Title: Parallel Computing and Distributed Systems Coursework

Module: SOFT354

Module leader: Dr Mario Gianni

Deadline: 24th January 2019

Student Number: 10528570

Chosen Algorithm: Implementation of a multi layer perceptron – Machine learning

1.0 Introduction

1.1 Multi Layer Perceptron definition

A multi layer perceptron is a feedforward neural network which at the minimum has an input layer, a hidden layer and an output layer.

The input layer has a size of 1xN where N constitutes the number of inputs – it contains the values of each neuron and is initialised to the input data passed to it. The hidden layer can constitute multiple layers, although in this implementation it's only one, it contains a series of interconnected neurons which have data passed through to them by the input layers and ultimately are outputted to either further hidden layers or to the output layer.

The hidden layer is defined by the number of neurons and is again a 1xN array but of weights – these are initialised randomly between values of 0 and 1 but are then offset by an initial bias of -0.5, this is done as to ensure all units fire and update initially to generate the best network performance increase. Each neuron has its own sigmoid function which maps the summative input value multiplied by the weights and maps it to a sigmoid curve to provide its new output value.

The output layer is similar to the hidden layers in that it is a layer composed of weighted neurons and a sigmoid output however unlike the hidden layer the output is the end point of the neural network and does not feed into any more neurons. In the case of the network implemented – there is a single output and thus the output layer constitutes a single 1x1 Neuron.

The network overall being feedforward makes the network easy to follow – it is linear in nature and passes the data forwards to an output rather than back into itself. As such the network has lots of parallel operations happening simultaneously during training and executions which are targets for acceleration.

The network undergoes training for every single iteration it performs. As this is a supervised learning task we have an idealised output for every piece of input data which we can use to check the health of our network and perform updates. These updates are performed using backpropagation in which the error value we have calculated is used to update the weights below it and then using calculus the weights below that are then updated accordingly as well. This is done using a method called gradient descent in which the minimal error is targeted and moved towards in increments of our learning rate. The learning rate is important as it helps us improve the network enough that it actually trains whilst also not overstepping the minima.

**References used included in references section – see end

1.2 Problem Implemented

The problem implemented for the network itself is a basic trigonometric problem. The problem is to predict an angle theta from two input points X and Y. The points are generated using a seeded random number generator which generates values between 1 and 0 with 4 decimal places. This random number is our ideal output angle theta in radians – as such this maps the arm to a perfect semi circle and as such X and Y can be modelled using Sin and Cos functions respectively making the data easy to generate.

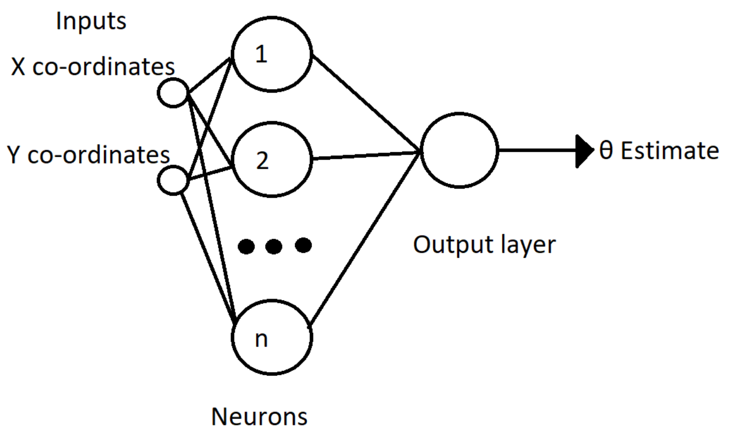


Figure 1 A design for the network implemented showing N neurons in the hidden layer and the X and Y data inputs

1.3 Mathematical formulations:

A multi layer perceptron is a complex algorithm and as such has a large variety of mathematical operations which comprise it. The full mathematical operations for various parts of the network are as such listed below separately.

Neuron Value

A neuron has inputs X, weights W and output net

The outputs are equal to the sum of the weight times each input

As such the overall equation is: NET = $\mathbf{\Sigma}$WX

Sigmoid Function

The sigmoid mapping function is the non linearity of each neurons outputs.

As such each neurons values are transformed by it.

The output for the sigmoid is denoted by the value a

$a = 1/\ 1+e^{-net}$

Differential of sigmoid

In order to go back and train our network we must find the rate of change with respect to the output

As such we differentiate with respect to output o

This resolves to $e^{-net}/(1+e^{-net})^2$ or o(1-o)

Error

Error e is defined as the positive difference between our ideal output t and our actual output o

$e = (t-o)^2$

Gradient Descent of first layer

By making use of the chain rule we can eventually find the differential of the Error e with respect to W2 or our second set of weights i.e the hidden layer which feeds into our output.

