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. Surya Prakash

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 $1 \to n$ do
 3:     for $j$ from $1 \to p$ do
 4:         Let $sum = 0$
 5:         for $k$ from $1 \to m$ 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);
```

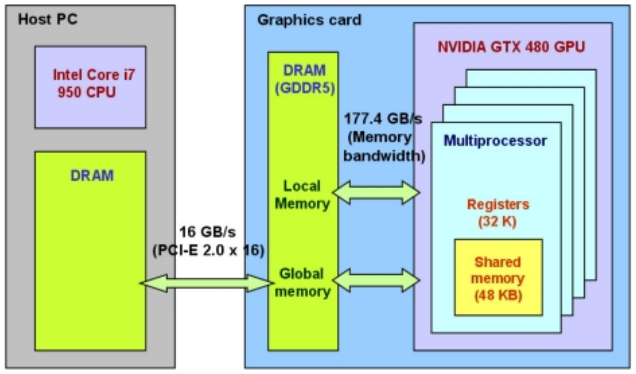


Figure 1: Host and Device Memory

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**.

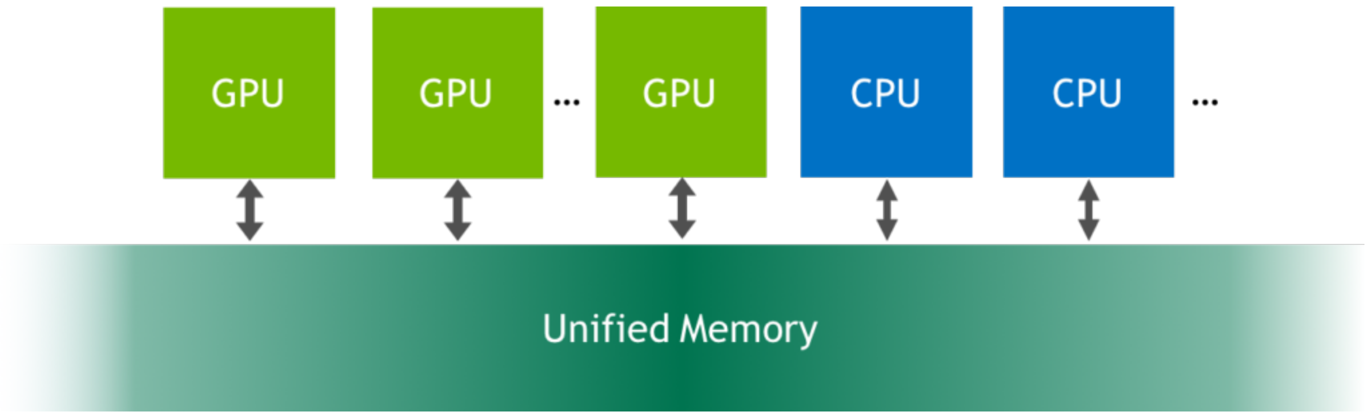


Figure 2: Unified Memory

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.

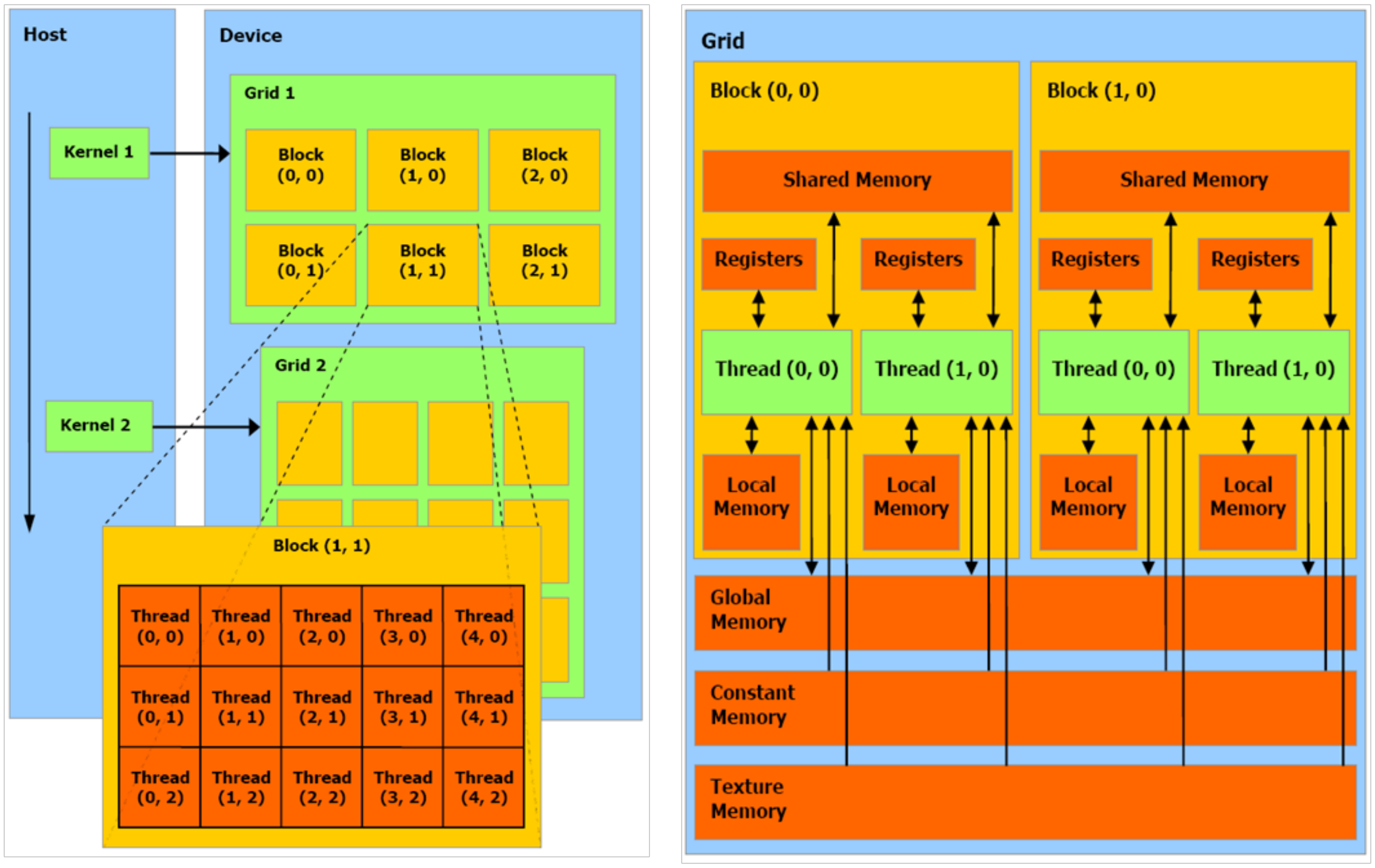


Figure 3: Grid, Block and Thread Model

The following code block demonstrate matrix multiplication using the grid/block-s/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.
