langle r_k, z_k \rangle$
8     $d_{k+1} := z_{k+1} + \beta_k d_k$
9 end
```

If we apply a nonsymmetric preconditioning matrix to the kernel matrix, the resulting matrix might not satisfy the symmetric positive definite property anymore. If the matrix does not satisfy the symmetric positive definite property, we cannot apply the conjugate gradient method. Other Krylov methods like MINRES or GMRES should be used for a better performance on solving the linear system. The details on MINRES and GMRES can be found in [7].

2.3 Z-order curve / Morton Code

As discussed in Section 1, to construct the preconditioner, we need to compute the local system($\hat{X}^i$), where i is the index of the point in training sample. The local system($\hat{X}^i$) is constructed with the $r-$nearest neighbors of the i^{th} point in the training sample. In this section, we will introduce the Z-order curve to find $r-$nearest neighbors of a point in the training samples.

The brute force approach to find the $r-$nearest neighbors is to calculate the distance from a data point to all other points in the training sample and choose the $r-$nearest points from the calculated distances. The time complexity of the brute force approach to find $r-$nearest neighbors of the points in the training sample is O(N^2), where N is the number of points in training sample [10]. Due to this time complexity, the cost of this approach is very expensive for the construction of the local system($\hat{X}^i$). Therefore, an efficient way to find r-nearest neighbors is required in the construction of preconditioner.

To address the issue of time complexity, methods like KD-tree [16] has been suggested for an efficient search of r-nearest neighbors in training data. The time complexity of the KD-tree approach for low-dimensional data is O($n \log n$), where n is the size of training sample [10]. The KD-tree algorithm for r-nearest neighbors is massively sequential [16]. Therefore, the parallelization of the KD-tree approach is not beneficial for efficiency. Since the space-filling curve has linear time complexity and can be parallelized efficiently, this approach is used in this thesis to find r-nearest neighbors of a point in the training sample.

As discussed in Section 1, we will use the Z-order curve [11] to find the r-nearest neighbors of points in the training sample. A Z-order curve is one of the fastest space-filling curves for computing one-dimensional data from the d-dimensional data. The obtained one-dimensional data is usually referred to as Morton index code. The Morton index codes need to be sorted in ascending order. Finally, the points in the training sample must be sorted according to their location in the sorted sequence of the Morton index code. The resulting points are sorted according to their nearest neighbors. In the sorted training sample, the $\frac{r}{2}$ points left and right of a point are its r-nearest neighbors. An example is shown in Fig. 1. In this example, the 2 dimensional data is first converted to Morton index code. The Morton index code is then sorted. The training points are then sorted according to the Morton index code. In Fig. 1, the obtained Morton index code suggests that the nearest neighbors of point $(0, 1)$ are $(0, 0)$ and $(1, 0)$. If we move along the sorted sequence of Morton index code, then a shape of ”Z” is formed as shown in Fig. 1. Therefore, this space-filling curve is referred to as the Z-order curve.

For encoding a Morton index code, the d-dimensional data will be converted to the binary number system. Then, each bit of the binary represented features is expanded by $(d - 1)$. Then, successive bits of all binary represented features are interleaved to obtain

Morton index code. Interleaving the successive bits means taking the first bits of all features of $d-$dimensional data and then appending it to the second bits of all features and so on, as done in Fig 1. For e.g., let $p = (a, b)$ be a 2-dimensional data with features, $a = 1$, $b = 7$. Converting a and b to binary representation we get $a = 001$ and $b = 111$. Finally, we expand the bits and interleave the successive bits of all features to construct the Morton index code as shown in Fig. 1. To obtain the Morton index code, we took first bits of 1 and 7 ,i.e., 01 and the appended it to second bits of the data ,i.e., 01 and then appended the resulting bits to third bits of the data ,i.e., 11. The obtained Morton index code for p is $p_{morton} = 010111$.

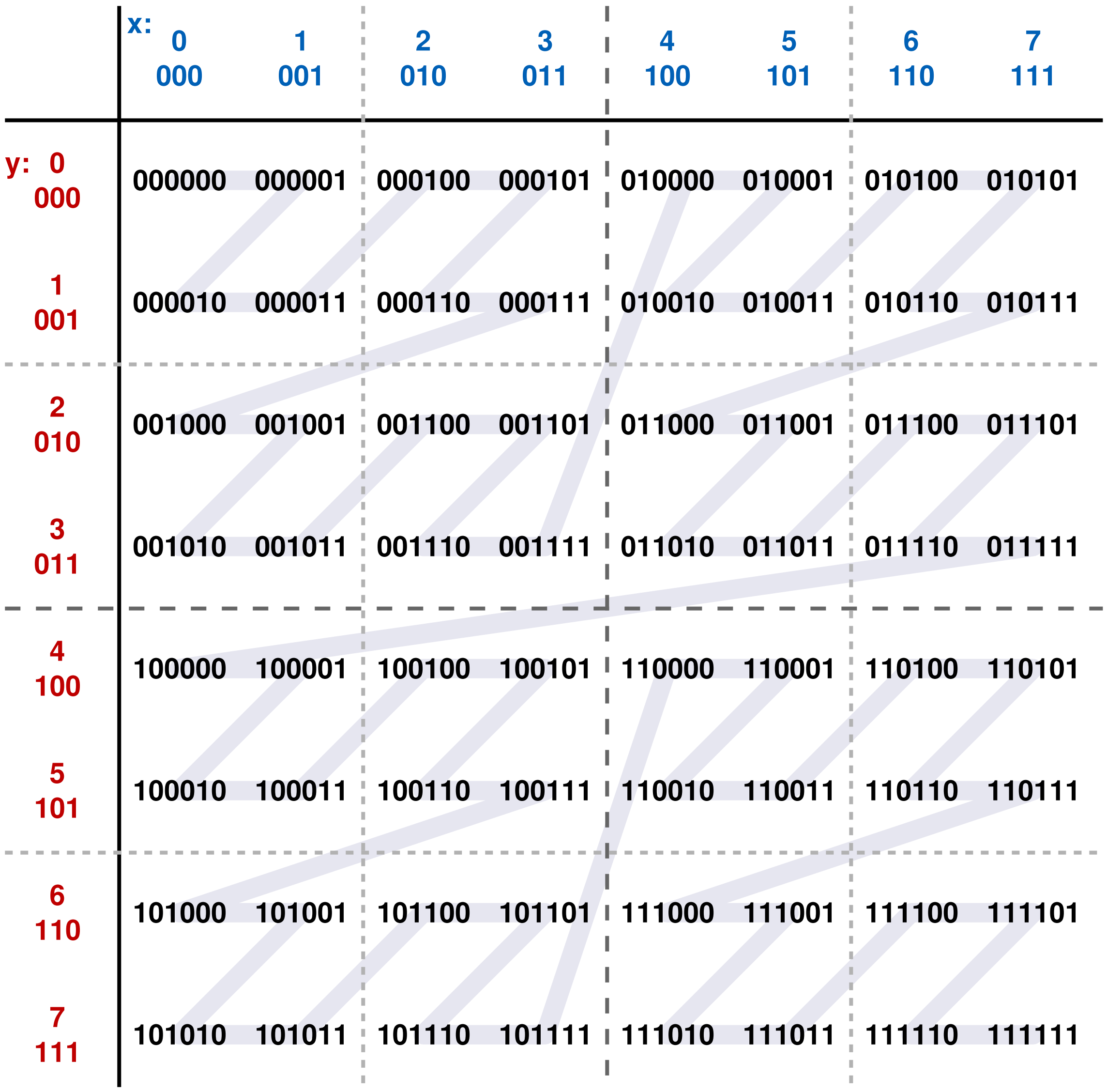

Figure 1: The Z-order curve formed by interleaving the binary representations of the coordinates (x,y) and sorting the resulting interleaved binary numbers. This figure is the work of David Eppstine [17].