Our learning rate is defined as α

This is derived as $de/dw_2 = -(t-o).o(1-o)$

The error gradient is defined as $de/dw_2 = d_3.a^t$

As such we can update our gradient using gradient descent

$W_2 \leftarrow W_2 - \alpha\, de/dw_2$

Training lower layers

The error of the upper layers d_3 is multiplied by the weights of this layer

The error of this layer is defined as d_2 and the output as a_2

$D_2 = (W_2.D3)\ .a_2(1-a_2)$

This is then used to calculate the error gradient and update it with the error gradient defined as

$De/dw_1 = d_2.X^t$

$W_1 \leftarrow W_1 - \alpha\, de/dw_1$

Matrix multiplication

Matrix multiplication is a key element of this algorithm – it involves multiplying two matricies (m,n) and (p,q)

Both N and P must be equal to perform a multiplication

This will create an array of size (M,Q)

2.0 Implementation

2.1 Key serial aspects of program

Creating Dataset

To begin we define global variables which will be used in generating the co-ordinates for our input, these include the length of the arm and the number of samples required.

```
const int radius = 3;
const int numSamples = 100;
```

Then in main we call the function to create an output with the parameter numSamples to produce a dataset of 100 doubles.

```
double* expectedOutputData = createOutput(numSamples);
```

The method itself creates random integers with 4 decimal places between the value of 0 and 1. Rather than pass back the array it returns a pointer to the new array of doubles.

```
double* createOutput(int dataSize) {
	double* dataSet = new double[dataSize];
	for (int i = 0; i < dataSize; i++) {
		double theta = rand() % 1000;
		theta = theta / 1000.0;
		dataSet[i] = theta;
	}
	return dataSet;
}
```

Then we generate input data using the parameters of a pointer to the expectedOutputData array and the constant number numSamples. We need the expected output data array as we need to iterate through it and generate the co-ordinates from it using basic trigonometry so the ideal output matches the input data. We then return a pointer to the new array of inputs.

```
double* inputData = createInput(numSamples, expectedOutputData);

double* createInput(int dataSize,double* inputData) {
	int setSize = dataSize * 2;
	double* dataSet = new double[setSize];
	for (int i = 0; i < dataSize; i++) {
		dataSet[i] = radius * cos(inputData[i]);
		dataSet[(dataSize + i)] = radius * sin(inputData[i]);
	}
	return dataSet;
}
```

Structure of layers/ Neurons

We define the number of neurons using a global static – this is done because it should not change during runtime and needs to be accessed throughout the program. It also makes the code more maintainable as altering the number of neurons is done by changing a single constant.

```
const int numNeurons = 20;
```

We must define the two arrays of layers. We don’t need a third layer for input as we can directly map input into first layer hence why it’s called the inputLayer despite it having the size of the number of neurons. This is done by increasing the row size to two to allow for both the inputs to be stored. This allows for us to have a hidden layer in two arrays rather than three as we can process all the weights necessary for the network in a smaller space. We also initialise the arrays with 0s in every position and declare them globally so they can be accessed anywhere.

```
double inputLayer[2][numNeurons] = {0};
double outputLayer[1][numNeurons] = {0};
```

The arrays are now initialised with random values to aid in training and offset with an initial bias of -0.5. Input layers are updated both at once to make the loop more efficient.

```
for (int ii = 0; ii < numNeurons; ii++) {
	inputLayer[0][ii] = { ((rand() % 1000) / 1000.0) - 0.5 };
	inputLayer[1][ii] = { ((rand() % 1000) / 1000.0) - 0.5 };
	outputLayer[0][ii] = { ((rand() % 1000) / 1000.0) - 0.5 };
}
```

Training perceptron

There is a separate method used for the training of the perceptron – this requires pointers to both the input data and the expected output data.

```
void trainMultiLayerPerceptron(double* inputData, double* expectedOutputData, int
maxiter) {
	for (int j = 0; j < maxiter; j++) {
```

It was important to reset the adjustment matrices with every iteration so as to prevent earlier adjustments which are no longer beneficial to the training of the network having an impact.

```
		double layerOneAdjustment[2][numNeurons] = { 0 };
		double layerTwoAdjustment[1][numNeurons] = { 0 };
		double errorSum = 0.0;
		for (int i = 0; i < numSamples; i++) {
```

For every iteration through a sample new arrays must be created so as to temporarily store data and ensure space for results from multiplications.

```
			double layer1output[1][numNeurons] = { 0 };
			double inputDataArray[1][2] = { inputData[i]
,inputData[numSamples + i] };
```

By automatically creating the array as a transpose by swapping the rows and columns this entirely saves on a method and computational time.

```
			double transposeInputData[2][1] = {
inputData[i],inputData[numSamples + i] };

			double layerOneAdjustmentTmp[2][numNeurons] = { 0 };
			double layerTwoAdjustmentTmp[1][numNeurons] = { 0 };
			double layer2delta[1][1] = { 0 };
			double layer1error[1][numNeurons] = { 0 };
			double layer1delta[1][numNeurons] = { 0 };
			double layer1outputSigmoid[1][numNeurons] = { 0 };
```

Matrix multiply and sigmoid matrix are used to calculate the results of layer 1 – these are subsequently stored in layer1output.

```
			MatrixMultiply(*inputDataArray, *inputLayer, *layer1output, 1, 2,
numNeurons);
			sigmoidMatrix(*layer1output, numNeurons, *layer1output);
```

Once more these methods are used to calculate and store the results of layer 2 – these are stored in layer2output.

```
			double layer2output = sigmoidScalar(dotProduct(*layer1output,
*outputLayer, numNeurons));
```

Error and delta values are calculated using appropriate scalar or matrix calls to most efficiently exploit gradient descent and update the system. More additional matrix multiplication calls are made – exploiting the parallelisation – in total 4 calls will be made per iteration.

```
                    double layer2error = expectedOutputData[i] - layer2output;
                    layer2delta[0][0] = layer2error * 
sigmoidDerivativeScalar(layer2output);
                    MatrixMultiply(*layer2delta, *outputLayer, *layer1error, 1, 1, 
numNeurons);
                    sigmoidDerivativeMatrix(*layer1output, numNeurons, 
*layer1outputSigmoid);
                    elementMultiply(*layer1error, *layer1outputSigmoid, *layer1delta, 
numNeurons);
```