The relevant code:

```
template<typename T> T par_mat_mult(const T& h_A,const T& h_B){
// Cacluating sizes of matrices as well as memory space
    ↪ needed to store the data in them.
        int size_A = h_A.n * h_A.m, size_B = h_B.n * h_B.m;
        int mem_size_A = size_A * sizeof(double), mem_size_B =
            ↪ size_B * sizeof(double);

// copying data from matrix h_A to unified memory array d_A
    for(int i = 0; i < h_A.n; i++){
        for(int j = 0;j < h_A.m; j++){
                d_A[i * h_A.m + j] = h_A[i][j];
        }
    }

// copying data from matrix h_B to unified memory array d_B
    for(int i = 0; i < h_B.n; i++){
        for(int j = 0;j < h_B.m; j++){
                d_B[i * h_B.m + j] = h_B[i][j];
        }
    }

// Cacluating sizes of matrices as well as memory space
    ↪ needed to store the data in them.
        int size_C = h_A.n * h_B.m;
    int mem_size_C = size_C * sizeof(double);
// Parameters for cublas function call
    const double alpha = 1.0;
    const double beta = 0.0;
// Function call to cublas to multiply matrices storen in d_A
    ↪ , d_B and store result in h_CUBLAS.
    cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, h_B.m, h_A.n,
        ↪  h_A.m, &alpha, d_B, h_B.m, d_A, h_A.m, &beta, d_C,
        ↪ h_B.m);



// Copying contents from h_CUBLAS into CPU-accessible data
    ↪ store, d_C
    cudaMemcpy(h_CUBLAS, d_C, mem_size_C,
        ↪ cudaMemcpyDeviceToHost);

//Copying contents from d_C to result matrix
    T result(h_A.n, h_B.m);
    for(int i = 0; i < h_A.n; i++){
        for(int j = 0;j < h_B.m; j++){
                result[i][j] = h_CUBLAS[i * h_B.m + j];
        }
    }

    return result;
}
```

4 Hardware

The hardware used is NVIDIA's Geforce 940MX dedicated GPU card.

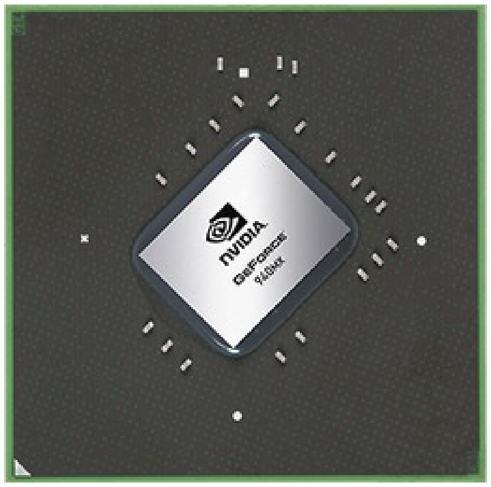
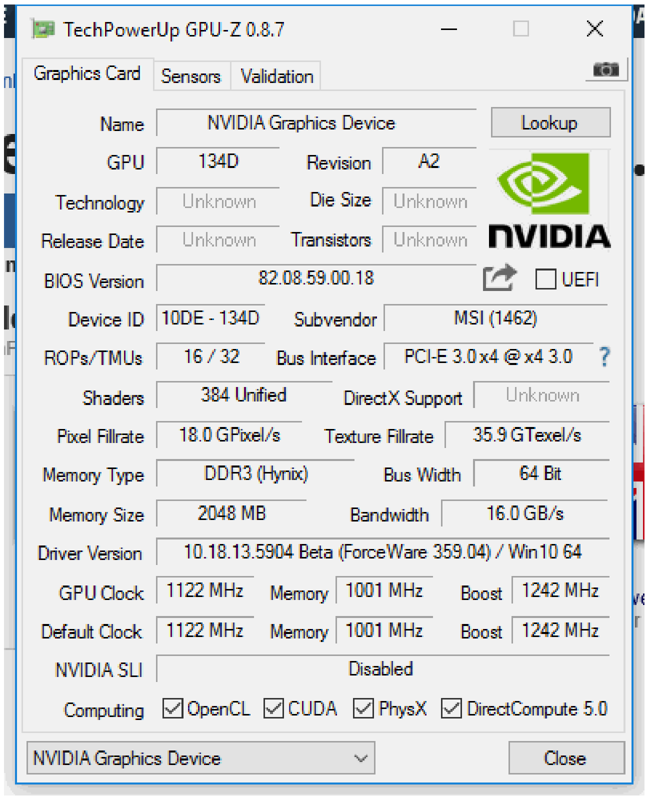


Figure 4: NVIDIA Geforce 940MX



Name	NVIDIA Graphics Device		
GPU	134D	Revision	A2
Technology	Unknown	Die Size	Unknown
Release Date	Unknown	Transistors	Unknown
BIOS Version	82.08.59.00.18	UEFI	☐
Device ID	10DE - 134D	Subvendor	MSI (1462)
ROPs/TMUs	16 / 32	Bus Interface	PCI-E 3.0 x4 @ x4 3.0
Shaders	384 Unified	DirectX Support	Unknown
Pixel Fillrate	18.0 GPixel/s	Texture Fillrate	35.9 GTexel/s
Memory Type	DDR3 (Hynix)	Bus Width	64 Bit
Memory Size	2048 MB	Bandwidth	16.0 GB/s
Driver Version	10.18.13.5904 Beta (ForceWare 359.04) / Win10 64		
GPU Clock	1122 MHz	Memory 1001 MHz	Boost 1242 MHz
Default Clock	1122 MHz	Memory 1001 MHz	Boost 1242 MHz
NVIDIA SLI	Disabled		
Computing	✓ OpenCL ✓ CUDA ✓ PhysX ✓ DirectCompute 5.0		

NVIDIA Graphics Device

Figure 5: NVIDIA Geforce 940MX Specifications

5 Datasets

5.1 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.

