Indian Institute of Technology Indore

Parallel Computing Project Report

Parallel Training of a Back-Propagation Neural Network Using CUDA

Authors:
GVS Akhil
Kumar Abhinav
Mahesh Kumar

Supervisor:
Dr. Kapil Ahuja

December 4, 2018

Contents

Abstract

In this project, we speed up the training phase of Multilayer Neural Networks by parallelising the matrix operations involved in the back-propagation step using NVIDIA's CUDA model.

1 Introduction

Training an artificial neural network is time-consuming due to the large number of weight updates required to reach an optimal performance.

The cost of implementing computational tractable algorithms for training/testing with large size inputs is extremely high. The bottle-neck in the training of a neural network is the forward/backward propagation step where matrix multiplication is involved. The complexity of matrix multiplication is $O(n^3)$.

2 Approach

The key to lower execution time is to speed up the layerwise parameter calculations/updates during propagation process. Since the propagation process can be modeled in matrix forms, we will speed up the matrix operations with a parallel scheme focusing on multiplications.

We use the GPU to perform the operations rather than the traditional CPU. This is because a CPU consists of a few cores (up to 24) optimized for sequential serial processing. It is designed to maximize the performance of a single task within a job. On the other hand, a GPU uses thousands of smaller and more efficient cores for a massively parallel architecture aimed at handling multiple functions at the same time.

Modern GPUs provide superior processing power, memory bandwidth and efficiency over their CPU counterparts. They are 50–100 times faster in tasks that require multiple parallel processes.

We use CUDA (Compute Unified Device Architecture), a parallel computing platform and an API model that was developed by Nvidia. Using

CUDA, we utilize the power of Nvidia GPUs to perform parallel computing operations.

3 Matrix Multiplication Algorithm

3.1 Sequential

Algorithm 1 Sequential Matrix Multiplication Algorithm

Require: Matrices A and B
1: Let C be a new matrix of the appropriate size
2: **for** i from $1 \rightarrow n$ **do**
3: **for** j from $1 \rightarrow p$ **do**
4: Let $sum = 0$
5: **for** k from $1 \rightarrow m$ **do**
6: Set $sum \leftarrow sum + A_{ik} \times B_{kj}$
7: **end for**
8: Set $C_{ij} \leftarrow sum$
9: **end for**
10: **end for**
11: **return** C

3.2 Parallel

In the parallel algorithm, we use NVIDIA's CUDA API to perform matrix multiplications using the GPU instead of the CPU.

3.2.1 Step 1: Memory Allocation

The matrix data initially lies in the host memory. To allow the processors in the GPU to perform operations on this data, we need to move it to the device(GPU) memory.
To do this, we can use the following command:

```
cudaMalloc((void **) &host_object, mem_size);
```

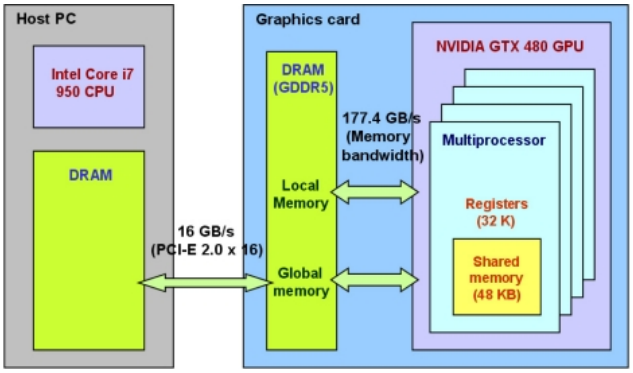


Figure 1: Host and Device Memory

where host_object is a pointer and mem_size is the space which needs to be allocated. However, to avoid constant copying from host to device memory and vice versa, we use the concept of **Unified Memory**.

