UNIVERSITÀ DI SIENA 1240

CPU/GPU Comparison of Multi Linear Regression Algorithm

Francesco Maccantelli
Duccio Meconcelli

High-Performace Computer Architecture Project
Professor Roberto Giorgi

26/05/2023

Abstract

This report presents the findings and analysis of a High-Performance Computer Architecture project focused on comparing the performance of linear regression algorithms on both CPU and GPU architectures. The initial approach involved converting a CPU-based linear regression algorithm to CUDA, the parallel computing platform and API model developed by NVIDIA. However, during the implementation, several challenges were encountered, hindering the successful conversion of the algorithm. As a result, a second algorithm was developed, leveraging the power of the GPU through the utilization of the gradient descent technique.

The results of our experiments demonstrate that the GPU outperforms the CPU in terms of speed and performance. Leveraging the parallel processing capabilities of the GPU significantly reduced the computation time for linear regression, especially when dealing with large datasets or complex models.

The parallelization of computations with CUDA allows the GPU to process vast amounts of data simultaneously, resulting in efficient utilization of computational resources.

All the code of this project can be found on this GitHub Page [1].

Contents

1 Introduction

The objective of the project was to explore the potential acceleration benefits offered by GPU computing for linear regression tasks.

Linear regression is a widely used statistical technique for modeling the relationship between a dependent variable and one or more independent variables. It assumes a linear relationship between the predictors and the target variable and aims to find the best-fitting line that minimizes the overall prediction error.

In this report, initially, the Least Squares Method was considered for computing regression. Later, we transitioned to Gradient Descent for reasons that will be explained further.

Traditional linear regression algorithms are typically implemented on CPUs, which may encounter limitations in handling large-scale datasets and complex computations. However, leveraging the computational power of Graphics Processing Units (GPUs) can offer several advantages for linear regression algorithms.

The parallel architecture of GPUs enables faster execution of matrix operations and calculations involved in linear regression. This speedup can significantly reduce the overall training time, making the algorithm more time-efficient and suitable for real-time or time-sensitive applications.

2 Challenges in Converting the Initial Algorithm

In this chapter, we discuss the difficulties encountered during the conversion of the initial CPU-based linear regression algorithm to a GPU implementation using CUDA. The original code provided by the repository [2] served as the starting point for the conversion process. However, we encountered specific challenges that impeded a smooth transition. Let's explore some of the major problems we faced:

2.1 Handling Shared Variables

The original code implemented a linear regression algorithm using the least squares method, which involved calculating the sum of squared residuals to determine the best-fit line. However, the reliance on shared variables, such as sumX, sumX2, sumXY, and sumY, presented difficulties in parallelizing the code for GPU implementation.

During the conversion process, it became apparent that the efficient utilization of parallel computing capabilities offered by the GPU architecture was hindered by the need for complex synchronization mechanisms. The shared variables, being updated at each iteration of the for loop, required careful coordination to ensure proper access and update by multiple threads.

The conversion of the code from CPU to GPU necessitated finding a balance between data parallelism and the correct synchronization of shared variables. Several challenges arose, such as race conditions, data hazards, and potential thread conflicts when accessing and modifying the shared variables simultaneously.

Considering the time constraints and the desire to allocate more resources to optimize and fine-tune the code, a decision was made to transition to a new algorithm that offered a more straightforward parallelization approach.

3 Transition to a New Algorithm

Due to the challenges encountered during the conversion of the initial code, as discussed in the previous chapter, it was decided to adopt a new algorithm for linear regression. This shift in approach aimed to overcome the difficulties posed by the original code and enable a more efficient and effective implementation on the GPU architecture.

By adopting the gradient descent algorithm as the foundation for the new approach, the reliance on shared variables was significantly reduced. The gradient descent method calculates the regression coefficients iteratively, adjusting them gradually. This technique simplifies the implementation and eliminates the complex synchronization requirements of the original least squares method.

3.1 The New Algorithm

The new Algorithm is based on a gradient descent method that allows us to compute the coefficients of the regression, this algorithm was implemented at first for CPU with the function `linear_regression_cpu()`. The algorithm aims to find the optimal regression coefficients for a linear relationship between the input variables, which can be just one (x) or a vector of features ($x1$,$x2$,$x3$,...), and a target variable (y). The algorithm iteratively updates the coefficients by calculating the prediction error and adjusting them based on the error.

The algorithm was initially designed with a single input variable; however, it was later extended to support multiple input variables, allowing for a more comprehensive analysis of complex data sets.