2.4 Parallel Preconditioned Conjugate Gradient Method

If a large dataset is provided in the kernel ridge regression, the construction of a prediction model requires solving a large linear system. The time taken by the preconditioned conjugate gradient method to approximate the exact solution of this linear system will still be very large. To further reduce the time taken for kernel ridge regression, GPUs will be used for executing the preconditioned conjugate gradient algorithm and constructing the preconditioner. This section will provide an introduction to the parallelization of the preconditioned conjugate gradient method and the construction of the preconditioner.

In the preconditioned conjugate gradient algorithm, there are several scaled vector addition, dot product, and matrix-vector multiplication(MVP) operations. These matrix-vector operations can be parallelized in GPU to reduce the time taken to solve the linear system. The time complexity for scaled vector addition and dot product can be reduced to O(n/p) and O($n \log n/p$) respectively, where p is the number of threads and n is the size of the vector [18]. The time complexity of MVP involving kernel matrix can be reduced to O(n^2/p), where n is the number of rows or columns of the kernel matrix and p is the number of processors [8].

The construction of the preconditioner that will be used in the conjugate gradient method must be as fast as possible. To construct the local preconditioner, the r-nearest neighbors and the solution of the linear system given in Eq(1) must be computed for every data point. To find r-nearest neighbors, the Morton index code for every data point needs to be computed. The process of assigning the Morton index code to a data point does not depend on other data point's Morton index code assignment task. Therefore, they can be executed parallelly on GPUs. Furthermore,the linear system, given in Eq(1), should be constructed and solved for every data point. This operation can also be parallelized on GPUs. The details of parallelization of this task in given in Section 4.

While programming for GPUs, the programmer must have a very good understanding of the underlying hardware and compilers. The use of standard libraries leads to the better performance of a task. Therefore, the use of standard libraries has always been a favourable choice for the programmers. In this thesis, we will use cuBLAS, cuRand and thrust libraries [19]. The details of the libraries and their usecases are given in Section 4.

2.5 Related work

There has been a lot of work done to reduce the time complexity of kernel ridge regression. One of the fields of research is the low-rank approximation for faster kernel regression. In low-rank approximation approaches, the kernel matrices are decomposed using decomposition methods like eigendecomposition to reduce the time complexity of the kernel ridge regression. The most widely used low-rank approximation method for faster kernel regression is Nyström method [20]. Williams et al.[20] reduced the time complexity of kernel regression to O(m^2n), where $m(\ll n)$ is the size of the subset of training samples taken to do the approximation and n is the size of the training samples. However, there is a drawback associated with this approach. Low-rank approximation methods assume that most of the eigenvalues of the kernel matrix are small or close to zero, however, this is not always true [21]. The violation of this assumption will have a negative effect on the performance.

Another approach is to accelerate the MVP in the conjugate gradient method. Improved Fast Gauss Transform(IFGT) [21] and GPUML [22] are used to accelerate the MVP. Yang et al.[21] suggested IFGT to accelerate the kernel matrix and vector product. The accelerated MVP is used in the conjugate gradient method for solving the linear system of regularized kernel-based classification [21]. In terms of efficiency and accuracy, the result showed that kernel-based classification with IFGT was better than kernel-based classification with full kernel evaluation approach, Nyström approximation approach, and Proximal Support Vector Machines(PSVM) [23]. The performance of this approach is mediocre for a kernel matrix constructed with a Gaussian kernel function with low variance [24].

Shrinivasan et al.[22], in contrast to Yang et al., provided a parallel approach to solve MVP and the linear system involving kernel matrices. GPUML [8] is an open-source package that has been proposed to reduce the time complexity of the kernel regression. Shrinivasan et al.[22] used GPUML's MVP for kernel matrices to accelerate each iteration of the conjugate gradient method to solve the dense linear system of the Gaussian Process regression [25]. The time taken in Gaussian process regression using GPU, compared to CPU, was relatively less for larger datasets. Since the computational process is similar, the reduction of time complexity of this method signifies that the time complexity of the kernel ridge regression can also be reduced. In addition to parallel MVP, our goal is to provide a method that will use a preconditioner for accelerating the convergence rate of the conjugate gradient method.

In parallel to our work, Shrinivasan et al.[8] have used preconditioner in the Krylov method for faster computation of Gaussian Process regression. Shrinivasan et al.[8] have used a regularized kernel matrix as the right preconditioner in flexible GMRES. The regularized kernel matrix has a low condition number. As given in Eq.(5), the use of the right preconditioner will leave the right-hand side unchanged. Shrinivasan et al. [8] have proposed to use the conjugate gradient method to solve $y = Px$ in Eq.(5) and flexible GMRES to solve Eq.(5). Moreover, the GPUML was used for MVP in each iteration of the Krylov method. The results showed that this approach was faster than the direct method, the conjugate gradient method, and incomplete LU-based preconditioner [7] for flexible GMRES, for larger datasets. In this thesis, the approach for the construction of the preconditioner is different from the approach used in [8]. Our assumption is that the local preconditioner will be a good approximation of the inverse of the kernel matrix. This will lead to significant decrease in number of iterations to solve the dense linear system involved in kernel ridge regression.

3 Preconditioner for iterative training

The primary work of this thesis is to investigate the performance of the conjugate gradient method with the use of local preconditioner. The mathematical foundations and the construction of this preconditioner is discussed in Section 3.1. Furthermore, we will discuss the construction of the tridiagonal preconditioner in Section 3.2. The reason for constructing this preconditioner is to compare it's result with the result of the local preconditioner. In section 3.2, we only discuss the algorithmic approach to construct the tridiagonal preconditioner. The details on this preconditioner can be found in [7].

To understand the mathematical foundations explained in Section 3.1.1, the reader should have the knowledge of linear algebra concepts like function space and basis function. These topics are explained in [26].

3.1 Local Preconditioner using Approximate Lagrange Basis functions

3.1.1 Mathematical foundations

Interpolation [6] is a method that is used for estimating the output of a point that lies in the range of the given data points with known output. There are different methods of interpolation. Some examples of interpolation methods are piecewise constant interpolation, linear interpolation, Lagrange interpolation, spline interpolation [6]. The piecewise constant interpolation for a point simply assigns the output of the nearest neighbors to that point. For example, the viscosity of water for various temperatures is given as shown in Table 1. The viscosity of water for the temperature in the range $[0, 15]$ can be computed using the concept of the interpolation. The estimation of the viscosity of water with piecewise interpolation at $9°c$ will be $1.308\ Cp$. There are many machine learning algorithms that uses the concept of interpolation. The kernel ridge regression is one of such algorithm.

Temperature ($°c$)	0	5	10	15
Viscosity (Cp)	1.792	1.519	1.308	1.140

Table 1: The viscosity of the the water measured for various temperature [6]. The temperature is given in Celsius and viscosity in specific heat capacity.

Given n output value $\{y_i\}_{i=1}^n \in R$ and corresponding points $\{x_i\}_{i=1}^n \in R$, the kernel interpolation is defined as the method of finding the coefficients $\{\alpha_i\}_{i=1}^n \in R$ such that for a given set of kernel basis functions $\{k_i(x)\}_{i=1}^n \in R$, the linear combination

$$\hat{Y}(x) = \sum_{i=1}^{n} \alpha_i k_i(x) \quad \text{where,} \quad k_i(x) = k(x, x_j) \quad \forall j = 1, \dots, n \tag{6}$$

is interpolating [9]. In Eq.(6), $\hat{Y}(x)$ is the kernel interpolation function and $k(x, x_j)$ is the kernel function. In kernel interpolation, α should be computed in such a way that the following equation should hold true

$$\hat{Y}(x_j) = \sum_{i=1}^{n} \alpha_i k_i(x) = y_j \quad \forall j = 1, \dots, n. \tag{7}$$

As you can see from Eq.(6) and Eq.(7), this is exactly the same problem as kernel regression. Therefore, kernel regression is also referred to as kernel interpolation.

