Osama Shamout CSC 447
Gtihub link: https://github.com/OsamaShamout/ParallelProject2023

Final Project: Single Layer Perceptron

Introduction

Artificial neural networks are computational models or algorithms inspired by the human brain's structure. They excel at learning from data, and as information processing systems, they have achieved significant success in various fields ranging from marketing to research. Among the many types of artificial neural networks, single-perceptron and multilayer algorithms hold a prominent position. A perceptron is a mathematical model based on a simplified representation of a single biological neuron's structure and function. It is a branch of Machine Learning focused on understanding the principles of binary classifiers.

Literature Review

Improvement of parallelization efficiency of batch pattern BP training algorithm using Open MPI: Designing an optimal parallel algorithm is necessary but not sufficient for achieving high parallelization efficiency. To develop efficient parallel algorithms for neural network training, it is crucial to implement the technical features correctly, minimize the number of MPI collective function calls, and use the advanced properties of the latest MPI releases to improve the performance of collective communications.

A Survey on Parallelization of Neural Network using MPI and Open MP:

In many neural network, special implementations have to be considered for the as selecting the many implementations can no longer be generalizable after certain amount of nodes (PPT achieved 10.8x speedup but diminishes significantly after 8 nodes). Therefore, parallelization approach has to be based on the neural network use and hardware platform. Also, the survey indicates that mixing MPI and Open MP might result in an unsuitable implementation in HPC stemming for example from MPI communication issues, and even different MPI interface calls can result in improvements over different use cases (MPI_Allreduce()).

Open MPI

For the Message Passing Interface (MPI). The algorithm works as follows: It divides the dataset among processes and trains a perceptron model on each portion, then averages the weights across all processes to obtain the final model.

The parallelism in this code is primarily at the data-level. The main constructs used for parallel processing are:

- MPI_Init: Initializes the MPI execution environment.
- MPI_Comm_rank: Determines the rank of the calling process in the communicator.
- MPI_Comm_size: Determines the number of processes in the communicator.
- MPI_Wtime: Returns an elapsed time on the calling processor.
- MPI_Bcast: Broadcasts a message from the master process to all other processes in the communicator.
- MPI_Scatter: Distributes data from one process to all others in a group.
- MPI_Reduce: Performs a collective reduction operation on values from all processes and stores the result on the master process.
- MPI_Gather: Gathers values from a group of processes and stores them in an array on the master process.
- MPI_Barrier: Synchronizes all processes in a group, ensuring that all processes have reached the same point in the code.
- MPI_Finalize: Terminates the MPI execution environment.

The data-level parallelism is achieved through several steps:

- Broadcasting the number of data points to all processes using MPI_Bcast.
- Dividing the dataset among processes and assigning equal portions of the dataset to each process.
- Scattering the dataset to all processes using MPI_Scatter.

Each process then works on its portion of the data concurrently, improving overall performance. The algorithm also uses MPI_Reduce to aggregate the correct predictions and total predictions across all processes. It collects the local_correct_predictions and local_total_predictions from all processes and sums them up on the master process (rank 0).

After the training, the code uses MPI_Gather to collect the weights from all processes and store them in an array on the master process (rank 0). This is useful for calculating the average weights of the model across all processes.

Finally, MPI_Barrier is used to synchronize all processes before printing the results and finalizing the MPI environment.

OpenMP

In this implementation of the Perceptron Training Algorithm, several levels of parallelism are exploited. The primary level of parallelism is loop-level parallelism, which is achieved by parallelizing the training loop that iterates over each data point in the training set. This allows for simultaneous processing of multiple data points, increasing the efficiency of the training process. Though, it is worth to mention that data-level parallelism also exists yet it is not explicitly performed through a directive and rather is performed through the #pragme omp for directive. However, it is important to note that the data-level parallelism in this implementation is limited by the shared-memory nature of OpenMP. As a result, it might not be as scalable as other parallel programming models, such as MPI or CUDA

The parallel technique used in this implementation is OpenMP. OpenMP is employed to parallelize the training loop using the #pragma omp parallel for directive. This directive specifies that the following loop should be executed in parallel by multiple threads, with each thread handling a portion of the loop iterations. In this case, the loop iterates over the training data points and updates the weights of the perceptron based on the calculated error.

