



Optimization of Heterogeneous Parallel Computing Systems using Machine Learning

Devi Abhisheshu Adurti
Mohit Battu

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author(s):

Devi Abhiseshu Adurti

E-mail: dead20@student.bth.se

Mohit Battu

E-mail: mobt20@student.bth.se

University advisor:

Senior Lecturer, Suejb Memeti

Department of Computer Science and Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background: Heterogeneous parallel computing systems utilize the combination of different resources CPUs and GPUs to achieve high performance and, reduced latency and energy consumption. Programming applications that target various processing units requires employing different tools and programming models/languages. Furthermore, selecting the most optimal implementation, which may either target different processing units (i.e. CPU or GPU) or implement the various algorithms, is not trivial for a given context. In this thesis, we investigate the use of machine learning to address the selection problem of various implementation variants for an application running on a heterogeneous system.

Objectives: This study is focused on providing an approach for optimization of heterogeneous parallel computing systems at runtime by building the most efficient machine learning model to predict the optimal implementation variant of an application.

Methods: The six machine learning models KNN, XGBoost, DTC, Random Forest Classifier, LightGBM, and SVM are trained and tested using stratified k-fold on the dataset generated from the matrix multiplication application for square matrix input dimension ranging from 16x16 to 10992x10992.

Results: The results of each machine learning algorithm's finding are presented through accuracy, confusion matrix, classification report for parameters precision, recall, and F-1 score, and a comparison between the machine learning models in terms of accuracy, run-time training, and run-time prediction are provided to determine the best model.

Conclusions: The XGBoost, DTC, SVM algorithms achieved 100% accuracy. In comparison to the other machine learning models, the DTC is found to be the most suitable due to its low time required for training and prediction in predicting the optimal implementation variant of the heterogeneous system application. Hence the DTC is the best suitable algorithm for the optimization of heterogeneous parallel computing.

Keywords: Application, Heterogeneous systems, Parallel computing, Machine learning, Optimization

Acknowledgments

We would like to express our sincere thanks of gratitude to our supervisor at BTH, Suejb Memeti for his guidance and encouragement throughout the project. He helped us grow professionally and pushed us to sharpen our thinking. We would also like to thank our parents and friends for their moral support and encouragement in the completion of the thesis.

Authors:

Devi Abhiseshu Adurti

Mohit Battu

Contents

| | |
|--|-----------|
| Abstract | i |
| Acknowledgments | ii |
| 1 Introduction | 1 |
| 1.1 Aims and objectives | 2 |
| 1.2 Research questions | 2 |
| 1.3 Background | 3 |
| 1.3.1 CUDA: Platform for Heterogeneous computing | 3 |
| 1.3.2 Machine Learning | 3 |
| 1.3.3 Supervised learning | 4 |
| 1.3.4 Machine Learning algorithms | 4 |
| 1.3.5 Independent and dependent variables in ML | 5 |
| 1.3.6 Stratified K-fold cross validation | 5 |
| 1.3.7 Matrix Multiplication | 5 |
| 1.4 Scope of the thesis | 6 |
| 1.5 Overview | 7 |
| 2 Related Work | 8 |
| 3 Method | 10 |
| 3.1 Literature Review | 10 |
| 3.2 Experiment | 11 |
| 3.2.1 Working Environment | 11 |
| 3.2.2 Data-set Generation | 12 |
| 3.2.3 Data-set Visualization | 13 |
| 3.2.4 Label Count | 14 |
| 3.2.5 Correlation Matrix | 14 |
| 3.2.6 Outlier Detection | 15 |
| 3.2.7 Data-set Split | 16 |
| 3.2.8 Model Implementation | 17 |
| 3.2.9 Performance Metrics | 18 |
| 3.3 Construction of Results | 20 |
| 4 Results and Analysis | 21 |
| 4.1 Literature Review Results | 21 |
| 4.2 Experiment Results | 22 |
| 4.2.1 KNN | 22 |

| | | |
|----------|--|-----------|
| 4.2.2 | Decision Tree Classifier | 24 |
| 4.2.3 | XGBoost | 25 |
| 4.2.4 | Random Forest Classifier | 27 |
| 4.2.5 | LightGBM | 28 |
| 4.2.6 | Support Vector Machine | 30 |
| 4.3 | Evaluation Results | 32 |
| 4.4 | Summary of Analysis | 33 |
| 5 | Discussion | 34 |
| 5.1 | Answering Research Questions | 34 |
| 6 | Conclusions and Future Work | 37 |
| A | Supplemental Information | 42 |

List of Figures

| | | |
|------|---|----|
| 3.1 | Importing Python Libraries | 13 |
| 3.2 | Multilabel Count for Winning Column | 14 |
| 3.3 | Correlation Matrix for Data-set Columns | 15 |
| 3.4 | Outlier Detection for Data-set Columns | 16 |
| 3.5 | Flow Chart of the Machine Learning Modelling Process | 18 |
| 4.1 | KNN Accuracy Scores | 23 |
| 4.2 | KNN Confusion Matrix | 23 |
| 4.3 | DTC Accuracy Score | 24 |
| 4.4 | DTC Confusion Matrix | 25 |
| 4.5 | XGBoost Accuracy Score | 26 |
| 4.6 | XGBoost Confusion Matrix | 26 |
| 4.7 | Random Forest Classifier Accuracy Score | 27 |
| 4.8 | Random Forest Confusion Matrix | 28 |
| 4.9 | LightGBM Accuracy Score | 29 |
| 4.10 | LightGBM Confusion Matrix | 29 |
| 4.11 | SVM Accuracy Score | 30 |
| 4.12 | SVM Confusion Matrix | 31 |
| 4.13 | DTC Outcome | 32 |
| A.1 | Input Size v/s Execution Time(ms) graph for the input dimensions ranging from 16 to 160 | 42 |
| A.2 | Input Size v/s Execution Time(ms) graph for the input dimensions ranging from the 1008 to 1328 | 43 |
| A.3 | Input Size v/s Execution Time(ms) graph for the input dimensions ranging from the 4464 to 8560 | 44 |

List of Tables

| | | |
|-----|---|----|
| 3.1 | Snippet of the data set | 13 |
| 3.2 | Format of the data set | 13 |
| 3.3 | confusion matrix | 20 |
| 4.1 | Literature Review findings | 22 |
| 4.2 | KNN Classification Report | 24 |
| 4.3 | DTC Classification Report | 25 |
| 4.4 | XGBoost Classification Report | 27 |
| 4.5 | Random Forest Classification Report | 28 |
| 4.6 | LightGBM Classification Report | 30 |
| 4.7 | SVM Classification Report | 31 |
| 4.8 | Models Evaluation Report | 32 |
| 5.1 | Comparison of Models | 35 |

The interest in parallel computing has grown steadily over the last several decades. Parallel computation is the parallel usage of multiple computing units (cores or computers) to perform concurrent calculations. The primary intention of parallel computing is to increase computation speed. Parallel computing systems have traditionally been used for technical and scientific computing [1]. In the early days, computing systems only had central processing units (CPUs) that were intended to perform general programming tasks. Over the last decade, modern parallel computing systems have evolved rapidly because of the advent of GPU-CPU heterogeneous architectures, which contributed to a significant paradigm change in parallel computing [2].

A heterogeneous system is a combination of CPUs and GPUs that reduces communication latency, energy consumption, and increased performance by running multiple devices in parallel mode. It utilizes and combines the different types of resources available in our computer system. The transition from homogeneous to heterogeneous systems is a pivotal event in the evolution of high-performance computing. Homogeneous computation executes an application or program on one or more processors of the same architecture. Heterogeneous computation, on the other hand, executes an application using a variety of processor architectures, assigning tasks to architectures that are best suited for them, resulting in improved performance [1].

Programming applications running on heterogeneous parallel computing platforms target various processing units and require employing different tools and programming models/languages. These applications can have various implementations in performing their task. For example, there are several benchmark applications, such as sorting, back-propagation, matrix multiplication, depth-first search, and so on, all of which are designed to operate on heterogeneous parallel computing systems [3][4]. Most of the time, several implementations of the same application exist, like in the sorting application, where we have quick sort, merge sort, bubble sort, and selection sort, among others. In general, irrespective of the heterogeneous system, the efficiency of these implementations varies in terms of execution time according to the length of the input. Also, a particular implementation of the same application can perform differently on two heterogeneous systems as parallel computation differs from system to system based on the processors it is running on. It is uncertain which application implementation will perform optimally at the run-time based on heterogeneous system resources and also the provided input size. To achieve the high-performance efficiency of heterogeneous parallel computing systems, we must know the optimal implementation variant at run-time.

To get the most out of these resources while being efficient in their usage, a significant understanding of applications running on heterogeneous parallel computing systems is needed [5]. Knowing which implementation variant for an application is optimal at run-time for a heterogeneous parallel computing system would demand having system-specific architectural knowledge, as well as programming skills for many programming models targeting different systems [6] [7]. Machine Learning models can be implemented by training on the available information on performance parameters such as run-time, resource usage, energy consumption etc for selecting an implementation variant of an application that is to be executed on the heterogeneous parallel computing system.

In this thesis, we investigate the use of machine learning models to select the most optimal implementation of a particular application that yields the highest performance based on execution times of the implementations in a given context, that is hardware architecture and input size. For experimentation, we will focus on using the matrix multiplication application. There exist various implementations of different algorithms for matrix multiplication, including Naive, Strassen, and Tiled matrix multiplication algorithms [8][9][10]. Furthermore, there exist various implementation variants that target different target architectures, including CPU and GPU. The execution times of matrix multiplication application implementations for a heterogeneous parallel computing system vary with input size is shown in the figures A.1, A.2, A.3. The machine learning model will be trained off-line (that is before the run-time) on execution times of implementations, and when the program is executed the machine learning model will be used to predict which implementation variant would result in the highest performance based on the available computing processing units (CPU or GPU) and the input size.

1.1 Aims and objectives

The aim of the thesis to build machine learning models that can be used in predicting the optimal implementation variant based on run-time performance of the considered matrix multiplication application in the given context, that is the processing units and the input size.

The objectives can be defined as follows:

1. To investigate which machine learning algorithms are suitable to predict the optimal implementation variant for heterogeneous parallel computing.
2. Evaluate the selected machine learning algorithms concerning accuracy.
3. Apply the machine learning model to select the optimal implementation variant of the considered matrix multiplication application.

1.2 Research questions

This project would focus on the following research questions to achieve the goal:

1. Which machine learning algorithms are suitable to predict the optimal implementation variant for heterogeneous parallel computing systems?
2. How selected machine learning algorithms can improve the performance of heterogeneous parallel computing?

1.3 Background

In this section, we describe the theme and underlying concepts that were used as part of the research for this thesis. Firstly, information on a heterogeneous computing platform is discussed. Following that, a short overview of machine learning as a concept, its approach of supervised learning, and the machine learning algorithms used in this study to select the suitable implementation variant. Finally, the use of the considered matrix multiplication application is discussed, as well as its various implementations.

1.3.1 CUDA: Platform for Heterogeneous computing

CUDA is a parallel computing framework and programming interface model designed for general computing that makes use of NVIDIA GPUs parallel compute engine to solve certain complicated computational problems more effectively. Using CUDA, traditionally done computation on CPU can be done with GPU computation. The CUDA platform acts as a layer of software that offers direct access to the parallel computing components and GPU's virtual instructions set, for the compute kernel execution. The CUDA platform is intended to be used with standard programming languages like C, C++, Python, and Fortran. A CUDA application program consists of a two parts host code that runs on CPU and device code for GPU. The CUDA uses a compiler named NVCC, which stands for Nvidia Cuda Compiler. When running a program in CUDA, the NVCC compiler splits the device and host code parts of the program during the compilation time and sends the host code to the CPU and the device code to the GPU for execution[1].

