

CSS 535 Final Project - Exploration of Multiple Regression Model Training using GPU

Jesse Leu, Khushaal Kurswani, Anthony Bustamante

March 16, 2023

Abstract

This paper explores the use of GPU technology in the training of multiple linear regression models. Multiple linear regression analysis is a commonly used statistical method in many fields, including finance, marketing, and social sciences. However, it can be computationally intensive, especially when working with large datasets. The use of GPUs in parallel computing has been shown to significantly speed up training time for machine learning models, and this project investigates the potential benefits of using GPUs in multiple linear regression analysis.

The paper presents experimental results comparing the training time and performance of multiple regression models on a CPU versus a GPU. Our results showed that the use of GPUs can obtain good accuracy for the predictions, nevertheless, this time we did not obtain better performance for the GPU implementation with CUDA. Still there are several more things that we could try in the future to try to improve the CUDA implementation and reach better performance.

Overall, this paper highlights the potential benefits of using GPU technology in multiple linear regression analysis and provides a good evaluation of different implementations of linear regression on CPU and GPU that can be used as reference in the future.

Introduction

Machine Learning is a field that is constantly getting more popular and more frequently used in current times. Machine Learning models identify patterns from large amounts of data and apply those patterns to make predictions on new data. This is extremely useful because the models can identify relationships and correlations in an immense dataset much more efficiently than humans. The models' predictions can automate tasks that a person needs to perform in their jobs and daily lives.

One of the biggest aspects of Machine Learning is the training phase. Without a large-sized, diverse, and well-formatted dataset, the model will not be able to fully understand the concept it is designed to predict. As a result, researchers spend a lot of effort and time training their models.

Many times the training phase can take several hours to days depending on the size of the training dataset. As a result, Machine Learning becomes inaccessible and an incredibly time-consuming process.

Several computations in the Machine Learning models' training process are parallelizable meaning that those tasks can be done simultaneously. Consequently, these computations can be performed on a GPU instead of a CPU to speed up the training process. This is because GPUs have a larger number of computing cores allotted for performing simple arithmetic operations in parallel when compared to CPUs.

The goal of this project is to accelerate the training process of a Linear Regression model (a type of Machine Learning model) by not only parallelizing the training computation operations on a GPU but also employing various High-Performance Computing techniques to make the computations more optimized for parallel computing. A Linear Regression model was chosen for this project out of all the various types of models since it is the most basic and foundational model.

Related Work

In NVIDIA's article titled "What is Regression," it is explained a regression model is a supervised machine learning algorithm that "estimates the relationship between a target outcome label and one or more feature variables to predict a continuous numeric value" (NVIDIA, n.d.). Linear regression models estimate the relationship between the outcome and the features by fitting a linear function to the data. The NVIDIA article further mentions that the GPUs can be used to perform the matrix multiplication operations and the loss function calculation required by linear regression models. Libraries and tools such as PyTorch, RAPIDS, and TensorFlow have APIs that can be used to assign the GPU operations related to the training and testing of various different machine learning models such as Linear Regression.

In Rory Mitchell and Eibe Frank's journal article titled "Accelerating the XGBoost algorithm using GPU computing," the researchers accelerate the XGBoost algorithm using parallel computing patterns such as tiling, reduction, and prefix sums. The article mentions that the GPU implementation can accelerate "decision tree construction within individual boosting iterations" (Mitchell & Frank, 2017). The results of their optimizations showed that the parallel implementations demonstrated a significant speedup when training the model on the GPU compared to training the model on the CPU while maintaining a similar level of accuracy. For example, when training the XGBoost model on a subset of the Higgs dataset, the CPU took 7,761 seconds to train the model whereas the GPU took 1,201 seconds to train the model which is a speed up of 6.62. Both the CPU-trained and GPU-trained models had an accuracy of 84.26%. This shows that training the machine learning model on the GPU is far more efficient than training the model on the CPU and there will be minimal loss in accuracy.

In Chetlur et al.'s paper titled "cuDNN: Efficient Primitives for Deep Learning," a Deep Learning library called cuDNN is introduced. This library provides APIs for training Deep Neural Network models. Although cuDNN is not for linear regression models, the paper provides great insight

into the considerations that need to be taken when training a machine learning model on the GPU. For example, GPUs have a smaller memory size than CPUs and there is a great latency in the data transfer between the CPU and the GPU. In addition, the paper also demonstrates how to take complex machine learning training operations such as performing convolutions on image data can be converted into simple mathematical operations such as matrix-matrix multiply (Chetlur et al., 2014).

In their paper "Database Processing by Linear Regression on GPU using CUDA," Kulkarni, Sawant, and Inamdar discuss the use of CUDA as a programming model to improve the performance of processing large amounts of data in a database. Linear regression and linear convolution techniques are used to predict data patterns, while Linear Time Invariance (LTI) systems and measures such as covariance and eigenvalues/vectors are used for predictive power. GPUs are low-cost devices that have been utilized for many general-purpose computations, and the CUDA programming model allows for parallel computing with CPUs, which can improve performance. The multithreading and Single Instruction Multiple Data (SIMD) approaches of CUDA can also be beneficial for performance improvement. The paper demonstrates the implementation of database processing using linear regression on a GPU with CUDA on an NVIDIA Graphics Processing Unit integrated with a Central Processing Unit.(Kulkarni, Sawant, & Inamdar, 2011)

Technical Specifications

Windows Computer

- **CPU:** Intel® Core™ i7-7500U, 2 cores, 2.7 GHz speed
- **GPU:** Nvidia GeForce 940MX, Maxwell microarchitecture, 5.0 Compute Capability
- **Operating System:** Windows 10
- CUDA version 12.0
- NVIDIA CUDA Compiler (nvcc)
- NVIDIA Visual Profiler (nvprof)
 - nvprof and Nvidia visual profile are the same tool, nvprof for command-line and the latter one is the GUI version.

Linux Computer

- **CPU:** Intel i9 10920X, 12 cores, 3.5 GHz speed
- **GPU:** Nvidia RTX 3070, Ampere microarchitecture, 8.6 Compute Capability
- **Operating System:** Rocky Linux (a Linux distribution)
- CUDA version 12.0

- NVIDIA CUDA Compiler (nvcc)
- NVIDIA Nsight Compute (ncu)
 - Nsight Compute is an interactive profiler for CUDA

Programming Languages and External Libraries

- C/C++
- CUDA
- Python
- Scikit-Learn
- NumPy
- Pandas

C/C++ and CUDA were used to implement the Linear Regression Algorithm on the GPU. Python, Scikit-Learn, NumPy, and Pandas were used to implement the Linear Regression model on CPU and to experiment with pre-built models.

Compilation and Profiling Instructions

Install CUDA. nvcc should come with the CUDA toolkit installation. Install Nsight Compute or NVPROF depending on GPU compute capability.

The source code for the naive GPU implementation and the optimized GPU implementation can be found in the regressor.cu and regressorOpt.cu files, respectively. The following command can be inputted into the Command Line Interface to compile this code.

```
$ nvcc <cuda_code_file>
```

If using a GPU with a microarchitecture that is deprecated in the installed CUDA version, then add the `--gpu-architecture=compute_#` flag to the compilation command. For example, the Maxwell microarchitecture is deprecated from CUDA 11 onwards so the following command would be used to compile the CUDA code files.

```
$ nvcc <cuda_code_file> --gpu-architecture=compute_50 -lcublas
```

It is suggested to run the compiled code in the following manner so that the extensive output is written to a text file rather than the terminal.

```
$ <executable_file> > output.txt
```

The following command can be inputted into the Command Line Interface to profile the kernel functions in the CUDA code using NSIGHT Compute.

```
$ ncu -o <profiler_output_file_name> --set full <executable_file>
```

Note that for the Windows machine we did use NVPROF because according to NVIDIA, Nsight compute is available for Volta architectures or superior[Windows 10 error with nsight], and due to we did have a Maxwell architecture in windows we could not use Nsight compute. For NVIDIA's recommendation we had to use Nvprof.

Additionally, to run the CPU implementation you have to do it with python:

```
python program.py
```

Design

Initial Project Ideas

Originally, the project was going to focus on accelerating the training process of a Convolutional Neural Network model (a type of Deep Neural Network) through the GPU. The cuDNN paper was the main influence behind that idea. As a result, the initial project design was to find a few open-source Machine Learning libraries, parallelize sections of their code, run the modified code on the GPU, and then analyze the modified code using a profiler. However, we could not find any suitable open-source libraries that would meet the project requirements.

Consequently, the project's goal shifted toward trying to implement the design of cuDNN's parallel implementation for the convolution layer for a Convolution Neural Network. Since cuDNN's code is not open source, we would have tried to develop the code ourselves purely based on the description provided in the paper. After that, we would have profiled the code to analyze it. This idea was dropped due to the complexity associated with implementing a convolution layer. Also, if we would have tried to simplify the project to simulate a single convolution layer, it would have been difficult to compare our implementation's performance with pre-existing models due to the fact that it would not have been possible to isolate and test just a single convolution layer from the pre-existing models.