Gradient descent is now fully exploited in these matrix multiplications as adjustment data is now created – this is then used in a for loop which utilises the adjustments to update the exitsing arrays then in another loop multiplied by the learning rate to adjust the layers weights themselves. Error sum is also calculated simultaneously so the network can be evaluated.

```
                    MatrixMultiply(*transposeInputData, *layer1delta, 
*layerOneAdjustmentTmp, 2, 1, numNeurons);
                    MatrixMultiply(*layer2delta, *layer1output, 
*layerTwoAdjustmentTmp, 1, 1, numNeurons);
                    for (int ii = 0; ii < numNeurons; ii++) {
                           layerOneAdjustment[0][ii] = layerOneAdjustment[0][ii] + 
layerOneAdjustmentTmp[0][ii];
                           layerOneAdjustment[1][ii] = layerOneAdjustment[1][ii] + 
layerOneAdjustmentTmp[1][ii];
                           layerTwoAdjustment[0][ii] = layerOneAdjustment[0][ii] + 
layerTwoAdjustmentTmp[0][ii];
                    }
                    errorSum = errorSum + (layer2error * layer2error);
                    for (int ii = 0; ii < numNeurons; ii++) {

                           inputLayer[0][ii] = inputLayer[0][ii] + learningRate * 
layerOneAdjustment[0][ii];
                           inputLayer[1][ii] = inputLayer[1][ii] + learningRate * 
layerOneAdjustment[1][ii];
                           outputLayer[0][ii] = outputLayer[0][ii] + learningRate * 
layerTwoAdjustment[0][ii];
                    }
```

Sigmoid functions

The sigmoid functions are very expensive computationally functions as they have to make use of an exponential function and in the case of sigmoidMatrix – repeatedly use exponential for every value in an array but as a negative. These methods unfortunately are not parallelisable but do come with a great cost whilst still being necessary to the training.

```
double sigmoidScalar(double inputVal) {
       double sigmoidValue;
       sigmoidValue = 1 / (1 + exp(-inputVal));
       return sigmoidValue;
}


void sigmoidMatrix(double *inputArray, int rows, double *output) {
       for (int i = 0; i < rows; i++) {
              output[i] = 1 / (1 + exp(-inputArray[i]))
```

2.2 Cuda

Matrix Multiplication

This code is invoked initially in main. This is done so that there is persistent memory allocated on the GPU for the arrays we create. As there are only 4 matrix matrix multiplications which take place and all of them have very easily calculatable sizes which don't change it makes sense rather than spending a large amount of computational time to allocate once all the memory needed and deallocate after the neural network has run rather than cost efficiency as this is not very memory expensive.

```
	double* expectedOutputData = createOutput(numSamples);
	double* inputData = createInput(numSamples, expectedOutputData);
	cudaMalloc(&matrixA, (1 * 2 * sizeof(double)));
	cudaMalloc(&matrixB, (2 * numNeurons * sizeof(double)));
	cudaMalloc(&matrixC, (1 * 1 * sizeof(double)));
	cudaMalloc(&matrixD, (1 * numNeurons * sizeof(double)));
	cudaMalloc(&matrixE, (2 * 1 * sizeof(double)));
	for (int i = 0; i < epochs; i++) {
		for (int ii = 0; ii < numNeurons; ii++) {
			inputLayer[0][ii] = { ((rand() % 1000) / 1000.0) - 0.5 };
			inputLayer[1][ii] = { ((rand() % 1000) / 1000.0) - 0.5 };
			outputLayer[0][ii] = { ((rand() % 1000) / 1000.0) - 0.5 };
		}
		trainMultiLayerPerceptron(inputData, expectedOutputData, 1000);
	}
	cudaFree(matrixA);
	cudaFree(matrixB);
	cudaFree(matrixC);
	cudaFree(matrixD);
	cudaFree(matrixE);
	return 0;
```

In this method memory is moved over to the GPU from the host to setup the multiplication. This is done using parameters pointing to the globals already allocated in the previous method to greatly increase efficiency. Subsequently the grid is setup to be correctly scaled relative to the size of the resultant array. Afterwards memory is copied from the result of the matrix multiplication back to the host – this is unfortunately very time consuming but has to be done.

```
void cudaMatrixMultiply(double *array1, double *array2, double *output, int arr1_rows,
int arr1_cols, int arr2_cols,double * matrixP1,double * matrixP2,double * matrixP3) {
	cudaMemcpy(matrixP1, array1, sizeof(double)*arr1_rows*arr1_cols,
cudaMemcpyHostToDevice);
	cudaMemcpy(matrixP2, array2, sizeof(double)*arr2_cols*arr1_cols,
cudaMemcpyHostToDevice);
	dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE,1);
	dim3 dimGrid((arr2_cols - 1) / BLOCK_SIZE + 1, (arr1_rows - 1) / BLOCK_SIZE +
1);
	MatMultKernel<<<dimGrid,dimBlock>>> (matrixP1, matrixP2, matrixP3, arr1_rows,
arr1_cols, arr2_cols);
	cudaMemcpy(output, matrixP3, sizeof(double)*arr1_rows*arr2_cols,
cudaMemcpyDeviceToHost);
}
```

In this method we implement the actual parallel aspect of the matrix multiplication. This kernel code begins by initialising the shared memory which will be used to host the sub arrays before subsequently setting up a multitude of variables which are used to identify threads specifically so rows and columns can be assigned more specifically and broken up amongst the blocks.