1.3.2 Machine Learning

Machine learning(ML) is a branch of artificial intelligence that enables computing systems to learn how to solve a problem from their experiences by the use of data or information [11]. Machine learning is defined as "a computer program learns from experience E about some types of Tasks T and measure of performance P, if its performance in T tasks, as calculated by P, is improved with E" [12]. Machine learning techniques construct a model from a data sample, referred to as "training data," in making predictions or decisions without being directly instructed to do so. There are widely three machine learning approaches classified into supervised learning, unsupervised learning, and reinforcement learning. This study would make use of supervised learning as the machine learning approach.

1.3.3 Supervised learning

The most common approach to machine learning is supervised learning. Supervised learning algorithms construct a mathematical model from the data set that has both inputs and expected outputs[8]. The data set involves labelled training data that contains examples of the training set. During the training process, algorithms gain knowledge or identify patterns in the data set. The algorithm is provided with new data after completion of the training process and provides the result based on the learned patterns during the training. The primary focus of supervised learning is to correctly predict the classification by analyzing the data it contains, using its training experience[13].

1.3.4 Machine Learning algorithms

The following machine learning algorithms are used in this research.

Decision Tree Classifier (DTC)

Decision tree classifier is a supervised learning method, used when solving problems with classification [13]. The classifier is tree-structured, which classify by sorting instances as per the feature values. In Decision Tree branches represent decision rules, internal nodes represent the data set features as an instance that to be classified and the outcome is represented by each leaf node.

XGBoost

XGBoost stands for Extreme Gradient Boosting model. XGBoost is a fast and efficient implementation of gradient boosted decision trees. It is particularly for structured or tabular data sets. XGBoost is open-source and free to use. Its impact is recognized in various data mining and machine learning challenges [14]. XGBoost system uses scalability to its advantage, running faster on a single machine than the existing implementations and scaling to many examples in the memory limited or distributed settings.

Random Forest Classifier

Random forest classifier is a technique of supervised learning method. The forest is created by the collection of decision trees, trained by a method called bagging. The basic concept of bagging is combining learning models improves the final result. Multiple decision trees are merged in providing more reliable and accurate prediction [15].

K-Nearest Neighbors (KNN)

K-Nearest Neighbors is a simple supervised learning technique used for both classification and regression problems. The KNN algorithm assumes the similarities of the new data and the existing data, then places the new data into a class that is more similar to the existing classes. The KNN algorithm collects the available data and a new data point is classified based on the similarities [16].

Light GBM

Light GBM is a Light Gradient Boosting Machine that is an open-source and free to use gradient boosting framework based on decision tree algorithms used for classification, ranking, and other tasks of machine learning [17]. For this study, the Gradient Boosting Decision Tree (GBDT) is used along with Light GBM. GBDT is widely used for multi-class classification, which builds a prediction model from the collection of weak decision tree models.

Support Vector Machine (SVM)

Support Vector Machine is a supervised algorithm of machine learning, mostly used for classification problems. In the SVM, each data object is plotted as a point in n -dimensional space with n being the number of features and each feature value being a particular coordinate value [18]. Then, the classification is performed by locating the hyper-plane that best distinguishes the two classes.

1.3.5 Independent and dependent variables in ML

In machine learning, independent and dependent variables are crucial terms. The independent and dependent variables both have an influence on one another, which means that if one variable changes, the other will also change. The dependent variable determines the independent variable. The variable can control its selection and manipulation in an independent variable which influences the dependent variable [19]. The dependent variable is the variable that is measured and affected in the experiment. It is referred to as the dependent variable since it is dependent on the independent variable. A dependent variable in an experiment cannot exist in the absence of an independent variable [11].

1.3.6 Stratified K-fold cross validation

Cross-validation (CV) is a statistical analysis procedure used for evaluation of the efficacy of the machine learning method and for re-sampling used for validating an algorithm where data is insufficient. The stratification process is the rearrangement of the data such that each fold is a good representation of the whole [20]. Data divided into folds can be managed by criteria such as ensuring that each fold has the same results ratio with a specific categorical value as the class result value. This is known as stratified k-fold cross-validation.

1.3.7 Matrix Multiplication

Matrix multiplication is a binary operation in mathematics that outputs a matrix C which is the result of the multiplication of two input matrices A and B . For matrix multiplication to be possible the columns in the first matrix must be equal to the rows in the second matrix. The resulting matrix product contains the number of rows and columns from the first and second matrices. In the square matrix, the number of rows equals the number of columns. As an operation, matrix multiplication has several

algorithm implementations such as Tiled, Naive, and Strassen that can operate on heterogeneous systems are considered for this study [8].

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} \\ a_{21} & a_{22} & \dots & a_{2j} \\ \dots & \dots & \dots & \dots \\ a_{i1} & a_{i2} & \dots & a_{ij} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1k} \\ b_{21} & b_{22} & \dots & b_{2k} \\ \dots & \dots & \dots & \dots \\ b_{j1} & b_{j2} & \dots & b_{jk} \end{pmatrix}$$

$$C = AB = \begin{pmatrix} a_{11}b_{11} + \dots + a_{1j}b_{j1}, & a_{11}b_{12} + \dots + a_{1j}b_{j2}, & \dots, & a_{11}b_{1k} + \dots + a_{1j}b_{jk} \\ a_{21}b_{11} + \dots + a_{2j}b_{j1}, & a_{21}b_{12} + \dots + a_{2j}b_{j2}, & \dots, & a_{21}b_{1k} + \dots + a_{2j}b_{jk} \\ \dots & \dots & \dots & \dots \\ a_{i1}b_{11} + \dots + a_{ij}b_{j1}, & a_{i1}b_{12} + \dots + a_{ij}b_{j2}, & \dots, & a_{i1}b_{1k} + \dots + a_{ij}b_{jk} \end{pmatrix}$$

Naive Matrix multiplication

Naive multiplication is the ordinary matrix multiplication algorithm. Uses the matrix multiplication function in which the formula for the calculation of element of the matrix product is applied [21]. Following the normal approach of matrix multiplication contains two parts of code as host and device for the CPU and GPU.

Tiled Matrix multiplication

Tiled matrix multiplication is an algorithm that uses the parallel nature of matrix multiplication to perform on GPUs. The tiling method is used to reduce global memory accesses by taking advantage of the shared memory on the GPU, thereby boosting the execution of kernel efficiency [10]. Tiled matrix multiplication is only meant for the GPU, often the CPU code part is placed with naive matrix multiplication.

Strassen Matrix Multiplication

Strassen algorithm is faster matrix multiplication than the standard naive matrix multiplication, mostly useful for larger matrix sizes. For the Strassen algorithm to work, the matrices must be square matrices of dimensions n and n should be the power of 2. In case not, matrices should be padded 0 to match conditions. Strassen multiplication is a recursive technique for matrix multiplication where the matrix is divided into 4 sub-matrices of dimensions $n/2$ in each recursive step [22]. Strassen algorithm program can be implemented for both host and device i.e for CPU and GPU.

1.4 Scope of the thesis

The focus of this thesis is on using machine learning algorithms to predict the optimal implementation variant for an application on a heterogeneous parallel computing system based on resource utilization and input size. For experimentation, matrix multiplication application is taken and machine learning algorithms are trained on run-time performance data collected from a single heterogeneous parallel computing

system. The thesis is not concerned with telling the performance of the application and its implementation because it would differ against different heterogeneous parallel computing systems, and the machine learning models would only predict for the heterogeneous system from which data is collected and for systems with similar processing units. The thesis contributes by presenting a machine learning approach for application optimization that be implemented on heterogeneous parallel computing systems.

1.5 Overview

The background is structured containing explanations of the fundamental concepts in the above section. The other part of the paper is structured as follows. First, the related works contain the previous works relating to optimization of heterogeneous systems and used machine learning approaches for that. The methods section describes the steps involved in collecting data, creating machine learning models, and analyzing the models. The results section then presents the models' outcomes and the analysis of the model using performance metrics. After that, answers to the research-related questions in the discussion segment. Finally, conclusions were drawn based on the findings, and recommendations for future work were made to enhance the project.

Vi Nfoc-Nha Tran et al [23]. have done experiments that propose REOH, a holistic tuning method that uses a probabilistic network to determine the most energy-efficient configuration of a heterogeneous system for running a given application. The study showed that the REOH solution outperforms the brute-force approach in terms of optimal energy consumption on heterogeneous systems.

Zheqi Yu et al [24]. have worked on an optimization approach in heterogeneous computer systems to optimize energy power usage and performance. The paper proposes a power measurement utility for a reinforcement learning (PMU-RL) technique that can dynamically change the resource utilization of heterogeneous systems to reduce energy consumption. The study shows that the PMU-RL approach significantly reduced power consumption without impacting the application's efficiency.

Suejb Memeti et al [2]. has done a systematic literature review on using meta-heuristics and machine learning for the optimization of parallel computing systems. The research follows determining the optimal parameters set in the given context by using the heuristic search or machine learning for software optimization at compile-time and run-time. The research contributes to a deeper understanding of cutting-edge approaches that use machine learning and meta-heuristics to cope with the complexities of software optimization for parallel computing systems.

Joseph L. Greathouse and Gabriel H. Loh [25]. have researched the use of machine learning for the power and performance modelling of heterogeneous systems. The study, in particular, was focused on the measure of the power and performance of large test applications running on different hardware configurations, and machine learning models are trained against these measurements. The study contributes to the use of machine learning algorithms in the prediction of the application's power and performance.

Ben Taylor et al [26]. has done a study on optimization on embedded heterogeneous systems for OpenCL programs. The study makes use of machine learning models to predict suitable processors for the OpenCL program to run on, and also processor frequency for operating. The paper contributes to present an automatic procedure for mapping OpenCL programs on embedded heterogeneous systems, which provided significantly better performance.

From the above-related works, we found that only a few research papers discussed

the use of machine learning for the performance of heterogeneous systems, but none of the research focuses on optimizing the heterogeneous parallel computing systems by choosing the optimal implementation variant of the program or application using machine learning. In this thesis, we focus on the optimization of heterogeneous parallel computing systems by the use of machine learning to predict the optimal implementation variant of an application.

With the mentioned two research questions in mind, we aim to investigate literature and experimentation methods for the optimization of heterogeneous parallel computing systems using machine learning. For research question 1, the following section will describe the systematic literature review in which we carefully examined the literature through which we identified suitable machine learning algorithms for prediction. Based on the results of research question 1, we performed an experiment with research question 2 in which the best performing machine learning model is chosen for optimization of the heterogeneous parallel computing system by evaluating each machine learning classifier model on performance metrics such as accuracy, recall, f-measure, support, and confusion matrix.

3.1 Literature Review

To analyze and answer research question 1, a systematic literature review was conducted using the guidelines of Yang Liu [27], P.C. Chaitra [28], and Chase E. Golden [29]. This section mainly focuses on the understanding of various machine learning algorithms as well as identifying appropriate machine learning algorithms for prediction. Several steps were taken in our research, which is as follows:

1. **Identifying the key words:** We have found the following keywords are Supervised, Classification, Heterogeneous systems, Parallel computing, Machine learning, and Optimization.
2. **Formulating the search strings:** To form the search string primary keywords were chosen from the above-mentioned keywords.
3. **Locating the literature:** We had searched on various digital database platforms, including Google Scholar, IEEE, and Science Direct by using a search string.
4. **Following the inclusion and Exclusion criteria for selection:** Based on the collected literature such as articles and conference papers, the inclusion and exclusion criteria were formed. These criteria are used to narrow the scope of our research.

Inclusion Criteria

- Research papers relevant to the supervised machine learning algorithms for classification.

- Every article must be written in English.

Exclusion Criteria

- Articles that have not been completed.
 - Articles that are not written in English weren't considered.
5. **Evaluating and selecting the literature:** Resulting from the application of the inclusion and exclusion criteria, further modification is achieved through careful assessment and selection of the gathered literature.
 6. **Summarizing the literature:** All the observations from the collected literature are compiled and expressed for analysis.

3.2 Experiment

An experiment process is carried out using the results of the Systematic Literature Review of research question 1, in which we identify the suitable machine learning models for optimization of a heterogeneous parallel computing system. The experiment is then continued to build a prediction model with the chosen algorithm to determine the research question 2, where the optimal performing machine learning model is chosen for optimization of the heterogeneous system.