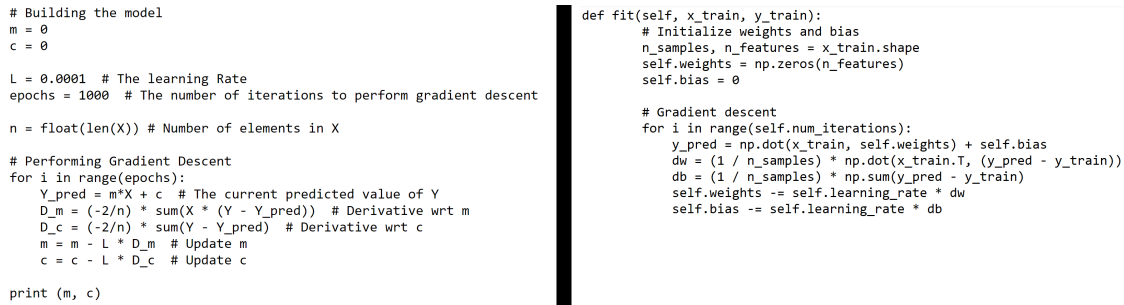
Thus, we decided to shift the project's goal towards implementing another exciting topic in machine learning model: Multiple Linear Regression.

Linear Regression Implementation

The machine learning model that was implemented for this project was a multiple Linear Regression model. The model has several parameters that are used to compute the prediction and the number of parameters is the same as the number of features in the dataset. To train the model, it is important to calculate the optimal value for the model's parameters. Gradient Descent is the algorithm that modifies the values of the parameter over several iterations based on the error of the prediction computed by using the parameter values. Since this algorithm was the biggest part of training a Linear Regression model, it was necessary to parallelize this algorithm for the GPU in order to accelerate the training process.

The gradient descent algorithm works by using the parameters' current values and the input data to predict the target variable's value. Then, it calculates the error for each prediction. It uses the learning rate, the error values, and each parameter's input value for each instance in the training dataset to calculate the new value for each parameter. There is an extra parameter called a bias which is not associated with any input values. The bias is trained similarly to the rest of the parameters. This process of modifying the parameters and the bias happens for several iterations. Ideally, the algorithm iterates until the parameter and bias values converge and have minimal change between iterations. There can also be a predefined number of iterations for which the algorithm to run.

Adarsh Menon's article titled "Linear Regression using Gradient Descent" provided a detailed explanation regarding the gradient descent algorithm and even provided Python code for single variable gradient descent (Menon, 2018). The code for this project adapts Menon's code for Linear Regression to perform gradient descent on multiple parameters.



```
# Building the model
m = 0
c = 0

L = 0.0001 # The learning Rate
epochs = 1000 # The number of iterations to perform gradient descent
n = float(len(X)) # Number of elements in X

# Performing Gradient Descent
for i in range(epochs):
    Y_pred = m*X + c # The current predicted value of Y
    D_m = (-2/n) * sum(X * (Y - Y_pred)) # Derivative wrt m
    D_c = (-2/n) * sum(Y - Y_pred) # Derivative wrt c
    m = m - L * D_m # Update m
    c = c - L * D_c # Update c

print (m, c)
```

```
def fit(self, x_train, y_train):
    # Initialize weights and bias
    n_samples, n_features = x_train.shape
    self.weights = np.zeros(n_features)
    self.bias = 0

    # Gradient descent
    for i in range(self.num_iterations):
        y_pred = np.dot(x_train, self.weights) + self.bias
        dw = (1 / n_samples) * np.dot(x_train.T, (y_pred - y_train))
        db = (1 / n_samples) * np.sum(y_pred - y_train)
        self.weights -= self.learning_rate * dw
        self.bias -= self.learning_rate * db
```

Figure 1: Menon's gradient descent code in Python (Left), Our adaptation of gradient descent for Multiple Linear Regression in Python (Right)

After that, we converted the code from Python into C/C++ and CUDA. In the code, we identified three mathematical operations which could be parallelized and run on the GPU: Matrix-Vector Multiplication, Vector-Vector Subtraction, and Vector-Constant Multiplication. Consequently, these three operations became our CUDA kernel functions. These kernel functions were profiled and then modified to optimize them through the use of shared memory and loop unrolling. Then, these modified kernel functions were profiled to determine if the optimizations were successful at making the training process more efficient.

Below is a screenshot of an example output that would be printed out upon running the C/C++ and CUDA code.

```

Top 10 rows of x_train:
1.61726e+09 1226 902 324
1.61728e+09 1103 833 270
1.6173e+09 1099 826 273
1.61732e+09 1115 848 267
1.61734e+09 1064 801 263
1.61736e+09 1059 791 267
1.61739e+09 1062 790 272
1.61741e+09 1062 790 272
1.61743e+09 1057 797 260
1.61745e+09 1064 796 268
...

Training Time: 215.223 milliseconds
Training FLOPS: 3.54414e+08 FLOPS

Regressor Model Weights: Feature[0] Weight: 0.508412 Feature[1] Weight: 0.480656
Feature[2] Weight: 0.491613 Feature[3] Weight: 0.400968 Bias: 0.00390181

Predict Value:      Actual Value:
predict[0]:         11.25   Actual[0]   10.00
predict[1]:         10.46   Actual[1]   11.00
predict[2]:          9.65   Actual[2]    9.00
predict[3]:          9.84   Actual[3]    9.00
predict[4]:         10.60   Actual[4]    9.00
predict[5]:         10.80   Actual[5]   10.00
predict[6]:         10.66   Actual[6]   10.00
predict[7]:         10.43   Actual[7]    9.00
predict[8]:         11.18   Actual[8]   12.00
predict[9]:         11.42   Actual[9]   12.00
...

Model r-squared value: 0.237471

```

Figure 2: Example C/C++ CUDA code output

Dataset

For the purpose of this final project, we are using our own dataset, the dataset is composed of 4 independent variables and 1 dependent variable. The dataset was obtained from a proxy server working on a private network and was obtained with help of Nagios software, it dates back to 2021 when the dataset was extracted. The independent variables/features are composed as the following table states.

Timestamp	Connected Clients	Active Clients	Idle Clients
1617256800	1226.17218	902.13341	324.03877
1617278400	1102.97019	833.0531	269.87489
1617300000	1098.53606	825.73575	272.80031
...
1624899600	1154.7704	787.35489	367.41556

Table 1: Independent variables for this project

And the dependent variables are composed by the following table:

Timestamp	CPU Consumption (%)
-----------	---------------------

1617256800	6.85147
1617278400	6.52981
1617300000	5.45794
...	...
1624899600	8.35956

Table 2: Dependent variable for this project

The objective of the project in regard to this dataset was to train our regression models with this dataset and see the computing time, execution rate, and accuracy for each type of implementation that we used.

The dataset was composed of 4330 samples in total for each feature(independent and dependent variables), and it was divided into two parts, the first part was for training our models and it was composed of 80 % of the dataset, and the rest 20 % was used for testing.

For more information, you can visit the GitHub link for this project in which we have included this dataset and some others as well for testing purposes.

Experiments

Naive vs. Optimized GPU Implementation

In our CUDA section, we have developed two implementations: a naive version and an optimized version. In the naive version, we reviewed the multiple linear regression algorithm using Python on the CPU and discovered that the training function's following calculations can be efficiently performed on the GPU:

```
y_predicted = np.dot(X, self.weights) + self.bias
dw = (1 / num_samples) * np.dot(X.T, (y_predicted - y))
db = (1 / num_samples) * np.sum(y_predicted - y)
self.weights -= self.learning_rate * dw
self.bias -= self.learning_rate * db
```

Figure 3: calculations can be efficiently performed on the GPU

Based on the Python code, we have the following function:

1. Matrix-vector multiplication: The function computes the row index for each thread, ensuring that each thread processes a single row of data. Then, it checks that the row index is within the boundaries of the matrix. Inside a loop, the function performs the multiplication of a row of the matrix with the vector, iterates over the columns of the matrix, and accumulates the result in the corresponding element of the result array. After the loop, the function multiplies each element in result by the factor and then adds the bias to the resulting vector.


```

__global__ void MVMult(float *matrix, float *vector, float *result, int M, int N, float bias, float factor)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M)
    {
        for (int i = 0; i < N; i++)
        {
            result[row] += matrix[row * N + i] * vector[i];
        }
        result[row] *= factor;
        result[row] += bias;
    }
}

```

Figure 4: Matrix-vector multiplication function

2. Vector-Vector subtraction: The function first calculates the index of each thread, ensuring that each thread processes a unique element of the vectors. Then, it checks whether the calculated index is within the bounds of the vectors to prevent accessing memory outside of the allocated space. Finally, the function performs element-wise subtraction of the corresponding elements in vec1 and vec2, and stores the result in the corresponding element of the res array.

```

__global__ void VVSub(float *vec1, float *vec2, float *res, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < N)
    {
        res[index] = vec1[index] - vec2[index];
    }
}

