A Simple CUDA Neural Network

Kyle Salitrik (kps168), Tomoki Takasawa (tmt5336)

Abstract

This document covers the general purpose and workings of a neural network implemented to recognize handwritten digits. The MNIST dataset collected by the U.S. Government was used for all training and testing data. A mathematical background of neural networks is discussed followed by explicit discussion of the code implementation.

1 Introduction and Purpose

This program is designed to identify hand written numbers (0 9) by neural network digit classification technique. Generally, this technique requires enourmous amuont of run time. For training process, it calculates 5 matrix multiplications, 2 matrix subtractions, and 5 elemnt-wise vector operations. For verification, it requires 2 matrix multiplications and 2 element-wise operations for sigmoid function. In the case of all of the operations listed, the matrices and vectors are all densely populated. We decided to approach this problem by processing these operations in parallel on NVDIA's GPU processors.

2 Related Work

Due to the age of neural networks, which were conceived in the 1940s, there is a large amount of documentation to reference. For the purposes of this paper, the first and second chapters of the online book Neural Networks and Deep Learning (NNDL) can be used a supplemental material (referenced at the end of the paper). Figures used in the Neural Network Structure and Forward Propagation sections were sourced from this book.

3 Neural Network Background

This portion of the document briefly covers the background on the mathematical methods employed for neural networks.

3.1 Neural Network Structure

The general purpose of the neural network is identifying handwritten digits 28x28 in size. An example of the handwritten data is shown below. This dataset, called MNIST, is commonly used to benchmark neural network performance. The handwriting samples were obtained by the U.S. National Institute of Standards and Technology (NIST) from both census workers and high school students. This data is divided into 50,000 training samples, 10,000 test samples, and 10,000 validation samples.

Figure 1: MNIST Data, source: NNDL Book

Although it will be explained in the next section in detail, it is important to know that each neuron within the network will output a value between 0 and 1. This value is bound by what is called the activation function of the neuron. The input to the neural network was a matrix of 28x28 reshaped into a single column vector of size 784, with each neuron (vector element) representing a single pixel in the image. Initially the data is defined by a value of 0-127 for pixel intensity, but it is then bound to a value between 0 and 1 by simply dividing each element by 127.

The hidden (or second) layer size of 128 neurons was arbitrarily chosen. While the objective was to train a simple neural network with only one hidden layer. A deep neural network may consist of a network with two or more hidden layers, typically each layer will reduce in size with each step forward in the network closer to the desired output layer size. These hidden layer sizes can be tuned (known as meta-parameters) in order to change the behavior of the network. The final (output) layer of the network consists of 10 neurons, with each neuron (0 through 9) representing the corresponding numerical digit. This format results in a vector similar to the following:

$$[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$$

This representation is known as a 1-hot format, where the element that has a value of 1 indicates the corresponding digit as true. Referencing the above example, it represents the digit 3 (as element 3 has a value of 1). With regard to the output layer, the neuron with the highest value is chosen as the ”answer” by the network. The structure for the neural network was as follows:

- Layer 1 (input): 784 neurons
- Layer 2 (hidden layer): 128 neurons
- Layer 3 (output): 10 neurons

3.2 Forward Propagation

The behaivor of the neural network is defined by a function chosen by the network implementor. When a set of inputs are passed into a neuron layer (pictured below), each neuron in the layer computes its activation using this predetermined function. For the case of this assignment (and most simple neural networks), the sigmoid function was used. The sigmoid function helps us to bind the output from our neurons between 0 and 1 because we are looking for a (mostly) boolean answer: true or false for identifying a digit. This is where a significant portion of our computational cost comes from, as it includes a floating point exponential function, division and addition.

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{Sigmoid Function} \tag{1}$$

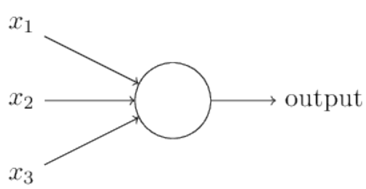

Figure 2: Neuron Diagram, source: NNDL Book

In our case, we use a densely connected neural network where every neuron in a layer is connected to every neuron in the following layer. It is useful to know that other types of layer relationships do exist such as sparsely connected layers, but are not explored here. These connections are defined by weight matrices where each element represents the connection from neuron A in layer L to neuron B in layer $L + 1$.

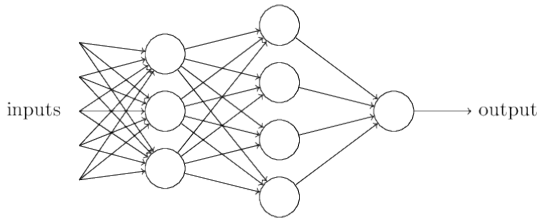

Figure 3: Network Diagram, source: NNDL Book

The majority of the computational costs arise from the vector-matrix multiplication between these dense weight matrices and the activations of the previous neurons. Each weight matrix is initalized to a random set of values and is adjusted during the final step (gradient descent) after each iteration through the network. In the following equations, $a^{(l)}$ represents the activation of layer l and $\theta^{(l)}$ represents the weights between $a^{(l)}$ and $a^{(l+1)}$

To compute a full pass of forward propagation, we must compute the following equations:

$$z^{(2)} = a^{(1)} * \theta^{(1)T}$$
$$a^{(2)} = \sigma\{z^{(2)}\}$$
$$z^{(3)} = a^{(2)} * \theta^{(2)T}$$
$$a^{(3)} = \sigma\{z^{(3)}\}$$

For our neural networks, the following are the weight matrix sizes:

- Weight Matrix 1: 128x784 elements
- Weight Matrix 2: 10x128 elements

At this point, the forward propagation has computed the estimated answer to a given input. Next, the result is evaluated in backpropagation which prepares for adjusting the weights to be more accurate.

3.3 Backward Propagation

The second step in training a neural network, as stated previously, is called backpropagation. This step evaluates the error in each layer’s activations. This information can be used simply as a metric evaluated by a defined cost function (not used for our purposes) or can be used to adjust the weights through various methods. The validation data is typically used for computing the cost function.

The first step in backpropagation is to evaluate how far off the network was from the expected result. To do this the following value is caluclated, where y is the known answer in one-hot format:

$$\delta^{(3)} = a^{(3)} - y$$

Because we know the error of the last layer, we can calculate the error contributed by each previous layer – excluding the input layer – by the following implicit function:

$$\delta^{(l)} = \widehat{\delta}^{(l+1)}\widehat{\Theta}^{(l)} \odot \sigma'(z^{(l)}) \tag{2}$$

In the previous equation $\odot$ represents the Hadamard product (element wise multiplication) for vectors, and $\sigma'(z)$ is the derivative of the sigmoid function:

$$\sigma'(z) = \quad \sigma(z) \odot (1 - \sigma(z)) \tag{3}$$

Knowing these functions, we must only evaluate (2) for the following, as we only have one more hidden layer:

$$\delta^{(2)} = \widehat{\delta}^{(3)}\widehat{\Theta}^{(2)} \odot \sigma'(z^2)$$

However, this is an extremely computationally costly function, including a floating point vector-matrix multiply, floating point subroutines, divides, additions, and multiplications, which is why parallel computing is very tempting to use. At this point we are ready to move on to the final step of training the network: gradient descent.