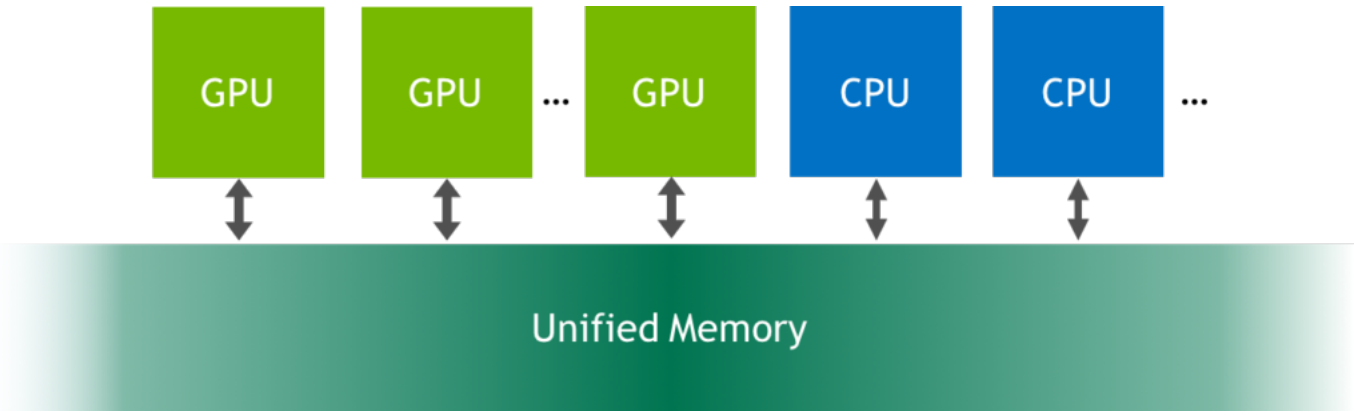


Figure 2: Unified Memory

In Unified Memory, both CPU and GPU are able to acess the same memory location. This sigficantly reduces the overhead.

3.2.2 Step 2: Matrix Multiplication

We use the same algorithm mentioned in the sequential section but here, we adapt it to allow multiple processors to work on the completion of the algorithm. NVIDIA's GPU's follows a Grid, Block and Thread model like in

Figure 3.