The following algorithm snippet represents the core of the new algorithm. It consists of a main for-loop that processes each input data point. Within each iteration of the loop, the algorithm computes the predicted output value (`y_pred`) based on the current intercept and slope. Subsequently, it calculates the error by comparing the predicted value to the actual value of y. This error is then accumulated in the variable $errors[0]$.

```
while(j_error > MAX_J_ERROR) {
...
    for (int i = 0; i < INPUT_SIZE; ++i) {
            // Predict output based on current
            float y_pred = intercept + slope * x[i];
            // Calculate J for this specific index
            errors[0] += 0.5f * pow((y[i] - y_pred), 2);
            // Calculate intercept error
            errors[1] += -(y[i] - y_pred);
            // Calculate slope error for this index
            errors[2] += -(y[i] - y_pred)*x[i];
        }
    j_error = errors[0] / INPUT_SIZE;
...
    // Update intercept and slope based on errors
    intercept_new = intercept - LEARNING_RATE * errors[1];
    slope_new = slope - LEARNING_RATE * errors[2];
...
```

Listing 1: Pseudo code of the gradient descent algorithm

After the accumulation of errors, the algorithm proceeds to calculate the intercept error and slope error. These errors will be used later in the algorithm to adjust the intercept and slope values, thus refining the model predictions. This adjustment step aims to optimize the performance of the model based on the calculated errors.

This process is repeated until the `j_error` (MSE : Mean Square Error), is not below a fixed `MAX_J_ERROR`, this allows us to set the accuracy of the algorithm.

3.2 Multi Linear Regresion

In order to obtain a better performance improvement from converting this code to its own GPU version, we decide to pass from a single linear regression, formula (1) to a multi-linear regression formula (2).

$$y = intercept + slope\, x \tag{1}$$

$$y = intercept + slope_1\, x_1 + slope_2\, x_2 + slope_3\, x_3 \tag{2}$$

In order to do it the main changes are, the `y_pred` computation, and the computation of the errors for each slope. Then we update each slope using its own error. In our case, we decide to have 3 independent variables and 1 dependent.

```
...
    y_pred = intercept + slope1*x1[i] + slope2*x2[i] + slope3*
   x3[I];
...
    errors[2] += -(y[i] - y_pred)*x1[i];
    errors[3] += -(y[i] - y_pred)*x2[i];
    errors[4] += -(y[i] - y_pred)*x3[I];
...
    intercept_new = intercept - LEARNING_RATE * errors[1];
    slope_new_1 = slope1 - LEARNING_RATE * errors[2];
    slope_new_2 = slope2 - LEARNING_RATE * errors[3];
    slope_new_3 = slope3 - LEARNING_RATE * errors[4];
...
```

Listing 2: Changes to obtain multi-linear regression

3.3 CUDA Version

Now we have converted this algorithm in order to use CUDA architecture. This conversion has unlocked its parallelization capabilities, allowing for more efficient and faster computations.

In order to convert our algorithm to its own CUDA version we create a '`linear_regression4d.cuh`' file where we implemented the CUDA kernel that takes input arrays (`d_x1, d_x2, d_x3, d_y`) representing the independent variables and dependent variable, along with the model parameters (`d_intercept, d_slope1, d_slope2, d_slope3`). The size of the input data (`in_size`) is also provided.

To leverage parallelization, we utilized CUDA's thread block architecture. Each thread within a block is responsible for processing a specific data point. By defining a shared memory array, "errors," of size `ERROR_DIMENSIONS` (set to 5), we maintained the collective sum of squared errors for each block.

Within each thread, the predicted value, `y_pred`, is calculated based on the current bias and intercept values. Subsequently, various error terms are computed using `atomicAdd` operations to ensure thread safety and avoid data races. These errors include the overall squared error, bias error, and intercept errors for each independent variable.

To synchronize the threads and ensure consistency, `__syncthreads()` is called before updating the output values. Only the first thread in each block (`threadIdx.x == 0`) writes the accumulated errors to the device memory, specifically to the `d_results` array. The results are stored in a sequential manner, with each block contributing five consecutive entries, and then in the main this information is decompressed in order to get back all the data stored in this array

Finally, the shared memory array, errors, is reset to zero to prepare for the next iteration of the algorithm. By harnessing the parallel processing capabilities of CUDA, this implementation significantly accelerates the linear regression computation, particularly when handling large datasets and complex regression problems.