3.4 Gradient Descent

Gradient descent computes the gradient of the weights for each weight matrix and is usually adjusted by 1 divided by the number of data samples (m). The weight matrices are adjusted by these gradients in order to find a local minima. Before proceeding it is worth noting that gradient descent is not the ONLY method for training a network's weights, nor is it the best. One common pitfall includes overshooting the minima and ending up in a divergent inflation of the weights. This is why the 1/m adjustment is applied to the gradient.

The following equation is the computation for the gradient of each weight matrix:

$$\nabla \frac{\partial J}{\partial \Theta^{(l)}} = \frac{1}{m}\delta^{(l+1)T} * a^{(l)} \tag{4}$$

For the investigated network, the following gradients were computed after each forward propagation for each data example (image).

$$\nabla^{(1)} = \frac{1}{m}\delta^{(2)T} * a^{(1)}$$
$$\nabla^{(2)} = \frac{1}{m}\delta^{(3)T} * a^{(2)}$$

The final step in gradient descent is to adjust the weight matrices by the gradients:

$$W1 := W1 - \nabla^{(1)}$$
$$W2 := W2 - \nabla^{(2)}$$

3.5 Evaluating the Network

Typically to train the network, the training data is cycled through in its entirety (known as one epoch) and then shuffled to be run for another epoch. The number of epochs is dependent on the accuracy desired and the time allowed for computation. To test the network after the training epochs, the test data is run through the network and the number of correct predictions out of the number of total testing examples is reported to determine the networks accuracy. Using more advanced methods, networks for the MNIST data can obtain $> 90\%$ performance.

4 Code Implementation

Due to the amount of calculations described in the previous section, the code for all of the neural network computations was parallelized using CUDA. This section will describe each function implemented. All variables (excluding counts such as the number of elements in a matrix) are floats.

4.1 CUDA Matrix Kernels

The matrix kernels described next were all used by a helper function that ran the kernel. This helper function took in the size of the matrices to be used in the calculations, pointers to the matrices to be used in the computation, a flag stating which kernel should be run, and scalar multipliers used in the kernels defined below. This helper function allocated memory on the GPU device, copied from the host to the device, ran the kernel, copied the result back to the host and then freed the device memory. For the CUDA implementations, a 'matrix' was created as a dynamically allocated single-dimension array of length rows * columns.

4.1.1 Matrix Add

This kernel is mostly self explanatory; it computed the following equation, where α and β are floating point scalar multipliers.

$$\mathbf{C} = \alpha\mathbf{A} + \beta\mathbf{B}$$

4.1.2 Matrix Hadamard Product

The Hadamard product operation (denoted by $\odot$) computes the *element-wise* product of two matrices. In the following equations, the second equation explicitly states this product.

$$\mathbf{C} = \alpha\mathbf{A} \odot \beta\mathbf{B}$$
$$\mathbf{C}_{ij} = \alpha\mathbf{A}_{ij} * \beta\mathbf{B}_{ij}$$

4.1.3 Matrix Sigmoid Function

Depending on the implementation of a neural network, the sigmoid function may be required to be computed for a matrix or a vector. This implementation arises if one wants to compute gradient decent based on more than one sample at a time.

$$\mathbf{C} = \sigma(\mathbf{A}) = \frac{1}{1 + e^{-\mathbf{A}}}$$

4.1.4 Matrix Sigmoid Function Derivative

Similar to the sigmoid function, the sigmoid derivative may be required to be performed on a matrix or a vector. The first equation references the mathematical function implemented. The following two equations define how the calculations were performed.

$$\sigma'(\mathbf{A}) = \mathbf{C} = \sigma(\mathbf{A}) \odot (1 - \sigma(\mathbf{A}))$$
$$z = \sigma(\mathbf{A}_{ij})$$
$$\mathbf{C}_{ij} = z * 1 - z$$

4.1.5 Matrix Multiplication using CUBLAS

The matrix-matrix multiplication was implemented using it's own separate driver function due to the fact that the nVidia optimized CUBLAS (CUDA Basic Linear Algebra Subroutine) library was used. The CUBLAS library performs the following matrix multiplication and addition operation:

$$\mathbf{C} = \alpha\mathbf{A} * \mathbf{B} + \beta\mathbf{C}$$

Using the input arguments, matrix A or B can be transposed by the calculation.

4.2 CUDA Vector Kernels

For the vector kernels, a kernel driver function similar to the matrix kernel driver was implemented. In short, this driver allocates the memory for the kernel on the GPU, runs the kernel, then frees the memory after copying the values to be used in the computation and the result between the host and device. The one restriction on all vector kernels is that all vectors must be of the same length.

4.2.1 Vector Add

This kernel is mostly self explanatory; it computed the following equation, where α and β are floating point scalar multipliers.

$$\hat{C} = \alpha\hat{A} + \beta\hat{B}$$

4.2.2 Vector Hadamard Product

The vector Hadamard product (denoted by $\odot$) is identical to the matrix version. It computes the *element-wise* product of two vectors. In the following equations, the second equation explicitly states this product.

$$\hat{C} = \alpha\hat{A} \odot \beta\hat{B}$$
$$\hat{C}_i = \alpha\hat{A}_i * \beta\hat{B}_i$$

4.2.3 Vector Dot Product

This computes the dot product of two vectors and returns a scalar value.

$$C = \alpha\hat{A} \cdot \beta\hat{B}$$

4.2.4 Vector Sigmoid Function

As explained above, the sigmoid function must be performed on a vector if computing forward propagation of data samples one at a time.

$$\hat{C} = \sigma(\hat{A}) = \frac{1}{1 + e^{-\hat{A}}}$$

4.2.5 Vector Sigmoid Function Derivative

Again, the sigmoid function derivative for a single data sample forward propagation is shown. The first equation references the mathematical function implemented. The following two equations define how the calculations were performed.

$$\sigma'(\hat{A}) = \hat{C} = \sigma(\hat{A}) \odot (1 - \sigma(\hat{A}))$$
$$z = \sigma(\hat{A}_{ij})$$
$$\hat{C}_{ij} = z * 1 - z$$

4.3 Extra Functions Used

Helper functions for creating the structures and memory initialization were created and explained in this section.

4.3.1 Structure Functions

The functions of SetVectorSize and SetMatrixSize took in a structure of VectorSize and MatrixSize, respectively, and their associated arguments, and filled out all of the information within the structure as well as performing compatibility checks for the dimensions. The vector size function required the length of the vectors and the matrix size initializer required the height and width of each matrix to be used as arguments.

4.3.2 Memory Initialization Functions

A helper function was created to actually initialize the memory on the GPU for both vectors and matrices. The inputs for the vector function are a VectorSize structure, pointers to vector A and B on the host to be copied to the device as well as pointers to reference the vectors A, B and C allocated on the GPU. The function returned all 3 pointers via reference to the host after allocating the required space on the GPU and copying Vector A and B.

The function for allocating the matrix memory is identical except for the fact that it required a MatrixSize structure passed into the function in order to create the required memory spaces.

4.3.3 Testing and Utility Functions

The final functions implemented were for testing the kernels and utility operations. A test function was implemented to run the kernel drivers for vectors and matrices then display the output in order to verify the code was working properly. After completing the kernels and verifying them, this function was no longer used.

The data that was used by the network was stored in a CSV format created by a Python script. This data was read line by line into the input (a1) and expected value (y) vectors using a utility function that takes the file to read from and number of elements to read as arguments.

In order to actually perform the forward propagation, weights were initialized using a helper function (InitializeWeights) that takes a MatrixSize structure and reference to the matrix to initialize as input arguments. It then assigns a value between 0 and 1 to each element using the rand() function.