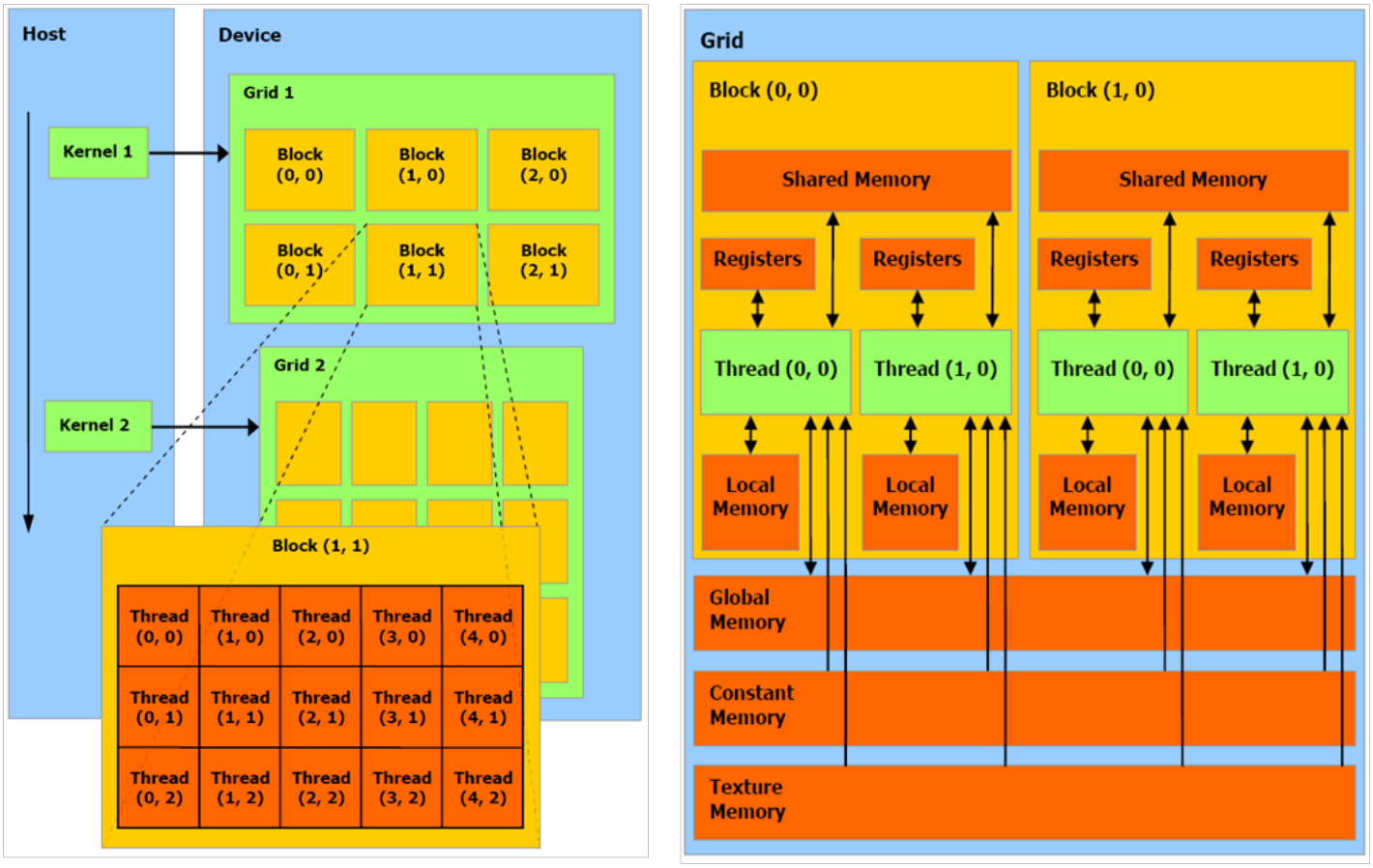


Figure 3: Grid, Block and Thread Model

The following code block demonstrate matrix multiplication using the grid/blocks/threads.

```
#include <math.h>
#include <iostream>
#include "cuda_runtime.h"
#include "kernel.h"
#include <stdlib.h>
```

```
using namespace std;

__global__ void matrixMultiplicationKernel(float* A, float* B, float* C, int N) {

    int ROW = blockIdx.y*blockDim.y+threadIdx.y;
    int COL = blockIdx.x*blockDim.x+threadIdx.x;

    float tmpSum = 0;

    if (ROW < N && COL < N) {
        // each thread computes one element of the block sub-matrix
        for (int i = 0; i < N; i++) {
            tmpSum += A[ROW * N + i] * B[i * N + COL];
        }
    }
    C[ROW * N + COL] = tmpSum;
}


void matrixMultiplication(float *A, float *B, float *C, int N){

    // declare the number of blocks per grid and the number of threads per block
    // use 1 to 512 threads per block
    dim3 threadsPerBlock(N, N);
    dim3 blocksPerGrid(1, 1);
        if (N*N > 512){
            threadsPerBlock.x = 512;
            threadsPerBlock.y = 512;
            blocksPerGrid.x = ceil(double(N)/double(threadsPerBlock.x));
            blocksPerGrid.y = ceil(double(N)/double(threadsPerBlock.y));
        }

    matrixMultiplicationKernel<<<blocksPerGrid,threadsPerBlock>>>(A, B, C, N);
}
```

The above algorithm requires the user to set the number of threads per block, number of blocks per grid and total number of grids.
This makes the algorithm perform poorly when run on different machines

because the parameters would be optimised for the original machine only.

To overcome this, we use the cuBLAS library which automatically sets the parameters based on the available GPU and performs an optimised matrix multiplication.

4 Datasets

4.1 MNIST

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. Due to minimal preprocessing required, we can quickly test our algorithm on this dataset.

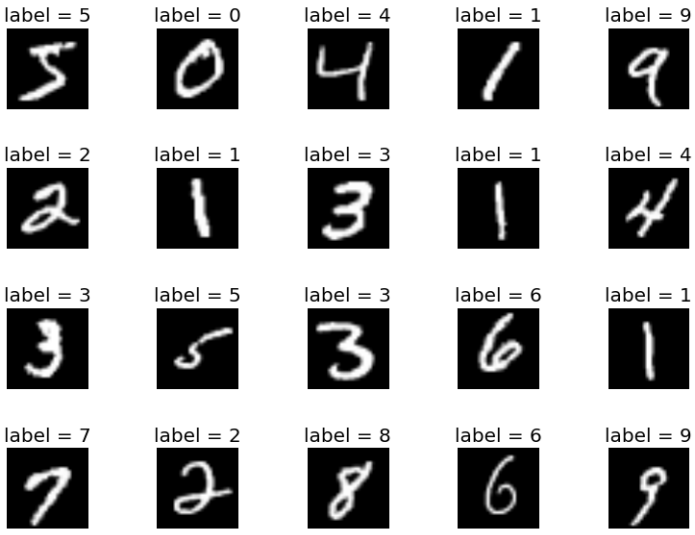
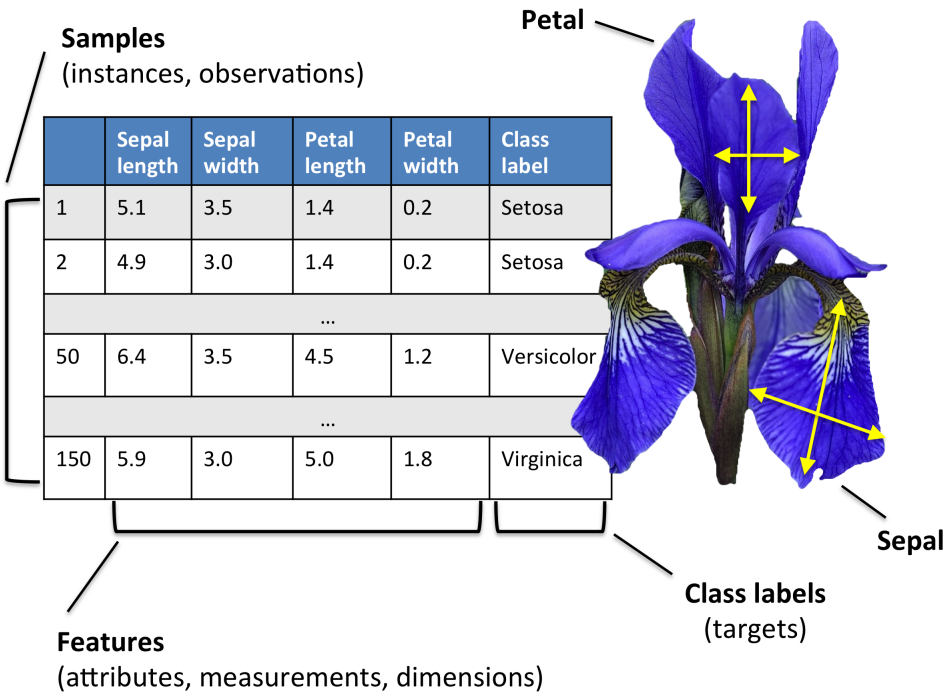