The code subsequently makes use of the subarrays in shared memory by assigning values based off of their block size or assigning 0 if the rows or columns assigned do not match ones within the array.

The code subsequently syncs all threads to ensure that the sub arrays are generated before processing them then syncs all threads once more to ensure that the result data is ready to be appended to the output.

```
__global__ void MatMultKernel(double *array1, double *array2, double *output, int
arr1_rows, int arr1_cols, int arr2_cols) {
	double result = 0;
	__shared__ double subArray1[BLOCK_SIZE][BLOCK_SIZE];
	__shared__ double subArray2[BLOCK_SIZE][BLOCK_SIZE];
	int bIDx = blockIdx.x, bIDy = blockIdx.y, tIDx = threadIdx.x, tIDy =
threadIdx.y;
	int row = bIDy * BLOCK_SIZE + tIDy;
	int col = bIDx * BLOCK_SIZE + tIDx;
	for (int i = 0; i < (arr1_cols-1)/BLOCK_SIZE+1; i++) {
		if (row < arr1_rows && i*BLOCK_SIZE+tIDx<arr1_cols) {
			subArray1[tIDy][tIDx] = array1[row*arr1_cols + i * BLOCK_SIZE +
tIDx];
		}else {
			subArray1[tIDy][tIDx] = 0;
		}
		if (col < arr2_cols && i*BLOCK_SIZE+tIDy<arr1_cols) {
			subArray2[tIDy][tIDx] = array2[(i * BLOCK_SIZE +
tIDy)*arr2_cols+col];
		}else {
			subArray2[tIDy][tIDx] = 0;
		}
		__syncthreads();
		for (int ii = 0; ii < BLOCK_SIZE; ii++) {
			result += subArray1[tIDy][ii] * subArray2[ii][tIDx];
		}
		__syncthreads();
	}
	if (row < arr1_rows&&col < arr2_cols) {
		output[row*arr2_cols + col] = result;
	}

}
```

2.3 MPI

Matrix Multiplication

```
void mpiMatrixMultiply(double *array1, double *array2, double *output, int arr1_rows,
int arr1_cols, int arr2_cols) {
    int rank, world_size, strip, workers, start, end, iii;
    double result;
    double * tmpResult;
    tmpResult = (double*)malloc(arr1_rows*arr2_cols * sizeof(double));
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    workers = world_size - 1;
    start = 0;
    iii = 0;
    end = arr1_rows;
```

The master rank is set up to control the MPI matrix multiplication. As such it sets up a large amount of variables including strip – a guess at the minimum number of items processed per process as well as start – which is the start position in the array and end – the end position for a given process. It then also sends off copies of array1 and array2 as during the process of training the network these may not be the same across processes due to not all processes receiving results from matrix multiplication calls.

```
    if (rank == 0) {
        strip = arr1_rows / workers;
        for (int reciever = 1; reciever <= workers; reciever++) {
            start = (reciever - 1)*strip;
            if ((((reciever + 1) == world_size) && ((arr1_rows % (world_size
- 1)) != 0))) {
                end = arr1_rows;
            }
            else {
                end = start + strip;
            }
            MPI_Send(&start, 1, MPI_INT, reciever, 1, MPI_COMM_WORLD);
            MPI_Send(&end, 1, MPI_INT, reciever, 1, MPI_COMM_WORLD);
            MPI_Send(array1, arr1_rows*arr1_cols, MPI_DOUBLE, reciever, 1,
MPI_COMM_WORLD);
            MPI_Send(array2, arr1_cols*arr2_cols, MPI_DOUBLE, reciever, 1,
MPI_COMM_WORLD);

        }
```

Rank 0 subsequently sets up to receive from every worker in order of rank – whilst this isn’t the most efficient way to receive as there can be long waits it ensures the data received is ordered by worker and is ready to process instantly into the array using simple arithmetic involving our counter variable iii * start.

```
        for (int i = 1; i <= workers; i++) {
            MPI_Recv(&start, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
            MPI_Recv(&end, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
            MPI_Recv(&iii, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
            MPI_Recv(&output[start*iii],iii, MPI_DOUBLE, i, 2,
MPI_COMM_WORLD, &status);


        }

    }
```

Iterate from the received start to finish of the array whilst increasing the counter variable. Each time it gains a different row and multiplies it by every column available – this is done using 1 dimensional array by multiplying parameters such as the total number of columns to reach the correct position in memory. This counter variable alongside our newly calculated matrix result are passed back – these can then be used to calculate and position the data within the total array in rank 0 completing the matrix calculation.

```
	if (rank > 0) {
		MPI_Recv(&start, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
		MPI_Recv(&end, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
		MPI_Recv(array1, arr1_rows*arr1_cols, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, 
&status);
		MPI_Recv(array2, arr1_cols*arr2_cols, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, 
&status);
		MPI_Send(&start, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
		MPI_Send(&end, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
		for (int row = start; row < end; row++) {
			for (int col = 0; col < arr2_cols; col++) {
				result = 0.0;
				for (int i = 0; i < arr1_cols; i++)
				{
					result = result + array1[row * arr1_cols + i] * 
array2[i * arr2_cols + col];
				}
				tmpResult[iii] = result;
				iii = iii + 1;

			}
		}
		MPI_Send(&iii, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
		MPI_Send(tmpResult, iii, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
```

The tmpResult memory is subsequently freed after the execution of all ranks so as to prevent a memory leak occurring.

```
	}
	free(tmpResult);

}
```