```

Figure 5: Vector-Vector subtraction function

3. Vector-Constant multiplication: The function first calculates the index of each thread, ensuring that each thread processes a unique element of the vectors. Next, the function checks if the calculated index is within the bounds of the vectors to avoid accessing memory outside of the allocated space. Inside the if statement, the function multiplies the corresponding element in vec by the scalar num and subtracts the result from the corresponding element in res, producing a new vector as output.

```

__global__ void VCMult(float *vec, float num, float *res, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < N)
    {
        res[index] -= vec[index] * num;
    }
}

```

Figure 6: Vector-Constant multiplication function

After starting with a naive CUDA implementation, we implemented our optimization version to improve performance using:

1. Shared memory: Shared memory is useful when multiple threads need to access the same data multiple times. By loading that data into shared memory once and then allowing each thread to access it from there, we can reduce the number of global memory accesses.
2. Loop unrolling: By unrolling a loop, we can reduce the number of branches and conditional statements that the CPU or GPU needs to execute, which can improve performance by reducing the time spent on these operations. Additionally, unrolling a loop can expose additional opportunities for instruction-level parallelism, allowing multiple instructions to be executed simultaneously. In our implementation, we have unrolled the loop to execute 4 instructions in each iteration, which can lead to issues when the size of the vector or matrix is not divisible by 4. To address this issue, we created another kernel function to handle the remaining elements after loop unrolling.

The optimized functions are:

1. Optimized Matrix-vector multiplication: The function performs a matrix-vector multiplication using shared memory to improve computation performance. Firstly, it copies a subset of the vector data into shared memory, where each thread reads a portion of the data. Then, the function checks if the row index is within the boundaries of the matrix and proceeds to compute the corresponding row of the matrix. The computation is done through loop unrolling, which iterates over the columns of the matrix and accumulates the result in the corresponding element of the result array. In case there are any unprocessed elements, the second loop takes care of the leftover part. Finally, the function multiplies each element in the result by the factor and adds the bias to the resulting vector.

```

__global__ void MVMult(float *matrix, float *vector, float *result, int M, int N, float bias, float factor,
int numShared)
{
__shared__ float cachedVector[MAX_SHARE_SIZE];
int row = blockIdx.x * blockDim.x + threadIdx.x;
int numRowsPerThread = numShared / blockDim.x;
int startIndex = threadIdx.x * numRowsPerThread;
int endIndex = startIndex + numRowsPerThread - 1;
for (int i = startIndex; i <= endIndex; i++)
{
cachedVector[i] = vector[i];
}
int numCopied = blockDim.x * numRowsPerThread;

__syncthreads();
if (row < M)
{
int i = 0;
for (; i < numCopied && i + 3 < N; i+=4)
{
result[row] += matrix[row * N + i] * cachedVector[i];
result[row] += matrix[row * N + i + 1] * cachedVector[i + 1];
result[row] += matrix[row * N + i + 2] * cachedVector[i + 2];
result[row] += matrix[row * N + i + 3] * cachedVector[i + 3];
}

for (; i < N; i++)
{
result[row] += matrix[row * N + i] * vector[i];
}

result[row] *= factor;
result[row] += bias;
}
}

```

Figure 7: Optimized Matrix-vector multiplication function

2. **Optimized Vector-Vector subtraction** This function performs an element-wise subtraction of two input vectors (vec1 and vec2) and stores the result in a third vector (res) using loop unrolling to process four elements at a time. The function computes the index of each thread, and each thread handles four consecutive elements in the vectors. If there are not enough elements left in the vectors to complete a group of four, those elements will not be processed, and another kernel function will be used to handle the remaining elements.

```

__global__ void VVSub(float *vec1, float *vec2, float *res, int N)
{
    int index = (threadIdx.x + blockIdx.x * blockDim.x) * 4;
    if (index + 3 < N) // loop unrolling
    {
        res[index] = vec1[index] - vec2[index];
        res[index + 1] = vec1[index + 1] - vec2[index + 1];
        res[index + 2] = vec1[index + 2] - vec2[index + 2];
        res[index + 3] = vec1[index + 3] - vec2[index + 3];
    }
}

```

Figure 8: Optimized Vector-Vector subtraction function

3. Optimized Vector-Constant multiplication: This function performs an element-wise multiplication of a vector (vec) with a scalar value (num) and subtracts the resulting product from the corresponding element in another vector (res). The function uses loop unrolling to process four elements at a time, and it computes the index of each thread accordingly. If there are not enough elements left in the vectors to complete a group of four, those elements will not be processed, and another kernel function will be used to handle the remaining elements.

```

__global__ void VCMult(float *vec, float num, float *res, int N)
{
    int index = (threadIdx.x + blockIdx.x * blockDim.x) * 4;
    if (index + 3 < N) // loop unrolling
    {
        res[index] -= vec[index] * num;
        res[index + 1] -= vec[index + 1] * num;
        res[index + 2] -= vec[index + 2] * num;
        res[index + 3] -= vec[index + 3] * num;
    }
}

```

Figure 9: Optimized Vector-Constant multiplication function

Block Size Experiment

Another experiment done in this project was to manipulate the block size, or the number of threads per block, for the two main CUDA kernel functions: MVMult and VVSub. These two kernels were profiled using Nsight Compute in order to observe the changes in performance as the block size was changed. This experiment was done on both the naive version and the optimized version of the kernels. The main motivation for performing this experiment was to determine if there was an execution configuration for which the optimized kernels would outperform the naive kernels.

The block sizes that were used for these experiments for both kernels in both versions were 128,

256, 512, 768, and 1024 threads per block. The experimental block sizes are multiples of 32 in order to increase occupancy and fill out the warp size of 32 threads. The profiler metrics that were tracked in this experiment are Floating-point Operations per Second (FLOPS), L1 cache hit ratio, and achieved occupancy.

Below is a screenshot of the code used to set up the experiment.

```
void blocksExperiment(int m, int n, float alpha, float *x_train, float *y_train)
{
    int numSizes = 4;
    int blockSizeList[numSizes] = {128, 256, 512, 1024};

    Regressor regressor(m, n);

    for (int i = 0; i < numSizes; i++)
    {
        regressor.fit(x_train, y_train, alpha, 1, blockSizeList[i], blockSizeList[i]);
    }
}
```

Figure 10: Block size experiment code

The for loop iterates through each block size and calls the regressor's fit function which will call the kernel functions. The regressor's fit function is provided the training dataset, expected predictions, alpha which is the learning rate, an iteration count of 1, the block size for MVMult, and the block size for VVSub.

Efficiency and Accuracy Comparison with CPU Implementations

For this part, we used three implementations to compare and analyze, first, we chose the CUDA naive version, second, we implemented a naive multiple linear regression algorithm on CPU with python, and lastly, we also implemented a version of linear regression with Sklearn. Furthermore, we normalized the dataset with Min-Max Scaler for all the implementations for our dataset can be fitted correctly to the regression models.

The dataset that we trained our models on was plotted in Figure 13, and we can see that this dataset does not follow a specific pattern doing our project a little more challenging. From this dataset our models are going to learn and predict future values for unseen data.

About the implementation:

Note that for all the implementations in this project, we used the same hyperparameters, for example: We set up the Learning rate at 0.01 and the number of iterations at 1000, and this was the standard not just for the comparison between GPU implementation and CPU implementation, but also for the optimizations on GPU.

A. For the implementation on CUDA: We used the naive version and details about it have been described previously.

B. For the implementation on CPU: We used the code on python in Figure 11, we created a class with two methods, one for training and another one for predicting future values, then we proceeded to normalize the dataset and fit it into the fit method for training, once the model is trained we can start predicting values with the predict method.

```
class MultipleLinearRegression:
    def __init__(self, learning_rate=0.01, num_iterations=1000):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        num_samples, num_features = X.shape

        # Initialize weights and bias to zeros
        self.weights = np.zeros(num_features)
        self.bias = 0

        # Gradient descent
        for i in range(self.num_iterations):
            y_predicted = np.dot(X, self.weights) + self.bias
            dw = (1 / num_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / num_samples) * np.sum(y_predicted - y)
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
        y_predicted = np.dot(X, self.weights) + self.bias
        return y_predicted

def train(X_train, y_train):
    regressor = MultipleLinearRegression()
    regressor.fit(X_train, y_train)
    return regressor
```

Figure 11: Naive implementation on CPU and Python

C. For the implementation with Sklearn: Sklearn is a very popular library for machine learning, and we used it for comparison purposes. We used the code on python in Figure 12, imported the Linear Regression algorithm from Sklearn, and proceeded to train the model with the normalized values, once the model was trained we started predicting values with the predict function.