Figure 4: Sample Data from MNIST Dataset

4.2 Iris Flower Dataset

The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from

each sample: the length and the width of the sepals and petals, in centimeters.

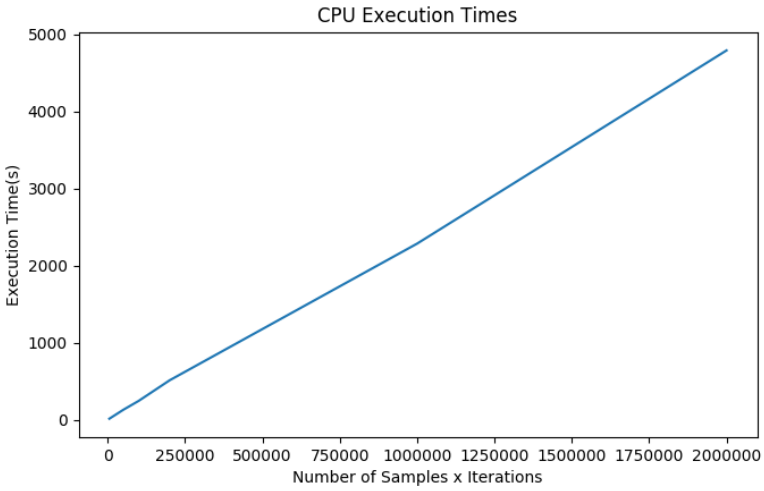


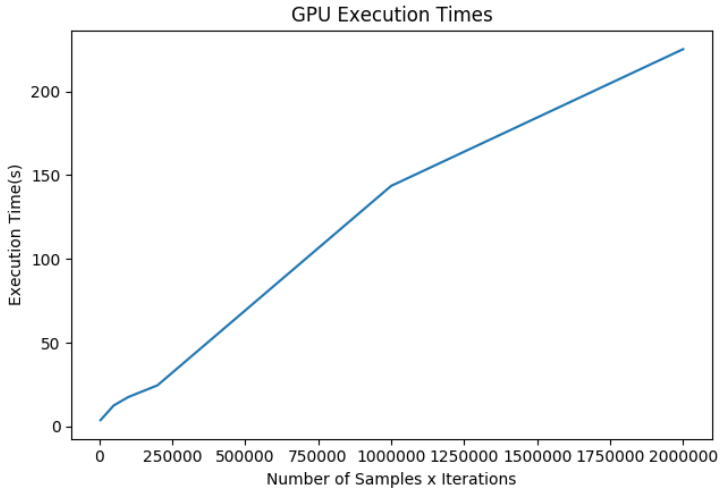
	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