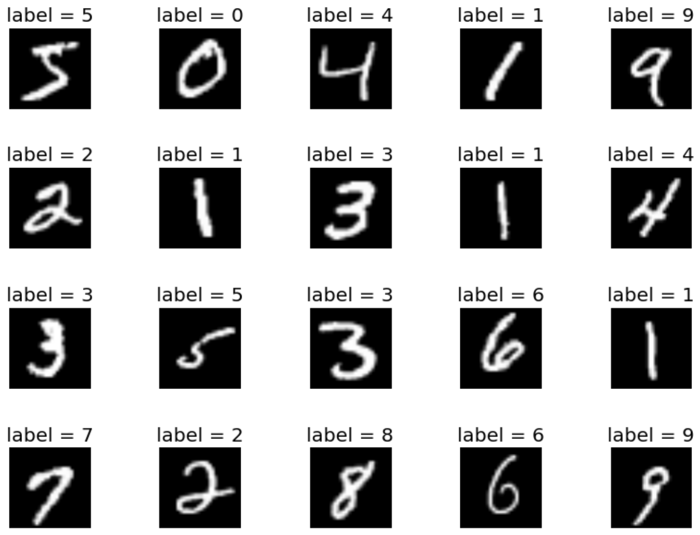
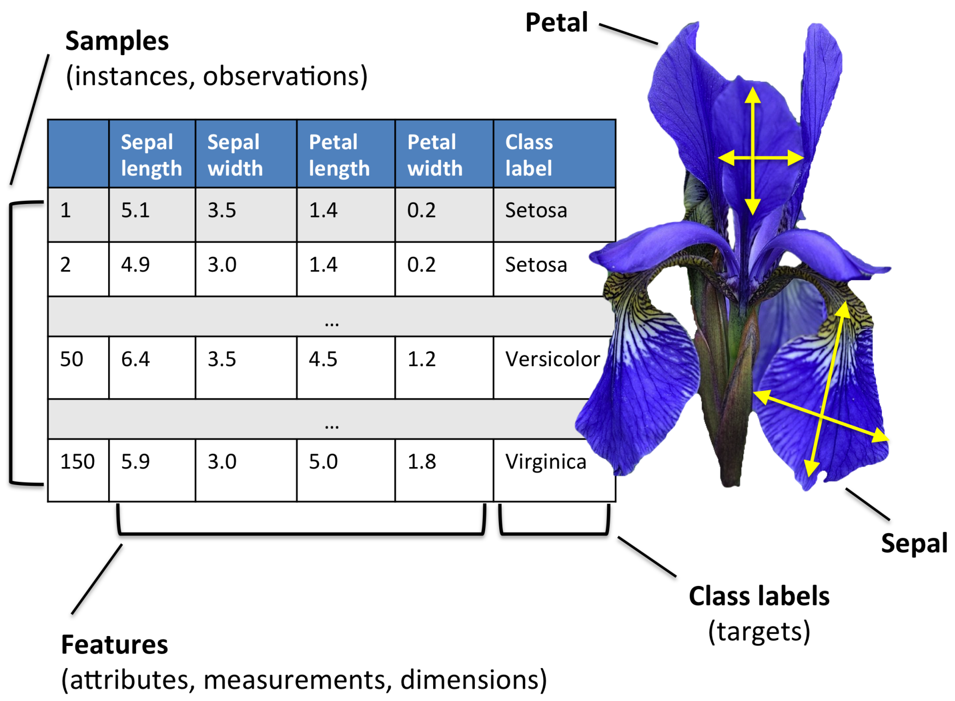


Figure 6: Sample Data from MNIST Dataset

5.2 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.