The OpenMP constructs used in this program include the omp_get_wtime() function and the #pragma omp parallel for directive. The omp_get_wtime() function is utilized to measure the elapsed time during the parallel execution of the training loop, providing an indication of the performance improvement achieved through parallelization. The #pragma omp parallel for directive, as previously mentioned, is used to parallelize the training loop and distribute its iterations among multiple threads.

Cuda C

In the Cuda C implementation, multiple levels of parallelism are exploited to improve performance which are the data-level, thread-level, and block-level. At the data level, each thread in the kernel processes an individual data point, allowing for concurrent execution across data points. Also, threads are organized into blocks with a fixed number of threads (in our case, 256), facilitating efficient execution on modern GPUs optimized for thread-level parallelism. Finally, blocks are organized into a grid, with the gridSize calculated as (num_lines + blockSize - 1) / blockSize to ensure that there are enough blocks to cover all data points, further improving GPU resource utilization.

The implementation employs various parallel techniques, such as executing the train function as a CUDA kernel on the GPU, processing multiple data points simultaneously, and using the parallel reduction technique within the sumWeightedInputs function to efficiently compute the sum of elements in an array. Atomic operations, like atomicAdd, are used to update weights while avoiding race conditions, ensuring concurrent updates to shared memory locations are performed in a mutually exclusive manner.

Some Cuda C constructs used in the parallel program include the CUDA kernel definition and invocation of the train function as a global function, thread and block indexing calculations using blockIdx.x, blockDim.x, and threadIdx.x, shared memory declaration and synchronization using the shared keyword and __syncthreads() function, and atomic operations like atomicAdd for thread-safe weight updates.

Results

The parallel implementation resulted in times of 0.030254, 0.068094, and 0.090869. For the dataset with number 100, 200, and 300, respectively.
Of course Speed Up Factor = Seq Exec / Parallelized and,
Efficiency = Speed Up Factor/# of threads or processes.

OpenMPI

Number of Threads	Dataset Size	Total Time in seconds	Speed Up	Efficiency
2	100	0.02139	1.41439925	0.70719963
2	200	0.013188	2.2940552	1.1470276
2	300	0.01656	5.48725845	2.74362923
4	100	0.013	2.32723077	0.58180769
4	200	0.017	4.00552941	1.00138235
4	300	0.03	3.02896667	0.75724167
8	100	0.01245	2.43004016	0.30375502
8	200	0.01872	3.6375	0.4546875
8	300	0.021383	4.2495908	0.53119885

OpenMP

Number of Threads	Dataset Size	Time in seconds	Speed Up Factor	Efficiency
2	100	0.0102	2.966078431	1.48303922
2	200	0.0195	3.492	1.746
2	300	0.0278	3.268669065	1.63433453
4	100	0.0081	3.735061728	0.93376543
4	200	0.0184	3.70076087	0.92519022
4	300	0.0259	3.508455598	0.8771139
8	100	0.0047	6.437021277	0.80462766
8	200	0.0136	5.006911765	0.62586397
8	300	0.0184	4.938532609	0.61731658

Cuda C

For Cuda C, I was not able to measure the time as the kernel ran so fast the value was beyond four decimal points. This could indeed be the case as the datapoints are only 100, 200, 300. Thus, the total time would be the sequential execution time of the CPU for the input, printing, etc, rather than the computationally intensive operation. Though, if we only assume that the value is 1 after the four decimal points, 0.00001, then truly, the parallelization for this problem and for these amounts of tested dataset is very significant. However, we face an issue when trying to find different GPU cores to perform as we are limited by Google Colabs resources. Thus, we have only tried it over the T4 with 2560 cores.