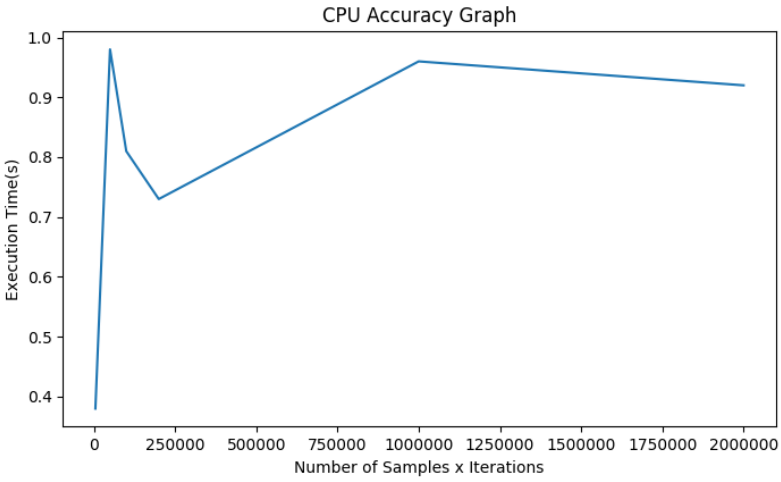
Figure 5: Iris Dataset Structure

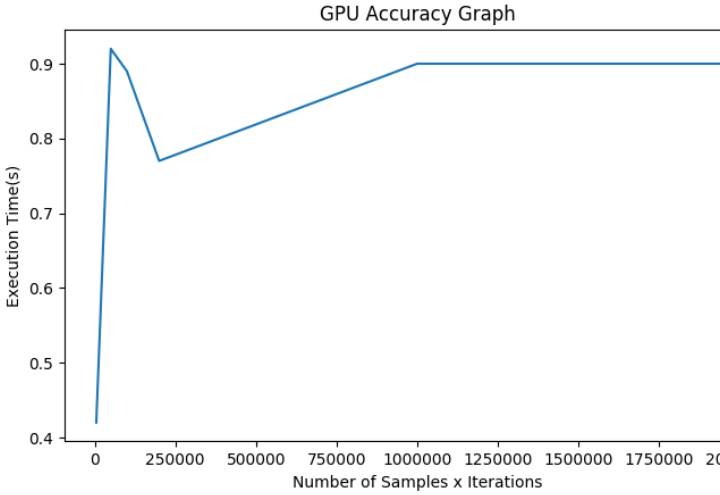
5 Results

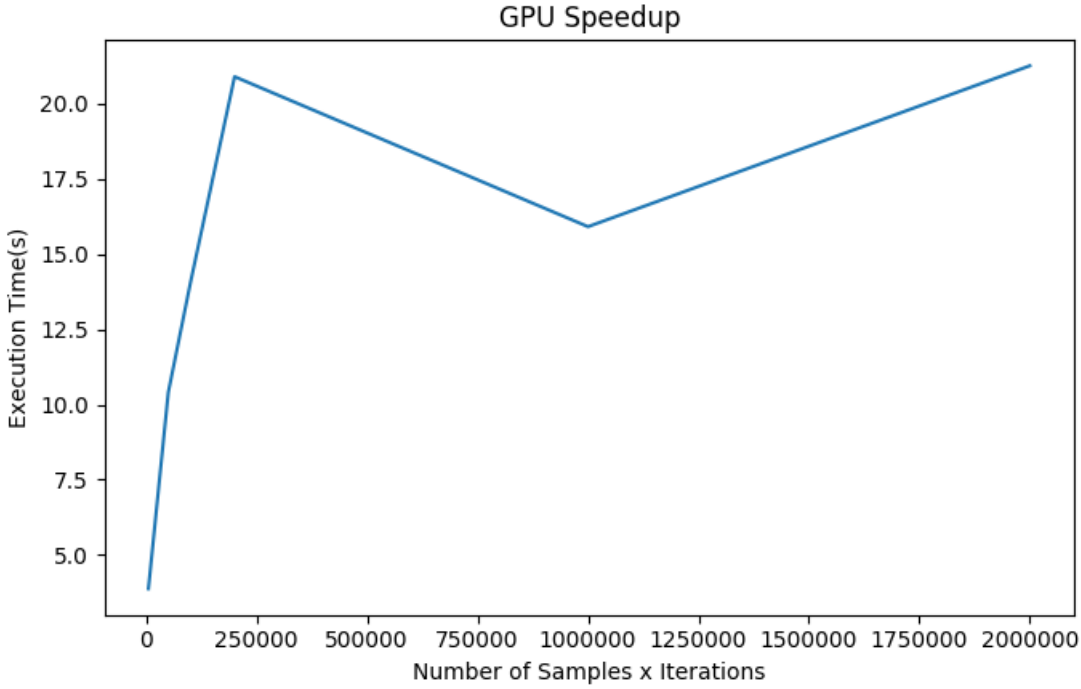
5.1 MNIST

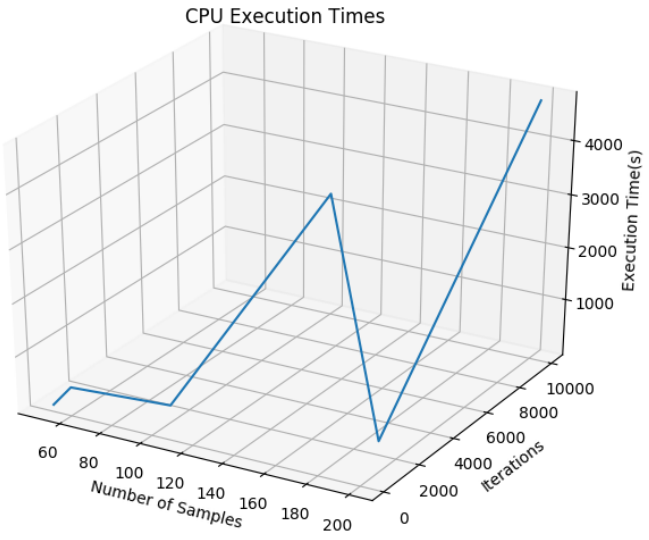

CPU Execution Times
Execution Time(s)
Number of Samples x Iterations
0
1000
2000
3000
4000
5000
0
250000
500000
750000
1000000
1250000
1500000
1750000
2000000