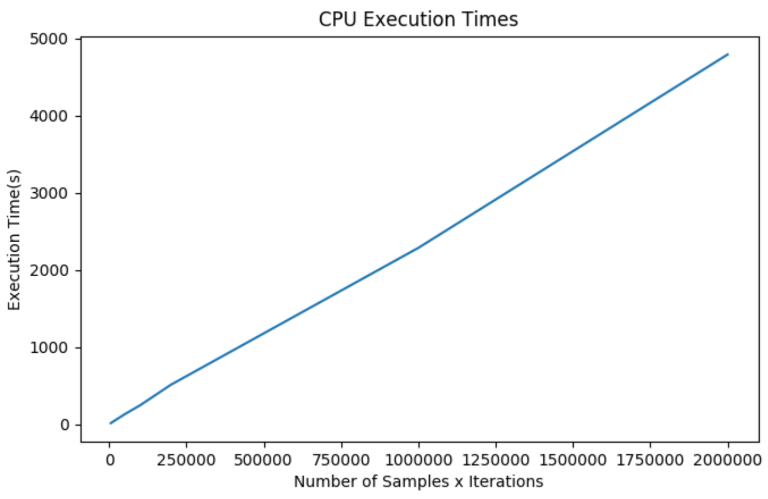
	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
…					
50	6.4	3.5	4.5	1.2	Versicolor
…					
150	5.9	3.0	5.0	1.8	Virginica

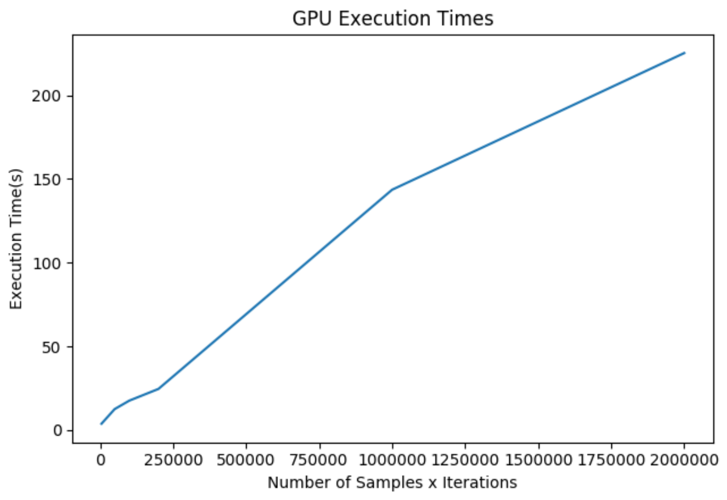
Samples
(instances, observations)
Petal
Sepal
Class labels
(targets)
Features
(attributes, measurements, dimensions)

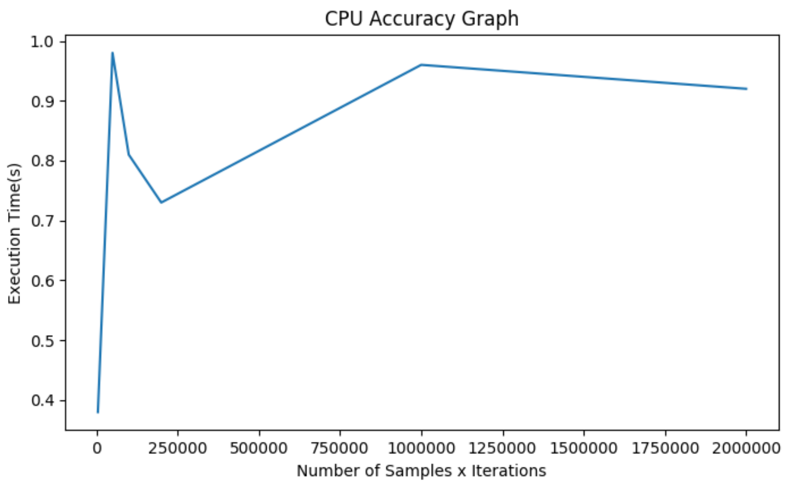
Figure 7: Iris Dataset Structure

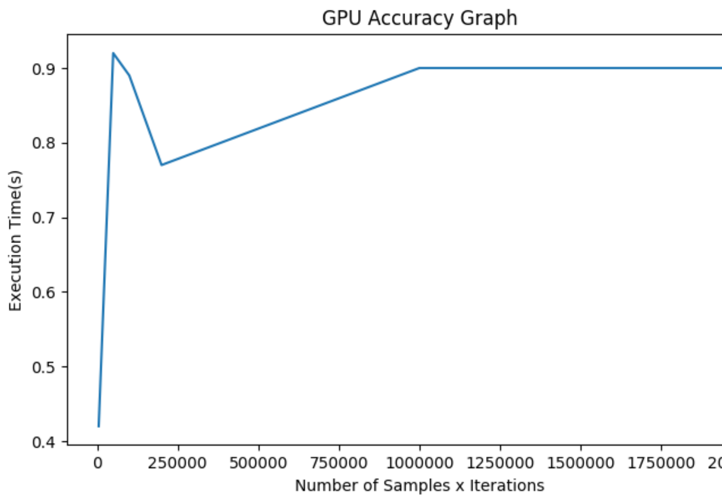
6 Results

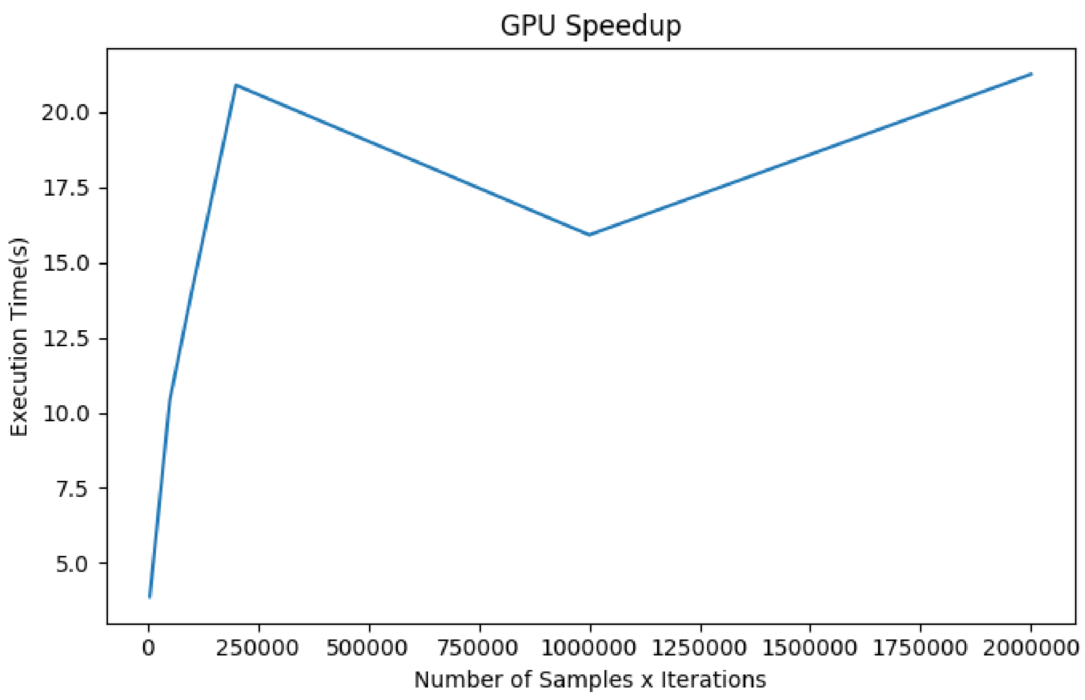
6.1 MNIST

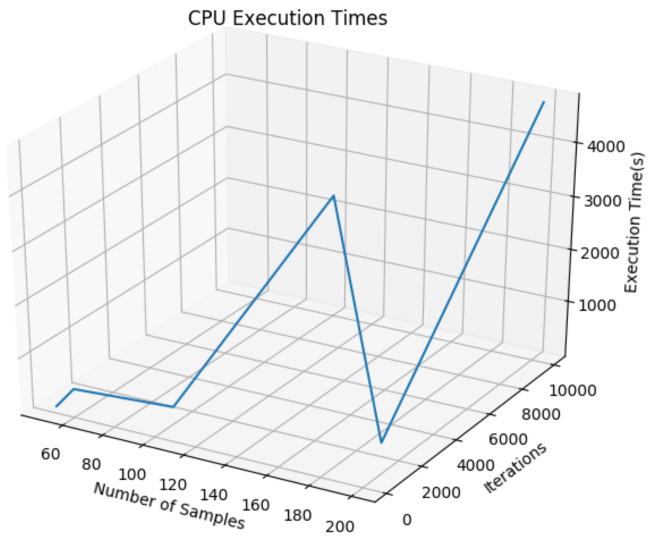

CPU Execution Times
Execution Time(s)
Number of Samples x Iterations
0
1000
2000
3000