It is also important to note that we used normalization min-max to standardize all the tests because in python the outputs for min-max are single-precision values, whereas some other popular types like standarscaler are double-precision, and due to we got the best accuracy with min-max normalization on CUDA over standarscaler for example, we chose this technique for all our experiments. Besides, it is worth to mention that we did not implement any type of encoding strategy because our dataset was numeric and not categorical.

```

def train(X_train, y_train):
    regressor = LinearRegression()
    regressor.fit(X_train, y_train)
    return regressor

def main():
    dataset = pd.read_csv('x_train.csv')
    dataset1 = pd.read_csv('y_train.csv')
    dataset2 = pd.read_csv('x_test.csv')
    dataset3 = pd.read_csv('y_test.csv')

    n=3633 #Number of samples
    m=4    #Number of features
    p=2    #Number of cores

    X_train = dataset.iloc[:n, :].values
    y_train = dataset1.iloc[:n, :].values
    X_test = dataset2.iloc[:, :].values
    y_test = dataset3.iloc[:, :].values

    scaler = MinMaxScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    regressor = train(X_train, y_train)
    y_pred = regressor.predict(X_test)

```

Figure 12: Sklearn implementation on CPU

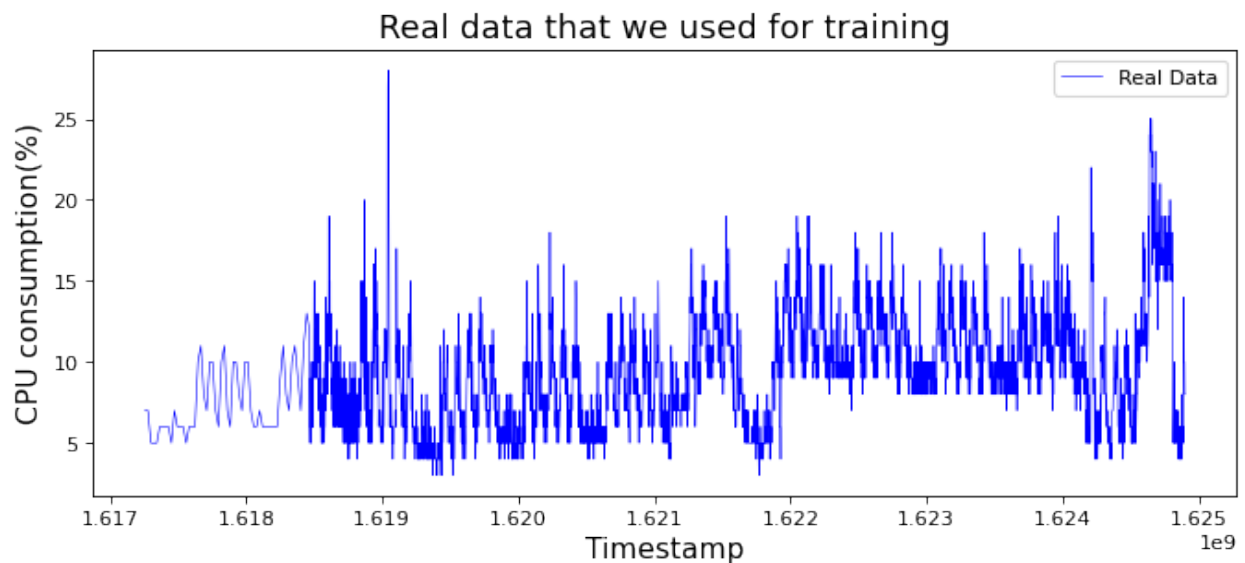


Figure 13: Plotting of the dataset we used for training

Results and Analysis

Naive vs. Optimized GPU Implementation

Naive vs optimized Matrix-vector multiplication:

Function	Elapsed time (ms)	Execution rate(Gflops)	Register per thread	Occupancy (%)	Shared memory per block	L1 hit rate	L2 hit rate
Naive	3.296	8.81	22	59.46	0	79.13	54.69
Optimized	3.84	7.56	39	31.85	48	84.60	72.53

Table 3: Naive vs optimized Matrix-vector multiplication

Naive vs optimized Vector-Vector subtraction:

Function	Elapsed time (ms)	Execution rate(Gflops)	Register per thread	Occupancy (%)	Shared memory per block	L1 hit rate	L2 hit rate
Naive	2.4	1.51	16	58.30	0	0	46.78
Optimized	2.668	0.73	39	31.85	0	84.60	72.53
Leftover function	2.304	—	16	56.74	0	0	61.37

Table 4: Naive vs optimized vector-vector subtraction

2nd Naive vs optimized Matrix-vector multiplication:

Function	Elapsed time (ms)	Execution rate(Gflops)	Register per thread	Occupancy (%)	Shared memory per block	L1 hit rate	L2 hit rate
Naive	270.784	0.107	22	2.08	0	90.08	103.99
Optimized	298.784	0.0972	39	2.09	0	90.28	90.84

Table 5: 2nd Naive vs optimized Matrix-vector multiplication

Naive vs optimized Vector-Constant multiplication:

Function	Elapsed time (ms)	Execution rate(Gflops)	Register per thread	Occupancy (%)	Shared memory per block	L1 hit rate	L2 hit rate
Naive	2.214	3.28	16	2.01	0	33.33	98.76
Optimized	2.272	3.19	16	2.13	0	83.33	100.92

Table 6: Naive vs optimized Vector-constant multiplication

Our attempts to optimize the implementation of our functions have resulted in worse performance than the original implementation. We suspect that the excessive use of loop unrolling may be a contributing factor. Loop unrolling can increase the register usage of each thread, which can limit the number of threads that can be executed simultaneously on a multiprocessor. This can lead to lower occupancy and lower overall performance. Additionally, implementing loop unrolling requires another kernel function to handle leftover data, which can increase processing time. Furthermore, the data size may not be large enough to see significant benefits from unrolling and shared memory.

When comparing the cache hit rates between the first and third tables, we noticed a significant difference. Although both tables represent the same function, the L1 and L2 cache hit rates were much lower in the third table. We believe this is because the data was loaded into cache during the first matrix-vector multiplication function call. However, the second function call had a much lower occupancy, which suggests that the kernel may have been accessing a small amount of data that entirely fits within the L1 cache. This can result in a low register usage and a low occupancy.

The second table, it shows that the naive implementation of vector-vector subtraction has a 0 L1 cache hit rate. This is likely due to the fact that each thread in the kernel accesses a different element of the two input vectors and writes the result to a different element of the output vector. As a result, there is no opportunity for the data to be cached in the L1 cache since each thread accesses a unique memory location that is not reused by any other thread.

In the case of the fourth table, both the naive and optimized implementations had very low occupancy. This may be due to the high register usage per thread in vector-constant multiplication. Each thread loads the vector and scalar constant into registers, performs multiplication, and stores the result back into memory. This process results in high register usage per thread, limiting the number of threads that can be executed simultaneously on a multiprocessor and leading to lower occupancy and lower overall performance. The optimized version has a much higher L1 cache hit rate, likely due to the fact that more data is stored in registers, which reduces the need for memory accesses. This reduction in memory accesses increases the likelihood of accessing data that is already in the L1 cache, leading to a higher L1 cache hit rate.