GPU Execution Times
Execution Time(s)
Number of Samples x Iterations
0
50
100
150
200
0
250000
500000
750000
1000000
1250000
1500000
1750000
2000000

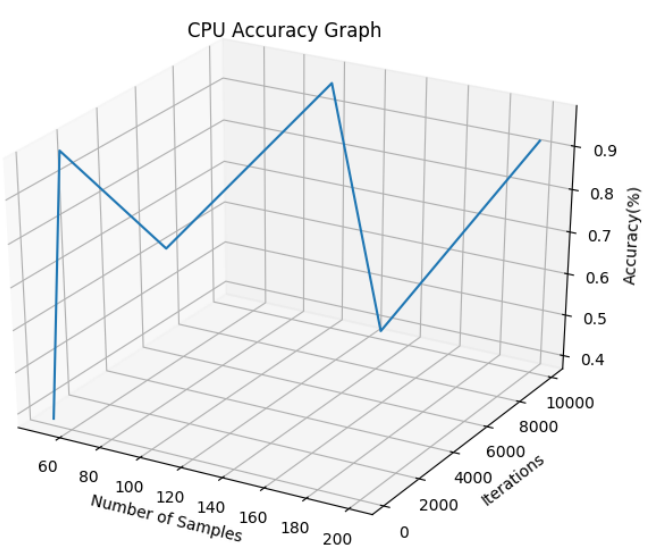

CPU Accuracy Graph
Execution Time(s)
Number of Samples x Iterations
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
250000
500000
750000
1000000
1250000
1500000
1750000
2000000

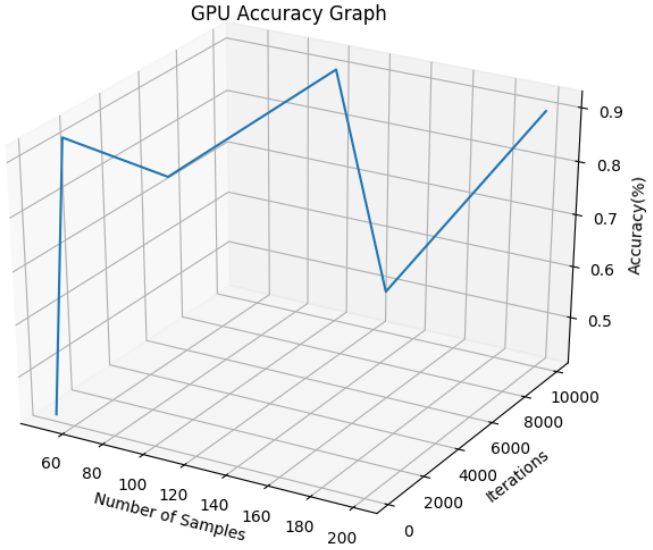

GPU Accuracy Graph
Execution Time(s)
Number of Samples x Iterations
0.4
0.5
0.6
0.7
0.8
0.9
0
250000
500000
750000
1000000
1250000
1500000
1750000