Given n output value $\{y_i\}_{i=1}^n \in R$ and corresponding points $\{x_i\}_{i=1}^n \in R$, the Lagrange interpolation [6] is defined as the method that uses the Lagrange basis function such that for a given set of Lagrange basis functions $\{L_i(x)\}_{i=1}^n \in R$, the linear combination

$$L(x) = \sum_{i=1}^{n} y_i L_i(x) \quad \text{where} \quad L_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{(x - x_j)}{x_i - x_j} \quad \forall i = 1, \dots, n, \tag{8}$$

is interpolating [6]. In Eq.(8) $L(x)$ is the Lagrange polynomial which is used for the prediction of a given point. Furthermore, the following equation must hold true

$$L(x_i) = y_i \quad \forall i = 1, \ldots, n. \tag{9}$$

In addition, the Lagrange basis functions $\{L_i(x)\}_{i=1}^n$ must fulfill the following the condition,

$$L_i(x_j) = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{else} \end{cases} \quad \forall j = 1, \ldots, n. \tag{10}$$

From the perspective of linear algebra, solving the Lagrange interpolation problem is equivalent to inverting the matrix formed by the basis functions. Therefore, the inverse of the kernel matrix can be computed by solving the kernel interpolation problem in terms of the Lagrange interpolation. Eq.(6) in terms of Lagrange interpolation is given as

$$\hat{Y}(x) = \sum_{i=1}^{n} y_i L_i(x) \quad \text{where,} \quad L_i(x) = \sum_{j=1}^{n} \alpha_j^i k(x, x_j) \quad \forall i = 1, \ldots, n \tag{11}$$

such that the Lagrange basis lies in function space created by $span(\{k_i(x)\}_{i=1}^n)$ [9]. Following the definition of Lagrange basis function given in Eq.(11), Eq.(10) can be rewritten as

$$\sum_{j=1}^{n} \alpha_j^i k(x_i, x_j) = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{else} \end{cases} \quad \forall i = 1, \ldots, n. \tag{12}$$

The coefficients α_j^i is needed for the computation of Lagrange polynomial $\hat{Y}(x)$, given in Eq.(11). For the computation of coefficients, we need to solve the following linear system

$$\hat{A}\alpha^i = L_i(x) \quad \forall i = 1, \ldots, n, \tag{13}$$

where $\hat{A}$ is the matrix formed from the basis function of kernel interpolation i.e. kernel matrix, α^i is a vector of unknowns, and $L_i(x)$ is the Lagrange basis function. The solution of this linear system gives a column of the inverse of the kernel matrix. Solving the linear system for all of the points, we get the exact inverse of the kernel matrix.

The time complexity to compute the exact inverse of the kernel matrix using the Lagrange interpolation is O(N^3). This time complexity is same as the time complexity of the direct method to solve the linear system. Therefore, this method is not beneficial for computing the exact inverse of the kernel matrix. However, we can approximate the inverse of the kernel matrix efficiently using the concept of the *approximate Lagrange basis function* [9].

Let $X = \{x_i\}_{i=1}^N$ be the given points. Let $\hat{X}^i$ be the subset of X, and r be the size of the subset. The subset is constructed in such a way that the following property is satisfied

$$\hat{X}^i \subseteq X, \quad \text{such that} \quad x_i \in \hat{X}^i \quad \forall i = 1, \ldots, N.$$

The approximate Lagrange basis function is the Lagrange basis function constructed from $\hat{X}^i$. Therefore, the size of this basis is $r \times 1$. Furthermore, the approximate Lagrange basis function $\{\hat{L}_i\}_{i=1}^N$ should fulfill the following condition

$$\hat{L}_i(x_j) = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{else.} \end{cases} \quad \text{where,} \quad \hat{L}_i(x_j) = \sum_{j=1}^{r} \hat{\alpha}_j^i k(x, x_j) \quad \forall i = 1, \ldots, N. \tag{14}$$

The $\hat{\alpha}^i$ is the vector of unknowns and $k(x, x_j)$ is the kernel function. The Lagrange interpolation for the approximate Lagrange basis function is given as,

$$\hat{L}(x) = \sum_{i=1}^{N} y_i \hat{L}_i(x), \tag{15}$$

where $\hat{L}(x)$ is the Lagrange polynomial.

To approximate the inverse of the kernel matrix, the locality of the approximate Lagrange basis function must be taken into account [9]. For this purpose, the *local Lagrange basis* is now introduced. The local Lagrange basis function $\{L_i^{loc}\}_{i=1}^N$ is the approximated Lagrange basis function with the subset that satisfy the following condition,

$$X_i^{loc} = \{\mathrm{x} \in X | \text{ x belongs to r nearest neighbors of the } i^{th} \text{ point}\}.$$

The Lagrange interpolation for local Lagrange basis is given as

$$L^{loc}(x) = \sum_{i=1}^{N} y_i L_i^{loc}(x), \quad \text{where,} \quad L_i^{loc}(x_j) = \sum_{j=1}^{r} \alpha_{j,loc}^i k(x, x_j). \tag{16}$$

As discussed previously, to compute the Lagrange polynomial($L^{loc}(x)$), we need to find the coefficients α_{loc}^i. For the computation of coefficients, we now only need to solve the local linear system constructed from r points. The local linear system is given as

$$\hat{A}_i^{loc} \alpha_{loc}^i = L_i^{loc}(x) \quad \forall i = 1, \ldots, N, \tag{17}$$

where $\hat{A}_i^{loc}$ is the kernel matrix built from the subset of the data points (X_i^{loc}), α_{loc}^i is the vector of unknowns, and $L_i^{loc}(x)$ is the local Lagrange basis function. The solution of the linear system, given in Eq.(17), gives an approximation of columns of the inverse of the kernel matrix. The increase in size of subset(X_i^{loc}) should improve the approximation of inverse of the kernel matrix. This method has been used by Zaspel [9] for the approximation of the inverse of the kernel matrix. Furthermore, Eq.(17) will be equivalent to Eq.(13), if the size of the subset of data points ($|X_i^{loc}|$) equals the number of given data points ($|X|$). In this case, we should get an exact inverse of the kernel matrix. Further details on approximate Lagrange basis can be found in [9].

In this thesis, the term *"local system"* indicates the subset of data points (X_i^{loc}) as discussed in the definition of the local Lagrange basis. The term *"local linear system"* indicates the linear system given by Eq.(17). Furthermore, the term *"local kernel matrix"* indicates the kernel matrix $\hat{A}_i^{loc}$ built from the local system (X_i^{loc}).

3.1.2 Construction of the preconditioner

In the previous section, we discussed a method to get an approximation of the inverse of the kernel matrix using the approximate Lagrange basis function. In this section, we will discuss the construction of the preconditioner using the solution of the local linear system, given in Eq.(17). Furthermore, we will discuss the efficiency of construction of such a preconditioner.