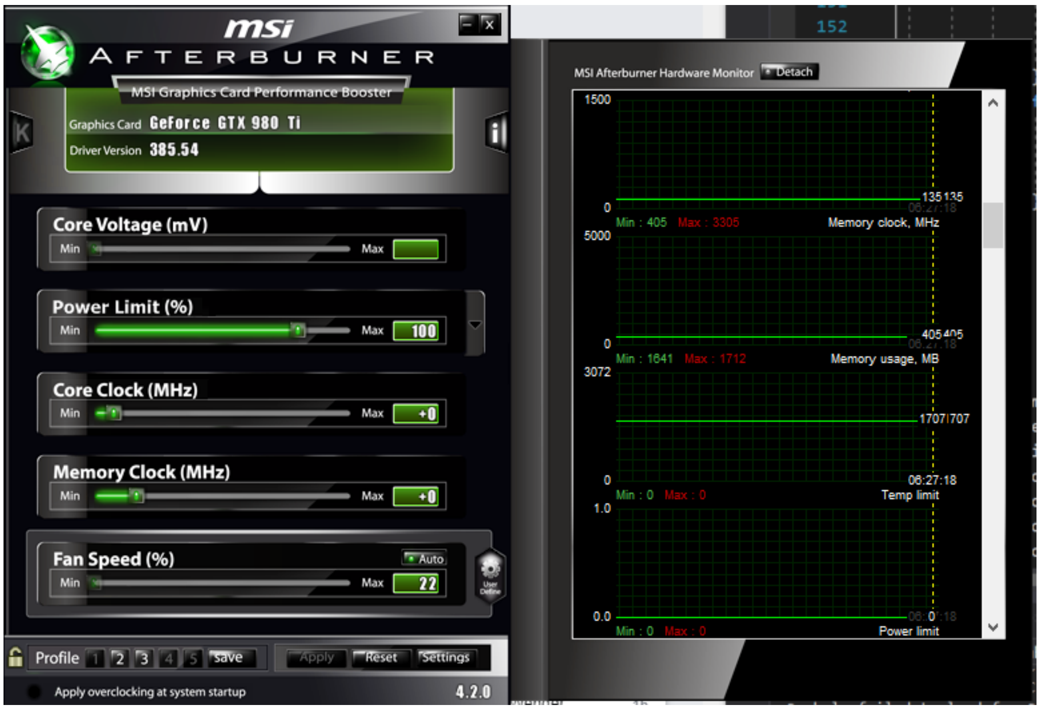
3.0 Evaluation

All testing is done on a Windows 10 64bit machine using a 5820K (12 threaded CPU) at stock clock speeds with a 980ti running driver 385.54 from Nvidia. The 980ti is cooled exceptionally (with a peak load temperature of 50c) to prevent thermal throttling but has any GPU boost facilities disabled to prevent vastly inconsistent results as a result of altering memory bandwidth and clock speeds.

To provide non trivial distinct tests which can be replicated easily on both CUDA and MPI I have opted to change the dimensions of the Neural network by altering the number of neurons. This will have a large impact on memory allocation, usage and on the accelerated matrix multiplications as they will have to deal with varying amounts of weight calculations. For each area of each program I will perform 5 tests using 2,4,8,16 and 20 neurons. It is worth noting that on the current parameters it is likely these will not successfully learn or output results as close to the default settings of the network however it will provide accurate performance metrics of the system. All tests take into account overhead and limit themselves on what they measure to be scoped as minimally as possible.

3.1 CUDA

All CUDA tests were performed using the visual studio debugger with timigns being managed in line with suggested nvidia methods as outlined (https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/). As for VRAM monitoring this was performed using MSI afterburner as this is the current GPU monitoring toolkit I have installed on this system.


msi
AFTERBURNER
MSI Graphics Card Performance Booster
MSI Afterburner Hardware Monitor
Detach
Profile
Save
Apply
Reset
Settings
Auto
User Define
4.2.0

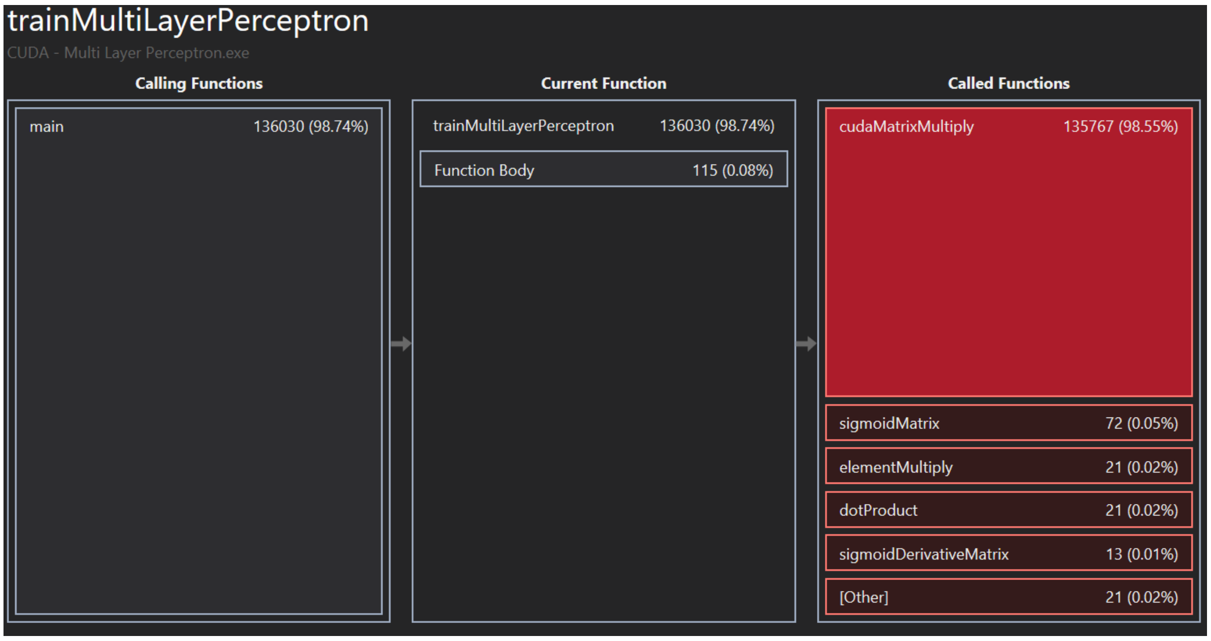
Graphics Card GeForce GTX 980 Ti
Driver Version 385.54
Core Voltage (mV)
Power Limit (%) 100
Core Clock (MHz) +0
Memory Clock (MHz) +0
Fan Speed (%) 22
Apply overclocking at system startup