Block Size Experiment

Matrix Vector Multiplication Kernel

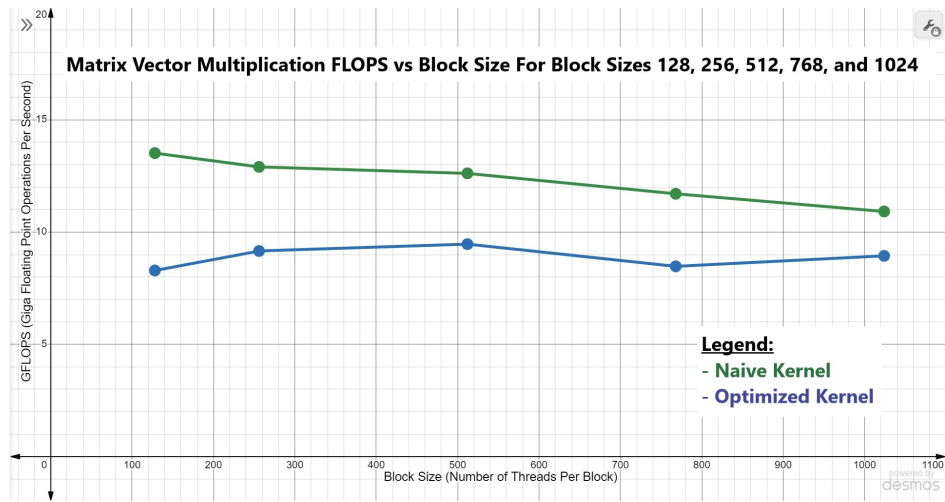


Figure 14: Matrix Vector Multiplication FLOPS vs Block Size Graph

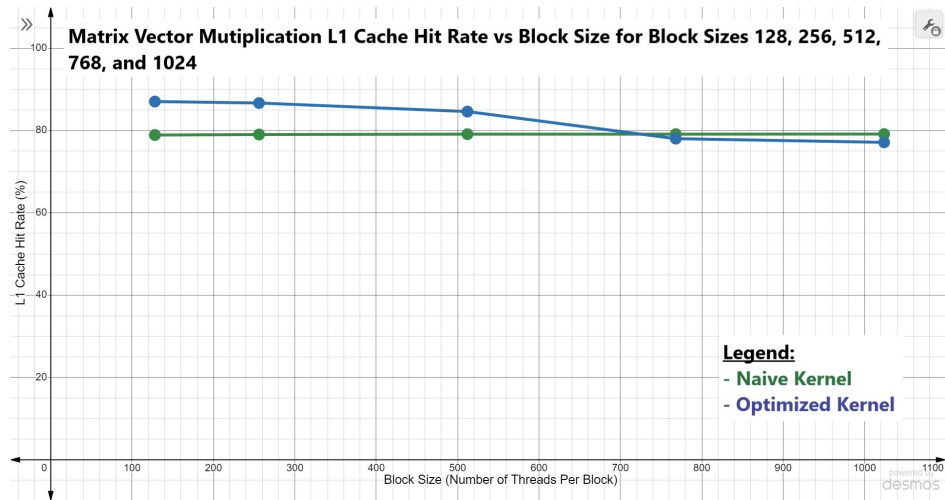


Figure 15: Matrix Vector Multiplication L1 Cache Hit Rate vs Block Size Graph

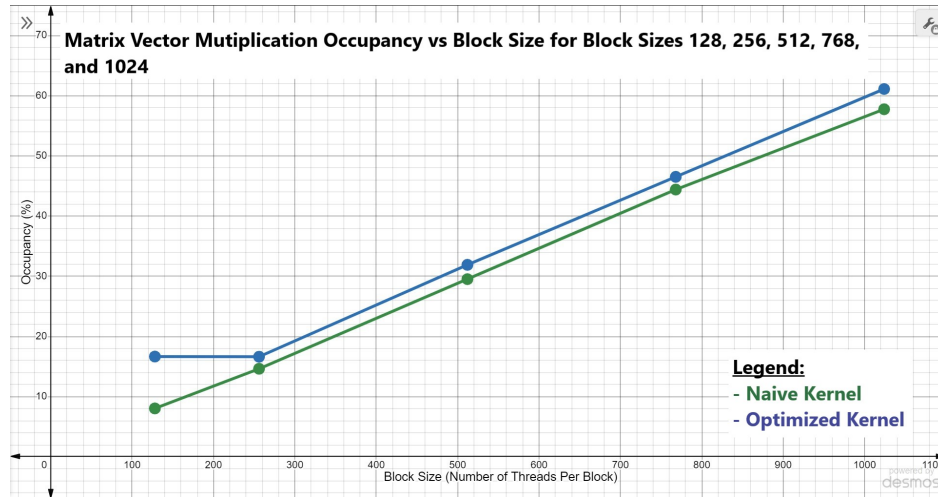


Figure 16: Matrix Vector Multiplication Occupancy vs Block Size Graph

From the FLOPS graph, it is apparent that the optimized MVMult kernel performed at a lower number of FLOPS than the naive version of the kernel despite the shared memory and loop unrolling techniques that were employed in the optimized version. A possible reason for the optimization's lack of improvement in performance could be that the dataset being used to was not large enough for the optimizations to make the computation process more efficient. In fact, the greater number of instructions in the optimized kernel may have led to longer wall time and as a result, lead to a lesser number FLOPS. Since the dataset was small, the impact of the greater number of instructions in the kernel may have outweighed the impact of the optimizations with respect to FLOPS.

In addition, the FLOPS remained relatively constant for both versions of the kernel regardless of the changes made to the block size. Similarly, the block size did not seem to have a large impact on the L1 cache hit rate of either kernel. The hit rate for both kernels remained relatively constant at around 80%. This means that there was no particular block size where either kernel would have had more FLOPS or a better L1 cache hit rate.

An interesting point to note is that the optimized kernel had a higher L1 cache hit rate at smaller block sizes than the naive kernel. This was because the loop unrolling allowed the optimized kernel to continue to reuse data that they stored in their registers and lower-level caches. This optimization's effect is more prominent in the smaller block sizes because having fewer threads per block allows each thread to have more registers and there is less contention over what is stored in the cache.

In the third graph, it is apparent that both versions of the kernel had greater occupancy as the block size increased. The optimized kernel seemed to slightly outperform the naive version regardless of the changes made to the block size. The reason for this might be that the optimizations allowed the threads to have faster access to data since they were frequently accessing data that was stored in registers and shared memory which is faster than global memory access.

Vector Vector Subtraction Kernel

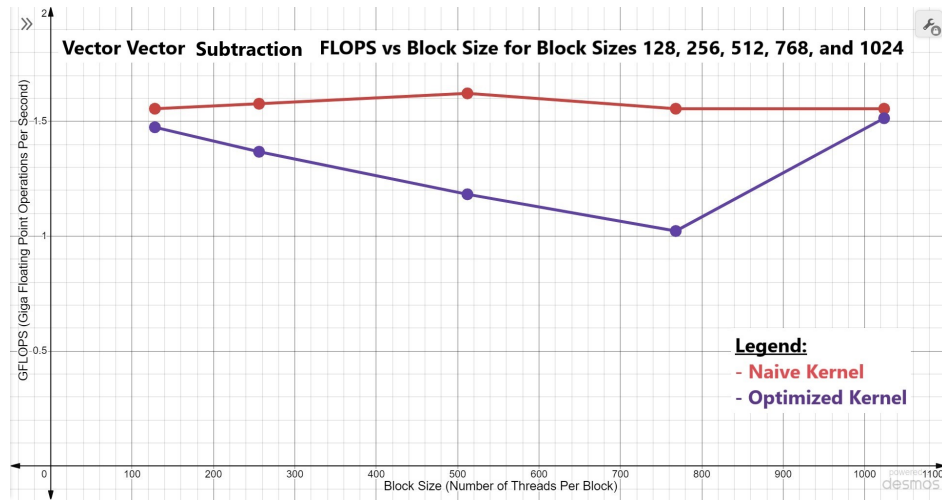


Figure 17: Vector Vector Subtraction FLOPS vs Block Size Graph

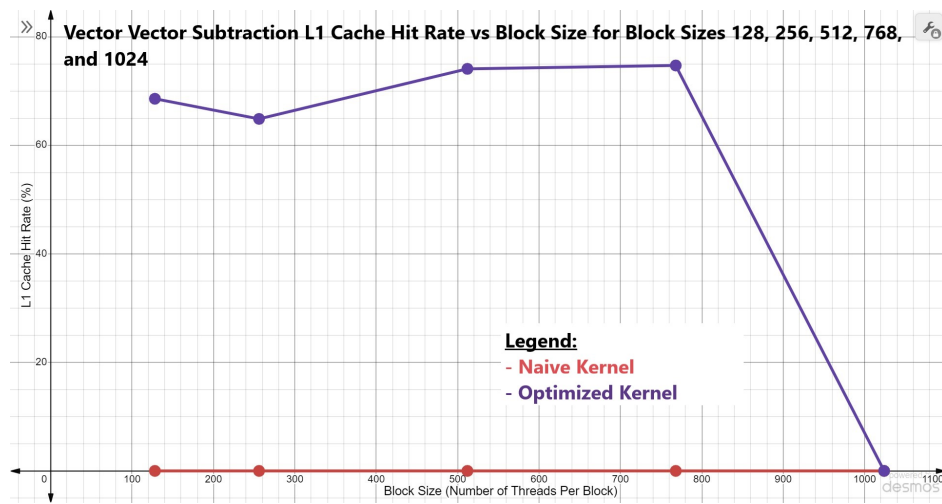


Figure 18: Vector Vector Subtraction L1 Cache Hit Rate vs Block Size Graph

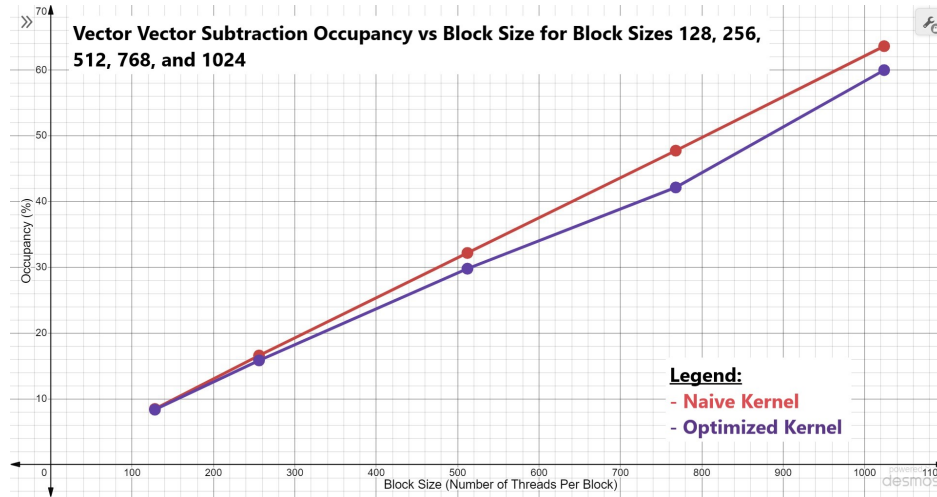


Figure 19: Vector Vector Subtraction Occupancy vs Block Size Graph

It is important to note that at block size 1,024, the optimized kernel resembles the performance of the naive kernel because, at that size, the code logic for the optimized version becomes the same as the naive version. In the optimized kernel's loop unrolling, each thread processes 4 threads and when the block size is 1,024, a total of 4,096 elements would need to be processed by each block for the kernel to run. However, the training dataset only has 3,633 elements which is less than 4,096 and as result, the optimized version of the kernel does not run and instead the kernel handling the leftover elements ends up handling the entire dataset.