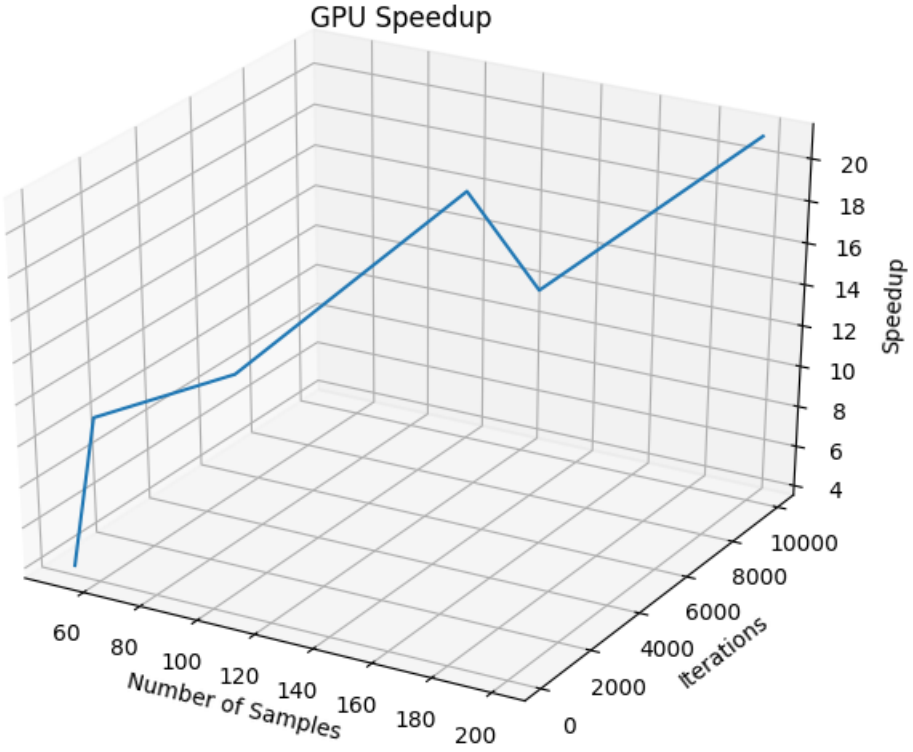

GPU Speedup
Execution Time(s)
20.0
17.5
15.0
12.5
10.0
7.5
5.0
0
250000
500000
750000
1000000
1250000
1500000
1750000
2000000
Number of Samples x Iterations


CPU Execution Times
Execution Time(s)
4000
3000
2000
1000
10000
8000
6000
4000
2000
0
Iterations
60
80
100
120
140
160
180
200
Number of Samples

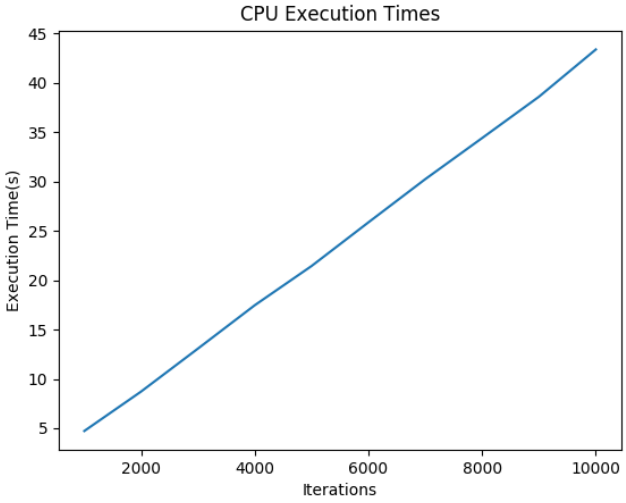

GPU Execution Times
Execution Time(s)
225
200
175
150
125
100
75
50
25
10000
8000
6000
4000
2000
0
Iterations
60
80
100
120
140
160
180
200
Number of Samples

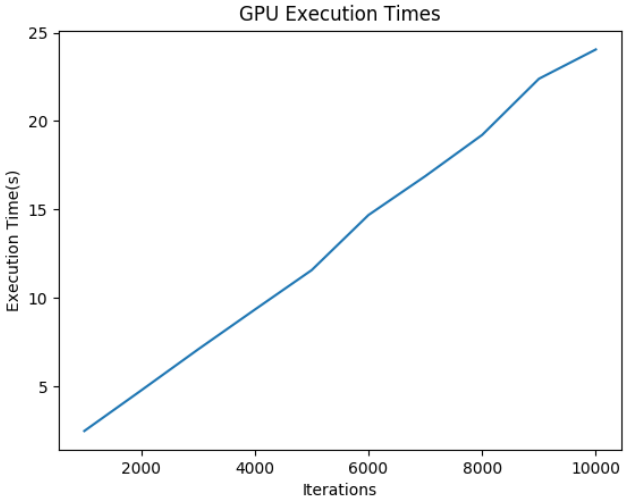

CPU Accuracy Graph
Accuracy(%)
0.9
0.8
0.7
0.6
0.5
0.4
Iterations
10000
8000
6000
4000
2000
0
Number of Samples
60
80
100
120
140
160
180
200