3.2.1 Working Environment

Every component used in this experiment is the most recent version and is labeled with the version number and description. The experimentation method and the preceding phases are carried out on a laptop with the following specifications.

- Microsoft Windows 10 64-bit Operating System.
- Intel Core i7-8750H with 6 Core(s), 12 Logical Processor(s) runs at 2.20GHz -2208 Mhz.
- 16.00 GB RAM that runs at 2667 MHz.
- NVIDIA GeForce GTX 1050 Ti graphic card with a dedicated GPU memory of 4GB and Driver Version of 470.14.
- Nvidia CUDA compiler V 11.2.152 - Using the NVIDIA Compiler SDK, developers can create or extend programming languages with GPU acceleration using an open-source compiler infrastructure.
- Python V 3.9.4 - Open source programming language.
- Jupyter Notebook Virtual Environment V 6.3.0 - Open source software.
- pandas V 1.2.4 - A powerful data analysis and manipulation library for python.
- seaborn V 0.11.1 - A data visualization library based on matplotlib.

- matplotlib V 3.4.1 - A comprehensive library for creating static, animated, and interactive visualizations in python.
- numpy V 1.19.5 - A python library used to perform mathematical computations in fields of matrices, linear algebra, and Fourier transform.
- scikit-learn V 0.24.2 - A library that provides many unsupervised and supervised learning algorithms.
- lightgbm V 3.2.1 - Open source gradient boosting library is used to train models on tabular data.
- xgboost V 1.4.1 - Open source software library is used for training the data.

3.2.2 Data-set Generation

To generate a data-set for a matrix multiplication benchmark application concerning CPU and GPU, we used the Nvidia Compiler SDK environment to run our various matrix multiplication implementation variants. Different matrix multiplication implementation variants were chosen namely Naive for CPU and GPU, Strassen for CPU and GPU, and Tiled for GPU. The above-mentioned matrix implementation variants were written in the form of CUDA code, which runs five times on the same input matrix dimensions, and the mean is calculated using the corresponding execution time of five results obtained from the same input matrix dimensions.

The calculated mean score is printed as the final execution time of that particular matrix input size. The matrix input dimensions range from 16 x 16 to 10,992 x 10,992 and are fed into the five different matrix multiplication implementations that generate the execution times. For each of the aforementioned implementations, we would create five data sets with the parameters Algorithm Name, Matrix Input Size, and Execution Time(milliseconds).

Each of the five data sets provided 798 items, with 266 rows and 3 columns in each, which were then merged into one, allowing us to store the execution time of each algorithm and created a new column indicating the winning label. In this case, we'd be mentioning another column with a header called winning label in a combined data set. The Winning column values are obtained by analyzing the performance of the algorithms on the same input size, with the winner being the algorithm that takes the least amount of time to execute.

A unique number is given for a winning algorithm which denotes that the algorithm is taking the least amount of time with that specified matrix input dimensions. These unique numbers frequently repeat in the winning column, transforming this data-set into a multi-classification data-set where each unique number represents the name of the corresponding algorithm thus making it suitable for applying the machine learning techniques to it [30].

The figure 3.1 shows how the data set looks, The data set has a shape of 1330 rows and 4 columns, with a total of 5,320 values are stored in the comma-separated value

| Algorithm Name | Input Dimensions | Execution Time(ms) | Winning |
|------------------------------------|------------------|--------------------|---------|
| Naive GPU | 16 | 0.691494 | 2 |
| Naive Matrix CPU | 16 | 0.065000 | 2 |
| Strassen Matrix Multiplication CPU | 16 | 0.114688 | 2 |
| Strassen Matrix Multiplication GPU | 16 | 0.469597 | 2 |
| Tiled Matrix Multiplication GPU | 16 | 0.536563 | 2 |

Table 3.1: Snippet of the data set

file(CSV). The Winning column is the target variable and the remaining columns like Algorithm Name, Matrix Input Size and Execution Time are the independent variables for the machine learning model. The format of the obtained data-set is represented below in the form of a table 3.2.

| Column Name | Type |
|--------------------|---------|
| Algorithm_Name | object |
| Input_Dimensions | Int64 |
| Execution_Time(ms) | float64 |
| Winning | int64 |

Table 3.2: Format of the data set

3.2.3 Data-set Visualization

Before we begin visualizing the data-set, there are a few preliminary steps that must be completed. First, we must import all of the necessary Python libraries into the Jupyter notebook as shown in figure 3.1.

Importing All The Necessary Python Libraries

```

In [2]: import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from IPython.core.pylabtools import figsize
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
import time
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import confusion_matrix
from sklearn.ensemble import AdaBoostClassifier
from sklearn import metrics
from sklearn import svm
import lightgbm as lgb
from xgboost import XGBClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier

```

Figure 3.1: Importing Python Libraries

The data-set is then loaded with the panda's library in the Jupyter notebook. The data-set consists of four columns, the independent variables being "Algorithm Name", "Input Dimensions", and "Execution Time". Finally, the "Winning" column represents a target variable. The target variable is filled with unique numbers, each representing the name of the corresponding algorithm [31].

3.2.4 Label Count

In the figure 3.2, the total label count of each matrix multiplication implementation variant used in the target variable is depicted as a bar graph. The label count, in this case, indicates how many times a specific matrix multiplication implementation variant is considered a winner in the target variable, i.e. winning column. The label count is a critical observation in determining which cross-validation strategy to use for a model to be trained and tested.

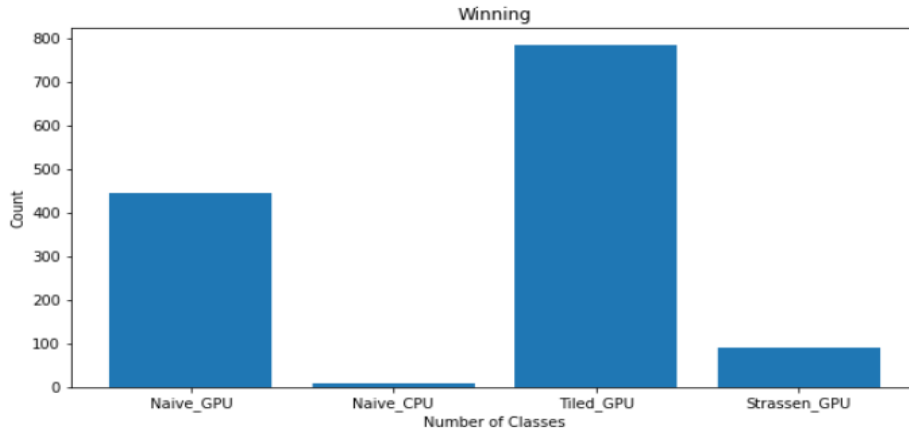


Figure 3.2: Multilabel Count for Winning Column

We can deduce from the above bar graph that the winning column is labeled with four different types of matrix multiplication implementation variants namely Naive Matrix Multiplication GPU, Naive Matrix Multiplication CPU, Tiled Matrix Multiplication GPU, and Strassen Matrix Multiplication GPU. In the figure 3.2, label 1 is named as Naive GPU and has a label count of 445, label 2 is named as Naive CPU and has a label count of 10, Label 3 is named as Tiled GPU and has a label count of 785, and Label 4 is named as Strassen GPU and has a label count of 90.

From the above figure 3.2, we can observe that the above-mentioned label names in a Winning column are not uniformly distributed as few of them have the label count of 445, 10, 785, and 90 respectively.

3.2.5 Correlation Matrix

A correlation matrix is a table that displays the coefficients of correlation between variables [12]. The correlation coefficients between the two variables are shown in

each cell of the table shown in the figure 3.3.

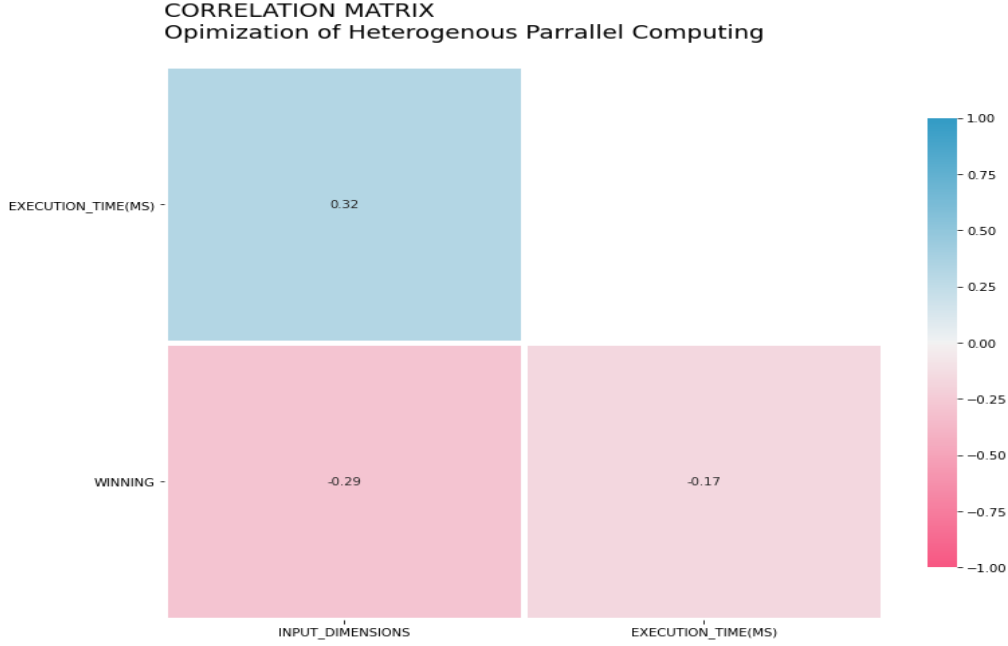


Figure 3.3: Correlation Matrix for Data-set Columns

From the above figure, we can observe that the variables like Input_Dimensions concerning Execution_Time indicate a positive correlation, where the values of both variables tend to increase together. For the other variable Input_Dimensions, Execution_Time concerning Winning variable shows a negative correlation where the values of one variable tend to increase when the values of the other variable decrease. The higher is the positive correlation between the two variables, the stronger is the relationship.

The main goal of the correlation matrix is to identify the patterns in a data set. In the above Figure 3.3, the observable pattern is that variables like Input_Dimensions and Execution_Time(ms) are highly correlated with each other.

3.2.6 Outlier Detection

A box plot can be used to check whether the data contains any outliers [32]. Boxplot provides the five-number summary of a particular set of data which contains the minimum value, lower quartile, median, upper quartile, and maximum value. A box plot is created by drawing a box from the lower quartile to the upper quartile. When there is an even number of data points, the median is calculated by taking the average of the two middle numbers. Median split the data-set into two equal parts namely the lower quartile which is the median of the lower half of the boxplot and the upper quartile which is the median of the upper half of the boxplot.

The lower whisker of the boxplot represents the minimum value in the data, while the upper-end whisker represents the maximum value in the data. These lower and upper whiskers are also known as minimum and maximum whiskers of a boxplot. Outliers are observations that are numerically dissimilar to the rest of the data. During the review of a boxplot, an outlier is defined as a data point that is located outside the minimum and maximum whiskers of the boxplot.

In the figure 3.4, each column in a data-set that is `Input_Dimensions`, `Execution_Time(ms)`, and `Winning` are depicted individually in the form of a boxplot. We can observe from the figure 3.4 that there are no outliers or data points present outside the maximum and minimum whiskers of the boxplot in any of the columns in a data-set.