From the FLOPS graph, it is apparent that the optimized kernel performed at a lower number of FLOPS than the naive version of the kernel even though the optimized version used loop unrolling. In fact, the number of FLOPS seemed to get worse as the size of the block increased. This may be due to the fact that multiple kernels are required to process the same amount of elements that the naive version is processing with just one kernel.

In the L1 cache hit graph, the optimized version of the kernel has an L1 cache hit rate of over 68% for all the block sizes (except for size 1024 but the reason for that was explained above). On the other hand, the L1 cache hit rate for the naive kernel is always 0. This is because, in the naive kernel, each thread processes one element of the vector and has no reuse of data so the subtraction operation always requires the threads to retrieve data from outside their registers and local memory. Consequently, the threads will never have an L1 cache hit. However, the optimized version of the kernel uses loop unrolling so there are several instances of reuse of data meaning that the L1 cache will have a high cache hit rate for that kernel.

Looking at the occupancy graph, the optimized kernel seems to have a higher occupancy than the naive kernel despite the changes to the block size. Overall, both kernel's occupancy increases as the block size increases. This is similar to the Matrix Vector Multiplication occupancy graph. The reason for this behavior is also similar to the reason described in the Matrix Vector Multiplication section.

Overall Analysis

After analyzing all six graphs, there does seem to be a single block size that would help the optimized kernels outperform the naive kernels. However, larger block sizes allow the optimized kernels to have a greater occupancy than the naive kernel. In addition, the optimized kernels have a greater L1 cache hit rate than the naive version regardless of the block size. Therefore, a block size of 1024 for the MVMult kernel and a block size of 768 for the VVSub kernel will provide the best GPU performance for the optimized kernel for the particular dataset used for this project.

Efficiency and Accuracy Comparison with CPU Implementations

About the configuration that we used: We used seven configurations in total for our experiments, we kept the number of features for training at 4(for the independent variables) and we varied the number of samples(rows) by 500 per configuration. Then, we obtained the measurement for the elapsed time, the execution rate, and the R square value, which is one of the ways to measure accuracy in regression models.

Some of the results that we got are represented in the following tables:

Configuration	Elapsed time (seconds)	Execution rate(MFLOPS)	R2
(501,4)	1.601	6.56215	-5.03688
(1001,4)	2.3870	8.80017	-0.169822
(1501,4)	3.125	10.0819	-0.0655614
(2001,4)	3.843	10.9305	-12.9929
(2501,4)	4.587	11.4467	0.232568
(3001,4)	5.293	11.9036	-28.4969
(3633,4)	6.65	11.4704	0.237825

Table 7: CUDA Implementation running on Windows 10

Configuration	Elapsed time (seconds)	Execution rate(MFLOPS)	R2
(501,4)	0.0216	0.1857	-1.8612
(1001,4)	0.0238	0.3361	0.3225
(1501,4)	0.0256	0.4686	0.3710
(2001,4)	0.0289	0.5539	-0.0958
(2501,4)	0.0849	0.2358	-1.1770
(3001,4)	0.0805	0.2984	-1.0733
(3633,4)	0.0835	0.3481	-1.0067

Table 8: Naive implementation on CPU and Windows 10

Configuration	Elapsed time (seconds)	Execution rate(MFLOPS)	R2
(501,4)	0.0008	4.9484	-2.2560
(1001,4)	0.0008	9.7428	0.2319
(1501,4)	0.0009	13.2712	-0.9125
(2001,4)	0.0009	18.7484	-0.4104
(2501,4)	0.0018	11.3940	-3.4737
(3001,4)	0.0011	22.2064	-2.5905
(3633,4)	0.0010	29.6732	-2.0380

Table 9: Implementation with sklearn on CPU and Windows

10

From tables 7, 8, and 9 we can tell that the best overall performance was obtained by the sklearn implementation, followed by the naive CPU implementation, and lastly the CUDA implementation.

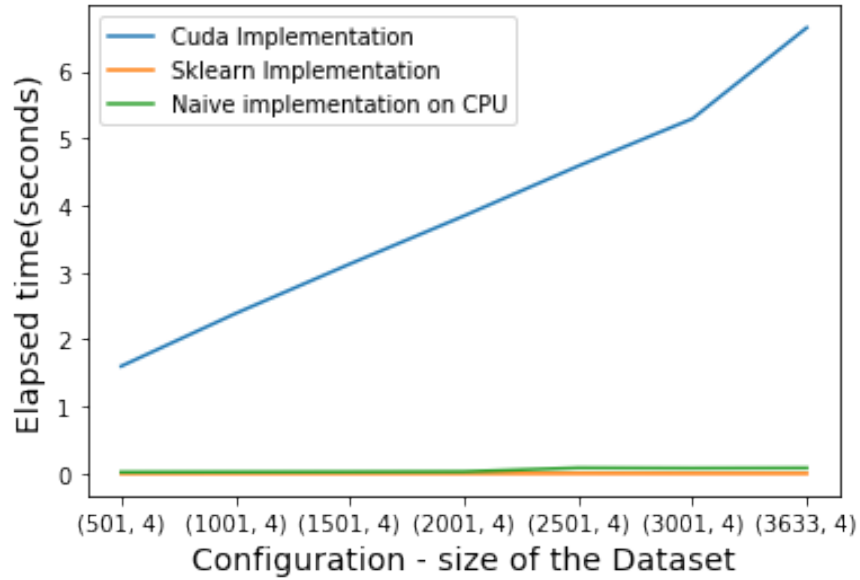


Figure 20: Comparison of Elapsed time for the implementations

From Figure 20, we observed that the CPU implementations achieved the best performance, while the CUDA implementation took several seconds to run. The reason our implementation is slower than the CPU implementation is because the CPU implementation uses the NumPy library, which takes advantage of BLAS and makes it significantly faster.

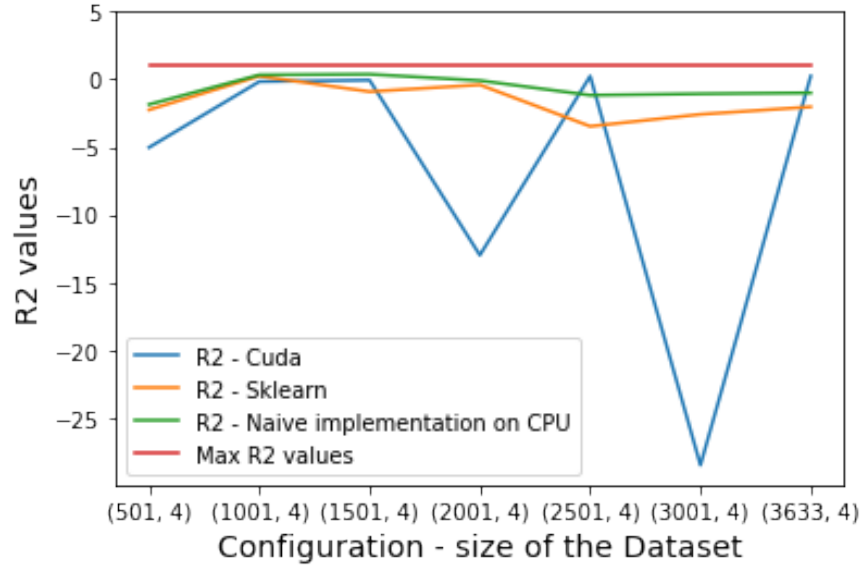


Figure 21: Comparison of R2 for the implementations

From Figure 21 we can see that the R2 results were mostly negative, this in a normal case would tell us that the regression algorithm did not fit well on our dataset, however, we must keep in mind that this is just a way of measurement and is not a rule of thumb. The reason why the R2 values were so poor is that the dataset is too complex and does not follow an easy pattern to predict as for normal regression problems, but that does not mean the multiple linear regression algorithm is bad for this case, in cases like this in which is more important predicting the patterns of consumption over the exact value in a precise frame of time we could use other measurement methods like plotting, a plot would tell us more exactly whether the algorithm predicted the patterns of consumption or not.