5 Network Training Results

For the network employed, we were only able to successfully run training with $\approx 20,000 - 30,000$ training samples and $10,000$ testing samples due to restrictions on the ACI-I cluster. One major issue encountered with the ACI-I nodes was that one person is capable of utilizing the entire CUDA device at once, so there was no way to guarantee obtaining enough memory for the computations. However on this limited subset of training data, the network was able to identify nearly 1/5th of the training data presented correctly on successful runs.

All things considered, using these standard methods with the full training samples and multiple (5-10) epochs typically results in a near 80% accuracy in digit identification, so 20% using only 20,000 samples is quite good. This is considering that if a network is trained using all 50,000 samples 10 times each, that results in a total of 500,000 iterations through backpropagation and gradient descent in order to train the network, so using only 4% of the number of training iterations to obtain nearly 25% of the performance is promising.

6 Alternate Approaches and Future Work

One alternate method is to train using the entire 50,000 samples at one time instead of iterating through forward propagation for each sample individually. It is possible that using this method with CUDA may improve performance significantly. It was not attempted in order to stay true to a traditional network training scenario.

Another change or approach is to include a bias term in all of of the network layers except the output layer. The bias term is set to be 1 and is not used when computing backpropagation performing gradient descent on the weights, however the weight matrices are increased by one in the dimension corresponding to the previous layer, shown below:

- Layer 1 (input): 784 neurons $\rightarrow$ 785 neurons
- Layer 2 (hidden layer): 128 neurons $\rightarrow$ 129 neurons
- Layer 3 (output): 10 neurons
- Weight Matrix 1: 128x784 $\rightarrow$ 128x785
- Weight Matrix 2: 10x128 elements $\rightarrow$ 10x129

The bias term was left out for simplicity in reading the data and calcuations.

Another common adjustment to the backpropagation and gradient descent methods used is called regularization. This regularization term λ is included to perturb the gradient in order to potentially find a lower local minima, especially if the gradient is small. A regularization term that is too large may cause divergence in the weights, so it is usually scaled by the number of samples in the dataset. Below are the equations for the gradient calcuation using regularization.

$$
\begin{aligned}
L^{[m \times m]} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\nabla \frac{\partial J}{\partial \Theta^{(l)}} &= \frac{1}{m} \Delta^{(l)} + \frac{\lambda}{m} \Theta^{(l)} L
\end{aligned}
$$

As one might expect, the computations used are fairly memory intensive. Future work would include increasing the memory efficiency as well as implementing bias terms and/or regularization.

References

Michael Nielsen. 2017. *Neural Networks and Deep Learning Book*. neuralnetworksanddeeplearning.com.