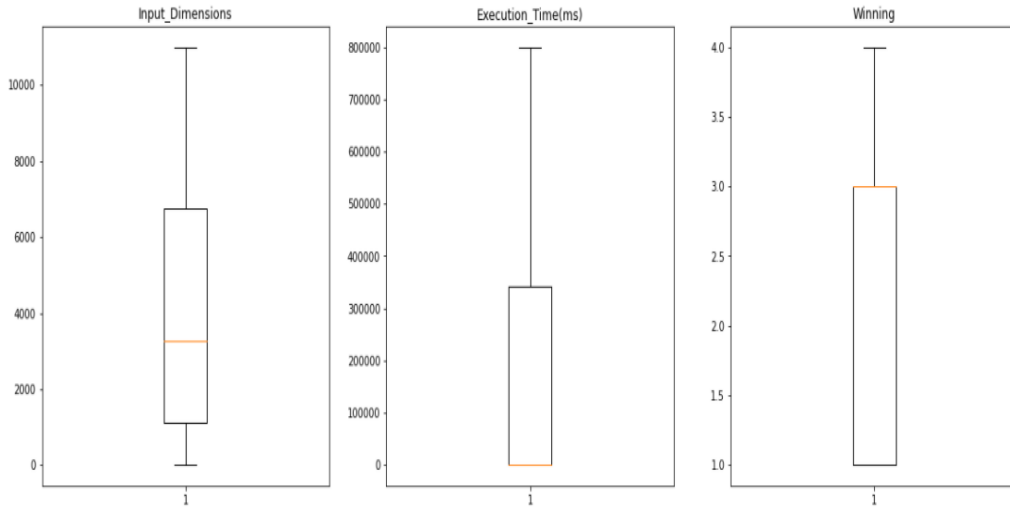


Figure 3.4: Outlier Detection for Data-set Columns

3.2.7 Data-set Split

From the above correlation matrix of Figure 3.3, we can observe that the independent variables such as `Input_Dimensions`, `Execution_Time(ms)` are not highly correlated concerning the target variable called `Winning` that adds noise in the data set which leads to a decrease in performance of machine learning model. As a result of the observation, we used the input dimensions and winning columns as an independent variable and as a dependent variable to feed the data into the model, and the remaining independent variables were eliminated from the data-set because they are irrelevant for a model to train. Separate python coding variables are introduced to store the data related to the input dimensions and winning columns so that both the independent and dependent variables are passed to the model during training, and only the independent variables are assigned to the model during the testing phase. Before we further divide the data-set into training and testing data-sets, a proper cross-validation strategy is needed so that the resulting estimate of model performance is not too optimistic or pessimistic.

We can see from the above visualizations of the data-set under the heading of label count that the winning label has four different class labels that are not evenly distributed among the classes. This observation indicates that the data-set has unequal class distributions. The Stratified K-fold strategy with classification tasks is the best cross-validation strategy for imbalanced class distributions. This strategy extends the regular K-fold strategy by ensuring that each fold of the data-set has the same proportion of observations with a given label. Whereas K-fold strategy is usually not preferred to measure the performance of a model for the imbalanced class labels [20][33]. The higher are the number of folds in the Stratified K-fold split the more duplicate data are generated which could lead to overfitting of the model and conversely if the number of folds is less then there is more chance of a machine learning model to get under fitted.

In this strategy, the number of splits was set to 5, the shuffle was set to true, and a random state was assigned to 42. The Stratified K-Fold cross-validation strategy involves randomly dividing the set of observations into K-folds, of approximately equal size. The first fold serves as a validation set for the model on which the model performance is assessed by training on the remaining K-1 folds. In this project, we calculated the training time, prediction time, training score, and prediction score of the machine learning model for each number of splits, and then used these variables to store in a unique list to produce an averaged score for the Stratified 5-fold splits. However, the traditional approach of selecting 80/20 Train-Test split ratio can also be considered here but Stratified K-fold gives more precision in assessing the skills of a machine learning model to predict on the unseen data set.

3.2.8 Model Implementation

The modelling process is described below in the form of a flow chart as shown in the figure 3.5.

KNN, DTC, XGBoost, Random forest classifier, Light GBM, and SVM are suitable models for the multilabel classification dataset [15][17][27][28][34]. The training data is used to fit these models. Appropriate hyperparameters are also provided to fine-tune the model while training the data, resulting in significantly improved performance.

Because we chose five splits in the Stratified K-fold strategy, the above-mentioned models are trained five times on the training dataset, and the time taken to train the machine learning model for each K-fold is recorded by appending the values in a list which is also called as model run-time training. Similarly, the models are predicted five times, with the prediction time values saved in a separate list, and the time taken to predict the values by the machine learning model for each K-fold are saved in a list which is called run-time prediction. The models are also used to compute the training and testing scores for each Stratified K-fold split iteration. These scores are derived from the training and testing datasets, which are then stored in different sorts of lists.

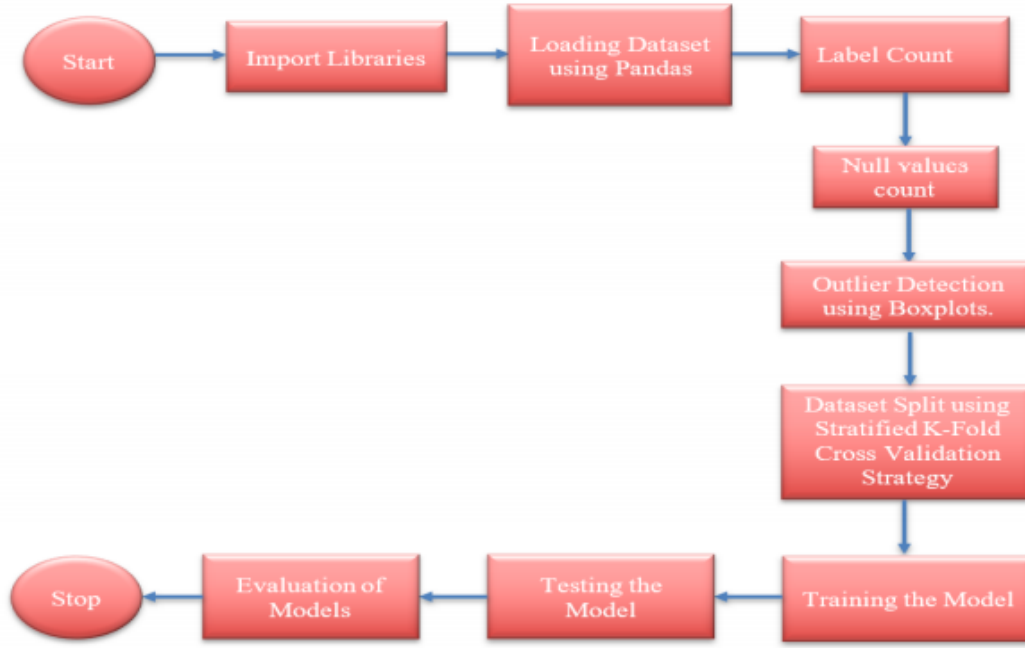


Figure 3.5: Flow Chart of the Machine Learning Modelling Process

The above-mentioned lists are saved to obtain the average score of the K-fold split. These calculated means concerning a selected machine learning model are saved in variables called training time, prediction time, testing accuracy score, and training accuracy scores.

3.2.9 Performance Metrics

There are a quite few performance metrics that are used for measuring the performance of a model. Before selecting the evaluation metric and assessing the model, it is necessary to understand how each metric measures. This thesis aimed to compare the performance of machine learning techniques by evaluating all performance metrics such as Accuracy score, F-1 score, Precision, Recall, Support, and Confusion Matrix [35][36].

Accuracy Score

Accuracy is formulated as a sum of true positives and true negatives divided by the total number of samples. This is true only if the model is balanced. If there is a class imbalance, it will produce inaccurate results.

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn} \quad (3.1)$$

Where in the above equation the terms "tp" is true positive, "tn" is true negative, "fp" is false positive, "tp" is true positive and "fn" is false negative.

Precision

Precision is defined as the ratio of true positive divided by the sum of a false positive and true positive. A classifier can avoid labeling a negative instance as positive. It provides the precision of positive predictions.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \quad (3.2)$$

Recall

The recall is calculated as the ratio of true positives to the sum of false negatives and true positives. The classifier model can find all positive instances.

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \quad (3.3)$$

F-1 Score

A weighted harmonic mean of precision and recall is used to calculate the F-1 score. The best F-1 score is denoted by 1.0 and the worst possible score is denoted by 0.0. It is used to compare the classifier models, not global accuracy.

$$Recall = 2 \times \frac{Recall \times Precision}{Recall + Precision} \quad (3.4)$$

Support

The number of actual occurrences of the class in the specified data set is referred to as support. Imbalanced support in the training data may indicate structural weaknesses in the classifier's reported scores, indicating the need for stratified sampling or re-balancing.

Confusion Matrix

A confusion matrix is a table that is used to measure the performance of a classification model on test data. It can be used to assess how well a classifier recognizes tuples of different classes. The confusion matrix contains data on actual and predicted classifications performed by a classification system. Table 3.3 represents the two-class classification problem with the two outcomes being "Positive" and "Negative".

| Actual | Predicted as Positive | Predicted as Negative |
|----------|-----------------------|-----------------------|
| Positive | True Positive | False Positive |
| Negative | False Negative | True Negative |

Table 3.3: confusion matrix

For a provided data point to predict, the classifier model will fall into any one of these two class labels. If we plot the predicted values against the actual values then we obtain a matrix with the following representative elements:

- **True Positive:** The data points whose actual outcomes were positive and the algorithm correctly identified it as positive.
- **True Negative:** The data points whose actual outcomes were negative and the algorithm correctly identified as negative.
- **False Positive:** The data points whose actual outcomes were negative but the algorithm has predicted incorrectly as positive.
- **False Negative:** The data points whose actual outcomes were positive but the algorithm has predicted incorrectly as negative.

3.3 Construction of Results

Firstly, the results of the literature review are represented in the form of a table. For experimentation, the results of the model's training and testing are presented in the form of a table. Accuracy, recall, precision, and the F-1 score are used to assess a model's performance metrics. The confusion matrix is used to assess the model's effectiveness. The parameters in the tables are named Model name, Testing Score (avg), Training (avg), Runtime Training (avg), and Runtime Prediction (avg). Because we chose five splits for the dataset in the Stratified K-fold strategy, the numerical values in the table represent the average values for each K-fold split of the training, testing, prediction time, and runtime. To view the model performance, the table is in CSV format.

4.1 Literature Review Results

In answering the first research question, a Systematic Literature Review (SLR) is conducted to knowing the machine learning algorithms that can be used to predict the optimal implementation variant. The findings from the SLR are listed in the form of a table.

| Title | Findings |
|---|---|
| Supervised machine learning algorithms: classification and comparison [13]. | The paper illustrates the various classification techniques for supervised machine learning, compares different algorithms, and evaluates the most effective classification algorithm based on a data set, features, and several instances. The algorithm SVM followed by Random forest classification is found to be most effective. |
| A Perspective of Supervised Learning Approaches in Data Classification [37]. | The strategies shall be examined according to the objectives, methodology, benefits, and drawbacks. Finally, provided a summary of the monitored ML approaches in the classification of results which includes SVM, Decision Tree classification. |
| Multi-class sentiment classification: The experimental comparisons of feature selection and machine learning algorithms [27]. | The paper presented a framework for the multi-class sentiment classification. The paper makes use of a multi-class classification dataset and selected algorithms Decision Tree, Naive Bayes, SVM, KNN, and Radial basis functions, out of which SVM was most efficient. |

| | |
|--|--|
| A review of multi-class classification algorithms [28]. | The paper focused on summarizing the significant classification methods and strategies for enhancing classification accuracy, the paper includes algorithms SVM, Random forest classifier, Decision tree classifier. |
| Analysis Accuracy of XGBoost Model for Multiclass Classification [34]. | The paper focuses on the analysis of the XGBoost algorithm for the applicant risk prediction for life insurance. The study found XGBoost to be better performing than Decision Tree, Random Forest particularly in the case of missing values in data. |
| LightGBM: A Highly Efficient Gradient Boosting Decision Tree [17]. | The paper proposes a modern Gradient Boosting Decision Tree (GBDT) algorithm called LightGBM for dealing with large numbers of data instances and features. The research shows XGBoost in terms of computational speed concerning low memory usage. |

Table 4.1: Literature Review findings

The research studies of machine learning algorithms for multi-class classification and supervised learning, as well as prominent performance algorithms, were listed in the Systematic Literature Review (SLR). Thus, from the Literature review, the 6 algorithms are chosen are SVM, KNN, Random Forest classifier, DTC, XGBoost, and Light GBM in predicting the optimal implementation variant for heterogeneous parallel computing systems.