Memory clock, MHz — Min : 405 Max : 3305 — 405
Memory usage, MB — Min : 1641 Max : 1712 — 1707
Temp limit — Min : 0 Max : 0
Power limit — Min : 0 Max : 0

Figure 2 – MSI afterburner running and displaying various GPU settings

Overall the VRAM monitoring must be taken with a large grain of salt as it only monitors system wide VRAM use and can not pick up executable specific VRAM use. As such minor fluctuations are expected perhaps throwing off results – however it was included for the sake of completeness.

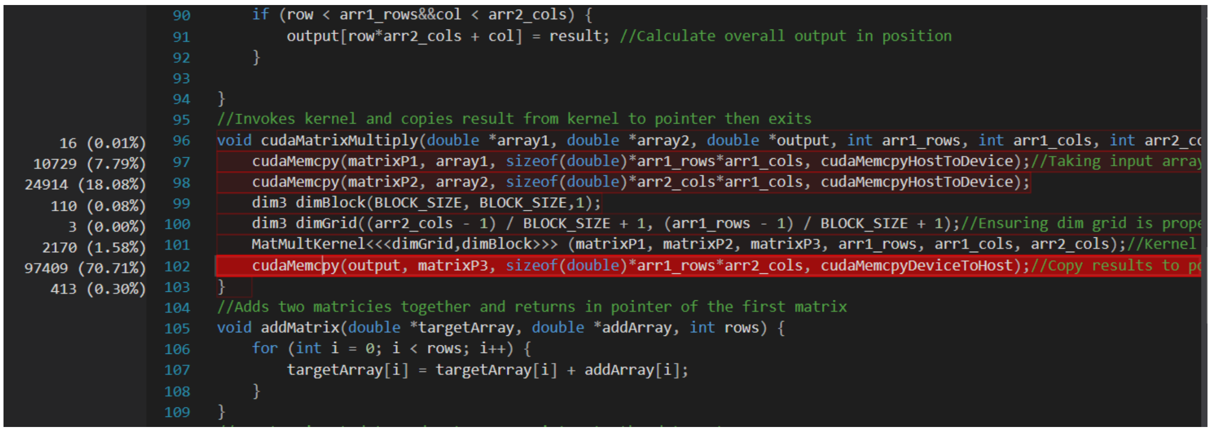
One way to weaknesses in the code which was used was to make use of visual studios built in code profiler. This profiler trawled through my code and found that the serial aspect of the code did not have a significant impact on the execution time – it did however find that some code in my Matrix multiply method in cuda was responsible for the majority of computation.



```
trainMultiLayerPerceptron
CUDA - Multi Layer Perceptron.exe

Calling Functions
main                          136030 (98.74%)

Current Function
trainMultiLayerPerceptron     136030 (98.74%)
Function Body                 115 (0.08%)

Called Functions
cudaMatrixMultiply            135767 (98.55%)
sigmoidMatrix                 72 (0.05%)
elementMultiply               21 (0.02%)
dotProduct                    21 (0.02%)
sigmoidDerivativeMatrix       13 (0.01%)
[Other]                       21 (0.02%)
```

Figure 3 – Visual studio code profiler

When delving deeper into the profiler it can be seen that cudaMemcpy from the GPU to the host is responsible for a large amount of the slow down – this unfortunately can not be helped and as such will have be accepted as is.



```
                        if (row < arr1_rows&&col < arr2_cols) {
                            output[row*arr2_cols + col] = result; //Calculate overall output in position
                        }

                    }
                    //Invokes kernel and copies result from kernel to pointer then exits
     16 (0.01%)     void cudaMatrixMultiply(double *array1, double *array2, double *output, int arr1_rows, int arr1_cols, int arr2_c
  10729 (7.79%)         cudaMemcpy(matrixP1, array1, sizeof(double)*arr1_rows*arr1_cols, cudaMemcpyHostToDevice);//Taking input array
  24914 (18.08%)        cudaMemcpy(matrixP2, array2, sizeof(double)*arr2_cols*arr1_cols, cudaMemcpyHostToDevice);
    110 (0.08%)         dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE,1);
      3 (0.00%)         dim3 dimGrid((arr2_cols - 1) / BLOCK_SIZE + 1, (arr1_rows - 1) / BLOCK_SIZE + 1);//Ensuring dim grid is prope
   2170 (1.58%)         MatMultKernel<<<dimGrid,dimBlock>>> (matrixP1, matrixP2, matrixP3, arr1_rows, arr1_cols, arr2_cols);//Kernel
  97409 (70.71%)        cudaMemcpy(output, matrixP3, sizeof(double)*arr1_rows*arr2_cols, cudaMemcpyDeviceToHost);//Copy results to p
    413 (0.30%)     }
                    //Adds two matricies together and returns in pointer of the first matrix
                    void addMatrix(double *targetArray, double *addArray, int rows) {
                        for (int i = 0; i < rows; i++) {
                            targetArray[i] = targetArray[i] + addArray[i];
                        }
                    }
```