4 Dataset

In this study experiments were carried out considering 2 datasets, one taken externally and one generated on the basis of our needs.

4.1 The Mock Dataset

In order to test our algorithm we decide to use an external dataset bigger enough to be able to see the improvements sprung from the conversion to the GPU version. We used the Mock [3] dataset, a big dataset that contains 160000 entries, about salary compared by skill level of people.

The dataset is composed as follows:

Name	Points	Skill	Assists	Salary
Eltron	8.6	low	4.7	598613.62
Tarrah	16	medium	1.7	6037179
...	...	...	...	...

This dataset cannot be directly used for our purpose because it contains some irrelevant data and some data that cannot be utilized in its current form. The 'Name' field is considered irrelevant, so we can drop it from the dataset. However, the 'Skill' field can be relevant in reconstructing the salary, so we need to convert it into a usable format. We performed the following conversion:

Skill	Converrted Value
Low	1
Medium	2
High	3

After that, we also decided to normalize the dataset by applying the formula (3) for each column of the dataset.

$$norm\_value = \frac{(value - min[column])}{(max - min[column])} \tag{3}$$

After all the changes we obtain a dataset that can be used to make a multi-liner regression.

X_1	X_2	X_3	Y
0.223	0	0.308	0.088
0.441	0.5	0.058	0.729
...	...	...	...

4.2 The Generative Dataset

To test the weight that the size of the dataset can have, we used a script to generate datasets of various sizes (100, 1000, 10k, 100k, 500k elements) with 3 input features according to a linear uniform distribution with the addition of noise.

5 Results and Performance

In order to measure the performance of the algorithm we decided to use two different computer configurations. The first one (from now on *laptop*)

- CPU: AMD Ryzen 5 5600H (6 core, 12 threads, 19.3 MB Cache, 3 MB L2, 3.30GHz)
- RAM: Crucial 32GB DDR4 (3200 MHz, dual channel)
- GPU: Nvidia RTX 3060 6 GB GDDR6 (laptop)
- OS: Ubuntu 22.04.1 LTS

The second one (from now on *desktop*)

- CPU: Intel i5 12400F (6 core, 12 threads, 18 MB Cache, 7.5 MB L2, 2.50 GHz)
- RAM: Corsair 16GB DDR4 (3600MHz, dual channel)
- GPU: Asus Dual NVIDIA RTX 3060 OC 12GB GDDR6 (Desktop)
- OS: Ubuntu 22.04.1 LTS

5.1 Measure performance

The performance of the algorithm was assessed by measuring the execution time of the CPU and GPU implementations. The `std::chrono` library was used to measure the elapsed time of several crucial parts. This enabled determining the total execution time of various components of the algorithm and the total elapsed time.

For the CPU version, we decided to save two different times, as reported here:

- `elapsed_cpu_run_time` : Time of execution of the while loop
- `elapsed_cpu_allocate` : Time for allocation

Meanwhile, for the GPU we decide to save more phases, this allows us to see the most time-consuming parts of the GPU implementation:

- `elapsed_gpu_allocate` : Time for allocation with `cudaMalloc` in the initialization.
- `elapsed_gpu_copy` : Time for the initial copy from Host to Device.
- `total_gpu_allocate_do_micro` : Time for Allocation in the do-loop.

- `total_gpu_copy_toDevice_do_micro` : Time for copy data to Device in the do-loop.

- `total_gpu_kernel_do_micro` : Time for the execution of the kernel.

- `total_gpu_copy_toHost_do_micro` : Time for copy data to Host in the do-loop.

- `total_gpu_get_results_update_do_micro` : Time to get results and update in the do-loop.

- `total_gpu_free_do_micro` : Time for the `freeCuda` in the do-cycle.

- `total_gpu_free` : Time for the `freeCuda` at the end of the algorithm.

5.2 Changing dataset size

The size of the dataset is a crucial parameter for achieving significant improvements when adopting the GPU version of the algorithm. It directly impacts the number of cycles the algorithm needs to complete in order to reach the desired `j_error` below the `MAX_J_ERROR` threshold.

To determine the optimal dataset size for obtaining the best improvement with the GPU version of the algorithm, we conducted several tests. We generated multiple datasets using the `gen_data.py` script, each consisting of varying numbers of samples [100, 1000, 10k, 100k, 500k]. We then tested these datasets and compared their execution times in Figures 3 and 4. The chosen parameters for these tests were: `th`:128 (number of threads), `max_j`:0.01, `lr`:10^{-}6 (learning rate), and a total of 20 executions.