Number of GPU Cores	Dataset Size	Time in seconds	Speed Up Factor	Efficiency
2560	100	0.00001	3025.4	1.18179688
2560	200	0.00001	6809.4	2.65992188
2560	300	0.00001	9086.9	3.54957031

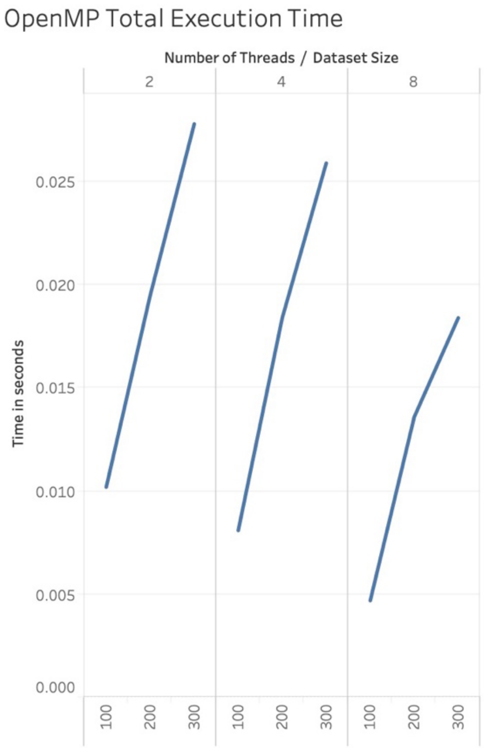
Discussion

First, we must comment about the efficiency number being higher than 1. While this might look abnormal, that isn't the case in the experiment at hand. For us, the type of the problem has an inherently parallel nature that allows for efficient distribution of the workload among the threads, resulting in a higher-than-expected speedup and thus reflecting back on efficiency. Also, it is evident that the implementation of Cuda C demonstrates a significant advantage in terms of speedup and efficiency compared to OpenMPI and OpenMP for the given problem and dataset sizes. This highlights the importance of selecting the right parallel processing approach based on the problem and available hardware resources. In fact, it came to no surprise that such an implementation would dominate the others. For OpenMP and OpenMPI, the efficiency values of OpenMP are generally higher than those of OpenMPI, especially for the 2-thread case. As the number of threads increases to 4 and 8, the efficiency decreases, although it remains relatively higher than the OpenMPI implementation. However, this suggests that the shared-memory parallelism approach might face challenges in load balancing and synchronization, leading to less effective performance on larger numbers of threads. For OpenMPI, for 2 processes, the efficiency is above 1 for dataset sizes 200 and 300, indicating good parallel performance. However, for 4 and 8 processes, the efficiency values are below 1, suggesting less efficient parallelization as the number of processes increases compared to that of OpenMP.