4000
5000
0
250000
500000
750000
1000000
1250000
1500000
1750000
2000000

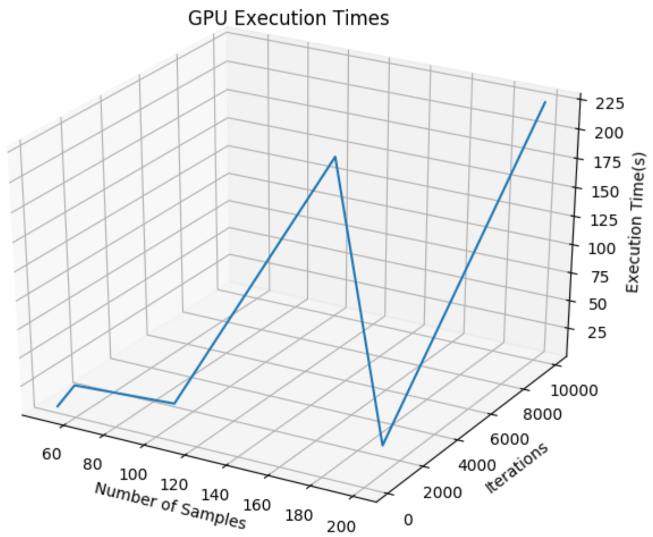

GPU Execution Times
Execution Time(s)
Number of Samples x Iterations
0
50
100
150
200
0
250000
500000
750000
1000000
1250000
1500000
1750000
2000000

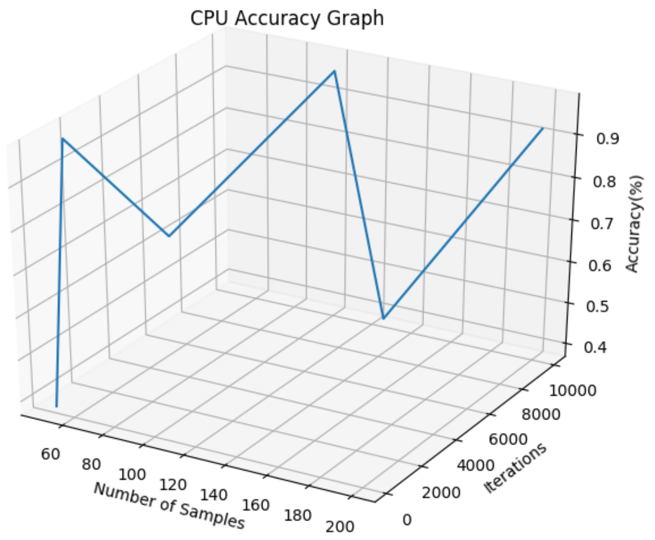

CPU Accuracy Graph
Execution Time(s)
Number of Samples x Iterations
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
250000
500000
750000
1000000
1250000
1500000
1750000
2000000

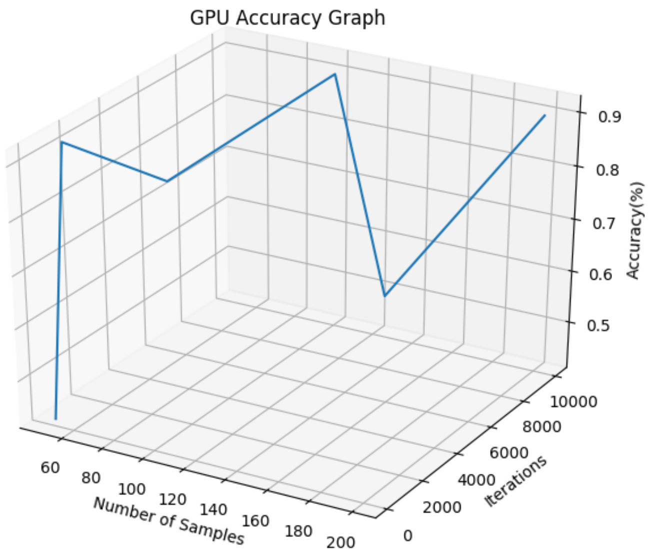

GPU Accuracy Graph
Execution Time(s)
Number of Samples x Iterations
0.4
0.5
0.6
0.7
0.8
0.9
0
250000
500000