Figure 4 – Visual studio line by line code profiler

Mean average of 3 tests of: Test on Initial memory allocation to VRAM

Number of neurons:	Peak VRam usage	Peak memory usage	Time to complete:
2	1751MB	83MB	365MS
4	1759MB	80MB	385MS
8	1755MB	85MB	436MS
16	1758MB	80MB	521MS
20	1762MB	94MB	614MS

This test was a test on the intial memory allocations in main to the GPU and testing the times. It identified a strong correlation between the size of the matrix and the time to complete. Memory used by the CUDA library seems very arbitrary however.

Mean average of 3 tests of: One multiplication in the kernel

Number of neurons:	Peak VRam usage	Peak memory usage	Time to complete:
2	1825MB	76MB	0.0235840008ms
4	1820MB	81MB	0.0236159991ms
8	1830MB	93MB	0.0237119999ms
16	1823MB	82MB	0.0240000002ms
20	1825MB	88Mb	0.0236159991ms

This test was a test of a single iteration of the matrix multiplication method – the results were too close and not did not provide enough information to say how much of an effect neuron number was having. Although it stands to reason the greater the number of neurons the greater the time to complete should be.

Mean average of 3 tests of: One entire training cycle (I.e an epoch without memory de-allocation)

Number of neurons:	Peak VRam usage	Peak memory usage	Time to complete:
2	1804MB	201MB	71.296414063 Seconds
4	1812MB	202MB	72.623523438 Seconds
8	1809MB	201MB	73.123820313 Seconds
16	1816MB	204MB	73.403054688 Seconds
20	1818MB	204MB	73.786375 seconds

This test was a full epoch encompassing both memory allocation and a large number of matrix multiplications. About 400000 multiplications to be precise. This does show more promising results than our prior experiment as it shows quite clearly that the larger the matrix the more time it takes as was expected from the theory in the maths.

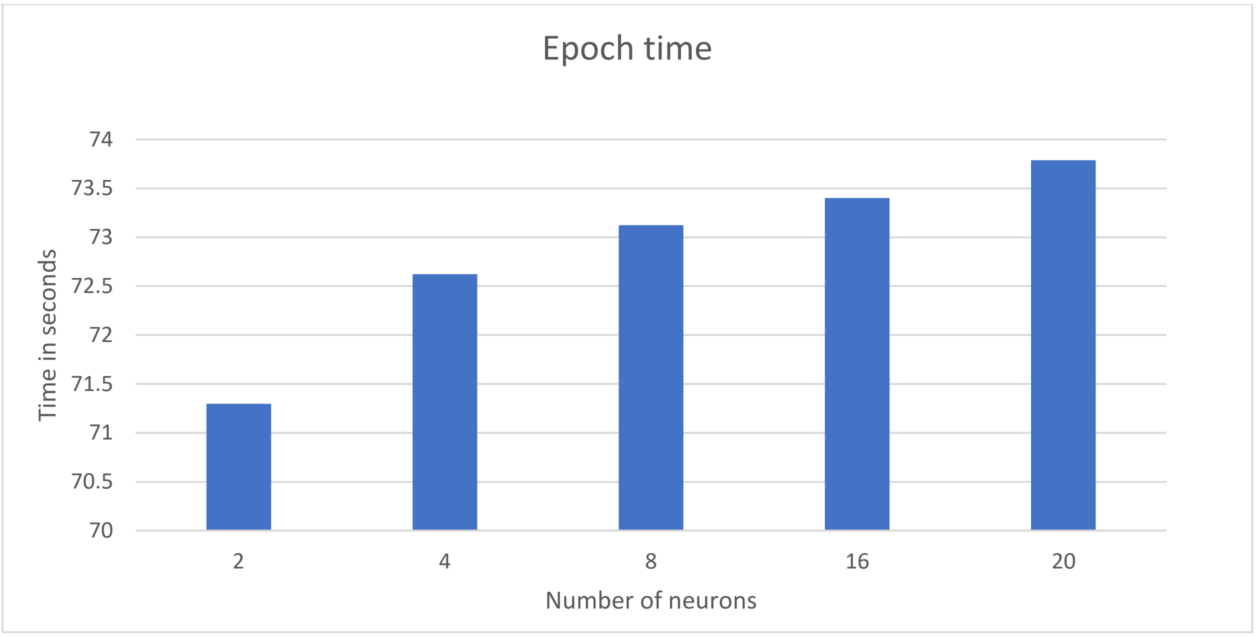


Figure 5 – A graph depicting the number of neurons versus time in seconds for execution – clearly showing the upward trend

3.2 MPI

Time to complete a full training cycle i.e epoch

Number of neurons	Number of ranks	Time (Seconds)
2	4	3.079045
8	4	2.906079
20	4	3.193069
2	8	9.475856
8	8	9.552496
20	8	10.391095

This result clearly shows that fewer ranks working more effectively can produce better results than greater numbers of ranks. In particular when the number of ranks is greater than the number of neurons as this causes the code to serialise largely to prevent any errors from occurring.