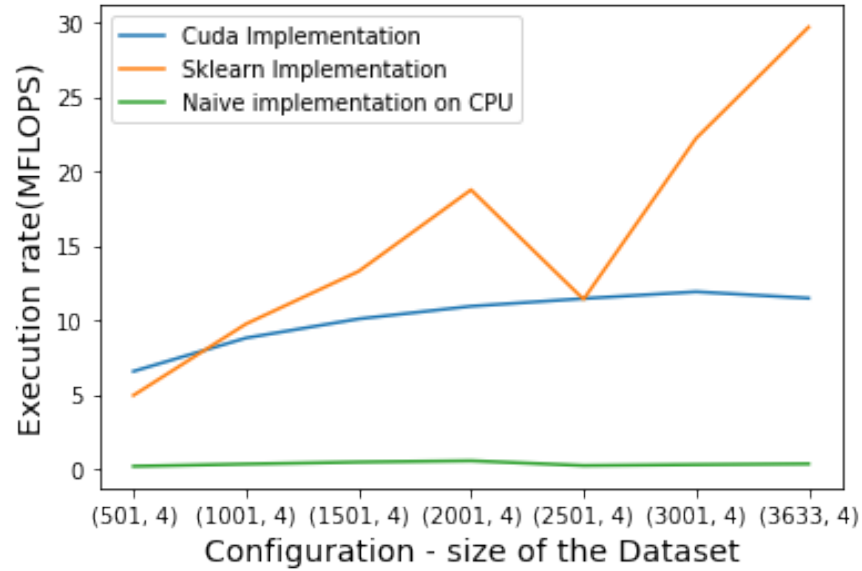


Figure 22: Comparison of Execution rate for the implementations

In Figure 22 we plotted the execution rate for all the implementations, this plot is just to show how much we got per implementation, objectively the execution rate for GPU should not be mixed with the CPU for being different processing units and different architectures, nonetheless, this graph still tell us the values we got for each implementation.

Note that for calculating the execution rate(FLOPS) on CPU we used this formula:

$FLOPS = (2 * n * m^2 * p) / t$; where: (Reference: Introduction to machine learning with python.)

n= Number of samples, m= Number of features, and p= Number of CPU cores.

As told previously, R2 square is not the best way to measure the accuracy of regression models working on dataset without a clear and distinguishable pattern like this, thus, we proceeded to plot different types of configurations to see the results about the prediction of patterns.

Note that for all the following results we used the same hyperparameters: Learning rate = 0.01 and Number of iterations = 1000. Except for sklearn that does not use gradient descent for Linear regression, for this implementation we used the default parameters of Sklearn.

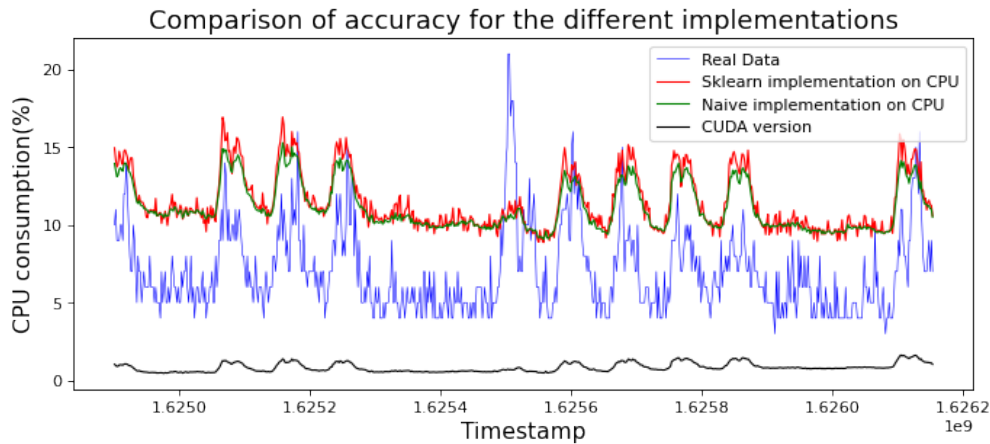


Figure 23: Plot for the first configuration (501,4)

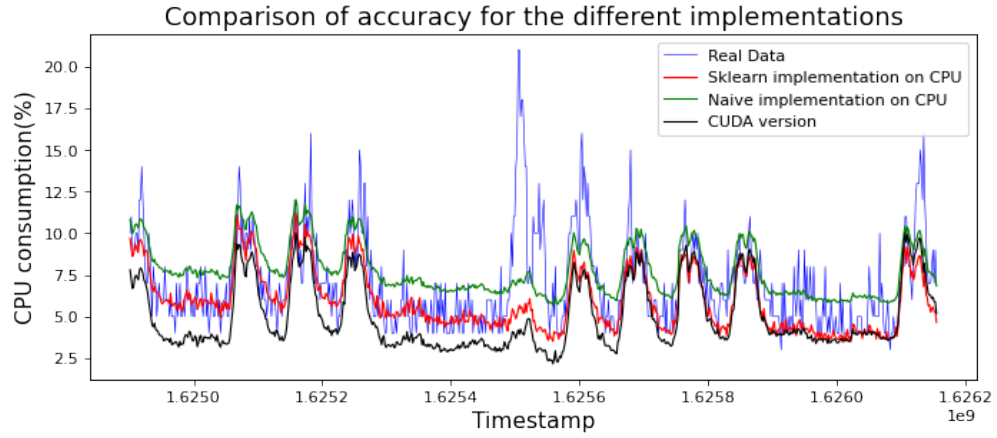


Figure 24: Plot for the second configuration (1001,4)

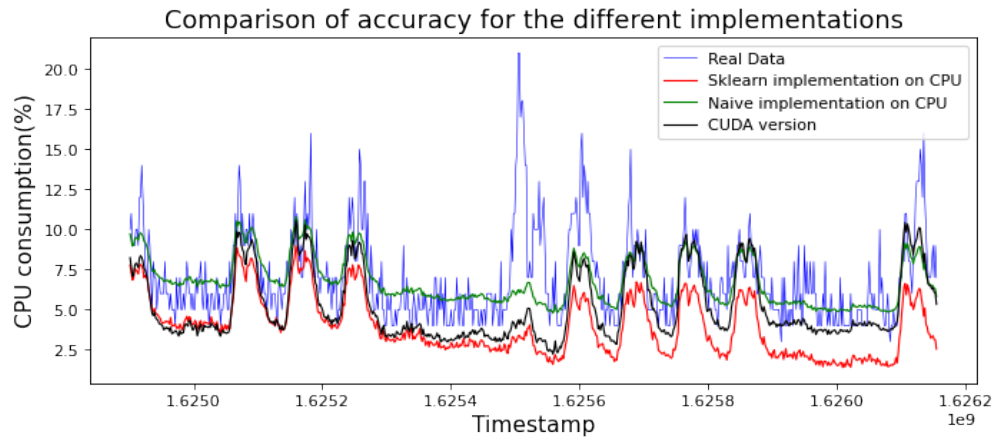


Figure 25: Plot for the third configuration (1501,4)

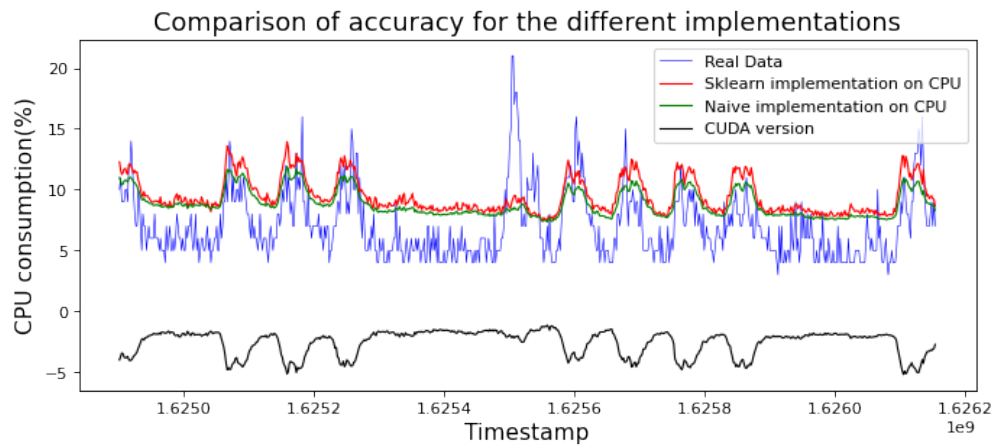


Figure 26: Plot for the fourth configuration (2001,4)