As discussed in the previous section, for each point in the training sample, we need to construct a local system such that it contains the $r-$nearest neighbors of the given point. For this purpose, we will use Morton code. After construction of the local system, we need to construct local kernel matrix from the points of the local system. This local kernel matrix is used for the construction of the local linear system, given in Eq.(17). This solution is a vector of size $r \times 1$, where r is the size of the local system. The size of the matrix built from the solutions of linear system, given in Eq.(17), is $r \times N$. However, we need a preconditioner of size $N \times N$, where N is the number of training points. To construct such a preconditioner, we will map the solution of the local linear system to i^{th} column of the preconditioner. Furthermore, we will map each entry of the solution to the original locations of data points in the local system. Then, we fill rest of the values with zeros. For example, let us suppose that

$$|X| = 10, \quad |X^{loc}| = 3, \quad \text{and,} \quad X_0^{loc} = \{x_0^0, x_5^0, x_8^0\},$$

where $|X|$ is number of given training points, $|X^{loc}|$ is the size of the local system, X_0^{local} is the local system for point x_0, and x_0, x_5, x_8 are the $3-$nearest neighbors of point x_0. Then using the concept of approximate Lagrange basis function, we construct the following local linear system

$$\begin{bmatrix} k(x_0^0, x_0^0) & k(x_0^0, x_5^0) & k(x_0^0, x_8^0) \\ k(x_5^0, x_0^0) & k(x_5^0, x_5^0) & k(x_5^0, x_8^0) \\ k(x_8^0, x_0^0) & k(x_8^0, x_5^0) & k(x_8^0, x_8^0) \end{bmatrix} \times \begin{bmatrix} \alpha_0^0 \\ \alpha_1^0 \\ \alpha_2^0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

This linear system is equivalent to the linear system given in Eq.(17). As we are approximating the inverse of the regularized kernel matrix, the local kernel matrix is also regularized. The obtained solution for the local linear system for x_0 is then mapped to 0^{th} column of the preconditioner. Finally, we map each entry of α^0 to the original position of the points of the local system. In this example, the α_0^0, α_1^0, and α_2^0 are mapped to 0^{th}, 5^{th} and 8^{th} position respectively. Then, we fill rest of the entries with 0. An illustration of mapping the solution to the 0^{th} column of preconditioner is given below

$$\begin{bmatrix} \alpha_0^0 \\ \alpha_1^0 \\ \alpha_2^0 \end{bmatrix} \longrightarrow \begin{bmatrix} \alpha_0^0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \alpha_1^0 \\ 0 \\ 0 \\ \alpha_2^0 \\ 0 \\ 0 \end{bmatrix}.$$

Following this process, for every local system constructed from the training points, we will get a preconditioner that is an approximation of the kernel matrix. The time complexity to construct this preconditioner is O(Nr^3), where N is the number of training points and r is the size of the local system. With the use of GPU, the preconditioner can be constructed very efficiently.

The constructed preconditioner might not satisfy the symmetric positive definite property. As conjugate gradient method can only be applied to symmetric positive definite matrix, we must make the obtained preconditioner symmetric. This is done as follows,

$$P^{'} = \frac{1}{2}(P + P^T),$$

where P is the non-symmetric preconditioner. The application of this preconditioner(P^i) in the conjugate gradient method should reduce the condition number of the regularized kernel matrix. Therefore, the number of iterations of the conjugate gradient method to solve the linear system of kernel ridge regression should be reduced.

3.2 Tridiagonal Preconditioner

A tridiagonal matrix is a matrix that has non-zero values in the main diagonal and the first diagonal above and below the main diagonal. An example of a tridiagonal matrix is given as

$$\begin{bmatrix} 1 & 7 & 0 & 0 \\ 1 & 7 & 5 & 0 \\ 0 & 3 & 4 & 9 \\ 0 & 0 & 2 & 8 \end{bmatrix}.$$

To construct a preconditioner, we build a matrix from the tridiagonal element of the kernel matrix. Let us suppose, the kernel matrix is given as

$$\hat{A} = \begin{bmatrix} 1 & 0.7 & 0.2 & 0.1 \\ 0.7 & 1 & 0.5 & 0.4 \\ 0.2 & 0.5 & 1 & 0.9 \\ 0.1 & 0.4 & 0.9 & 1 \end{bmatrix}.$$

The tridiagonal matrix constructed from $\hat{A}$ will be

$$P = \begin{bmatrix} 1 & 0.7 & 0 & 0 \\ 0.7 & 1 & 0.5 & 0 \\ 0 & 0.5 & 1 & 0.9 \\ 0 & 0 & 0.9 & 1 \end{bmatrix}.$$

The inverse of this tridiagonal matrix(P) is given as

$$P^{-1} = \begin{bmatrix} 0.391901 & 0.868713 & -2.286087 & 2.05747 \\ 0.868713 & -1.241018 & 3.265839 & -2.939255 \\ -2.286087 & 3.265839 & 3.265839 & 2.998040 \\ 2.05747 & -2.939255 & 2.998040 & -1.698236 \end{bmatrix}.$$

The matrix(P^{-1}) should give an approximation of the inverse of the kernel matrix. Therefore, this preconditioner is assumed to reduce the number of iterations in the conjugate gradient method.

4 Implementation

The algorithms discussed in the previous sections are implemented in the CUDA programming language. In this thesis, we are using *CUDA toolkit 11.1*. This toolkit provides various libraries for efficient performance of a task in the GPU. The implementation of this thesis was executed on NVIDIA® Quadro RTX 4000 GPU and Intel® Xeon® Silver 4210

CPU.

In the implementation, we have used cuRand [19] library, cuBlas [19] library, and thrust [19] library for the efficient performance of the tasks to be executed in the GPU. The cuRAND library provides functions to compute random numbers in a parallel manner. This library is used for generating training and testing points randomly. The cuBlas library is used for an efficient computation of basic linear algebra operations. In the implementation, this library is used for matrix-vector multiplication, matrix addition, and solving the linear system. The thrust is a parallel algorithm library which enhances the performance of the algorithms provided in C++ Standard Template Library(STL) [27] with the use of GPU. This library provides a parallel approach to standard algorithms like sorting and finding minimum and maximum values in a dataset. So, this library is used for computing the local system efficiently.

The input data of the training and testing samples are generated using `curandGenerateUn-iformDouble` function of cuRand library. This function will generate uniform samples with double precision in the $[0, 1]$ domain. The randomly generated dataset contains 3-dimensional input data and 1-dimensional output data. The output data is computed as

$$y = sin(X_1),$$

where y is the output data, X_1 is the first dimension of the input data. This dataset is provided to kernel ridge regression for the construction of the model and prediction of the output data.

As discussed in Section 2.4, the parallelization of matrix operations will reduce the time taken to perform this task significantly. As given in Algorithm 1 and Algorithm 2, there are several matrix operations in the kernel ridge regression and conjugate gradient method. The `cublasDgemm` function of cuBLAS library is used in the implementation for performing all matrix multiplications. Furthermore, we used `cublasDgeam` for all matrix addition in the implementation. The use of these functions enhances the performance of kernel ridge regression and conjugate gradient method significantly.

The implementation of generating the Morton code and finding the nearest neighbors is taken from Hierarchical matrices on GPUs library (hmglib) [28]. We need to allocate a specific number of bits to represent a feature of d-dimensional data. Every feature of the given points might not be represented by the number of bits allocated. Therefore, we need to scale the provided input data. The input data is scaled according to the minimum and maximum value of the input dataset. In the work by Zaspel [28], thrust library has been used to search the minimum and maximum value in the given input data. Furthermore, this library is also used for sorting the Morton code. To compute the minimum and maximum value in the input dataset, the `minmax_element` function of the thrust library is used. The `sort_by_key` function of the thrust library is used for sorting the Morton code.