Code Appendix

```
// Compile using nvcc <file> -lcublas -o <output>
#include <cublas_v2.h>
#include <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream>
#include <sstream>
#include <iostream>

// Define block size for thread allocation
#define NUM_THREADS 32 // 32 is max for N^2 threads: 32*32 = 1024
#define LOGGING 0

//=============================================================================
//=== Structure definitions
//=============================================================================
typedef struct _kernelParams {
    int block_size;
    int grid_size;
} sKernelParams;

typedef struct _matrixSize // Optional Command-line multiplier for matrix sizes
{
    unsigned int A_height, A_width, B_height, B_width, C_height, C_width;
} MatrixSize;

typedef struct _vSize // Optional Command-line multiplier for matrix sizes
{
    unsigned int len_A, len_B, len_C;
} VectorSize;

//=============================================================================
//=== Structure functions
//=============================================================================

/**
 * @brief - sets values of vector size structure
 *
 * @param vector_size - pointer to vector size struct
 * @param len - length of all vectors
 */
void SetVectorSize(VectorSize *vector_size, unsigned int len) {
    vector_size->len_A = len;
    vector_size->len_B = len;
    vector_size->len_C = len;

    if (LOGGING == 1)
        fprintf(stdout, "Vector A(%u), Vector B(%u), Vector C(%u)\n",
                vector_size->len_A,
                vector_size->len_B,
                vector_size->len_C);

    if (vector_size->len_A != vector_size->len_B ||
        vector_size->len_B != vector_size->len_C ||
        vector_size->len_C != vector_size->len_A) {
        fprintf(stderr, "ERROR: Vector lengths do not match!\n");
        exit(-1);
    }
}

/**
 * @brief - sets values of matrix size structure
 *
 * @param matrixSize - reference to matrix size struct
 * @param widthA - width of matrix A
 * @param heightA - height of matrix A
 * @param widthB - width of matrix B
 * @param heightB - height of matrix B
```

```
 * @param widthC - width of matrix C
 * @param heightC - height of matrix C
 */
void SetMatrixSize(MatrixSize *matrixSize,
                   unsigned int widthA, unsigned int heightA,
                   unsigned int widthB, unsigned int heightB,
                   unsigned int widthC, unsigned int heightC) {
    matrixSize->A_height = heightA;
    matrixSize->A_width = widthA;
    matrixSize->B_height = heightB;
    matrixSize->B_width = widthB;
    matrixSize->C_height = heightC;
    matrixSize->C_width = widthC;

    if (LOGGING == 1)
        fprintf(stdout, "Matrix A(%u x %u), Matrix B(%u x %u), Matrix C(%u x %u)\n",
                matrixSize->A_width,
                matrixSize->A_height,
                matrixSize->B_width,
                matrixSize->B_height,
                matrixSize->C_width,
                matrixSize->C_height);

}

//==============================================================================
//=== GPU memory initialization functions
//==============================================================================

/**
 * @brief - allocates memory on GPU for vectors A, B, and C then copies the values for vector A and B
 *          from host PC onto the device
 *
 * @param argc - from compiler
 * @param argv - from compiler
 * @param devID - device ID number
 * @param vector_size - reference to vector size structure
 * @param host_vA - pointer to host vector A (with values)
 * @param host_vB - pointer to host vector B (with values)
 * @param dev_A - pointer to vector A device memory reference
 * @param dev_B - pointer to vector B device memory reference
 * @param dev_C - pointer to vector C device memory reference
 */
void VectorInitCUDA(int argc, char **argv, int devID, VectorSize *vector_size, float *host_vA, float *host_vB,
                    float *&dev_A, float *&dev_B, float *&dev_C) {
    // Assign CUDA variables
    cudaError_t err;

    // Assign size variables
    size_t size_A = vector_size->len_A * sizeof(float);
    size_t size_B = vector_size->len_B * sizeof(float);
    size_t size_C = vector_size->len_C * sizeof(float);

    // Allocate memory on GPU
    err = cudaMalloc((void **) &dev_A, size_A);
    if (err != cudaSuccess) fprintf(stderr, "ERROR allocating vector A: %s\n", cudaGetErrorString(err));
    err = cudaMalloc((void **) &dev_B, size_B);
    if (err != cudaSuccess) fprintf(stderr, "ERROR allocating vector B: %s\n", cudaGetErrorString(err));
    err = cudaMalloc((void **) &dev_C, size_C);
    if (err != cudaSuccess) fprintf(stderr, "ERROR allocating vector C: %s\n", cudaGetErrorString(err));

    // Copy data from host PC to GPU
    err = cudaMemcpy(dev_A, host_vA, size_A, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) fprintf(stderr, "ERROR copying vector A to GPU: %s\n", cudaGetErrorString(err));
```

```
    err = cudaMemcpy(dev_B, host_vB, size_B, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) fprintf(stderr, "ERROR copying vector B to GPU: %s\n", cudaGetErrorString(err));

}

/**
 * @brief - allocates memory on GPU for matrices A, B, and C then copies the values for matrices A, B and C
 *          from host PC onto the device
 *
 * @param argc - from compiler
 * @param argv - from compiler
 * @param devID - device ID number
 * @param matrixSize - reference to vector size structure
 * @param host_matrixA - pointer to host matrix A (with values)
 * @param host_matrixB - pointer to host matrix B (with values)
 * @param host_matrixC - pointer to host matrix C (with values)
 * @param dev_matrixA - pointer to matrix A device memory reference
 * @param dev_matrixB - pointer to matrix B device memory reference
 * @param dev_matrixC - pointer to matrix C device memory reference
 */
void MatrixInitCUDA(int argc, char **argv, int &devID, MatrixSize *matrixSize,
                    float *host_matrixA, float *host_matrixB, float *host_matrixC,
                    float *&dev_matrixA, float *&dev_matrixB, float *&dev_matrixC) {
    // Assign CUDA variables
    cudaError_t err;

    // Assign size variables
    size_t matrixA_size = matrixSize->A_height * matrixSize->A_width * sizeof(float);
    size_t matrixB_size = matrixSize->B_height * matrixSize->B_width * sizeof(float);
    size_t matrixC_size = matrixSize->C_height * matrixSize->C_width * sizeof(float);

    // Allocate memory on GPU
    err = cudaMalloc((void **) &dev_matrixA, matrixA_size);
    if (err != cudaSuccess) fprintf(stderr, "ERROR allocating matrix A: %s\n", cudaGetErrorString(err));
    err = cudaMalloc((void **) &dev_matrixB, matrixB_size);
    if (err != cudaSuccess) fprintf(stderr, "ERROR allocating matrix B: %s\n", cudaGetErrorString(err));
    err = cudaMalloc((void **) &dev_matrixC, matrixC_size);
    if (err != cudaSuccess) fprintf(stderr, "ERROR allocating matrix C: %s\n", cudaGetErrorString(err));

    // Copy data from host PC to GPU
    err = cudaMemcpy(dev_matrixA, host_matrixA, matrixA_size, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix A to GPU: %s\n", cudaGetErrorString(err));
    err = cudaMemcpy(dev_matrixB, host_matrixB, matrixB_size, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix B to GPU: %s\n", cudaGetErrorString(err));
    err = cudaMemcpy(dev_matrixC, host_matrixC, matrixC_size, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix C to GPU: %s\n", cudaGetErrorString(err));
}

//==========================================================================
//=== CUDA Vector Kernels
//==========================================================================
/**
 * @required ALL VECTORS MUST BE THE SAME LENGTH
 * @brief - kernel for GPU computation of a vector addition
 * @param dev_vecA - pointer to device memory for vector A
 * @param dev_vecB - pointer to device memory for vector B
```

```
 * @param dev_vecC - pointer to device memory for vector C
 * @param alpha - multiplier for values in vector A