750000
1000000
1250000
1500000
1750000

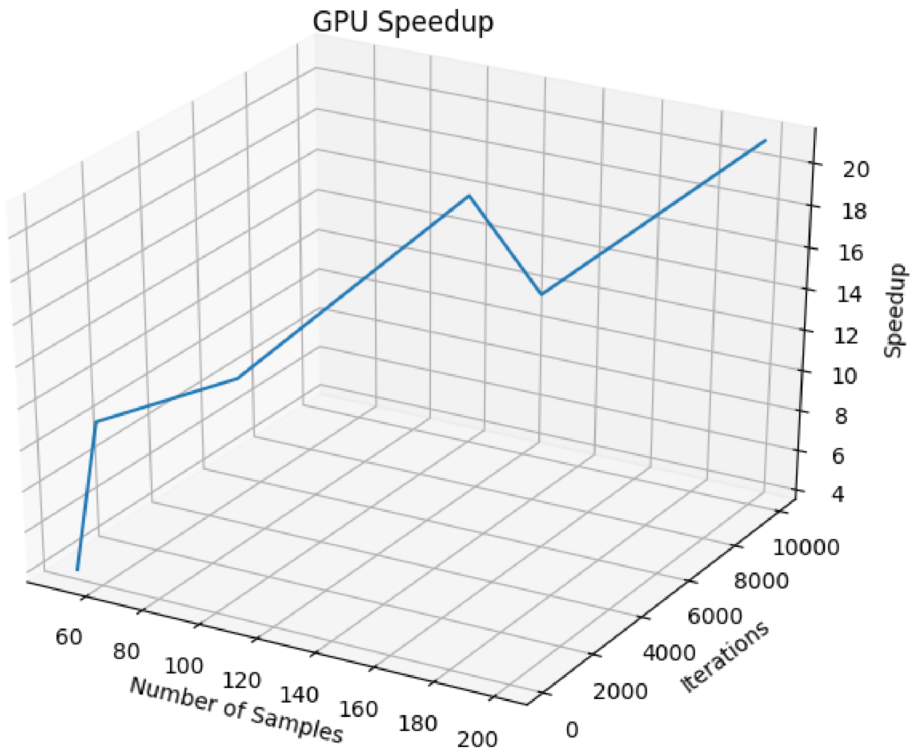

GPU Speedup
Execution Time(s)
Number of Samples x Iterations


CPU Execution Times
Number of Samples
Iterations
Execution Time(s)

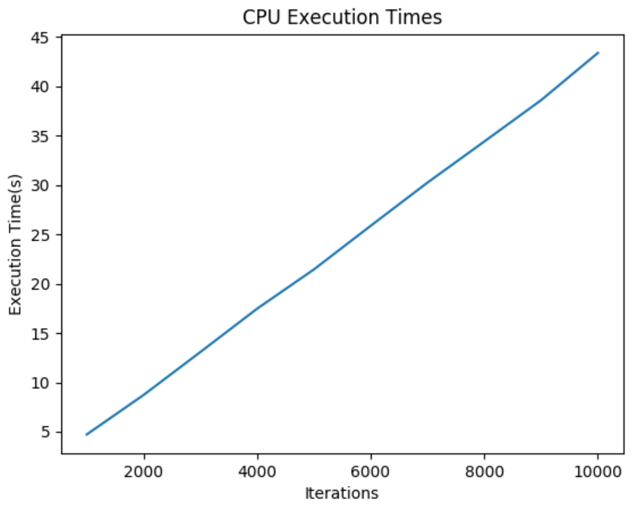

GPU Execution Times
Number of Samples
Iterations
Execution Time(s)

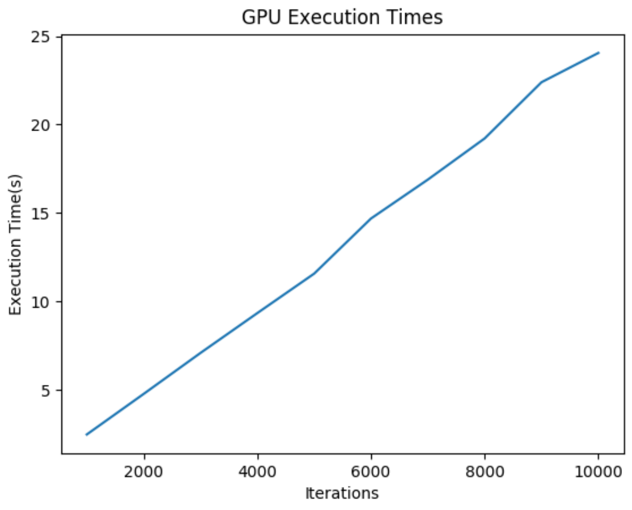

CPU Accuracy Graph
Accuracy(%)
0.4
0.5
0.6
0.7
0.8
0.9
Number of Samples
60
80
100
120
140
160
180
200
Iterations
0
2000
4000
6000
8000
10000

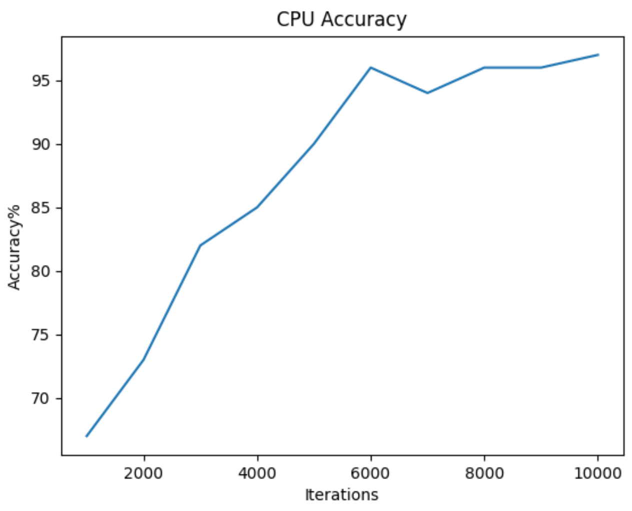

GPU Accuracy Graph
Accuracy(%)
0.5
0.6
0.7
0.8
0.9
Number of Samples
60
80
100
120
140
160