4.2 Experiment Results

The algorithms KNN, XGBoost, DTC, Random forest classifier, LightGBM, and SVM are trained with a data set using a Stratified K-fold cross-validation method. This section then presents the performance results of model outcomes and the analysis of the model using performance metrics.

4.2.1 KNN

The KNN model was implemented on the data-set, and accuracy scores for the KNN model were generated, which achieved 99 percent accuracy in training and 98 percent accuracy in testing data-sets as shown in Figure 4.1.

```
In [104]: print('KNN Training set score: {:.4f}'.format(acc_tr_knn))
          print('KNN Test set score: {:.4f}'.format(acc_knn))

KNN Training set score: 0.9992
KNN Test set score: 0.9880
```

Figure 4.1: KNN Accuracy Scores

Using the confusion matrix technique, The KNN model's performance is evaluated, and the results are interpreted to determine the misclassification count and accurate prediction count of the various class labels. The model misclassified and predicted the Naive GPU label four times with the actual class label name Tiled CPU. The confusion matrix for KNN is shown in the Figure 4.2 below.

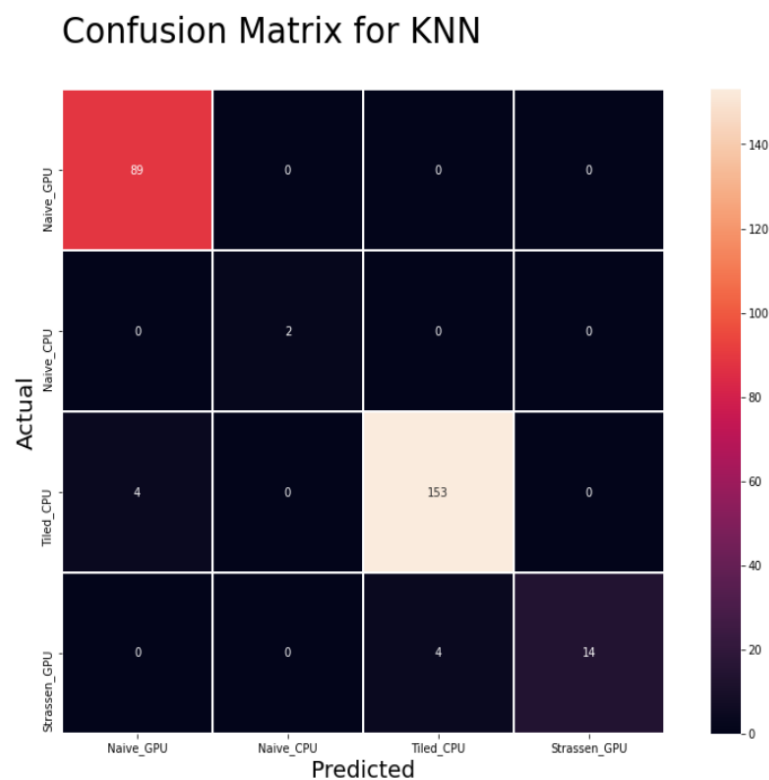


Figure 4.2: KNN Confusion Matrix

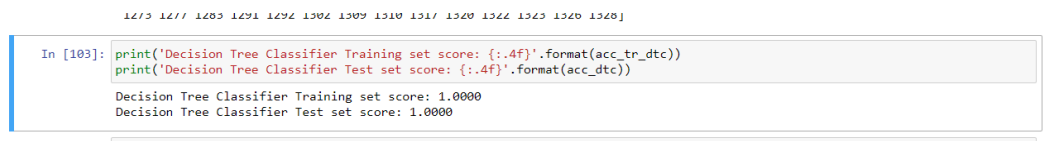
The classification report is analyzed to know the correctness of the KNN model thereby, evaluated precision, recall, f1 score, support for the class labels of this Naive Matrix GPU, Naive Matrix CPU, Tiled Matrix CPU, and Strassen Matrix GPU. The classification report for the KNN model is shown in Table 4.2 below.

| | precision | recall | f1-score | support |
|---------------------|-----------|--------|----------|---------|
| Naive_Matrix_GPU | 0.96 | 1.00 | 0.98 | 89 |
| Naive_Matrix_CPU | 1.00 | 1.00 | 1.00 | 2 |
| Tiled_Matrix_CPU | 0.97 | 0.97 | 0.97 | 157 |
| Strassen_Matrix_GPU | 1.00 | 0.78 | 0.88 | 18 |
| accuracy | | | 0.97 | 266 |
| macro_avg | 0.98 | 0.94 | 0.96 | 266 |
| weighted_avg | 0.97 | 0.97 | 0.97 | 266 |

Table 4.2: KNN Classification Report

4.2.2 Decision Tree Classifier

The DTC model was implemented on the data-set, and accuracy scores for the DTC model were generated, which achieved 100 percent accuracy in training and 100 percent accuracy in testing data-sets as shown in the Figure 4.3.



```

1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427
In [103]: print('Decision Tree Classifier Training set score: {:.4f}'.format(acc_tr_dtc))
          print('Decision Tree Classifier Test set score: {:.4f}'.format(acc_dtc))
          Decision Tree Classifier Training set score: 1.0000
          Decision Tree Classifier Test set score: 1.0000

```

Figure 4.3: DTC Accuracy Score

Using the confusion matrix technique, the DTC model's performance is evaluated and the results are interpreted to determine the misclassification count and accurate prediction count of the various class labels. The confusion matrix for DTC is shown in Figure 4.4 below.

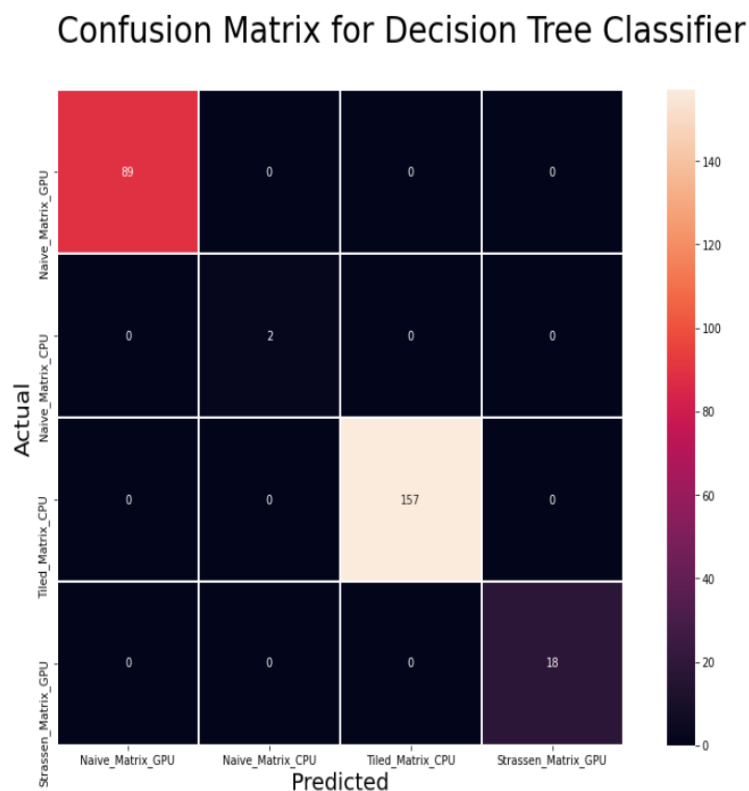


Figure 4.4: DTC Confusion Matrix

The classification report is analyzed to know the correctness of the DTC model thereby, evaluated precision, recall, f1 score, support for the class labels of Naive Matrix GPU, Naive Matrix CPU, Tiled Matrix CPU, and Strassen Matrix GPU. The classification report for the DTC model is shown in Table 4.3 below.

| | precision | recall | f1-score | support |
|---------------------|-----------|--------|----------|---------|
| Naive_Matrix_GPU | 1.00 | 1.00 | 1.00 | 89 |
| Naive_Matrix_CPU | 1.00 | 1.00 | 1.00 | 2 |
| Tiled_Matrix_CPU | 1.00 | 1.00 | 1.00 | 157 |
| Strassen_Matrix_GPU | 1.00 | 1.00 | 1.00 | 18 |
| accuracy | | | 1.00 | 266 |
| macro_avg | 1.00 | 1.00 | 1.00 | 266 |
| weighted_avg | 1.00 | 1.00 | 1.00 | 266 |

Table 4.3: DTC Classification Report

4.2.3 XGBoost

The XGBoost model was implemented on the dataset, and accuracy scores for the XGBoost model were generated, which has achieved 100 percent accuracy in both training and testing datasets as shown in Figure 4.5.

```
In [107]: print('XGBoost Training set score: {:.4f}'.format(acc_tr_xgb))
          print('XGBoost Test set score: {:.4f}'.format(acc_xgb))

XGBoost Training set score: 1.0000
XGBoost Test set score: 1.0000
```

Figure 4.5: XGBoost Accuracy Score

Using the confusion matrix technique, the performance of the XGBoost model is evaluated and the results are interpreted to determine the misclassification count and accurate prediction count of the various class labels. The confusion matrix for the XGBoost model is shown in Figure 4.6.

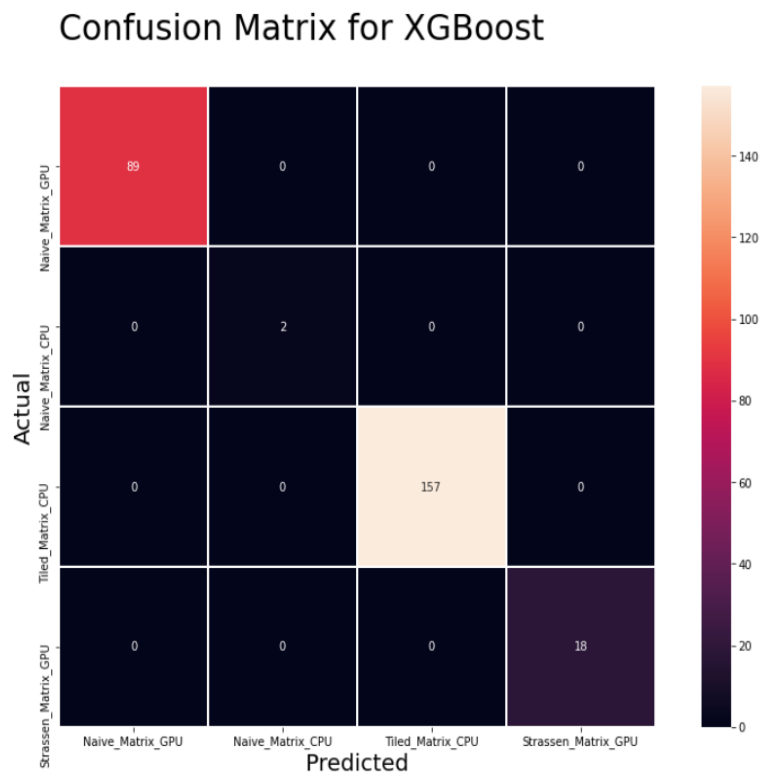


Figure 4.6: XGBoost Confusion Matrix

The classification report is analyzed to know the correctness of the XGBoost model thereby, evaluated precision, recall, f1 score, support for the class labels of Naive Matrix GPU, Naive Matrix CPU, Tiled Matrix CPU, and Strassen Matrix GPU. The classification report for the XGBoost model is shown in Table 4.4 below.

| | precision | recall | f1-score | support |
|---------------------|-----------|--------|----------|---------|
| Naive_Matrix_GPU | 1.00 | 1.00 | 1.00 | 89 |
| Naive_Matrix_CPU | 1.00 | 1.00 | 1.00 | 2 |
| Tiled_Matrix_CPU | 1.00 | 1.00 | 1.00 | 157 |
| Strassen_Matrix_GPU | 1.00 | 1.00 | 1.00 | 18 |
| accuracy | | | 1.00 | 266 |
| macro_avg | 1.00 | 1.00 | 1.00 | 266 |
| weighted_avg | 1.00 | 1.00 | 1.00 | 266 |

Table 4.4: XGBoost Classification Report

4.2.4 Random Forest Classifier

The Random forest classifier model was implemented on the data-set, and accuracy scores for the Random forest classifier model were generated, which has achieved 100 percent accuracy in both training and testing data-sets as shown in the Figure 4.7.

```
In [108]: print('Random Forest Classifier Training set score: {:.4f}'.format(acc_tr_rfc))
          print('Random Forest Classifier Test set score: {:.4f}'.format(acc_rfc))

Random Forest Classifier Training set score: 1.0000
Random Forest Classifier Test set score: 1.0000
```

Figure 4.7: Random Forest Classifier Accuracy Score

Using the confusion matrix technique, the performance of the Random forest classifier model is evaluated and the results are interpreted to determine the misclassification count and accurate prediction count of the various class labels. The confusion matrix of the Random forest classifier is shown in Figure 4.8.