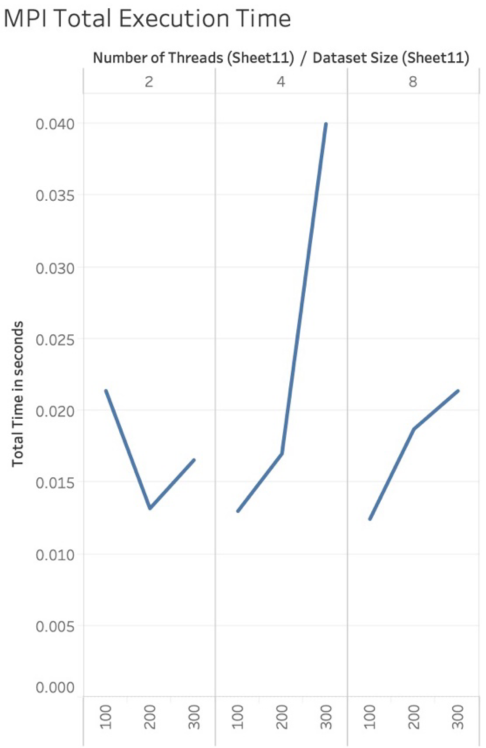
Regarding scalability, for OpenMPI and OpenMP, as the number of threads increases, we observe a general increase in speed-up factor and efficiency, although not entirely linear. This indicates that the parallelization of the problem provides some level of scalability. However, the efficiency tends to decrease as the number of threads increases, suggesting that there might be a limit to the scalability provided by these implementations. In the case of Cuda C, the results show an impressive speed-up factor and efficiency, even though the experiment was conducted with a fixed number of GPU cores (2560). This suggests that the Cuda C implementation exhibits strong scalability for this problem, especially when considering the small dataset sizes (100, 200, and 300). However, it's important to note that the scalability of the Cuda C implementation might also be influenced by factors such as GPU architecture, available memory, and problem size which have made it ideal in our case. Also, it is important to mention that tiling could be another approach for this problem which could be explored later on.

The limitations faced in this project primarily stemmed from the necessity to use different devices for testing, an approach that is not ideal in real-world experimentation. However, the constraint of available resources made this necessary. Future work should prioritize ensuring consistency between devices and runtimes in order to eliminate confounding factors. Additionally, gathering a larger number of data points from the code would help to identify and eliminate outliers, leading to a more reliable dataset for graphical representation and analysis. The current dataset includes outliers that distort the plots. One way to address this limitation is to calculate the average for each run, considering the number of threads and dataset size. Unfortunately, due to time constraints, this approach was not feasible for the current project.