 * @param beta - multiplier for values in vector B
 * @param vecLen - length of all vectors
 */
__global__ void VectorAdditionKernel(float *dev_vecA, float *dev_vecB, float *
    dev_vecC,
                                     float alpha, float beta, int vecLen) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < vecLen) {
        dev_vecC[i] = alpha * dev_vecA[i] + beta * dev_vecB[i];
    }
}

/**
 * @required ALL VECTORS MUST BE THE SAME LENGTH
 * @brief - kernel for GPU computation of a vector hadamard product
 * @param dev_vecA - pointer to device memory for vector A
 * @param dev_vecB - pointer to device memory for vector B
 * @param dev_vecC - pointer to device memory for vector C
 * @param alpha - multiplier for values in vector A
 * @param beta - multiplier for values in vector B
 * @param vecLen - length of all vectors
 */
__global__ void VectorHadamardKernel(float *dev_vecA, float *dev_vecB, float *
    dev_vecC,
                                     float alpha, float beta, int vecLen) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < vecLen) {
        dev_vecC[i] = alpha * dev_vecA[i] * beta * dev_vecB[i];
    }
}

/**
 * @required ALL VECTORS MUST BE THE SAME LENGTH
 *           REMEMBER: Call kernel using: <<<grid, threads, vecLen>>>
 * @brief - kernel for GPU computation of a vector dot product
 * @param dev_vecA - pointer to device memory for vector A
 * @param dev_vecB - pointer to device memory for vector B
 * @param result - pointer to a single float value where the result will be returned
 * @param alpha - multiplier for values in vector A
 * @param beta - multiplier for values in vector B
 * @param vecLen - length of all vectors
 */
__global__ void VectorDotProduct(float *dev_vecA, float *dev_vecB, float *result,
                                 float alpha, float beta, int vecLen) {
    extern __shared__ float temp[];
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < vecLen) {
        temp[i] = alpha * dev_vecA[i] * beta * dev_vecB[i];
    }
    __syncthreads();
    if (threadIdx.x == 0) {
        float sum = 0.0;
        for (int j = 0; j < vecLen; j++) {
            sum += temp[j];
        }
        result[0] = sum;
    }
}

/**
 * @required INPUT AND OUTPUT VECTORS MUST BE THE SAME LENGTH
 * @brief - kernel for GPU computation of the vector sigmoid function
 * @param dev_matrixA - pointer to device memory for vector A
 * @param dev_matrixC - pointer to device memory for vector C
 * @param vecLen - length of all vectors
 */
__global__ void VectorSigmoid(float *dev_vecA, float *dev_vecC, int vecLen) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
```

```
    if (index < vecLen) {
        float exp = 1 + expf(-dev_vecA[index]);
        dev_vecC[index] = 1 / exp;
    }
}

/**
 * @required INPUT AND OUTPUT VECTORS MUST BE THE SAME LENGTH
 * @brief - kernel for GPU computation of the vector sigmoid derivative function
 * @param dev_matrixA - pointer to device memory for vector A
 * @param dev_matrixC - pointer to device memory for vector C
 * @param vecLen - length of all vectors
 */
__global__ void VectorSigmoidDerivative(float *dev_vecA, float *dev_vecC, int vecLen) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index < vecLen) {
        float exp = 1 + expf(-dev_vecA[index]);
        float sig = 1 / exp;
        dev_vecC[index] = sig * (1 - sig);
    }
}

//==========================================================================
//=== CUDA Vector Kernel Drivers
//==========================================================================

/**
 * @brief driver function for computing vector operations
 * @param argc - from compiler
 * @param argv - from compiler
 * @param devID - device ID number
 * @param vectorSize - reference to vector size structure
 * @param operation - switch-case value for which matrix operation to perform
 *                     1: Vector addition
 *                     2: Vector Hadamard product
 *                     3: Vector dot product
 *                     4: Vector sigmoid function
 *                     5: Vector sigmoid derivative
 * @param host_vectorA - pointer to host vector A (with values)
 * @param host_vectorB - pointer to host vector B (with values)
 * @param host_vectorC - pointer to host vector C (with values)
 * @param alpha - multiplier for values in vector A
 * @param beta - multiplier for values in vector B
 */
void RunVectorKernel(int argc, char **argv, int &devID, VectorSize *vectorSize, int operation,
                     float *host_vectorA, float *host_vectorB, float *host_vectorC, float alpha, float beta) {
    // Assign CUDA variables
    cudaError_t err;
    dim3 threads(NUM_THREADS, NUM_THREADS);
    int gridX = (int) ceil((float) vectorSize->len_C / (float) threads.x);
    int gridY = (int) ceil((float) vectorSize->len_C / (float) threads.y);
    dim3 grid((unsigned int) gridX, (unsigned int) gridY);

    // Assign computation variables
    float *dev_vectorA = NULL;
    float *dev_vectorB = NULL;
    float *dev_vectorC = NULL;

    size_t vectorC_size = vectorSize->len_C * sizeof(float);

    // Initialize memory on GPU
    VectorInitCUDA(argc, argv, devID, vectorSize, host_vectorA, host_vectorB, dev_vectorA, dev_vectorB, dev_vectorC);

    switch (operation) {
        case 1: {
            // Compute vector addition
```

```
            VectorAdditionKernel<<<grid, threads>>>(dev_vectorA, dev_vectorB, dev_vectorC, alpha, beta,
                    vectorSize->len_C);
            err = cudaGetLastError();
            if (err != cudaSuccess) fprintf(stderr, "ERROR in Vector Add Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        case 2: {
            // Compute vector Hadamard Product
            VectorHadamardKernel<<<grid, threads>>>(dev_vectorA, dev_vectorB, dev_vectorC, alpha, beta,
                    vectorSize->len_C);
            err = cudaGetLastError();
            if (err != cudaSuccess)
                fprintf(stderr, "ERROR in Vector Hadamard Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        case 3: {
            // Compute vector dot product
            VectorDotProduct<<<grid, threads, vectorSize->len_C>>>
                                                   (dev_vectorA, dev_vectorB, dev_vectorC, alpha, beta, vectorSize->len_C);
            err = cudaGetLastError();
            if (err != cudaSuccess)
                fprintf(stderr, "ERROR in Vector Dot product Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        case 4: {
            // Compute sigmoid function
            VectorSigmoid<<<grid, threads>>>(dev_vectorA, dev_vectorC, vectorSize->len_C);
            err = cudaGetLastError();
            if (err != cudaSuccess)
                fprintf(stderr, "ERROR in Vector Sigmoid Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        case 5: {
            // Compute sigmoid derivative
            VectorSigmoidDerivative<<<grid, threads>>>(dev_vectorA, dev_vectorC, vectorSize->len_C);
            err = cudaGetLastError();
            if (err != cudaSuccess)
                fprintf(stderr, "ERROR in Vector Sigmoid Derivative Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        default: {
            fprintf(stderr, "ERROR: No vector kernel selected. Operation Aborted");
            break;
        }
    }

    // Make sure device is finished
    err = cudaDeviceSynchronize();
    if (err != cudaSuccess)
        fprintf(stderr, "ERROR synchronizing Vector Kernel calculation: %s\n", cudaGetErrorString(err));

    // Copy data from GPU to host PC
    err = cudaMemcpy(host_vectorC, dev_vectorC, vectorC_size, cudaMemcpyDeviceToHost);
    if (err != cudaSuccess)
        fprintf(stderr, "ERROR copying vector C to Host: %s\n", cudaGetErrorString(err));

    // Free GPU memory
```

```
    err = cudaFree(dev_vectorA);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing vector A on GPU: %s\n", cudaGetErrorString(err));