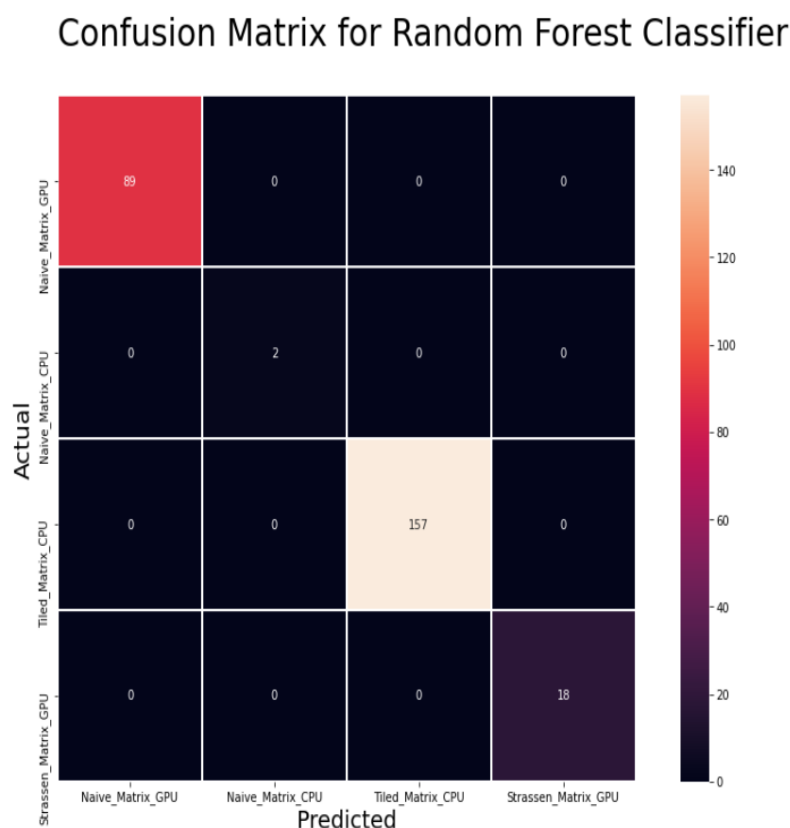


Figure 4.8: Random Forest Confusion Matrix

The classification report is analyzed to know the correctness of the Random forest classifier model thereby, evaluated precision, recall, f1 score, support for the class labels of Naive Matrix GPU, Naive Matrix CPU, Tiled Matrix CPU, and Strassen Matrix GPU. The classification report for the Random forest classifier model is shown in Table 4.5 below.

| | precision | recall | f1-score | support |
|---------------------|-----------|--------|----------|---------|
| Naive_Matrix_GPU | 1.00 | 1.00 | 1.00 | 89 |
| Naive_Matrix_CPU | 1.00 | 1.00 | 1.00 | 2 |
| Tiled_Matrix_CPU | 1.00 | 1.00 | 1.00 | 157 |
| Strassen_Matrix_GPU | 1.00 | 1.00 | 1.00 | 18 |
| accuracy | | | 1.00 | 266 |
| macro_avg | 1.00 | 1.00 | 1.00 | 266 |
| weighted_avg | 1.00 | 1.00 | 1.00 | 266 |

Table 4.5: Random Forest Classification Report

4.2.5 LightGBM

The Light GBM was implemented on the data-set, the average accuracy scores obtained from five splits for the Stratified K-Fold Cross-validation strategy of the

Light GBM, which has achieved 93 percent accuracy in both training and testing data-sets as shown in the Figure 4.9.

```
In [116]: # print the scores on training and test set
print('LightGBM Training set score: {:.4f}'.format(acc_tr_lgb))
print('LightGBM Test set score: {:.4f}'.format(acc_lgb))

LightGBM Training set score: 0.9301
LightGBM Test set score: 0.9301
```

Figure 4.9: LightGBM Accuracy Score

Using the confusion matrix technique, the performance of the Light GBM is evaluated and the results are interpreted to determine the misclassification count and accurate prediction count of the various class labels. The confusion matrix of the Light GBM is shown in the Figure 4.10.

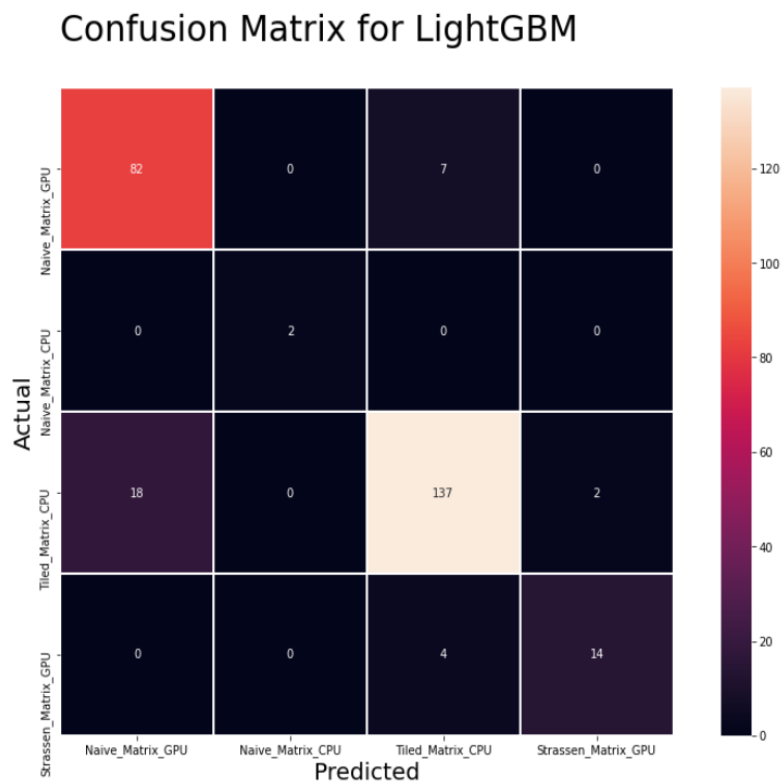


Figure 4.10: LightGBM Confusion Matrix

The above-mentioned confusion matrix figure shows that the LightGBM model misclassified the 27 class labels, 18 of which were labeled as Naive Matrix GPU, 7 as Tiled Matrix CPU, and 2 as Strassen Matrix CPU. The corresponding actual labels for the 18 misclassified labels are Tiled Matrix CPU, 7 are the Naive Matrix GPU, and 2 are the Tiled Matrix CPU.

The classification report is analyzed to know the correctness of the LightGBM model thereby, evaluated precision, recall, f1 score, support for the class labels of Naive Matrix GPU, Naive Matrix CPU, Tiled Matrix CPU, and Strassen Matrix GPU. The classification report for the LightGBM model is shown in Table 4.6 below.

| | precision | recall | f1-score | support |
|----------------------------|------------------|---------------|-----------------|----------------|
| Naive_Matrix_GPU | 0.82 | 0.92 | 0.87 | 89 |
| Naive_Matrix_CPU | 1.00 | 1.00 | 1.00 | 2 |
| Tiled_Matrix_CPU | 0.93 | 0.87 | 0.90 | 157 |
| Strassen_Matrix_GPU | 0.88 | 0.78 | 0.82 | 18 |
| accuracy | | | 0.88 | 266 |
| macro_avg | 0.91 | 0.89 | 0.90 | 266 |
| weighted_avg | 0.89 | 0.88 | 0.88 | 266 |

Table 4.6: LightGBM Classification Report

4.2.6 Support Vector Machine

The SVM model was implemented on the data-set, accuracy scores for the SVM model were generated, which achieved 74 percent accuracy in training and 73 percent accuracy in testing data-sets as shown in the Figure 4.11.

```
In [132]: print('SVM Training set score: {:.4f}'.format(acc_tr_svc))
          print('SVM Test set score: {:.4f}'.format(acc_svc))

SVM Training set score: 0.7421
SVM Test set score: 0.7346
```

Figure 4.11: SVM Accuracy Score

Using the confusion matrix technique, the performance of the SVM model is evaluated and the results are interpreted to determine the misclassification count and accurate prediction count of the various class labels. The confusion matrix of the SVM model is shown in the Figure 4.12.

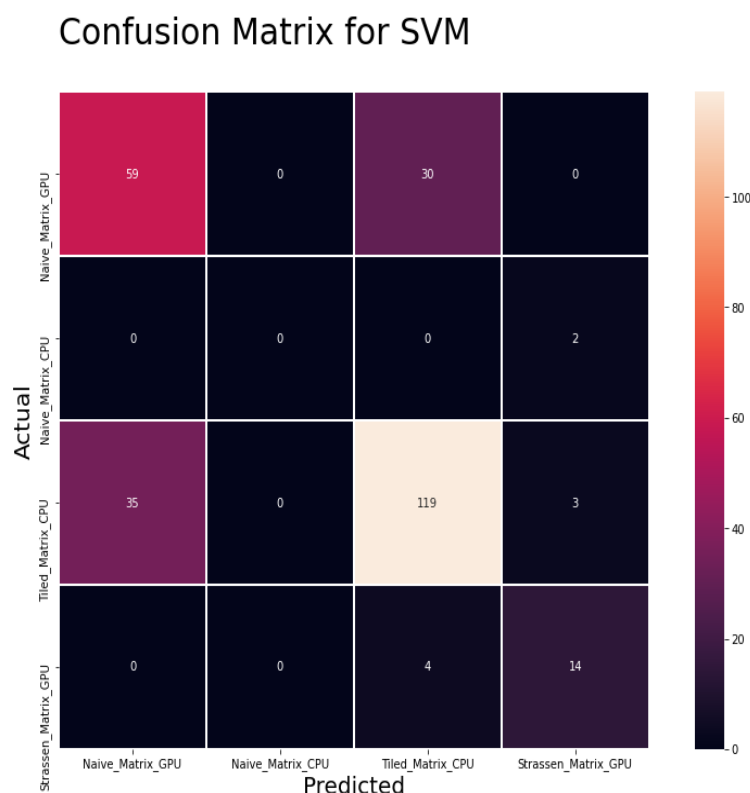


Figure 4.12: SVM Confusion Matrix

The above-mentioned confusion matrix figure shows that the SVM model misclassified the 70 class labels, 35 of which were labeled as Naive Matrix GPU, 30 as Tiled Matrix CPU, and 5 as Strassen Matrix CPU. The corresponding actual labels for the 35 misclassified labels are Tiled Matrix CPU, 30 are the Naive Matrix GPU, and out of 5 class labels 3 are the Tiled Matrix CPU and 2 are the Naive Matrix CPU.