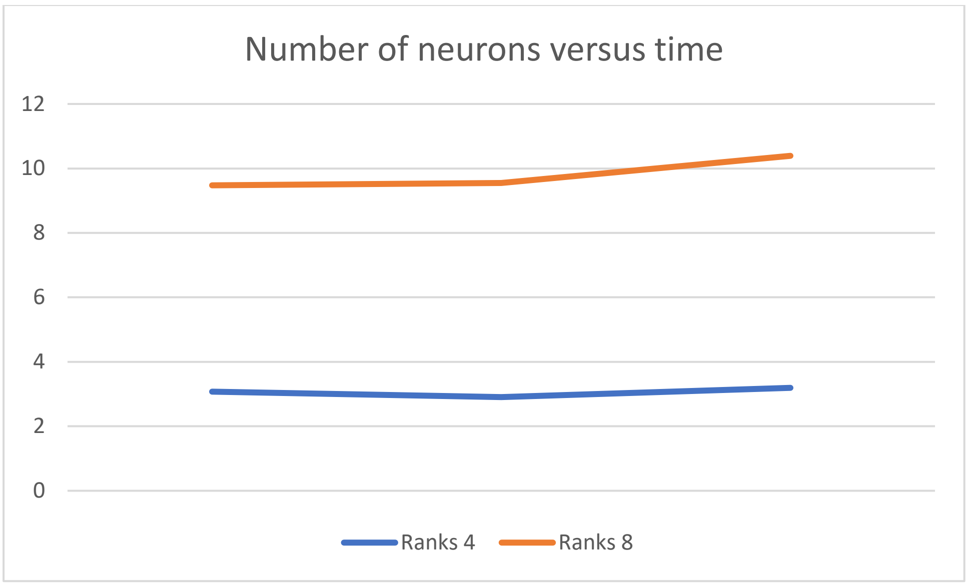


Figure 6 – A depiction of ranks 4 and ranks 8 plotted as neurons versus time in seconds.

Time for workers to complete their allotted matrix multiplications

Number of neurons	Number of ranks	Time (Seconds)
2	4	0.000006
8	4	0.000005
20	4	0.000011
2	8	0.000028
8	8	0.000031
20	8	0.000029

Once more this data shows us a strong connection between a sensibly small number of ranks and a well performing network. It also shows a connection between the number of neurons and the time taken to process in seconds. This can be seen in the ranks 4 data as 20 neurons takes just as long as both of the prior tests.

Time for master to finish allocating and sending out data

Number of neurons	Number of ranks	Time (Seconds)
2	4	0.000006
8	4	0.000007
20	4	0.000006
2	8	0.000045
8	8	0.000085
20	8	0.000017

This data shows no real correlation at the lower end however it shows the dangers of having ranks greater than the number of neurons. This is shown with the number of neurons 8 and ranks 8 performing extraordinarily poorly whereas neurons 20 performs very well as it can evenly divide.

4.0 Conclusions

4.1 CUDA

In conclusion CUDA scales fairly well however the parallelisation is massively handicapped due to limits in memory transfer speed - these limits as seen in the code profile take up a vast amount of computational time which should otherwise be spent computing the array. Whilst some problems were overcome with CUDA by taking a more memory heavy approach and pre-declaring arrays to increase speed ultimately memory transfer speeds between the CPU and GPU are a major downside. One upside however is that the system clearly has a lot of headroom – currently using the tiling technique resources are scarcely used – on my 980ti I observed roughly 15% utilisation at peak during load meaning that should the input node become larger if the algorithm were applied to say a deep neural network with more matrix multiplications of greater size this power could be leveraged.

Another con of the CUDA implementation is that it does not perform very well when presented with these narrow matrices – the tiling algorithm simply does not perform as well as it could when presented with the 1xN multiplications which are very common. As such further improvements to the algorithm could be made so as to tile better in these circumstances but as it currently stands with a low amount of inputs – CUDA performs poorly on the matrix multiplication acceleration.

One major pro of the CUDA implementation is that it did not hog CPU time – as the kernel call allowed for the CPU to not be doing the multiplication this left us with a roughly 12% utilisation which is very efficient and could allow for other tasks in the background simultaneously.

4.2 MPI

MPI performed very amicably and ultimately was consistently faster than CUDA in terms of outputting a working network. One of the main weaknesses of MPI however is that should the number of workable ranks available be greater than the number of divisible work it resorts to serialising large amounts of the algorithm massively slowing down the code but not releasing resources. Another downside is that MPI is a massive resource hog and lacks the control to intelligently invoke itself the same way the CUDA kernel can – as such it leads to more data being processed than necessary and unnecessary passing of data to ensure the multiplication produces the correct results. This also leads to with any large number of Hosts 100% CPU utilisation which is unfriendly to any other processes. One other positive beside the speed is the memory efficiency is greater in this instance – as the NVidia library seems to constantly force 200mb of ram instead MPI uses multiple hosts with a single MB of ram.

References:

AINT 351: Machine learning Lecture 7, Dr Ian Howard, 2018, Plymouth University - https://dle.plymouth.ac.uk/mod/folder/view.php?id=645674

https://skymind.ai/wiki/multilayer-perceptron

https://machinelearningmastery.com/neural-networks-crash-course/

https://en.wikipedia.org/wiki/Multilayer_perceptron