Observing the results, we can see that the execution time of the CPU remains relatively constant regardless of the dataset size. On the other hand, the GPU experiences significant variations in execution time as the dataset size changes.

Furthermore, we identified that the most time-consuming tasks for the GPU version are the data transfer operations between the host and the device. By increasing the dataset size, we can reduce the number of iterations required for these data transfers, ultimately decreasing the overall execution time.

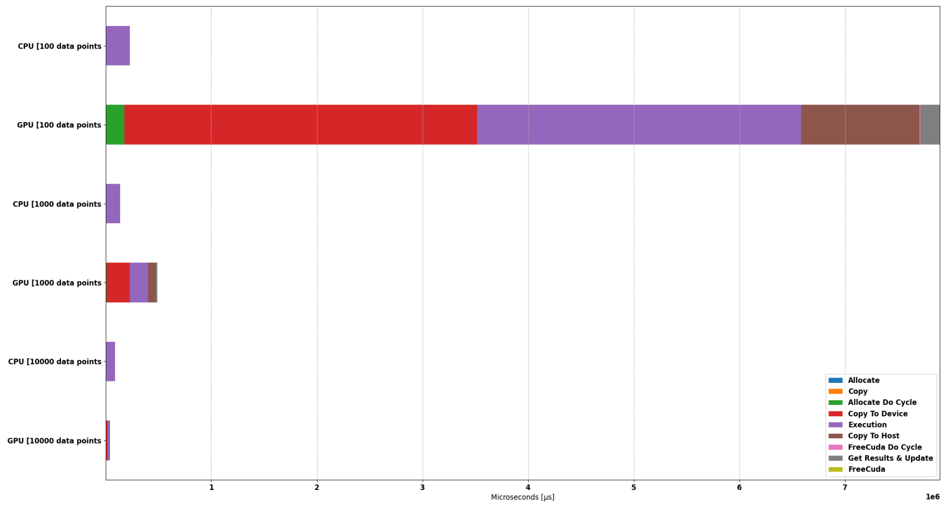

Figure 2: Time execution on *laptop* for datasets [100 - 1k - 10k]

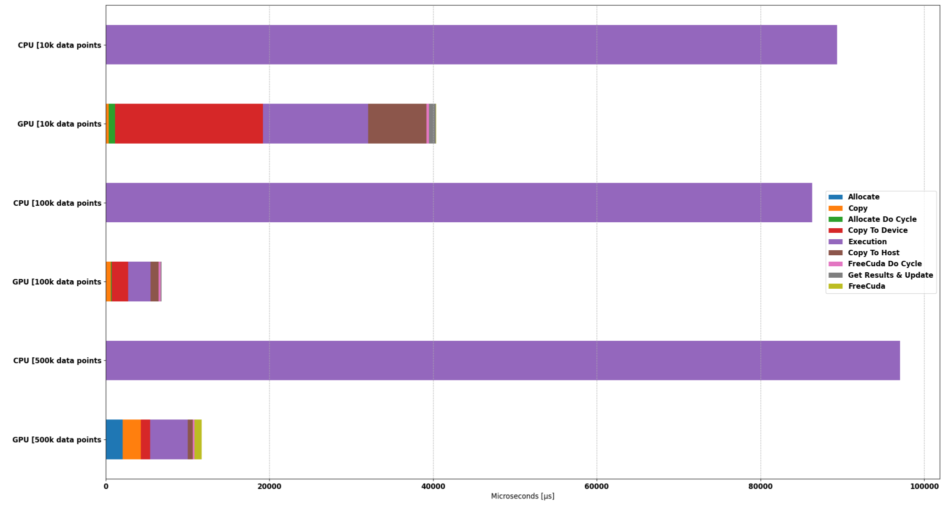

Figure 3: Time execution on *laptop* for datasets [10k - 100k - 500k]

5.3 Changing number of Threads

One important hyperparameter to consider is the number of threads to allocate when launching the GPU kernel. We made several tests to determine the optimal choice for this parameter. When executing the kernel, we also need to specify the number of blocks to use, which is determined as follows:

$$numBlocks = \frac{(INPUT\_SIZE + NUM\_OF\_THREADS - 1)}{NUM\_OF\_THREADS}$$

After our tests, we discovered that the best performance is in the range of [128 - 512] threads as we can see in the following image (figure 2). These data were performed on *laptop* with `MAX_J_ERROR = ` 0.01. The time reported is the mean after 20 executions. The CPU execution time for every test session is almost equal so in the graph is reported only for the first test (1 thread).