The construction of local kernel matrices is implemented as a *kernel function*. The kernel function is the function that is executed in GPU for parallelization of a given task. Each thread of GPU executes the kernel function simultaneously. The threads that execute the kernel function are grouped together as a *block*. The maximum number of threads that is allowed in a block is 1024. The blocks are grouped in the grid. In the implementation, each local kernel matrix is stored as an 1-dimensional array. Then, we constructed a matrix with each row representing a local kernel matrix. In the implementation, the 2-dimensional block is used to group the threads. The thread with id (i, j) is computing the kernel value of the j^{th} entry of i^{th} local kernel matrix. Each thread is executed simultaneously. An illustration of the computation is given as follows

$$\begin{bmatrix} t_{00} \to k_{00} & t_{01} \to k_{01} & \dots & t_{0r^2} \to k_{rr} \\ t_{10} \to k_{10} & t_{11} \to k_{11} & \dots & t_{1r^2} \to k_{rr} \\ \vdots & \vdots & \vdots & \vdots \\ t_{N0} \to k_{N0} & t_{N0} \to k_{N1} & \dots & t_{Nr^2} \to k_{rr} \end{bmatrix}.$$

In the illustration, $t_{i,j}$ represents the thread with id (i, j) and k_{ij} represents kernel values of i^{th} local kernel matrix at j^{th} entry. The r^2 represents number of entries in local kernel matrix. The thread t_{ij} computes the kernel value k_{ij}. All of the thread t_{ij} is executed simultaneously.

The construction of tridiagonal matrix is also implemented as a kernel function. In this case, we used an one dimensional block to group the threads. Each thread assigns one element of the main diagonal of kernel matrix to it's respective position in the main diagonal of the tridiagonal matrix. Furthermore, this thread also assigns elements of the first diagonal above and below the main diagonal of the kernel matrix to the same index in the tridiagonal matrix. For example, the thread with id i will assign the kernel values at index $(i, i-1)$, (i, i), and $(i, i+1)$ to the same index in the tridiagonal matrix. Therefore, we are parallelizing over the elements of main diagonal and first diagonals in the implementation of this task.

In the implementation, we used `cublasDgetrfBatched` and `cublasDgetriBatched` function of the cuBLAS library to invert the matrix. The `cublasDgetrfBatched` function computes the LU factorization of the matrix. Following the LU factorization, `cublasDgetriBat -ched` inverts the matrix. These functions are used to invert the local kernel matrices and the tridiagonal matrix. Each thread of GPU inverts one local kernel matrix. In the implementation, the parallelization of the tridiagonal matrix is very inefficient. An efficient way to parallelize the inverse of this matrix should be used in the future.

5 Results and Benchmarks

In the previous section, we discussed the implementation aspect of this thesis. In this section, we will analyze the results obtained in the construction of the local preconditioner. Furthermore, we will compare the result of the unpreconditioned conjugate gradient method with the preconditioned conjugate gradient method. We will also analyze the performance of the local preconditioner in the conjugate gradient method.

In the implementation, we set $\sigma = 2$ and $\lambda = 1 \times 10^{-9}$, where σ is the kernel width and λ is the shrinkage parameter. Furthermore, the tolerance in conjugate gradient method is set as $tol = r_0 \times 10^{-8}$, where r_0 is the initial residual. To analyze the efficiency of all of the methods used in this thesis, the time taken by a task is measured for increasing number of randomly generated training points. Initially, we provide 1000 training points to kernel ridge regression algorithm. Then, we increase the number of training points by 2000 until 11000 training points are provided to the kernel ridge regression algorithm. The time taken to execute a specific task is measured using the `cudaEventCreate`, `cudaEventRecord`, and `cudaEventSynchronize` function of CUDA runtime application programming interface.

5.1 Construction of the preconditioner

As discussed in Section 3, there are several steps in the construction of the local preconditioner. In this section, we will analyze the time taken by each step involved in the construction of the preconditioner.

In the construction of the preconditioner, the first step is to sort the training points according to their nearest neighbors. To perform this task, we need to construct the Morton code, sort the Morton code, and reorder the training points according to the sorted Morton code. The time taken for sorting the training points for increasing size of the training sample is shown in Table 2. The maximum time taken to generate the Morton code, sort the Morton code, and reorder the training points is 0.022560 milliseconds, 0.363072 milliseconds, and 0.051008 milliseconds respectively. The reason for the efficient performance of sorting the training points for increasing size of training sample is the use of GPU. In the implementation, each thread of the GPU generates and sorts exactly one Morton code. In addition, each thread of the GPU reorders exactly one training point according to the sorted Morton code. Furthermore, each thread of the GPU is executed simultaneously. Therefore, the increase in time taken to sort the training points according to the sorted Morton code is insignificant for increasing size of the training sample.

Size	Generate	Sort	Reorder
1000	0.020448	0.094080	0.051008
3000	0.019610	0.334656	0.032512
5000	0.017568	0.349536	0.027424
7000	0.016384	0.345664	0.027296
9000	0.022560	0.363072	0.036064
11000	0.018528	0.359360	0.031296

Table 2: Time taken for generating the Morton Code, sorting the Morton code, and reordering data points. Generate denotes the time taken(in ms) for generating the Morton code, Sort denotes the time taken(in ms) for sorting the morton code, Reorder denotes the time taken(in ms) for reordering the training points according morton code.

After sorting the training points, we need to construct a local system from this training points. This step includes the construction of a mapping array containing the index of data points of the local system which is then used for constructing the local system. The results for each step of construction of the local system is shown in Table 3. The maximum time taken to construct the mapping array and the local system are 0.087328 milliseconds and 0.194080 milliseconds respectively. The reason for such an efficient performance is due to the parallelization of these tasks. In the implementation, we parallelize task of construction of mapping array over each map index computation. Furthermore, each thread of the GPU will map the training points to the local system simultaneously. Therefore, time taken to perform these tasks is very small.

As discussed in Section 3, we need to construct local kernel matrices for the construction

Size	r=5		r=55		r=105	
	Map	LS	Map	LS	Map	LS
1000	0.017120	0.016640	0.016384	0.016384	0.018656	0.022208
3000	0.014752	0.014080	0.022496	0.032736	0.033056	0.067328
5000	0.014400	0.012768	0.032864	0.059904	0.047936	0.095168
7000	0.014464	0.013760	0.039264	0.079584	0.066144	0.131072
9000	0.016384	0.014816	0.046656	0.096256	0.073984	0.163808
11000	0.016480	0.016416	0.055296	0.108512	0.087328	0.194080

Table 3: Time taken for construction of mapping array and local system. ”r” denotes the size of neighbors, Map denotes the time taken(in ms) for generating mapping array, LS denotes the time taken(in ms) for constructing the local system for the training points.

of the local linear system. The time taken to construct the local kernel matrices is shown in Table 4. For the neighborhood size of 5, the minimum and the maximum time taken to construct the local kernel matrices are 0.241408 milliseconds and 2.28032 milliseconds respectively. However, the increase in time taken to construct the local kernel matrices is very significant if we increase the neighborhood size. The minimum time taken for the construction of the local kernel matrices for the neighborhood size 55 and 105 are 18.456160 milliseconds and 66.340866 milliseconds respectively. The reason for such an increase in time is due to the increase in number of data points. For a neighborhood size and training sample size of 5 and 11000 respectively, the number of entry in the matrix containing all local kernel matrices is $5 \times 5 \times 11000$. As the neighborhood size increases to 55, the number of entry of this matrix will be $55 \times 55 \times 11000$. This is a very large number of data points, even for GPU. This is the reason for increase in time taken to construct the matrix containing all local kernel matrices for increasing neighborhood size. Therefore, to construct the preconditioner efficiently, the neighborhood size should be small.