    err = cudaFree(dev_vectorB);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing vector B on GPU: %s\n", cudaGetErrorString(err));
    err = cudaFree(dev_vectorC);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing vector C on GPU: %s\n", cudaGetErrorString(err));
    err = cudaDeviceSynchronize();
    if (err != cudaSuccess) fprintf(stderr, "ERROR synchronizing Vector Kernel End: %s\n", cudaGetErrorString(err));
    if (LOGGING == 1) fprintf(stdout, "Vector Kernel finished.\n");
}

//==================================================================================
//=== CUDA Matrix Kernels
//==================================================================================

/**
 * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
 * @brief - kernel for GPU computation of matrix additions
 * @param dev_matrixA - pointer to device memory for matrix A
 * @param dev_matrixB - pointer to device memory for matrix B
 * @param dev_matrixC - pointer to device memory for matrix C
 * @param alpha - multiplier for values in matrix A
 * @param beta - multiplier for values in matrix B
 * @param matrix_width - width of all matrices
 * @param matrix_height - height of all matrices
 */
__global__ void MatrixAddKernel(float *dev_matrixA, float *dev_matrixB, float *dev_matrixC,
                                float alpha, float beta, int matrix_width, int matrix_height) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    int index = col + row * matrix_height;
    if (col < matrix_width && row < matrix_height) {
        dev_matrixC[index] = alpha * dev_matrixA[index] + beta * dev_matrixB[index];
    }
}

/**
 * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
 * @brief - kernel for actual GPU computation for the matrix Hadamard product
 * @param dev_matrixA - pointer to device memory for matrix A
 * @param dev_matrixB - pointer to device memory for matrix B
 * @param dev_matrixC - pointer to device memory for matrix C
 * @param alpha - multiplier for values in matrix A
 * @param beta - multiplier for values in matrix B
 * @param matrix_width - width of all matrices
 * @param matrix_height - height of all matrices
 */
__global__ void MatrixHadamardKernel(float *dev_matrixA, float *dev_matrixB, float *dev_matrixC,
                                     float alpha, float beta, int matrix_width, int matrix_height) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    int index = col + row * matrix_height;
    if (col < matrix_width && row < matrix_height) {
        dev_matrixC[index] = alpha * dev_matrixA[index] * beta * dev_matrixB[index];
    }
}

/**
 * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
 * @brief - kernel for GPU computation of matrix sigmoid function
 * @param dev_matrixA - pointer to device memory for matrix A
 * @param dev_matrixC - pointer to device memory for matrix C
```

```
 * @param matrix_width - width of all matrices
 * @param matrix_height - height of all matrices
 */
__global__ void MatrixSigmoid(float *dev_matrixA, float *dev_matrixC,
                              int matrix_width, int matrix_height) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    int index = col + row * matrix_height;
    if (col < matrix_width && row < matrix_height) {
        float exp = 1 + expf(-dev_matrixA[index]);
        dev_matrixC[index] = 1 / exp;
    }
}

/**
 * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
 * @brief - kernel for GPU computation of the matrix sigmoid derivative function
 * @param dev_matrixA - pointer to device memory for matrix A
 * @param dev_matrixC - pointer to device memory for matrix C
 * @param matrix_width - width of all matrices
 * @param matrix_height - height of all matrices
 */
__global__ void MatrixSigmoidDerivative(float *dev_matrixA, float *dev_matrixC,
                                        int matrix_width, int matrix_height) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    int index = col + row * matrix_height;
    if (col < matrix_width && row < matrix_height) {
        float exp = 1 + expf(-dev_matrixA[index]);
        float sig = 1 / exp;
        dev_matrixC[index] = sig * (1 - sig);
    }
}

//==============================================================================
//=== CUDA Matrix Kernel Drivers
//==============================================================================

/**
 * @brief -  Uses CUBLAS library to perform alpha(A x B) + beta(C) matrix
    multiplication and addition
 * @param argc - from compiler
 * @param argv - from compiler
 * @param devID - device ID number
 * @param matrixSize - reference to vector size structure
 * @param host_matrixA - pointer to host matrix A (with values)
 * @param host_matrixB - pointer to host matrix B (with values)
 * @param host_matrixC - pointer to host matrix C (with values)
 * @param alpha - value for alpha in CUBLAS function
 * @param beta - value for beta in CUBLAS function
 * @param transposeA - true if A should be transposed
 * @param transposeB - true if B should be transposed
 */

void MatrixMultiplyCUBLAS(int argc, char **argv, int &devID, MatrixSize *matrixSize,
                          float *host_matrixA, float *host_matrixB, float *
    host_matrixC,
                          float alpha, float beta, bool transposeA, bool transposeB)
     {
    // Assign CUDA variables
    cublasHandle_t handle;
    cudaError_t err;
    cublasCreate(&handle);
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, devID);
    dim3 threads(NUM_THREADS, NUM_THREADS);
    dim3 grid(matrixSize->C_width / threads.x, matrixSize->C_height / threads.y);

    // Assign computation variables
    float *dev_matrixA = NULL, *dev_matrixB = NULL, *dev_matrixC = NULL;
```

```
    int m = matrixSize->A_height;
    int n = matrixSize->B_width;
    int k = matrixSize->A_width;
    cublasOperation_t transA = CUBLAS_OP_N, transB = CUBLAS_OP_N;
    if (transposeA) transA = CUBLAS_OP_T;
    if (transposeB) transB = CUBLAS_OP_T;
    size_t matrixC_size = matrixSize->C_height * matrixSize->C_width * sizeof(float);

    // Initialize memory on GPU
    MatrixInitCUDA(argc, argv, devID, matrixSize,
                   host_matrixA, host_matrixB, host_matrixC,
                   dev_matrixA, dev_matrixB, dev_matrixC);

    // Perform matrix multiplication
    // SGEMM PARAMS: (handle, transposeA, transposeB, m, n, k, alpha, matrix A, k, matrix B, n, beta, matrix C, n)
    cublasSgemm(handle, transA, transB, m, n, k, &alpha, dev_matrixA, m,
                dev_matrixB, n, &beta, dev_matrixC, m);
    err = cudaGetLastError();
    if (err != cudaSuccess) fprintf(stderr, "ERROR in SGEMM: %s\n", cudaGetErrorString(err));

    // Make sure device is finished
    err = cudaDeviceSynchronize();
    if (err != cudaSuccess) fprintf(stderr, "ERROR synchronizing SGEMM calculation: %s\n", cudaGetErrorString(err));

    // Copy data from GPU to host PC
    err = cudaMemcpy(host_matrixC, dev_matrixC, matrixC_size, cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix C to Host: %s\n", cudaGetErrorString(err));

    // Free GPU memory
    err = cudaFree(dev_matrixA);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix A on GPU: %s\n", cudaGetErrorString(err));
    err = cudaFree(dev_matrixB);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix B on GPU: %s\n", cudaGetErrorString(err));
    err = cudaFree(dev_matrixC);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix C on GPU: %s\n", cudaGetErrorString(err));
    err = cudaDeviceSynchronize();
    if (err != cudaSuccess) fprintf(stderr, "ERROR synchronizing SGEMM end: %s\n", cudaGetErrorString(err));
    if (LOGGING == 1) fprintf(stdout, "Matrix Kernel finished.\n");
}

/**
 * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
 * @brief driver function for computing the matrix operations
 * @param argc - from compiler
 * @param argv - from compiler
 * @param devID - device ID number
 * @param matrixSize - reference to matrix size structure
 * @param operation - switch-case value for which matrix operation to perform