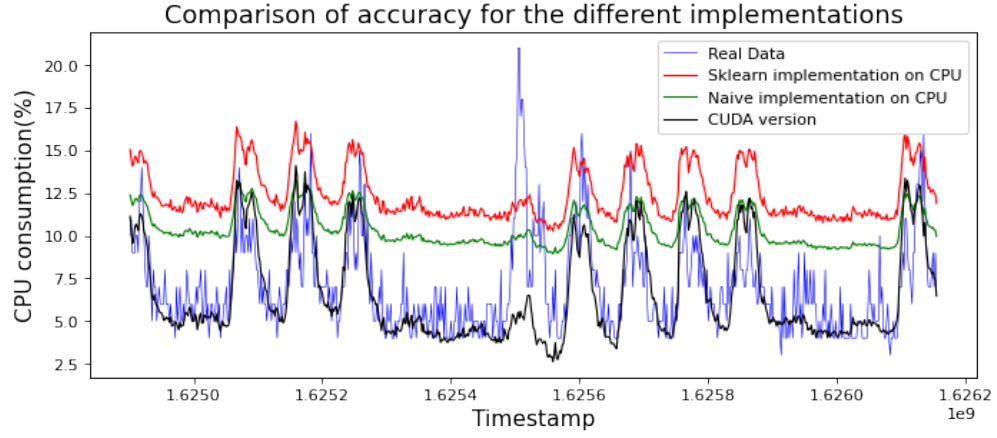


Figure 27: Plot for the fifth configuration (2501,4)

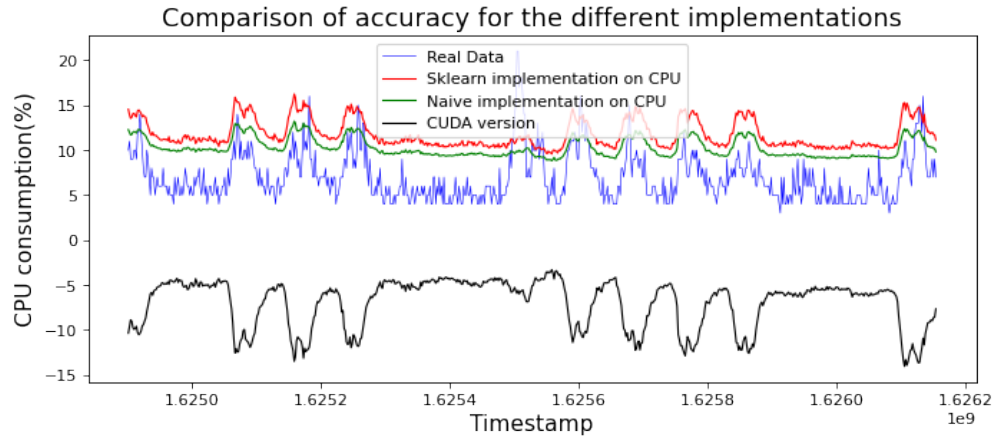


Figure 28: Plot for the sixth configuration (3001,4)

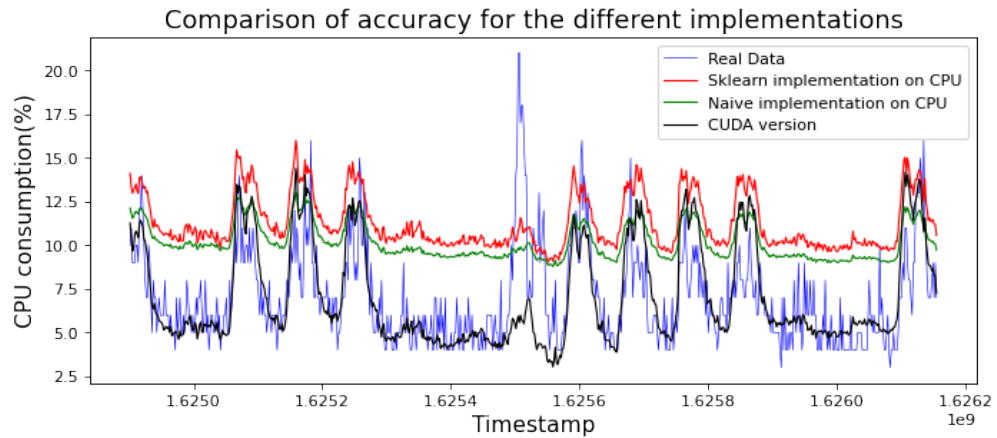


Figure 29: Plot for the seventh configuration (3633,4)

From the graphs just shown we can see that our implementations were actually very good at predicting the patterns of CPU consumption, therefore, from a functional point of view we met the expectations/objectives perfectly, of course, some graphs are closer to the real values but this type of behavior is expected in any regression/machine learning model, that is why usually in a real environment we would be looking for the best configuration/hyperparameters to solve a problem with methods like grid-search or random-search, still the objective of these plots was to show that the different implementations that we designed and coded were very accurate when compared with the real data, and with a little adjustment on the CUDA implementation to improve more the performance we could come up with truly useful multiple linear regression model.

Conclusions

Overall Outcomes and Takeaways

In this project, we attempted to speed up the training process for Multiple Linear Regression using the GPU and using several High-Performance Computing techniques such as shared memory and loop unrolling. Our results demonstrated that the GPU implementation of the training process was not in fact faster than the CPU implementation. However, the reason for this was that the NumPy that was used in the CPU implementation is much more efficient at performing mathematical computations than a naive GPU approach, also, another thing to consider is that sklearn does not use gradient descent for their Linear regression algorithm, instead, it uses the closed-form solution for linear regression, which involves computing the inverse of the matrix product of the feature matrix and its transpose. This method is also known as Ordinary Least Squares (OLS) regression and it can be more computationally efficient for small to medium-sized datasets, but it can become computationally expensive for large datasets or when the number of features is very large. Finally, the GPU approach's execution time included the data transfer time between the CPU and the GPU might be another factor due to which the GPU implementation might perform poorly.

An interesting thing to note was the GPU implementation produced a model that was more accurate than the CPU implementation for certain dataset sizes. But in other dataset sizes, the accuracy of the GPU implementation was worse than the accuracy of the CPU version. Each dataset size might have an optimal learning rate and the number of iterations associated with it. Experimenting with those hyperparameters might be helpful in improving the accuracy of the model.

The optimizations such as shared memory and loop unrolling seemed to slow down the training process even more than the GPU naive implementation. There also did not seem to be any sort of GPU execution configuration where the optimizations would perform better than the naive version. This might be because the dataset did not have enough instances and features for the optimizations to have an effect on the training process.

Overall, a larger dataset may expose the limitations of the CPU implementation and the GPU implementation's parallelism might outweigh its communication latency. In addition, the impact of loop unrolling and shared memory might be more observable with a larger dataset. Currently, our GPU implementations perform poorly with the small dataset used for this project.

Future Work

As stated above, a possible improvement for this project would be to use a dataset that is larger with respect to the number of instances and the number of features. Another improvement would be to experiment with using pinned memory to reduce the CPU and GPU data transfer time. Also, we can experiment with more complex regression models or even other Machine Learning models such as CNNs since they might benefit more from a parallelized, GPU implementation than a Multiple Linear Regression model.

Note: All the code, dataset, and rest of the components for this project are in Github, please consider seeing them to understand better the project: (<https://github.com/khushaalkurswani/CSS535-Project>).

References

- Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E. (2014). *cudnn: Efficient primitives for deep learning*. arXiv. Retrieved from <https://arxiv.org/abs/1410.0759> doi: 10.48550/ARXIV.1410.0759
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction*. Springer. Retrieved from <https://hastie.su.domains/Papers/ESLII.pdf>
- Kulkarni, J. B., Sawant, A. A., & Inamdar, V. S. (2011). Database processing by linear regression on gpu using cuda. *2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies*. doi: 10.1109/icscn.2011.6024507
- Kurswani, K. (2022). *CSS535-Project*. Retrieved from <https://github.com/khushaalkurswani/CSS535-Project>
- Menon, A. (2018, Sep). *Linear regression using gradient descent*. Towards Data Science. Retrieved from <https://towardsdatascience.com/linear-regression-using-gradient-descent-97a6c8700931>
- Mitchell, R., & Frank, E. (2017). Accelerating the xgboost algorithm using gpu computing. *PeerJ Computer Science*, 3, e127.
- NRIGroupIndia. (2023). *Introduction to machine learning with python*. Retrieved from [https://www.nrigroupindia.com/e-book/Introduction%20to%20Machine%20Learning%20with%20Python%20\(%20PDFDrive.com%20\)-min.pdf](https://www.nrigroupindia.com/e-book/Introduction%20to%20Machine%20Learning%20with%20Python%20(%20PDFDrive.com%20)-min.pdf) (Accessed on March 15, 2023)
- NVIDIA. (n.d.). *What is regression?* Retrieved from <https://www.nvidia.com/en-us/glossary/data-science/linear-regression-logistic-regression/>