180
200
Iterations
0
2000
4000
6000
8000
10000

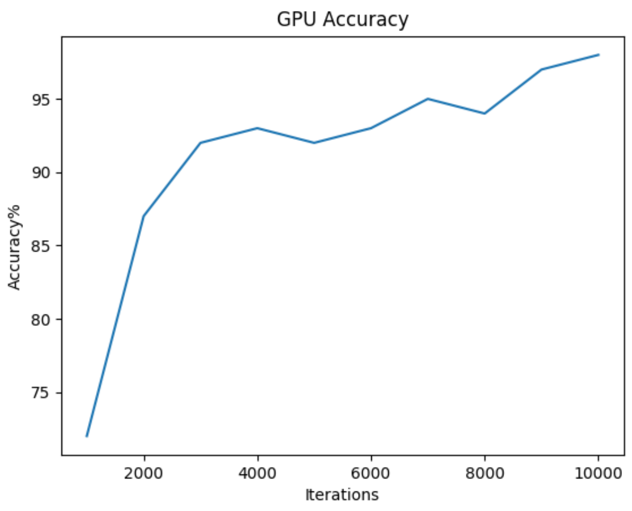

GPU Speedup
Speedup
4
6
8
10
12
14
16
18
20
Number of Samples
60
80
100
120
140
160
180
200
Iterations
0
2000
4000
6000
8000
10000

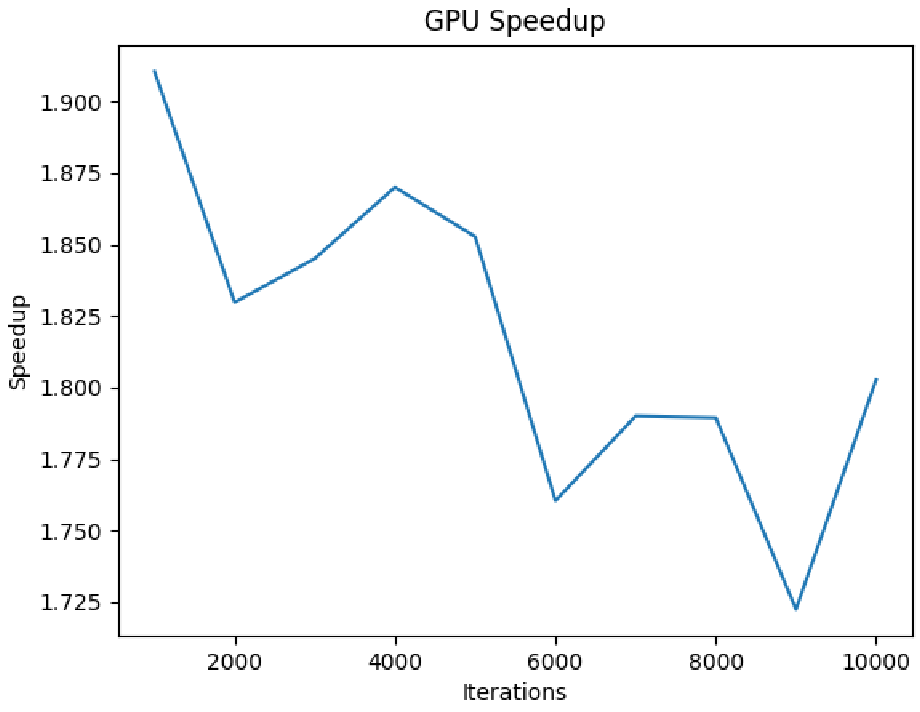
6.2 IRIS


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


GPU Execution Times
Execution Time(s)
Iterations
5
10
15
20
25
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

7 Result Analysis (Conclusion)

7.1 Accuracy

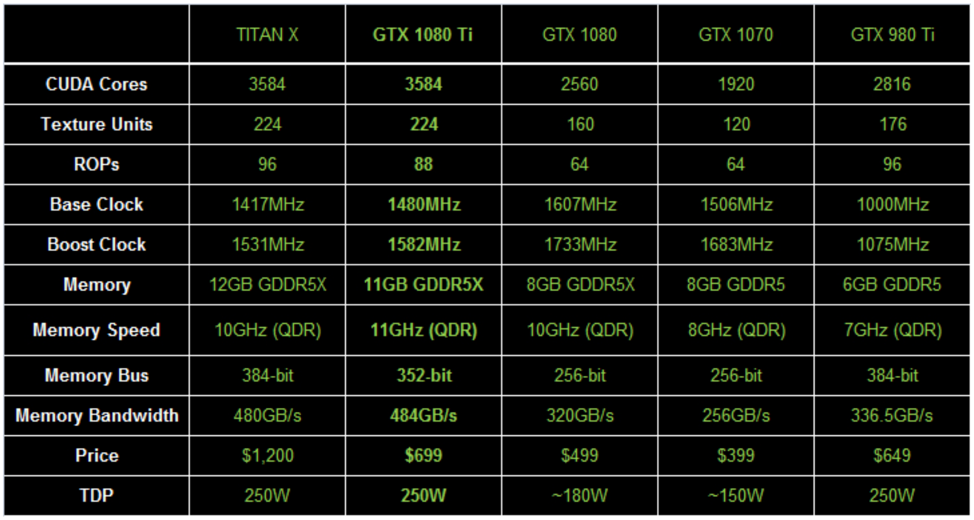
From the accuracy graphs of both the Iris and MNIST dataset, we observe that the curve is very similar.