 *                    1: Matrix addition
 *                    2: Matrix Hadamard product
 *                    3: Sigmoid function
 *                    4: Sigmoid derivative
 * @param host_matrixA - pointer to host matrix A (with values)
 * @param host_matrixB - pointer to host matrix B (with values)
 * @param host_matrixC - pointer to host matrix C (with values)
 * @param alpha - multiplier for values in matrix A
 * @param beta - multiplier for values in matrix B
 */
void RunMatrixKernel(int argc, char **argv, int &devID, MatrixSize *matrixSize, int operation,
```

```
                    float *host_matrixA, float *host_matrixB, float *host_matrixC, float alpha, float beta) {
    // Assign CUDA variables
    cudaError_t err;
    dim3 threads(NUM_THREADS, NUM_THREADS);
    int gridX = (int) ceil((float) matrixSize->C_width / (float) threads.x);
    int gridY = (int) ceil((float) matrixSize->C_height / (float) threads.y);
    dim3 grid((unsigned int) gridX, (unsigned int) gridY);

    // Assign computation variables
    float *dev_matrixA = NULL, *dev_matrixB = NULL, *dev_matrixC = NULL;
    size_t matrixC_size = matrixSize->C_height * matrixSize->C_width * sizeof(float);

    // Initialize memory on GPU
    MatrixInitCUDA(argc, argv, devID, matrixSize,
                   host_matrixA, host_matrixB, host_matrixC,
                   dev_matrixA, dev_matrixB, dev_matrixC);

    switch (operation) {
        case 1: {
            // Compute Matrix Addition
            MatrixAddKernel<<<grid, threads>>>(dev_matrixA, dev_matrixB, dev_matrixC, alpha, beta,
                matrixSize->C_width, matrixSize->C_height);
            err = cudaGetLastError();
            if (err != cudaSuccess) fprintf(stderr, "ERROR in Matrix Add Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        case 2: {
            // Compute Hadamard Product
            MatrixHadamardKernel<<<grid, threads>>>(dev_matrixA, dev_matrixB, dev_matrixC, alpha, beta,
                matrixSize->C_width, matrixSize->C_height);
            err = cudaGetLastError();
            if (err != cudaSuccess)
                fprintf(stderr, "ERROR in Matrix Hadamard Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        case 3: {
            // Compute Sigmoid function
            MatrixSigmoid<<<grid, threads>>>(dev_matrixA, dev_matrixC, matrixSize->C_width, matrixSize->C_height);
            err = cudaGetLastError();
            if (err != cudaSuccess)
                fprintf(stderr, "ERROR in Matrix Sigmoid Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        case 4: {
            // Compute Sigmoid derivative function
            MatrixSigmoidDerivative<<<grid, threads>>>
                                        (dev_matrixA, dev_matrixC, matrixSize->C_width, matrixSize->C_height);
            err = cudaGetLastError();
            if (err != cudaSuccess)
                fprintf(stderr, "ERROR in Matrix Sigmoid Derivative Computation: %s\n", cudaGetErrorString(err));
            break;
        }
        default: {
            fprintf(stderr, "ERROR: No matrix kernel selected. Operation Aborted");
            break;
        }
    }

    // Make sure device is finished
    err = cudaDeviceSynchronize();
```

```
    if (err != cudaSuccess)
        fprintf(stderr, "ERROR synchronizing Matrix Kernel calculation: %s\n",
    cudaGetErrorString(err));

    // Copy data from GPU to host PC
    err = cudaMemcpy(host_matrixC, dev_matrixC, matrixC_size, cudaMemcpyDeviceToHost
    );
    if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix C to Host: %s\n",
    cudaGetErrorString(err));

    // Free GPU memory
    err = cudaFree(dev_matrixA);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix A on GPU: %s\n",
    cudaGetErrorString(err));
    err = cudaFree(dev_matrixB);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix B on GPU: %s\n",
    cudaGetErrorString(err));
    err = cudaFree(dev_matrixC);
    if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix C on GPU: %s\n",
    cudaGetErrorString(err));
    err = cudaDeviceSynchronize();
    if (err != cudaSuccess) fprintf(stderr, "ERROR synchronizing Matrix Kernel end:
    %s\n", cudaGetErrorString(err));
    if (LOGGING == 1) fprintf(stdout, "Matrix Kernel finished.\n");

}

//==================================================================================
//=== Test Function
//==================================================================================

void runTest(int argc, char **argv, int devID) {
    int N = 10;
    float *host_A, *host_B, *host_C, *host_D;
    float *host_vA, *host_vB, *host_vC, *host_vD, *host_vE;

    // Create matrices
    MatrixSize *testMatrixSize = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
    size_t calcSize = N * N * sizeof(float);
    host_A = (float *) calloc(calcSize, 1);
    host_B = (float *) calloc(calcSize, 1);
    host_C = (float *) calloc(calcSize, 1);
    host_D = (float *) calloc(calcSize, 1);
    SetMatrixSize(testMatrixSize, N, N, N, N, N + 1, N + 1);

    // Create vectors
    VectorSize *testVectorSize = (VectorSize *) calloc(sizeof(VectorSize), 1);
    size_t calcSize_V = N * sizeof(float);
    host_vA = (float *) calloc(calcSize_V, 1);
    host_vB = (float *) calloc(calcSize_V, 1);
    host_vC = (float *) calloc(calcSize_V, 1);
    host_vD = (float *) calloc(calcSize_V, 1);
    host_vE = (float *) calloc(calcSize_V, 1);
    SetVectorSize(testVectorSize, N);

    // Initialize matrix values
    for (int i = 0; i < N * N; i++) {
        host_A[i] = (float) i;
        host_B[i] = (float) i;
    }

    // Initialize vector values
    for (int i = 0; i < N; i++) {
        host_vA[i] = (float) i;
        host_vB[i] = (float) i;
    }

    // MATRIX TESTS

    if (LOGGING == 1) {
```

```
            fprintf(stdout, "Matrix A:\n");
            for (int i = 0; i < N + 1; i++) {
                for (int j = 0; j < N + 1; j++) {
                    fprintf(stdout, "%6.0f ", host_A[i * j]);
                }
                fprintf(stdout, "\n");
            }

            fprintf(stdout, "\nMatrix B:\n");
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    fprintf(stdout, "%6.0f ", host_B[i * j]);
                }
                fprintf(stdout, "\n");
            }
        }

        RunMatrixKernel(argc, argv, devID, testMatrixSize, 3, host_A, host_B, host_C,
        1.0, 1.0);
        RunMatrixKernel(argc, argv, devID, testMatrixSize, 4, host_A, host_B, host_D,
        1.0, 1.0);

        if (LOGGING == 1) {
            fprintf(stdout, "\nMatrix C:\n");
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    fprintf(stdout, "%6.10f ", host_C[i * j]);
                }
                fprintf(stdout, "\n");
            }
            fprintf(stdout, "\nMatrix D:\n");
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    fprintf(stdout, "%6.10f ", host_D[i * j]);
                }
                fprintf(stdout, "\n");
            }
        }

        // VECTOR TESTS

        if (LOGGING == 1) {
            fprintf(stdout, "Vector A:\n");
            for (int i = 0; i < N; i++) {
                fprintf(stdout, "%6.0f ", host_vA[i]);
            }
            fprintf(stdout, "\n");
            fprintf(stdout, "\nVector B:\n");

            for (int i = 0; i < N; i++) {
                fprintf(stdout, "%6.0f ", host_vB[i]);
            }
            fprintf(stdout, "\n");
        }

        RunVectorKernel(argc, argv, devID, testVectorSize, 3, host_vA, host_vB, host_vC,
         1.0, 1.0);
        RunVectorKernel(argc, argv, devID, testVectorSize, 4, host_vA, host_vB, host_vD,
         1.0, 1.0);
        RunVectorKernel(argc, argv, devID, testVectorSize, 5, host_vA, host_vB, host_vE,
         1.0, 1.0);

        if (LOGGING == 1) {
            fprintf(stdout, "Vector C:\n");
            for (int i = 0; i < N; i++) {
                fprintf(stdout, "%6.0f ", host_vC[i]);
            }
            fprintf(stdout, "\n");
            fprintf(stdout, "\nVector D:\n");