Size	r=5	r=55	r=105
	LK Construct	LK Construct	LK Construct
1000	0.241408	18.456160	66.340866
3000	0.692832	55.321663	170.256958
5000	1.145248	90.980385	293.222260
7000	1.598016	127.195648	332.845123
9000	1.828832	165.654495	383.368500
11000	2.28032	171.165695	471.797913

Table 4: Time taken for computation of local kernel matrix. ”r” denotes the size of neighbors, LK Construct denotes the time taken(in ms) for construction of the local kernel matrices.

The solution of local linear system is required for the construction of the preconditioner. To solve the local linear system, we need to invert the local kernel matrices. The time taken to invert the local kernel matrices is shown in Table 5. For a neighborhood size of 5, the maximum and the minimum time taken for LU decomposition are 0.038880 milliseconds and 0.157224 milliseconds respectively. For a neighborhood size of 5, the maximum and the minimum time taken to invert the local kernel matrices following the LU decomposition are 0.029568 milliseconds and 0.139744 milliseconds respectively. This result suggests that inverting the local kernel matrix is an efficient task for small local system. However, the time taken to invert the local kernel matrices increases with the increase in the neighborhood size. The minimum time taken for LU decomposition and inverting the matrix following the LU decomposition for a neighborhood size of 55, are 3.369024 milliseconds and 5.17792 milliseconds respectively. Furthermore, the minimum time taken for LU decomposition and inverting the matrix following the LU decomposition for a neighborhood size of 105, are 22.889120 milliseconds and 27.692160 milliseconds respectively. The reason for increase in time taken is due to the increase in size of the local kernel matrix. The increase in the size of the local kernel matrix would add the number of operations in each thread to be executed in the GPU. This results in increase in time taken to invert the local kernel matrices. Therefore, the large neighborhood size results in increase in time taken to solve the linear system.

Size	r=5		r=55		r=105	
	LU fact.	Mat inv.	LU fact.	Mat inv.	LU fact.	Mat inv.
1000	0.038880	0.029568	3.369024	5.517792	22.889120	27.692160
3000	0.076800	0.057760	8.983712	16.367935	60.492512	46.505920
5000	0.082240	0.075136	13.707520	22.371679	101.672798	73.431778
7000	0.105248	0.105440	20.006912	36.786015	142.412125	102.711746
9000	0.159040	0.122496	23.370625	34.154144	183.561798	134.128922
11000	0.157224	0.139744	28.210400	37.459137	224.722458	164.374725

Table 5: Time taken for solving the local linear system. ”r” denotes the size of neighbors, LU fact. denotes the time taken(in ms) for LU decomposition, Mat inv. denotes the time taken(in ms) to compute the inverse of the local kernel matrices.

The final step, in the construction of the preconditioner, is to map the solution of the local linear system to the preconditioner. The time taken to map the solution of the local linear system with increasing number of training points and neighborhood sizes is shown in Table 6. The maximum and the minimum time taken to map the solution to the preconditioner is 0.15008 milliseconds and 0.592224 milliseconds, respectively. The results shown in Table 6 suggests that the time taken to map the solution does not change significantly with the increase in number of training points and neighborhood sizes. The reason for such a small time taken is due to the parallelization of the mapping task. Every thread maps one element of the solution to the preconditioner. Therefore, the time taken to map the solution to the preconditioner is very small.

Fig. 2 illustrates the time taken to invert and construct the local kernel matrices. This figure shows the increase in time taken with the increase in the number of training points for neighborhood size of 55. For this neighborhood size, Fig. 2 shows that the time taken is increasing rapidly with the increase in the number of data points. Due to this time taken in the construction and inversion of local kernel matrices, smaller neighborhood size must be considered for the construction of the preconditioner.

Size	r=5	r=55	r=105
	Map Pre.	Map Pre.	Map Pre.
1000	0.017760	0.030880	0.034976
3000	0.015008	0.089664	0.139296
5000	0.020896	0.149568	0.248416
7000	0.024256	0.236224	0.366528
9000	0.040704	0.261408	0.473216
11000	0.325408	0.325408	0.592224

Table 6: Time taken to map the solution of the local linear system to the pre-conditioner. ”r” denotes the size of neighbors, Map Pre. denotes the time taken(in ms) to map the solution of the local linear system to the preconditioner.

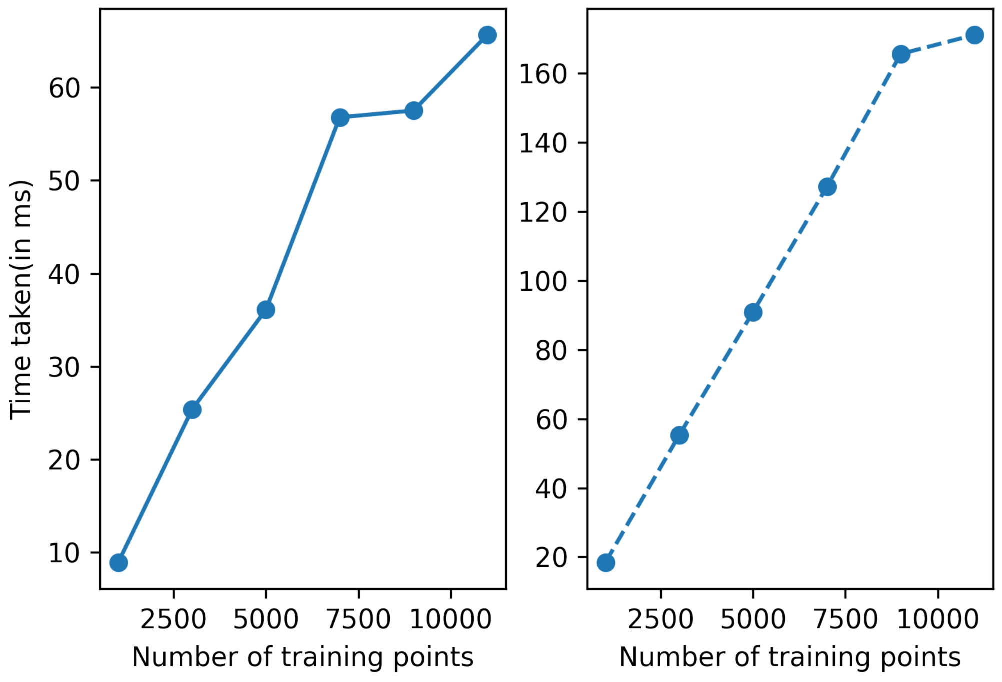

Figure 2: The plot represents the increase in time taken with the increase in number of training points for a neighborhood size ($r = 55$). The plot to the left represents the total time taken to invert the local kernel matrices, while the plot to the right represents the time taken to construct the local kernel matrices

5.2 Conjugate Gradient Method

The number of iterations and time taken to solve the unpreconditioned conjugate gradient method is shown in Table 7. Furthermore, the number of iterations and the time taken to solve the conjugate gradient method with the use of tridiagonal preconditioner and local preconditioner is shown in Table 8 and Table 9 respectively. The purpose for testing the conjugate gradient method for increasing number of training points is to investigate the performance of this method. As discussed in Section 2.2.1, the number of iterations is only influenced by the condition number of the regularized kernel matrix. The size has no impact in the number of iterations of the conjugate gradient method.

Table 7 shows that the number of iterations for unpreconditioned conjugate gradient method is significantly less than the number of training points. The maximum and minimum number of iterations of the conjugate gradient method without the use of the preconditioner is 187 and 129 respectively. The reason for this, is uniform distribution of the training points. As the training points are uniformly distributed, the condition number of the kernel matrix is small. Furthermore, the regularization of the kernel matrix will further reduce the condition number. Therefore, the number of iterations is small for the given training samples.