This indicates that the results obtained from GPU multiplications is the same as the results obtained from CPU multiplications.
There is zero or extremely minimal error in calculations.

7.2 Speed-up

We see a noticeable speed-up of upto 20x on the MNIST dataset and 2x on the Iris dataset.

The Iris dataset has a smaller speed-up due to the fact that the matrices involved in the multiplication operations have small dimensions. The speed-up gained due to faster multiplication is diminished by the overhead of copying data from host memory to device memory.

7.3 Improving Speed-up



	TITAN X	GTX 1080 Ti	GTX 1080	GTX 1070	GTX 980 Ti
CUDA Cores	3584	3584	2560	1920	2816
Texture Units	224	224	160	120	176
ROPs	96	88	64	64	96
Base Clock	1417MHz	1480MHz	1607MHz	1506MHz	1000MHz
Boost Clock	1531MHz	1582MHz	1733MHz	1683MHz	1075MHz
Memory	12GB GDDR5X	11GB GDDR5X	8GB GDDR5X	8GB GDDR5	6GB GDDR5
Memory Speed	10GHz (QDR)	11GHz (QDR)	10GHz (QDR)	8GHz (QDR)	7GHz (QDR)
Memory Bus	384-bit	352-bit	256-bit	256-bit	384-bit
Memory Bandwidth	480GB/s	484GB/s	320GB/s	256GB/s	336.5GB/s
Price	$1,200	$699	$499	$399	$649
TDP	250W	250W	~180W	~150W	250W

Figure 17: GPU Comparisions

Speed-up is largely dependent on the GPU. The number of CUDA cores determine the number of jobs that can be run in parallel and the memory bandwidth determines how fast the data can be accessed and written by the cores.

The 940MX has 384 CUDA cores and 16.02 GB/S memory bandwidth. The high-end graphics card such as NVIDIA 1080 TI (Figure 17) have as many as 3584 CUDA cores and 484 GB/S memory bandwidth. It has almost 10x the cores and 30x the memory bandwidth of 940MX.
Using these cards, we can obtain as much as a 200x computation speedup for the MNIST dataset.

8 File Details

- matrix.cpp
  - designed a modular operator overloaded matrix class for C++, which was used in nn.cpp
  - Variables
    - data -> stores the matrix data
    - _n -> stores no of rows
    - _m -> stores no of cols
  - Functions
    - operators [],+.-,*,/,= -> overloaded all mathematical operators to work for matrix-matrix as well as matrix-scalar operations.
    - seqmult -> Performs sequential matrix multiplication.
    - hadamard -> elementwise multiplication of matrices, c[i][j] = a[i][j]*b[i][j]
    - randomize() -> initialize values of matrices to random values
    - toVector() -> converts a column matrix to C++ STL vector and returns
    - activate(func) -> applies the passed function on every element of the matrix, a[i][j] = func(a[i][j])
    - coladd(a,b) -> add a column vector(b) to every column of matrix a

        * collapse(a) -> breaks matrix to column vector, and returns sum of all vectors
        * to2DVector() -> converts the matrix to C++ STL nested vector format and returns

- nn.cpp
    - The neural network library, fully modular, multiple activation function as well as multi-hidden layer support
    - Variables
        * inputsize, outputsize -> input and output layer sizes respectively
        * alpha -> learning rate of gradient descent
        * activation -> stores activation function of each layer
        * diff -> stores the corresponding differention function for each layer
        * weights -> layer transition weights for each layer transition
        * biases -> layer transition bias for each layer transition
    - Functions
        * addLayer(size,type) -> adds a layer to the neural network model of the given size and activation function type
        * outputActivation(type) -> sets output layer activation function
        * predict(input) -> returns the predict output for the given inputs in matrix form
        * train(input,results,iters) -> Performs batch training using gradient descent for iters no. of times, trying to fit the output to results
        * sigmoid, tanh, relu, nil -> activation functions
        * sigmoiddiff, tanhdiff, relu, nildiff -> differention of activation functions

- chkmatrix.cpp
    - A small test code to verify that all matrix functions are working properly

- chknnmnist.cpp

  - Test code that verifies performance of the neural network on the MNIST dataset

- chknniris.cpp
  - Test code that verifies performance of the neural network on the IRIS dataset

- readMNIST.cpp
  - CPP code to read binary format data of MNIST dataset

- kernel.cu
  - Contains all CUDA related code.
  - Variables
    * handle -> Used to start and initialise cublas library
    * d_A, d_B, d_C -> Unified memory variables to store initial matrices to be multiplied as well as final result matrix.
    * h_CUBLAS -> Device memory variable to store temporary result produced by cublas library.
    * size_A, size_B, size_C, mem_size_A, mem_size_B, mem_size_C -> Size variables
    * result -> Final result to be returned.
  - Functions
    * initiate() -> Initialises cublas library
    * matrixMulCPU(pointer,pointer,pointer,size,size,size) -> Test function to multiply matrices stored in 1D arrays using sequential algorithm
    * initialise_memory(size,size) -> Initialise memory in GPU to d_A, d_B, d_C, h_CUBLAS in the beginning to avoid repetetive allocation.
    * free_memory() -> Free memory allocated to device variables.
    * par_mat_mult(matrix,matrox) -> Multiples two matrices using parallel operations and returns the resultant matrix.