Graphs

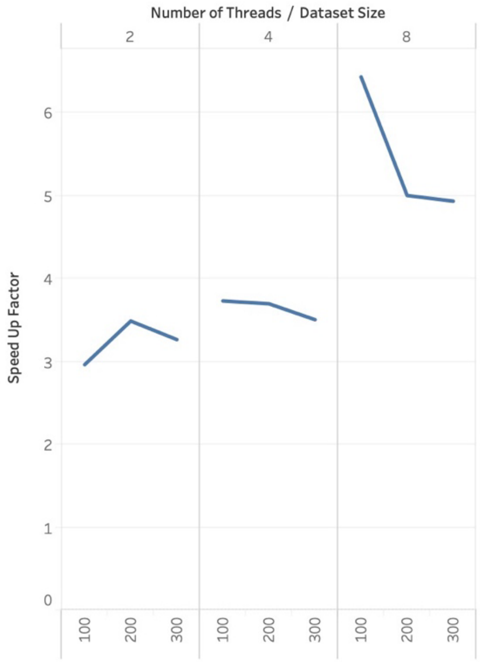


OpenMP Total Execution Time

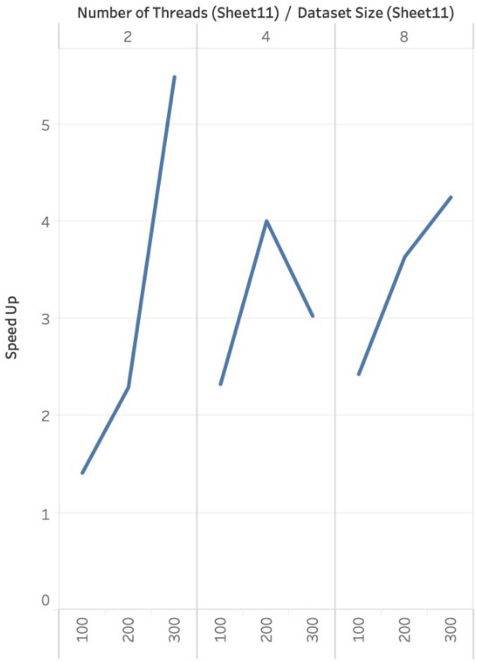


MPI Total Execution Time

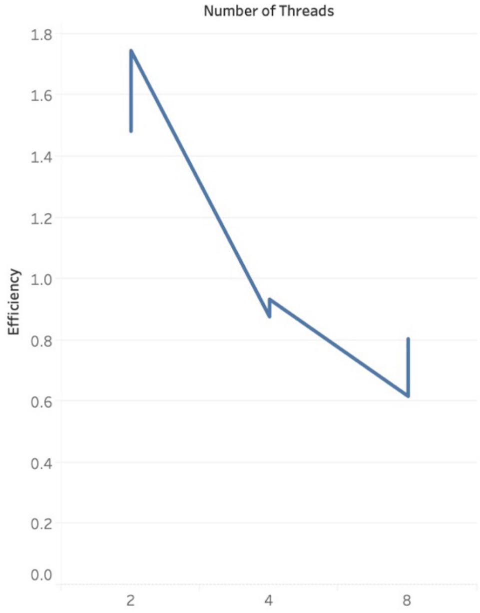
Open MP Speed Up Factors



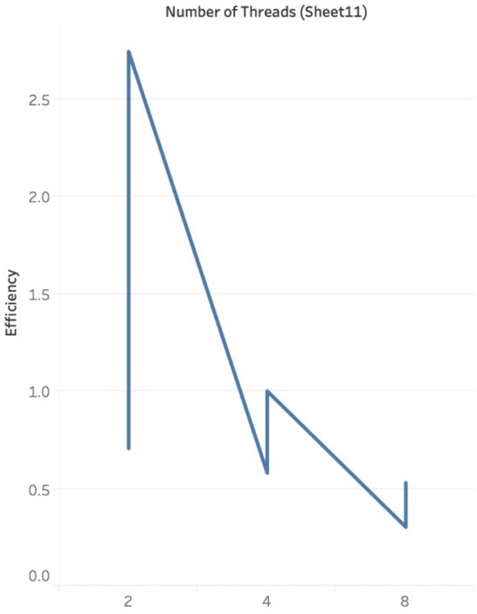
MPI Speed Up Factor



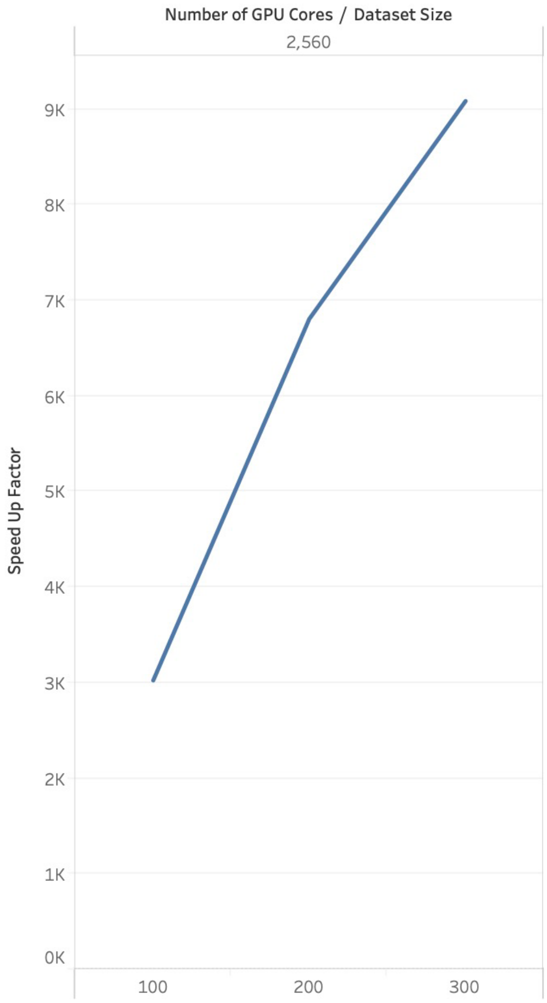