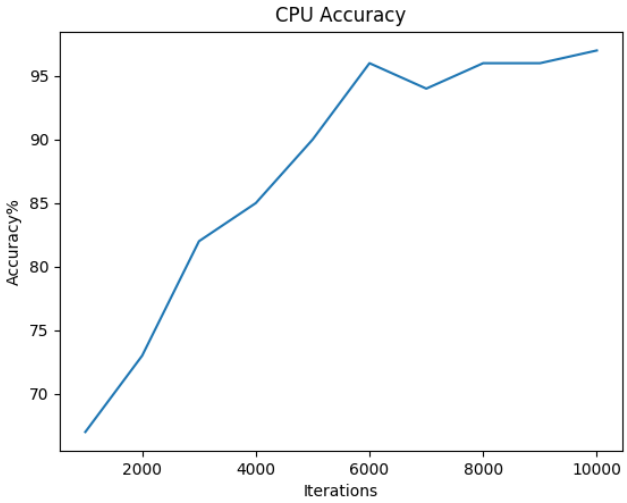

GPU Accuracy Graph
Accuracy(%)
0.9
0.8
0.7
0.6
0.5
Iterations
10000
8000
6000
4000
2000
0
Number of Samples
60
80
100
120
140
160
180
200

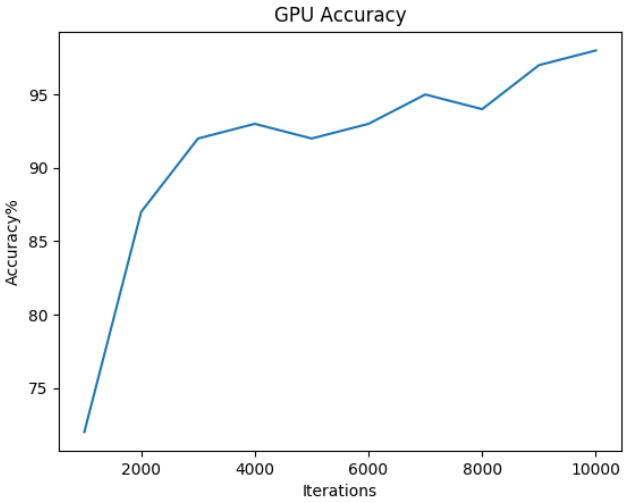

GPU Speedup
Speedup
20
18
16
14
12
10
8
6
4
Iterations
10000
8000
6000
4000
2000
0
Number of Samples
60
80
100
120
140
160
180
200

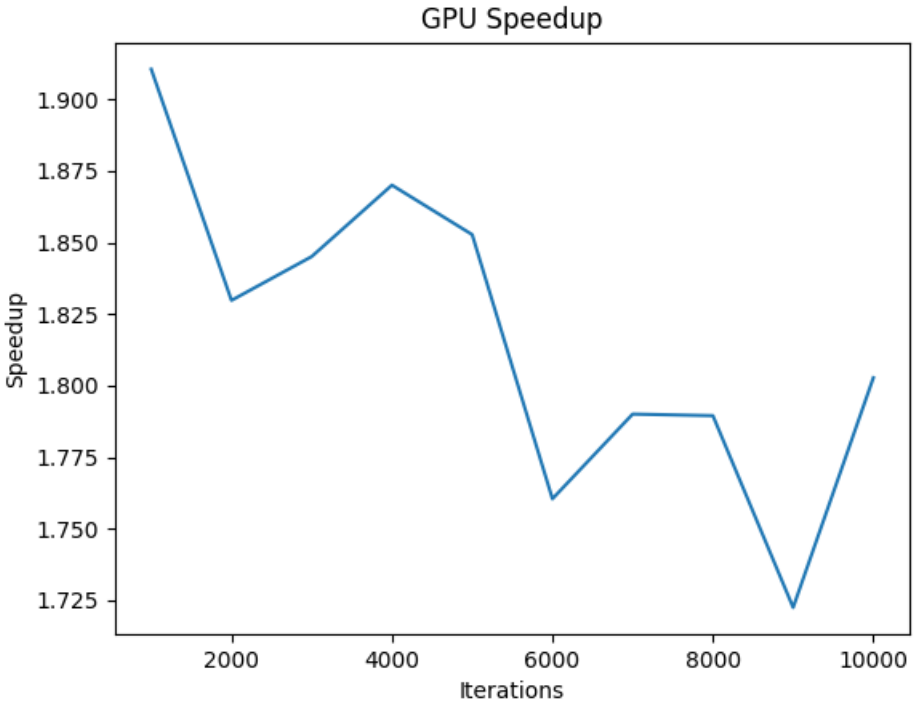
5.2 IRIS


CPU Execution Times
Execution Time(s)
Iterations
45
40
35
30
25
20
15
10
5
2000
4000
6000
8000
10000


GPU Execution Times
Execution Time(s)
Iterations
25
20
15
10
5
2000
4000
6000
8000
10000


CPU Accuracy
Accuracy%
Iterations
95
90
85
80
75
70
2000
4000
6000
8000
10000


GPU Accuracy
Accuracy%
Iterations
95
90
85
80
75
2000
4000
6000
8000
10000


GPU Speedup
Speedup
Iterations
1.900
1.875
1.850
1.825
1.800
1.775
1.750
1.725
2000
4000
6000
8000
10000