Size	No. of itr	Time taken
1000	187	26.02
3000	170	81.50
5000	164	181.11
7000	168	361.09
9000	129	440.19
11000	129	655.01

Table 7: The number of iterations and time taken by the conjugate gradient method without preconditioner. The No. of itr denotes the number of iterations and Time taken denotes the time taken (in ms).

The number of iterations and the time taken by the conjugate gradient method with the use of tridiagonal preconditioner is given in Table 8. The results show that the number of iterations is significantly larger than the unpreconditioned system. This suggests that the condition number of the regularized kernel matrix has increased with the application of the preconditioner. This means that the tridiagonal matrix is not approximating the regularized kernel matrix. The exact reason for this is unknown. Furthermore, due the implementation issue, we are only able to get the inverse of the regularized kernel matrix until the data size of 9000.

The number of iterations and the time taken by the conjugate gradient method with the use of local preconditioner is given in Table 9. The results show that the number of iterations of the conjugate gradient method increases with the application of this preconditioner. The minimum number of iterations for the training sample of size 1000, is 1357. This suggests that the time complexity to solve the linear system for this training sample is O(n^3), where n is the size of training sample. The exact reason of the failure of this preconditioner to decrease number of iterations of conjugate gradient method is unknown.

Size	No. of itr	Time taken
1000	1440	149.65
3000	1564	662.50
5000	2013	2193.58
7000	2179	4673.91
9000	3391	11800.83

Table 8: The number of iterations and time taken by the conjugate gradient method with the use of tridiagonal preconditioner. The No. of itr denotes the number of iterations and Time taken denotes the time taken (in ms).

Size	r=5		r=55		r=105	
	Itrs	TT(in ms)	Itrs	TT(in ms)	Itrs	TT(in ms)
1000	2370	298.92	2650	311.91	1357	185.56
3000	1917	829.70	3978	1695.71	2294	967.44
5000	1615	1771.45	2445	2653.37	1735	1883.58
7000	1860	3973.39	3132	6709.26	2814	6013.30
9000	4033	13698.42	3056	10383.45	1945	6601.97
11000	3246	16373.28	2347	11837.06	1877	9451.74

Table 9: Number of iterations and time taken for conjugate gradient method with the use of local preconditioner."r" denotes the size of neighbors, Itrs denotes the number of iterations required to approximate the exact solution, TT denotes the time taken(in ms) by conjugate gradient method.

Fig. 3 provides the number of iterations of conjugate gradient method for different sizes of neighborhoods. Our initial assumption was that the increase in size of neighborhood must decrease the number of iterations. However, as seen in Fig. 3, this does not seem to hold true. We assume the reason for this phenomenon is the approximation of the nearest neighbors calculated with Z-order curve. However, we are not certain if this assumption is true

The number of iterations of the conjugate gradient method with and without preconditioner is shown in Fig. 4. This figure clearly shows that the number of iterations in unpreconditioned conjugate gradient method is very small compared to the preconditioned conjugate gradient method. Therefore, the use of the preconditioner makes the conjugate gradient method very inefficient. The performance of the conjugate gradient method with the use of local preconditioner is very random. Furthermore, in Fig. 4, we can see that the number of iteration is increasing with the increase in number of training points. The reason for such a performance of local preconditioner and tridiagonal preconditioner is unknown. Further investigation of these preconditioner and their use in conjugate gradient method is required.

The time taken by conjugate gradient method is increasing with the increase in number of data points. The reason for increase in time is due to the matrix operations involved in the conjugate gradient method. The increase in number of training points will increase the size of the matrix and vectors involved in the conjugate gradient method. As the size of the matrix and vectors increases, the computation to perform the matrix operation will also increase. Therefore, each thread in the GPU has to do more computation.

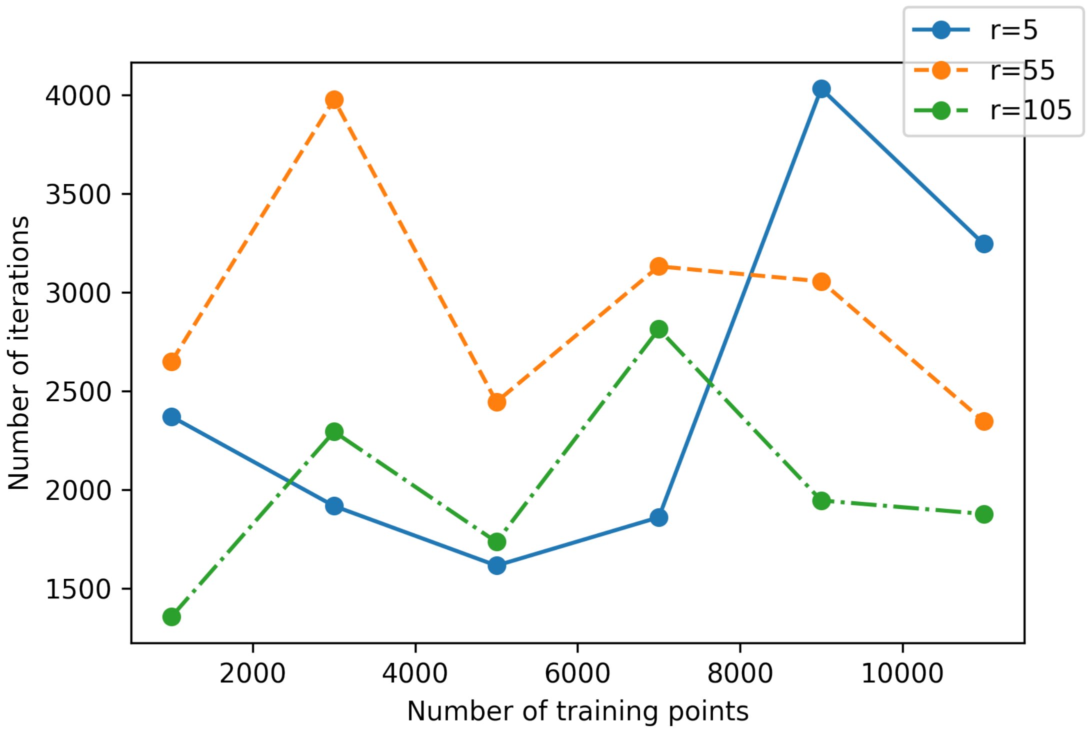

Figure 3: The number of iterations of conjugate gradient method with the use local preconditioner. The local preconditioner is constructed using different size of neighborhood.The x-axis represents the number of training points and the y-axis represents the number of iterations.