The classification report is analyzed to know the correctness of the SVM model thereby, evaluated precision, recall, f1 score, support for the class labels of Naive Matrix GPU, Naive Matrix CPU, Tiled Matrix CPU, and Strassen Matrix GPU. The classification report for the SVM model is shown in Table 4.7 below.

| | precision | recall | f1-score | support |
|---------------------|-----------|--------|----------|---------|
| Naive_Matrix_GPU | 0.63 | 0.66 | 0.64 | 89 |
| Naive_Matrix_CPU | 0.00 | 0.00 | 0.00 | 2 |
| Tiled_Matrix_CPU | 0.78 | 0.76 | 0.77 | 157 |
| Strassen_Matrix_GPU | 0.74 | 0.78 | 0.76 | 18 |
| accuracy | | | 0.72 | 266 |
| macro_avg | 0.54 | 0.55 | 0.54 | 266 |
| weighted_avg | 0.72 | 0.72 | 0.72 | 266 |

Table 4.7: SVM Classification Report

4.3 Evaluation Results

Based on the experimental results of each algorithm, the algorithm's accuracy scores and other performance metrics like runtime training and runtime prediction times are tabulated.

| Model | Testing Score | Training Score | Runtime Training | Runtime Prediction |
|---------------------------------|---------------|----------------|------------------|--------------------|
| Decision Tree Classifier | 100% | 100% | 0.01253 | 0.000122 |
| XGBoost | 100% | 100% | 0.96 | 0.006078 |
| Random Forest Classifier | 100% | 100% | 0.140082 | 0.011788 |
| KNN | 98.7% | 99.9% | 0.036943 | 0.005745 |
| LightGBM | 93% | 93% | 0.190915 | 0.001620 |
| SVM | 73.4% | 74.2% | 0.023791 | 0.014848 |

Table 4.8: Models Evaluation Report

The table and the figure show the three of the model's DTC, XBoost, Random Forest Classifier has accuracy scores of 100. In terms of performance parameters Runtime Training and Runtime Prediction, the DTC model has the lowest values, which makes the DTC is best performing model among those.

The below Figure 4.13 shows the DTC model's predicted outcome for the input dimension size 160.

```

In [157]: X = [[160]]#Giving Input Dimensions as input to the Model

In [158]: getting=dtc.predict(np.array(X))#Model Predicting the right matrix and its respective computing resources.
           print(getting)
           tellAlgo(getting[0])

[4]
The Suitable Resource that you need to use for this dimensions are: GPU
The suitable algorithm should be used here is: Strassen_Matrix

```

Figure 4.13: DTC Outcome

4.4 Summary of Analysis

The primary objective of this research is to find the best-performing machine learning model for optimization of the heterogeneous parallel computing system by evaluating the performance of each machine learning classifier model, namely KNN, SVM, Light GBM, XGBoost, DTC and Random forest classifier. These Models are measured with the performance metrics of accuracy, recall, f-measure, support, and confusion matrix by experimenting. In this section, the performance of the models is mentioned. A comparison of the models is performed based on the obtained results. The model that is best suited for classifying the data-set is identified and mentioned in this chapter.

5.1 Answering Research Questions

RQ1) Which machine learning algorithms are suitable to predict the optimal implementation variant for heterogeneous parallel computing systems?

The machine learning algorithms are selected according to the data-set type. Here, the data-set will be generated for a heterogeneous parallel computing system by considering a matrix multiplication benchmark application. The data-set would consist of the execution times for the benchmark application implementations for a range of input values, with the winning label column telling the optimal implementation variant in the data-set. Here, the winning label is a target variable, and the values under it are filled with a unique number consisting of 1 to 5, each representing the winning implementation variant for the corresponding input value.

These numbers repeat often under this column, which shows that this data-set is set to be a multi-class classification data-set. In finding which algorithms to be chosen for this followed through investigating the already existing literature works. The research study “Multi-class sentiment classification” suggested the selection of algorithms Decision Tree, Naive Bayes, Support Vector Classifier, K-Nearest Neighbour, and Radial basis function network over multi-class classification data-set [27]. From the above-stated algorithms, we consider three algorithms based on the advantages mentioned in the research “review of the multi-class classification algorithms“. For Decision Tree, handling both numerical and categorical data and computationally easy to understand and interpret; For K-Nearest Neighbour, ease of implementation, and flexible classification scheme; For Support Vector Machine, having largest flexibility over other classification [28].

Since We already decided on the Decision Tree algorithm, other tree based on algorithms like random forest and light gradient algorithm also be considered, the random forest can provide an advantage in giving high accuracy over the decision tree in case of any over-fitting and Light gradient boosting help get faster training speed and higher efficiency with least memory usage [17]. Also, the XGBoost algorithm is considered for being a high-performance model in most machine learning [34].

Thus, noting all the above discussion we have considered a total of 6 algorithms for this project: Support Vector Classifier, K-Nearest Neighbour, Random Forest, Decision tree classifier, XGBoost, and Light Gradient Boosting Model in predicting the optimal implementation variant for heterogeneous parallel computing systems.

RQ2) How selected machine learning algorithms can improve the performance of heterogeneous parallel computing?

The results obtained by answering research question 1 are compared to six machine learning models, namely, Decision Tree Classifier, XGBoost, KNN, LightGBM, SVC, and Random Forest Classifier. The above-mentioned models are then assessed for performance using a stratified cross-validation strategy. To overcome model overfitting, a 5-fold stratified cross-validation technique is used to prepare a model for final testing.

According to the experiment results, the accuracy values for Decision Tree Classifier, XGBoost, and Random Forest Classifier have achieved 100%. The remaining algorithms, such as KNN, Light GBM, and SVC, achieved 98%, 93%, and 73% accuracy, respectively.

Decision Tree Classifier, XGBoost, Random Forest Classifier is identified as the efficient model with good accuracy, recall, precision, and f1 score. The above-mentioned algorithms are compared in Table 5.1: Comparison of Models.

| Model Name | Accuracy | Precision | F1-Score | Recall | Runtime Training | Runtime Prediction |
|---------------------------------|----------|-----------|----------|--------|------------------|--------------------|
| DTC | 100% | 100% | 100% | 100% | 0.01253 | 0.000122 |
| XGBoost | 100% | 100% | 100% | 100% | 0.96 | 0.006078 |
| Random Forest Classifier | 100% | 100% | 100% | 100% | 0.140082 | 0.011788 |
| KNN | 98% | 98% | 95% | 93% | 0.036943 | 0.005745 |
| LightGBM | 93% | 90% | 89% | 89% | 0.190915 | 0.001620 |
| SVM | 73% | 53% | 54% | 55% | 0.023791 | 0.014848 |

Table 5.1: Comparison of Models

In comparison with other machine learning models, the Decision Tree Classifier is not only good in terms of performance metrics but it has achieved the lowest Runtime Training and Prediction. As a result of the experimental observations, We believe that the Decision Tree Classifier is the best fit for predicting the best implementation variant.

In the result section, The figure 4.13 depicts how, given an input size dimension of 160, the most efficient machine learning model was found for the matrix multiplication application. The decision tree classifier predicts that Strassen matrix multiplication with GPU resource usage is the best implementation variant. Thus, knowing this heterogeneous parallel computing application can improve its performance by choosing the optimal variant that is Strassen matrix multiplication and its corresponding resource to be implemented in GPU.

Chapter 6

Conclusions and Future Work

In this research, we used a machine learning approach to optimize the heterogeneous parallel computing system for application over execution time based on resource utilization, such as CPU or GPU, and the input size. For this, six machine learning algorithms are used: KNN, XGBoost, DTC, Random Forest Classifier, LightGBM, and SVM. We investigated the performance of these machine learning algorithms in predicting the optimal implementation variant of the considered matrix multiplication application, such that it increases run-time performance on heterogeneous parallel computing systems.

According to the results, the algorithms DTC, XGBoost, and Random Forest Classifier outperformed the others in terms of accuracy ranking achieving a 100% accuracy score. Based on the parameters of run-time prediction and run-time training, the algorithm DTC is shown to be the most efficient in the selection of the optimal implementation variant for an application on a heterogeneous parallel computing system.

In future work, the research may be advanced further by taking into account application implementations written in different languages such as Fortran, Python, and as well as on heterogeneous parallel computing platforms other than CUDA.

Also, the research can be advanced use comparing the performance outcomes of these machine learning algorithms to other parallel computing applications such as sorting, depth for search, and so on, and analyzing these results to determine whether there is a better machine learning algorithm overall.

Bibliography

- [1] O'Reilly Media. Professional CUDA C Programming.
- [2] Suejb Memeti, Sabri Pllana, Alécio Binotto, Joanna Kołodziej, and Ivona Brandic. Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review. *Computing*, 101(8):893–936, August 2019.
- [3] Agnieszka Bier and Zdzisław Sroczyński. Efficiency Comparison of Modern Computer Languages: Sorting Benchmark. In Radek Silhavy, Petr Silhavy, and Zdenka Prokopova, editors, *Intelligent Systems in Cybernetics and Automation Control Theory*, Advances in Intelligent Systems and Computing, pages 299–310, Cham, 2019. Springer International Publishing.
- [4] Lizy Kurian John and Lieven Eeckhout. *Performance Evaluation and Benchmarking*. CRC Press, October 2018. Google-Books-ID: Ge_LBQAAQBAJ.
- [5] Chao Jin, Bronis R. de Supinski, David Abramson, Heidi Poxon, Luiz DeRose, Minh Ngoc Dinh, Mark Endrei, and Elizabeth R. Jessup. A survey on software methods to improve the energy efficiency of parallel computing. *The International Journal of High Performance Computing Applications*, 31(6):517–549, 2017. Publisher: Sage Publications Sage UK: London, England.
- [6] Sabri Pllana, Siegfried Benkner, Eduard Mehofer, Lasse Natvig, and Fatos Xhafa. Towards an Intelligent Environment for Programming Multi-core Computing Systems. In Eduardo César, Michael Alexander, Achim Streit, Jesper Larsson Träff, Christophe Cérin, Andreas Knüpfer, Dieter Kranzlmüller, and Shantenu Jha, editors, *Euro-Par 2008 Workshops - Parallel Processing*, Lecture Notes in Computer Science, pages 141–151, Berlin, Heidelberg, 2009. Springer.
- [7] Lu Li. *Programming Abstractions and Optimization Techniques for GPU-based Heterogeneous Systems*. Ph.D., Linköping University, Linköping, Sweden, April 2018. ISBN: 9789176853702.
- [8] Homin Kang, Hyuck Chan Kwon, and Duksu Kim. HPMaX: heterogeneous parallel matrix multiplication using CPUs and GPUs. *Computing*, 102(12):2607–2631, December 2020.
- [9] Wangdong Yang, Kenli Li, and Keqin Li. A hybrid computing method of SpMV on CPU–GPU heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 104:49–60, June 2017.
- [10] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In

- Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.
- [11] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, March 2020. Google-Books-ID: tZnSDwAAQBAJ.
 - [12] Xian-Da Zhang. Machine Learning. In Xian-Da Zhang, editor, *A Matrix Algebra Approach to Artificial Intelligence*, pages 223–440. Springer, Singapore, 2020.
 - [13] Babcock University, Osisanwo F.Y, Akinsola J.E.T, Awodele O, Hinmikaiye J. O, Olakanmi O, and Akinjobi J. Supervised Machine Learning Algorithms: Classification and Comparison. *IJCTT*, 48(3):128–138, June 2017.
 - [14] Candice Bentéjac, Anna Csörg\Ho, and Gonzalo Martínez-Muñoz. A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, 54(3):1937–1967, 2021. Publisher: Springer.
 - [15] Archana Chaudhary, Savita Kolhe, and Raj Kamal. An improved random forest classifier for multi-class classification. *Information Processing in Agriculture*, 3(4):215–222, December 2016.
 - [16] Ömer Faruk Ertuğrul and Mehmet Emin Tağluk. A novel version of k nearest neighbor: Dependent nearest neighbor. *Applied Soft Computing*, 55:480–490, 2017. Publisher: Elsevier.
 - [17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. page 9.
 - [18] Yuantao Chen, Jie Xiong, Weihong Xu, and Jingwen Zuo. A novel online incremental and decremental learning algorithm based on variable support vector machine. *Cluster Computing*, 22(3):7435–7445, 2019. Publisher: Springer.
 - [19] Sriramakrishnan Chandrasekaran. A Machine Learning Implementation of Predicting the Real Time Scenarios in a better way. page 12.
 - [20] Mohammad Uzair. Cross Validation Improvements in TMVA. Technical report, 2018.
 - [21] University of Ljubljana, Faculty of computer and information science, Slovenia., Tomaž Dobravec, and Patricio Bulić. Comparing CPU and GPU Implementations of a Simple Matrix Multiplication Algorithm. *IJCEE*, 9(2):430–438, 2017.
 - [22] Mehdi G. Duaimi, Abbas FJ AL-Gburi, Ehsan A. Al-Zubaidi, and Ibraheem Al-Jadir. Implementing Multithreaded Programs using CUDA for GPGPU to Solve Matrix Multiplication. *Journal of Xi'an University of Architecture & Technology*, 12:3083–3089.
 - [23] Vi Ngoc-Nha Tran, Tommy Oines, Alexander Horsch, and Phuong Hoai Ha. REOH: Using Probabilistic Network for Runtime Energy Optimization of Heterogeneous Systems. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 381–388, December 2018. ISSN: 1521-9097.
 - [24] Zheqi Yu, Pedro Machado, Adnan Zahid, Amir M. Abdulghani, Kia Dashtipour, Hadi Heidari, Muhammad A. Imran, and Qammer H. Abbasi. Energy and Performance Trade-Off Optimization in Heterogeneous Computing via Reinforce-

- ment Learning. *Electronics*, 9(11):1812, November 2020. Number: 11 Publisher: Multidisciplinary Digital Publishing Institute.
- [25] Machine learning for performance and power modeling of heterogeneous systems.
- [26] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. Adaptive optimization for OpenCL programs on embedded heterogeneous systems. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2017, pages 11–20, New York, NY, USA, June 2017. Association for Computing Machinery.
- [27] Yang Liu, Jian-Wu Bi, and Zhi-Ping Fan. Multi-class sentiment classification: The experimental comparisons of feature selection and machine learning algorithms. *Expert Systems with Applications*, 80:323–339, September 2017.
- [28] P. C. Chaitra and R. Saravana Kumar. A review of multi-class classification algorithms. *International Journal of Pure and Applied Mathematics*, 118(14):17–26, 2018.
- [29] Chase E. Golden, Michael J. Rothrock Jr, and Abhinav Mishra. Comparison between random forest and gradient boosting machine methods for predicting *Listeria* spp. prevalence in the environment of pastured poultry farms. *Food research international*, 122:47–55, 2019. Publisher: Elsevier.
- [30] Gaurush Hiranandani, Shant Boodaghians, Ruta Mehta, and Oluwasanmi Koyejo. Multiclass Performance Metric Elicitation. page 10.
- [31] Sebastian Raschka and Vahid Mirjalili. Python Machine Learning: Machine Learning and Deep Learning with Python. *Scikit-Learn, and TensorFlow. Second edition ed*, 2017.
- [32] M. L. Walker, Y. H. Dovoedo, S. Chakraborti, and C. W. Hilton. An Improved Boxplot for Univariate Data. *The American Statistician*, 72(4):348–353, October 2018. Publisher: Taylor & Francis _eprint: <https://doi.org/10.1080/00031305.2018.1448891>.
- [33] Anita Rácz, Dávid Bajusz, and Károly Héberger. Effect of Dataset Size and Train/Test Split Ratios in QSAR/QSPR Multiclass Classification. *Molecules*, 26(4), February 2021.
- [34] Widya Fajar Mustika, Hendri Murfi, and Yekti Widyaningsih. Analysis Accuracy of XGBoost Model for Multiclass Classification - A Case Study of Applicant Level Risk Prediction for Life Insurance. In *2019 5th International Conference on Science in Information Technology (ICSITech)*, pages 71–77, October 2019.
- [35] Amalia Luque, Alejandro Carrasco, Alejandro Martín, and Ana de las Heras. The impact of class imbalance in classification performance metrics based on the binary confusion matrix. *Pattern Recognition*, 91:216–231, July 2019.
- [36] Nesime Tatbul, Tae Jun Lee, Stan Zdonik, Mejbah Alam, and Justin Gottschlich. Precision and Recall for Time Series. *arXiv:1803.03639 [cs]*, January 2019. arXiv: 1803.03639.
- [37] R. Saravanan and Pothula Sujatha. A State of Art Techniques on Machine Learning Algorithms: A Perspective of Supervised Learning Approaches in Data Classification. In *2018 Second International Conference on Intelligent Comput-*

ing and Control Systems (ICICCS), pages 945–949, June 2018.

Appendix A

Supplemental Information

The graph below represents the comparative analysis of the matrix input size dimensions varying from 16 to 160 plotted against the Execution time in milliseconds.

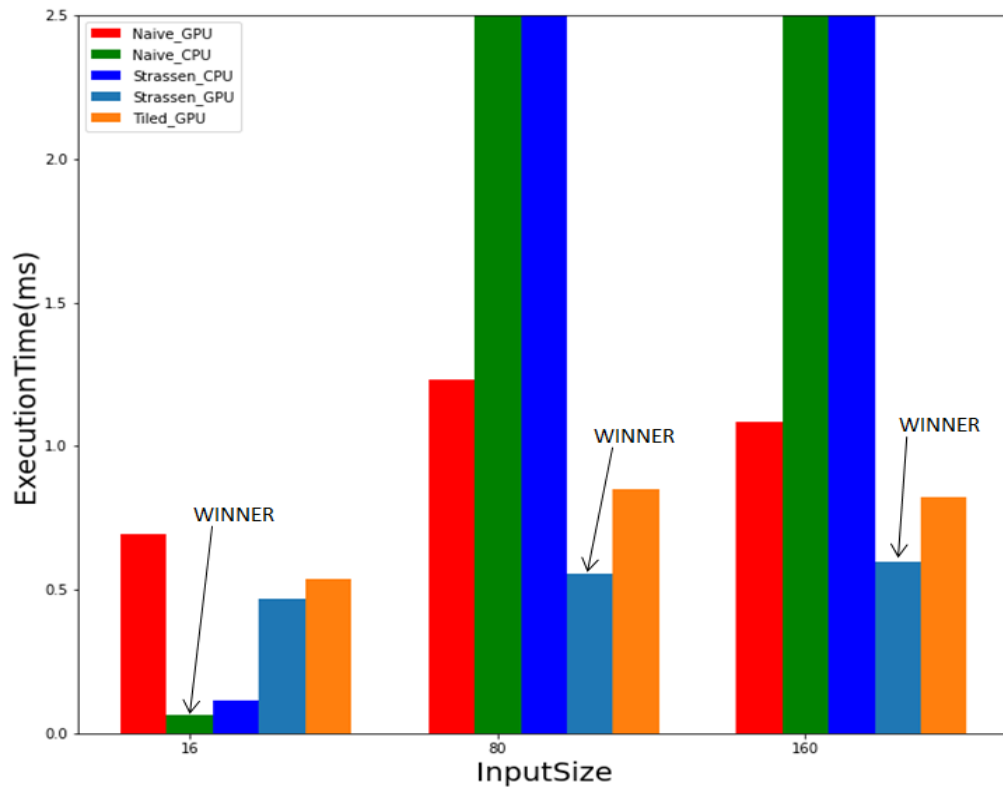


Figure A.1: Input Size v/s Execution Time(ms) graph for the input dimensions ranging from 16 to 160

The graph below represents the comparative analysis of the matrix input size dimensions varying from 1008 to 1328 plotted against the Execution time in milliseconds.

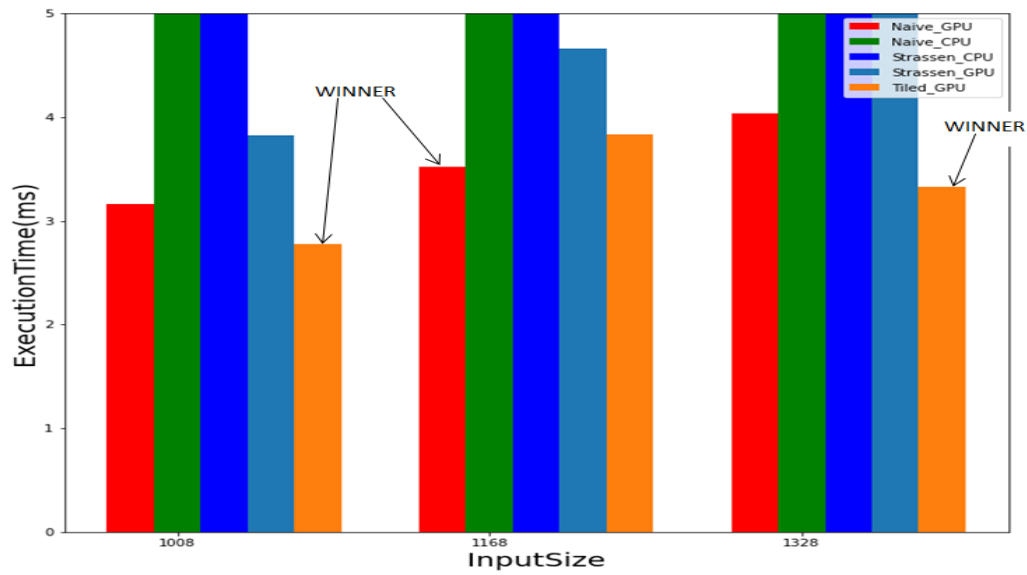


Figure A.2: Input Size v/s Execution Time(ms) graph for the input dimensions ranging from the 1008 to 1328

The graph below represents the comparative analysis of the matrix input size dimensions varying from 4464 to 8560 plotted against the Execution time in milliseconds.

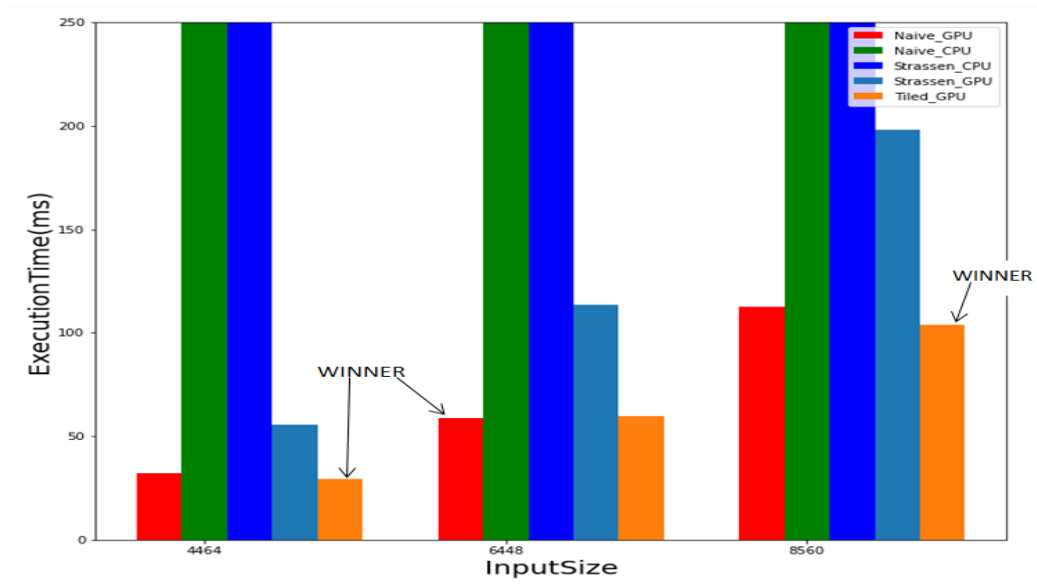


Figure A.3: Input Size v/s Execution Time(ms) graph for the input dimensions ranging from the 4464 to 8560