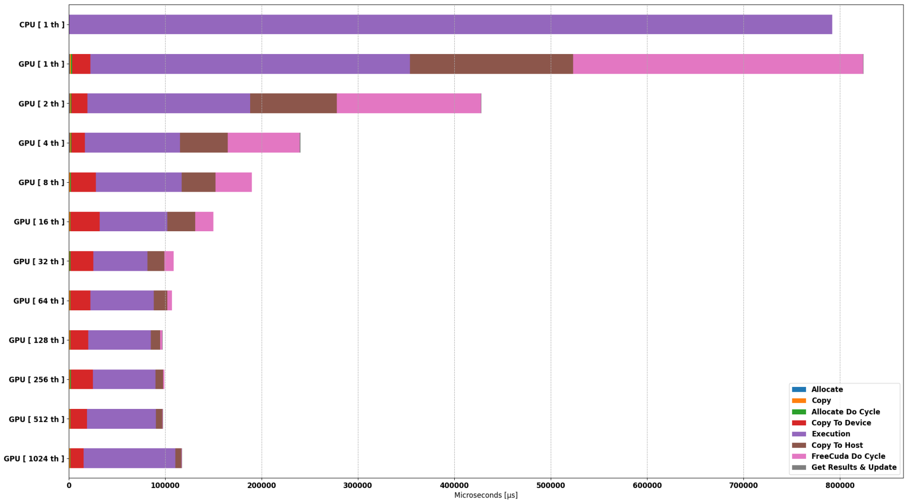

Figure 4: CPU/GPU (*laptop*) Execution time based on threads

5.4 Accuracy

In order to determine the actual accuracy achieved by the regression, we decided to create a Python script '`test.py`' that allows us to compute the accuracy of the predicted regression. In order to do it we divided the *Mockdataset*[3] into two parts: the first one (90%) for training, and the second one (10%) for testing.

The algorithm achieves different levels of accuracy based on the value of `MAX_J_ERROR`. A summary table detailing these accuracies is provided below.

j_error	Accuracy (%)
0.1	67.1
0.05	75.4
0.01	88.8
0.005	91.5
0.004	92.22
0.00385	92.44

5.5 Comparison *laptop* - *desktop*

We decided to also compare the performance of the two builds *laptop* and *desktop*. To make this comparison, we ran the same tests on each machine and plotted the average times of 20 executions.

We decided to compare the execution times of the two builds by varying the number of threads. Our goal was to determine if there were any advantages to using one configuration over the other. We conducted two different test runs with the following parameters (tables 1 and 2). The main difference between the two configurations was the choice of `MAX_J_ERROR`, which was 10 times greater in the second session. This allowed the algorithm in the second session to run faster, about 1000 times, while still achieving a decent level of accuracy (78%). On the other hand, by using a lower `MAX_J_ERROR` we can reach a higher accuracy at the expense of longer run times.

As we can see the images 5 and 6, the execution times of the algorithm for the two configurations, *laptop* and *desktop* are almost equal, we can notice just that the CPU execution on the *desktop* is almost twice faster as the *laptop*, meanwhile, the GPU execution is practically equal. Even if the laptop has a higher base clock frequency than the desktop, the difference in CPU execution time is probably due to the different target of the device: a laptop, with a lower TDP (45W); and a desktop with a higher TDP (60W). As far as the GPU is concerned, the differences on paper are marginal and mainly concern the memory of the graphics card which in these experiments is not put under strain.

Table 1: Parameters session 1

Session 1	
Number of threads	[32 - 1024]
Max J error	0.00386
Learning rate	0.000001
Number of repetitions	20

Table 2: Parameters session 2

Session 2	
Number of threads	[32 - 1024]
Max J error	0.0386
Learning rate	0.000001
Number of repetitions	20

Max j_error	Accuracy [%]	Mean time CPU [μs]	Mean time GPU [μs]
0.00386	92.4	1,8e07	1.4e06
0.0386	78.0	4,2e04	4,3e03