OpenMP Effeciency



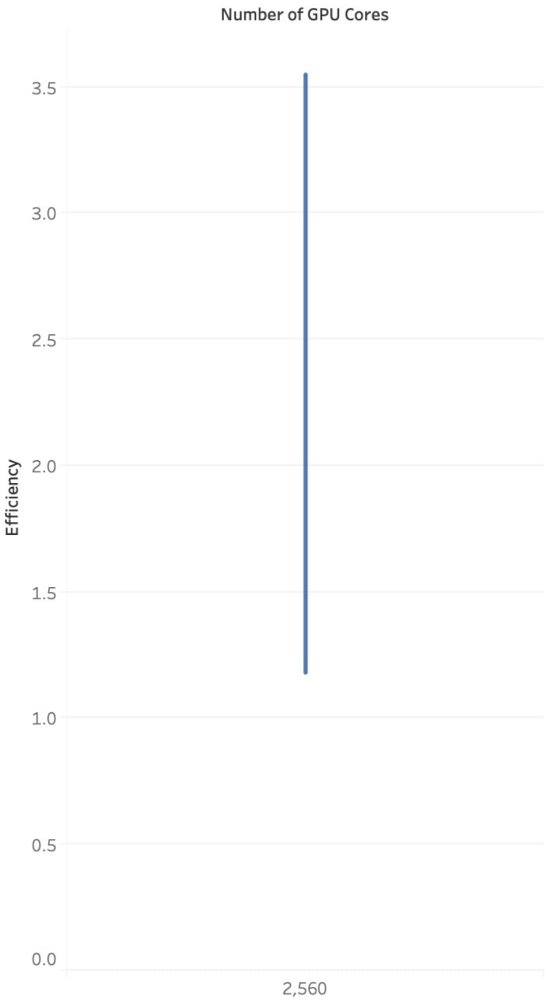
MPI Effeciency



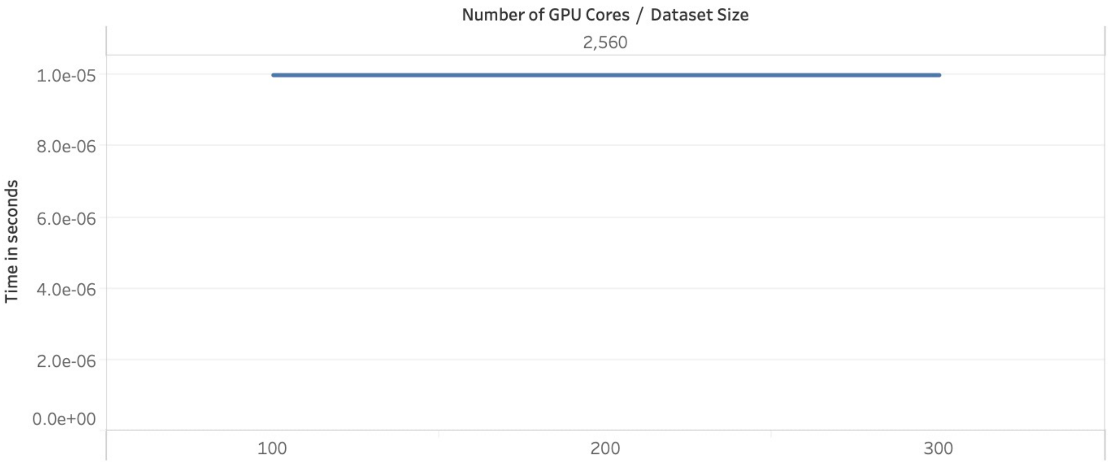
Cuda C Speed Up Factor



Cuda C Effeciency



Cuda C Total Execution Time


Number of GPU Cores / Dataset Size
Time in seconds
1.0e-05
8.0e-06
6.0e-06
4.0e-06
2.0e-06
0.0e+00