```

```
        for (int i = 0; i < N; i++) {
            fprintf(stdout, "%6.10f ", host_vD[i]);
        }
        fprintf(stdout, "\n");
        fprintf(stdout, "\nVector E:\n");

        for (int i = 0; i < N; i++) {
            fprintf(stdout, "%6.10f ", host_vE[i]);
        }
        fprintf(stdout, "\n");
    }
}

//==============================================================================
//=== Utility Functions
//==============================================================================

void ReadCSV(std::ifstream &file, int elements, float *array)
{
    std::string csvData;
    getline(file, csvData);

    std::istringstream dataStream(csvData);

    for (int col = 0; col < elements; col++){
        std::string value;
        getline(dataStream, value, ',');
        if ( !dataStream.good() )
            break;
        std::istringstream convertor(value);
        convertor >> array[col];
    }
}

void InitializeWeights(float *weights, MatrixSize *dims)
{
    int cols = dims->C_width;
    int rows = dims->C_height;
    int numEl = cols*rows;
    for(int idx = 0; idx < numEl; idx++)
    {
        weights[idx] = ((float) rand() / (RAND_MAX));
    }
}

//==============================================================================
//=== Main Function
//==============================================================================

/**
 * @brief computes weight matrices for a shallow neural network
 * @param argc - from compiler
 * @param argv - from compiler
 * @return 0 if success
 */
int main(int argc, char **argv) {
    // Assign CUDA variables
    int devID = 0;
    cudaGetDevice(&devID);
    cudaError_t mainErr;
    //runTest(argc, argv, devID);

    // Define NN layer lengths
    unsigned int layer_1 = 784;
    unsigned int layer_2 = 128;
    unsigned int layer_3 = 10;

    // Allocate memory for matrices and vectors
    float *a1, *a2, *a3;     // Activation vectors
    float *z2, *z3;          // Pre-sigmoid intermediary vectors
```

```
    float *W1, *W2;          // Weight matrices
    float *y;                // One-hot result vector
    float *del3, *del2;      // Error vectors
    float *scratch1, *scratch2;      // Error vectors
    float *Del2, *Del1;      // Error gradients

    a1 = (float *) calloc((size_t) layer_1, sizeof(float));
    a2 = (float *) calloc((size_t) layer_2, sizeof(float));
    a3 = (float *) calloc((size_t) layer_3, sizeof(float));
    z2 = (float *) calloc((size_t) layer_2, sizeof(float));
    z3 = (float *) calloc((size_t) layer_3, sizeof(float));
    y = (float *) calloc((size_t) layer_3, sizeof(float));
    W1 = (float *) calloc((size_t) layer_2 * layer_1, sizeof(float));
    W2 = (float *) calloc((size_t) layer_3 * layer_2, sizeof(float));
    del3 = (float *) calloc((size_t) layer_3, sizeof(float));
    del2 = (float *) calloc((size_t) layer_2, sizeof(float));
    scratch1 = (float *) calloc((size_t) layer_2, sizeof(float));
    scratch2 = (float *) calloc((size_t) layer_2, sizeof(float));
    Del2 = (float *) calloc((size_t) layer_3 * layer_2, sizeof(float));
    Del1 = (float *) calloc((size_t) layer_2 * layer_1, sizeof(float));

    // Initialize vector and matrix size structures for computation
    MatrixSize *inter2 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
    MatrixSize *inter3 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
    MatrixSize *grad1 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
    MatrixSize *grad2 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
    MatrixSize *backprop1 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
    MatrixSize *backprop2 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);

    VectorSize *activation2 = (VectorSize *) calloc(sizeof(VectorSize), 1);
    VectorSize *activation3 = (VectorSize *) calloc(sizeof(VectorSize), 1);
    VectorSize *delta2 = (VectorSize *) calloc(sizeof(VectorSize), 1);
    VectorSize *delta3 = (VectorSize *) calloc(sizeof(VectorSize), 1);

    if (LOGGING == 1) fprintf(stdout, "Intermediate 2: ");
    SetMatrixSize(inter2, 1, layer_1, layer_2, layer_1, 1, layer_2);
    if (LOGGING == 1) fprintf(stdout, "Intermediate 3: ");
    SetMatrixSize(inter3, 1, layer_2, layer_3, layer_2, 1, layer_3);
    if (LOGGING == 1) fprintf(stdout, "Grad 1: ");
    SetMatrixSize(grad1, 1, layer_2, 1, layer_1, layer_2, layer_1);
    if (LOGGING == 1) fprintf(stdout, "Grad 2: ");
    SetMatrixSize(grad2, 1, layer_3, 1, layer_2, layer_3, layer_2);
    if (LOGGING == 1) fprintf(stdout, "Backprop 1: ");
    SetMatrixSize(backprop1, layer_2, layer_1, layer_2, layer_1, layer_2, layer_1);
    if (LOGGING == 1) fprintf(stdout, "Backprop 2: ");
    SetMatrixSize(backprop2, layer_3, layer_2, layer_3, layer_2, layer_3, layer_2);

    if (LOGGING == 1) fprintf(stdout, "Activation 2: ");
    SetVectorSize(activation2, layer_2);
    if (LOGGING == 1) fprintf(stdout, "Activation 3: ");
    SetVectorSize(activation3, layer_3);
    if (LOGGING == 1) fprintf(stdout, "Delta 2: ");
    SetVectorSize(delta2, layer_2);
    if (LOGGING == 1) fprintf(stdout, "Delta 3: ");
    SetVectorSize(delta3, layer_3);

    // Set number of epochs and samples
    int epochs = 1; // Number of training epochs (iterations through data)
    int num_train = 20000;   // Number of samples;
    int num_test = 5000;

    // Initalize weights
    InitializeWeights(W1, grad1);
    InitializeWeights(W2, grad2);

    //Perform neural network training
    for (int epoch = 0; epoch < epochs; epoch++) {

        // Open training data files
        std::ifstream x_train_data("./data/train_img.csv");
```

```
    std::ifstream y_train_data("./data/train_res.csv");

    for (int sample = 0; sample < num_train; sample++) {
        // LOAD a1 AND y VECTORS:
        ReadCSV(x_train_data, layer_1, a1);
        ReadCSV(y_train_data, layer_3, y);

        // FORWARD PROPOGATION:
        MatrixMultiplyCUBLAS(argc, argv, devID, inter2, a1, W1, z2, 1.0, 1.0, false, true); // Compute z2
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "z2 Computation: %s\n", cudaGetErrorString(mainErr));
        RunVectorKernel(argc, argv, devID, activation2, 4, z2, z2, a2, 1.0, 1.0);           // Compute a2
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "a2 Computation: %s\n", cudaGetErrorString(mainErr));
        MatrixMultiplyCUBLAS(argc, argv, devID, inter3, a2, W2, z3, 1.0, 1.0, false, true); // Compute z3
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "z3 Computation: %s\n", cudaGetErrorString(mainErr));
        RunVectorKernel(argc, argv, devID, activation3, 4, z3, z3, a3, 1.0, 1.0);           // Compute a3
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "a3 Computation: %s\n", cudaGetErrorString(mainErr));

        // BACKWARD PROPOGATION:
        RunVectorKernel(argc, argv, devID, delta3, 1, z3, y, del3, 1.0, (float)-1.0);           // Compute del3
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "del3 Computation: %s\n", cudaGetErrorString(mainErr));

        MatrixMultiplyCUBLAS(argc, argv, devID, inter3, del3, W2, scratch1, 1.0, 1.0, false, false); // Compute del2 lhs
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "pre-del2 lhs Computation: %s\n", cudaGetErrorString(mainErr));

        RunVectorKernel(argc, argv, devID, delta2, 5, z2, y, scratch2, 1.0, (float) -1.0);           // Compute del2 rhs
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "pre-del2 rhs Computation: %s\n", cudaGetErrorString(mainErr));

        RunVectorKernel(argc, argv, devID, delta2, 2, scratch1, scratch2, del2, 1.0, (float) -1.0);  // Compute del2
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "del2 Computation: %s\n", cudaGetErrorString(mainErr));

        MatrixMultiplyCUBLAS(argc, argv, devID, grad1, del2, a1, Del1, 1.0, 1.0, true, false);   // Compute Del1
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "Del1 Computation: %s\n", cudaGetErrorString(mainErr));

        MatrixMultiplyCUBLAS(argc, argv, devID, grad2, del3, a2, Del2, 1.0, 1.0, true, false);   // Compute Del2
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "Del2 Computation: %s\n", cudaGetErrorString(mainErr));

        // Gradient descent
        RunMatrixKernel(argc, argv, devID, backprop1, 1, W1, Del1, W1, 1.0,
                        (float) -1.0 / (float) num_train); // Compute new W1
        RunMatrixKernel(argc, argv, devID, backprop2, 1, W2, Del2, W2, 1.0,
```

```
                            (float) -1.0 / (float) num_train); // Compute new W2
            cudaDeviceSynchronize();
            if( (sample % 1000) == 0) printf("Iteration: %d\n", sample);
            if( (sample % 5000) == 0) cudaDeviceReset();
        }
        // Close training data files
        x_train_data.close();
        y_train_data.close();

    }

    // Open verification data files
    std::ifstream x_test_data("./data/tests_img.csv");
    std::ifstream y_test_data("./data/tests_res.csv");

    int correct = 0;

    for(int test_sample = 0; test_sample < num_test; test_sample++)
    {
        // LOAD a1 AND y VECTORS:
        ReadCSV(x_test_data, layer_1, a1);
        ReadCSV(y_test_data, layer_3, y);

        // FORWARD PROPOGATION:
        MatrixMultiplyCUBLAS(argc, argv, devID, inter2, a1, W1, z2, 1.0, 1.0, false, true); // Compute z2
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "z2 Computation: %s\n", cudaGetErrorString(mainErr));
        RunVectorKernel(argc, argv, devID, activation2, 4, z2, z2, a2, 1.0, 1.0); // Compute a2
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "a2 Computation: %s\n", cudaGetErrorString(mainErr));
        MatrixMultiplyCUBLAS(argc, argv, devID, inter3, a2, W2, z3, 1.0, 1.0, false, true); // Compute z3
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "z3 Computation: %s\n", cudaGetErrorString(mainErr));
        RunVectorKernel(argc, argv, devID, activation3, 4, z3, z3, a3, 1.0, 1.0); // Compute a3
        mainErr = cudaGetLastError();
        if (mainErr != cudaSuccess) fprintf(stderr, "a3 Computation: %s\n", cudaGetErrorString(mainErr));

        float a3max = 0.0;
        int a3max_idx = 0;
        int ymax_idx = 0;

        for(int i = 0; i < layer_3; i++)
        {
            if(a3[i] > a3max)
            {
                a3max = a3[i];
                a3max_idx = i;
            }
            if(y[i] == 1) ymax_idx = i;
        }
        if(ymax_idx == a3max_idx) correct++;
    }
    x_test_data.close();
    y_test_data.close();
    printf("The network correctly identified %d of %d samples\n", correct, num_test);
    return 0;
}
```