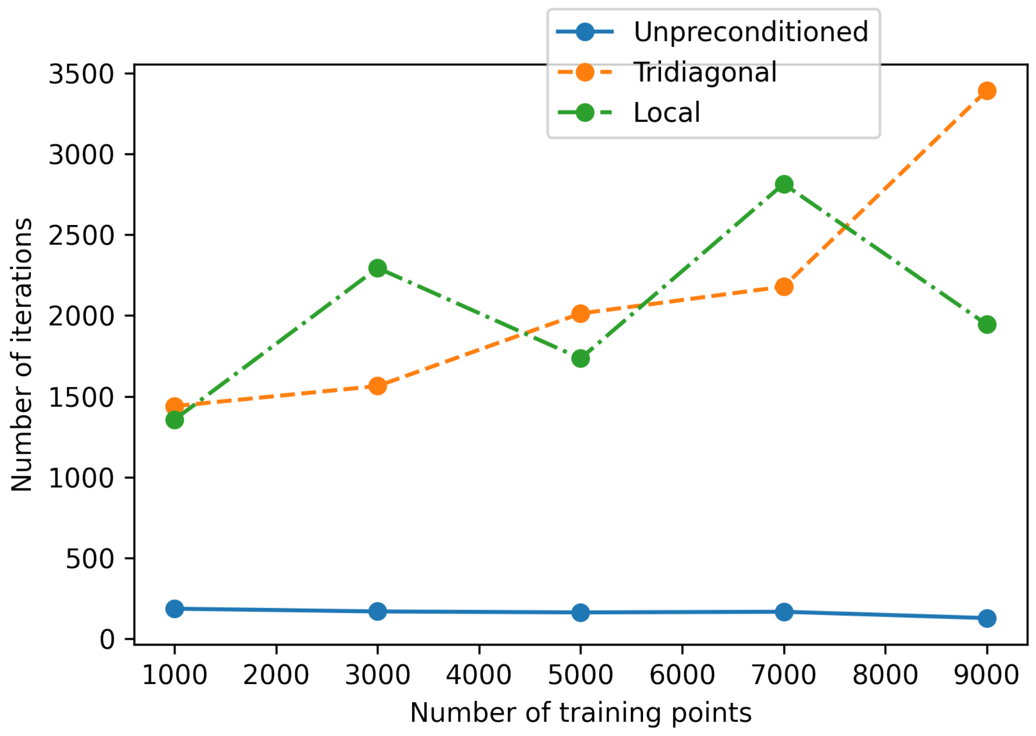

Figure 4: The number of iterations of the unpreconditioned conjugate gradient method and the conjugate gradient method with the use of the tridiagonal preconditioner. The x-axis represents the number of training points and the y-axis represents the number of iterations.

6 Conclusion and Future Work

This research aimed to reduce the time complexity of kernel ridge regression. For this purpose, we used the conjugate gradient method to approximate the exact solution of the linear system involved in the kernel ridge regression. The use of the conjugate gradient method reduced the time complexity of kernel ridge regression to O(kN^2), where k is the number of iterations in the conjugate gradient method and N is the number of training points. However, for an ill-conditioned system, the number of iterations could be equal to the number of training points. To reduce the condition number of the regularized kernel matrix, we proposed a preconditioner constructed with the concept of approximate Lagrange basis function. In the approximate Lagrange basis function, we used Z-order curve to approximate the nearest neighbors of a point. Furthermore, the GPU is used to enhance the performance of the algorithms used in this thesis.

In this thesis, we presented an effective approach to construct the local preconditioner. The experimental findings showed that the construction of the local preconditioner would be very efficient for a small neighborhood size. Furthermore, we presented the result of the application of this preconditioner and tridiagonal preconditioner in the conjugate gradient method. We also compared the result of the conjugate gradient method with and without the use of preconditioner. The findings suggested that the use of preconditioner proposed in this thesis would not decrease the number of iterations in the conjugate gradient. The number of iterations is rather increased with the use of preconditioner. We assume that the reason for the increase in number of iterations with the use of local preconditioner is due to the approximation of the nearest neighbors. However, the exact reason for this phenomenon unknown.

The future research work that we are purposing can be summerized as 1) Use the exact nearest neighbors search algorithm in the construction of local preconditioner 2) Use the obtained preconditioner in the Krylov method like GEMRES, which works for non-symmetric matrices.

References

[1] M. M. Abu Ghosh and A. Y. Maghari. “A Comparative Study on Handwriting Digit Recognition Using Neural Networks”. In: *2017 International Conference on Promising Electronic Technologies (ICPET)*. 2017, pp. 77–81.

[2] J. Akhil, S. Samreen, and R. Aluvalu. “The Future of Health care: Machine Learning”. In: *International Journal of Engineering and Technology(UAE)* 7 (Sept. 2018), pp. 23–25.

[3] F. Esposito and D. Malerba. “Machine Learning in Computer Vision”. In: *Intelligent Data Analysis* (Apr. 2001).

[4] L. Wang and C. Alexander. “Machine Learning in Big Data”. In: *International Journal of Mathematical, Engineering and Management Sciences* Vol. 1 (Sept. 2016), pp. 52–61.

[5] T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. 2nd ed. Springer, 2017.

[6] D. Kincaid and W. Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole Publishing Co., 1991. ISBN: 0534130143.

[7] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[8] B. V. Srinivasan et al. “Preconditioned Krylov solvers for kernel regression”. In: abs/1408.1237 (2014). arXiv: 1408.1237.

[9] P. Zaspel. “Parallel RBF Kernel-Based Stochastic Collocation for Large-Scale Random PDEs”. PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, July 2015.

[10] S. Sieranoja and P. Fränti. “Constructing a High-Dimensional kNN-Graph Using a Z-Order Curve”. In: *ACM J. Exp. Algorithmics* 23 (Oct. 2018). ISSN: 1084-6654.

[11] H. Sagan. *Space-filling curves*. 1st ed. Universitext Series. Springer-Verlag, 1994. ISBN: 9780387942650.

[12] J. Wang and J. Shan. “Space-filling curve based point clouds index”. In: *Proceedings of the 8th International Conference on GeoComputation* (2005), pp. 551–562.

[13] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN: 0131387685.

[14] Y. M. Kadah, K. Z. Abd-Elmoniem, and A. A. Farag. “Parallel Computation in Medical Imaging Applications”. In: *International Journal of Biomedical Imaging* (2011), pp. 1–2.

[15] C. Lauterbach et al. “Fast BVH Construction on GPUs”. In: *Computer Graphics Forum* 28.2 (2009), pp. 375–384.

[16] J. Friedman, J. Bentley, and R. Finkel. “An Algorithm for Finding Best Matches in Logarithmic Expected Time”. In: *ACM Trans. Math. Softw.* 3 (Sept. 1977), pp. 209–226.

[17] D. Eppstein. *User:David Eppstein/Gallery*. URL: https://commons.wikimedia.org/wiki/User:David_Eppstein/Gallery (visited on 05/15/2021).

[18] D. L. Michels. “Sparse-Matrix-CG-Solver in CUDA”. In: *In proceedings of Central European Seminar on Computer Graphics for Students (CESCG 2011)* (May 2011).

[19] D.B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123814723.

[20] C. Williams and M. Seeger. “Using the Nyström Method to Speed Up Kernel Machines”. In: *Advances in Neural Information Processing Systems*. Ed. by T. Leen, T. Dietterich, and V. Tresp. Vol. 13. MIT Press, 2001.

[21] C. Yang, R. Duraiswami, and L. S. Davis. “Efficient Kernel Machines Using the Improved Fast Gauss Transform”. In: *Advances in Neural Information Processing Systems*. Ed. by L. Saul, Y. Weiss, and L. Bottou. Vol. Vol. 17. MIT Press, 2005.

[22] B.V. Srinivasan, Q. Hu, and R. Duraiswami. “GPUML: Graphical processors for speeding up kernel machines”. In: *Workshop on High Performance Analytics Algorithms, Implementations, and Applications* (2010).

[23] G. Fung and O. L. Mangasarian. “Proximal Support Vector Machine Classifiers”. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’01. Association for Computing Machinery, 2001, pp. 77–86. ISBN: 158113391X.

[24] M. Griebel and D. Wissel. “Fast Approximation of the Discrete Gauss Transform in Higher Dimensions”. In: *Journal of Scientific Computing* 55 (2013), pp. 149–172.

[25] C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, Jan. 2006, p. 248.

[26] B. Cooperstein. *Advanced Linear Algebra*. Textbooks in Mathematics. CRC Press, 2016. ISBN: 9781439829691.

[27] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. 2nd. Addison-Wesley Professional, 2012. ISBN: 0321623215.

[28] P. Zaspel. “Algorithmic patterns for H-matrices on many-core processors”. In: *CoRR* abs/1708.09707 (2017). arXiv: 1708.09707.