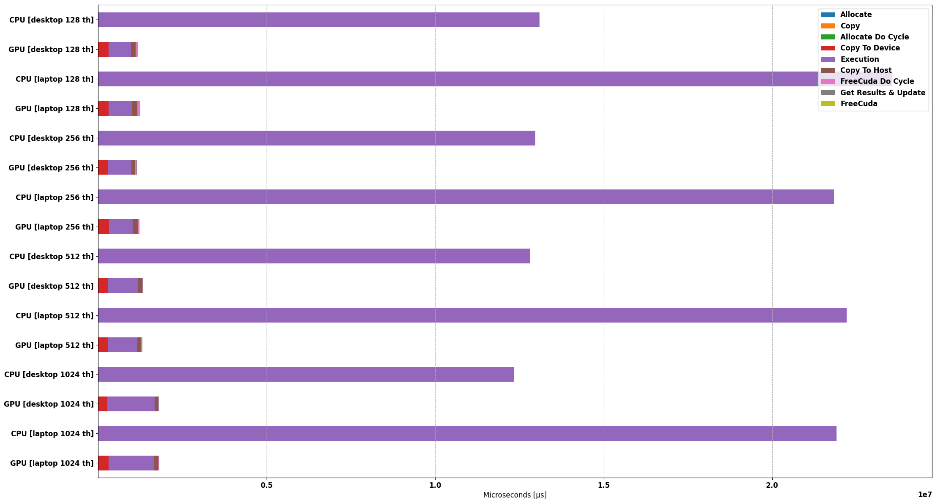

Figure 5: Time execution of session 1

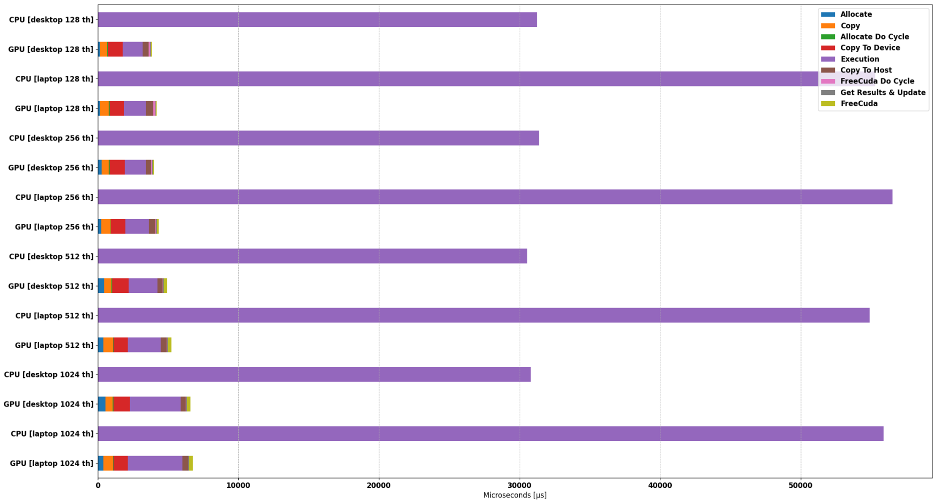

Figure 6: Time execution of session 1

6 Conclusion

In conclusion, we can say that we reached our purpose, by converting the CPU-based algorithm to a GPU-based algorithm with a reduction of time execution from 9 to 19 times depending on the configuration used to run the algorithm and the parameters chosen.

We also notice that if we want to increase the accuracy of the regression by about 14% [from 78% to 92%] the time needed increases about 1000 times, independently by the used platform [CPU - GPU] but obviously the GPU remains the faster device (about 10 times).

It is important to note that the specific speed gains observed in this study may vary depending on the hardware configuration and dataset characteristics. However, the fundamental principle remains that the parallel processing capabilities of the GPU offer significant advantages over the CPU for linear regression tasks.

In conclusion, our investigation clearly indicates that the implementation of linear regression using the CUDA library and GPU acceleration can yield substantial improvements in terms of speed and performance. The ability to harness the parallel computing power of the GPU opens up new opportunities for efficient and scalable data analysis and machine learning.

References

[1] D. Meconcelli and F. Maccantelli, “Hpca-linear-regression,” GitHub, 05 2023. [Online]. Available: https://github.com/Duccioo/HPCA-Linear-Regression

[2] M. Technologies, “Linear regression,” GitHub, 06 2017. [Online]. Available: https://github.com/maxeler/Linear-Regression/blob/master/ORIG/LinearRegressionCPUOnly.c

[3] L. Ling, “Mock dataset,” GitHub, 01 2023. [Online]. Available: https://github.com/lianaling/dspc-cuda/blob/main/dspc-cuda/mock.csv