

# A Simple CUDA Neural Network

Kyle Salitrik (kps168), Tomoki Takasawa  
(tmt5336)

## Abstract

## 1 Credits

Images throughout the forward propagation section were obtained from: [neuralnetworksanddeeplearning.com/chap1.html](http://neuralnetworksanddeeplearning.com/chap1.html)

## 2 Introduction and Objectives

This program is designed to identify hand written numbers (0-9) by neural network digit classification technique. Generally, this technique requires enormous amount of run time. For training process, it calculates 5 matrix multiplications, 2 matrix subtractions, and 5 element-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 NVIDIA's GPU processors.

## 3 Neural Network Structure

All figures in this section were obtained from the source credited in section 1. 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.

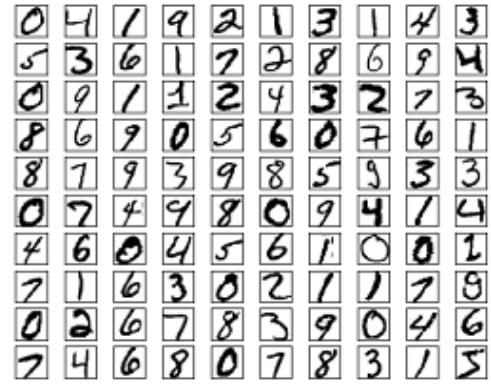


Figure 1: MNIST Data

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:

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

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

#### 4 Forward Propagation

The behavior 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.

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

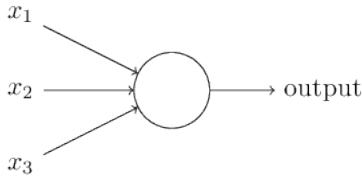


Figure 2: Neuron Diagram

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.

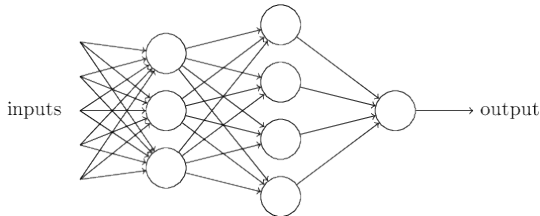


Figure 3: Neuron Diagram

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 initialized 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:

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

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.

#### 5 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 calculated, 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:

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

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

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

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.

## 6 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:

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

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

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

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

$$\begin{aligned} W1 &:= W1 - \nabla^{(1)} \\ W2 &:= W2 - \nabla^{(2)} \end{aligned}$$

## 7 Testing 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.

## 8 Performance

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.

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.

## 9 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. Other future work would include making the computations more memory efficient.

## 10 Code Function Explanations

- **setVectorSize:**
  - This function sets the vector size and store them in struct, `_vSize`.
- **setMatrixSize:**
  - This function sets the matrix size and store them in struct, `_matrixSize`.
- **VectorInitCUDA / MatrixInitCUDA:**
  - This function allocates memory in GPU, and copy the value from local machine.
- **VectorAdditionKernel:**
  - CUDA Kernel to add two vectors
- **VectorHadamardKernel:**
  - CUDA Kernel to compute the Hadamard product of two vectors
- **VectorDotProduct:**
  - CUDA Kernel to compute the dot product of two vectors
- **VectorSigmoid:**
  - CUDA Kernel that computes the sigmoid function for each item in a vector
- **VectorSigmoidDerivative:**
  - CUDA Kernel that computes the sigmoid function derivative for each item in a vector
- **MatrixAddKernel:**
  - CUDA Kernel to add or subtract two matrices
- **MatrixHadamardKernel:**
  - CUDA Kernel to compute the Hadamard product of two matrices
- **MatrixSigmoidMatrix:**
  - CUDA Kernel that computes the sigmoid function for each item in a matrix
- **SigmoidDerivative:**
  - CUDA Kernel that computes the sigmoid function derivative for each item in a matrix
- **RunVectorKernel:**
  - This function allocate and initialize memory in GPU and call functions for vector operations, `VectorAdditionKernel`, `VectorHadamardKernel`, `VectorDotProduct`, `VectorSigmoid`, `VectorSigmoidDerivative` functions. Then, it frees memory.
- **MatrixMultiplyCUBLAS:**
  - This allocates and initialize memory in GPU and perform matrix multiplication by calling `cublasSgemm`, which is part of NVIDIA's library. Then, it frees memory.
- **RunMatrixKernel:**
  - This function allocate and initialize memory in GPU and calls functions for matrix operations, `MatrixAddKernel`, `MatrixHadamardKernel`, `MatrixSigmoidMatrix`, and `SigmoidDerivative`. Then, it frees memory.
- **runTest:**
  - This is a test function that was used for testing the kernels before the network was implemented.
- **main:**
  - It allocates memory for activation vectors, pre-sigmoid intermediary vectors, weight matrices, one-hot result vectore, error vectors, and error gradients in local machine, and initialize them. Then, it calls `RunVectorKernel` and `MatrixMultiplyCUBLAS`

## References

Michael Nielsen. 2017. *Neural Networks and Deep Learning Book*. [neuralnetworksanddeeplearning.com](http://neuralnetworksanddeeplearning.com).

## Code Appendix

```
1 // Compile using nvcc <file> -lcublas -o <output>
2 #include <cublas_v2.h>
3 #include <cuda_runtime.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <fstream>
7 #include <sstream>
8 #include <iostream>
9
10 // Define block size for thread allocation
11 #define NUM_THREADS 32 // 32 is max for N^2 threads: 32*32 = 1024
12 #define LOGGING 0
13
14 //=====
15 //=== Structure definitions
16 //=====
17 typedef struct _kernelParams {
18     int block_size;
19     int grid_size;
20 } sKernelParams;
21
22 typedef struct _matrixSize // Optional Command-line multiplier for matrix sizes
23 {
24     unsigned int A_height, A_width, B_height, B_width, C_height, C_width;
25 } MatrixSize;
26
27 typedef struct _vSize // Optional Command-line multiplier for matrix sizes
28 {
29     unsigned int len_A, len_B, len_C;
30 } VectorSize;
31
32 //=====
33 //=== Structure functions
34 //=====
35
36 /**
37  * @brief - sets values of vector size structure
38  *
39  * @param vector_size - pointer to vector size struct
40  * @param len - length of all vectors
41  */
42 void SetVectorSize(VectorSize *vector_size, unsigned int len) {
43     vector_size->len_A = len;
44     vector_size->len_B = len;
45     vector_size->len_C = len;
46
47     if (LOGGING == 1)
48         fprintf(stdout, "Vector A(%u), Vector B(%u), Vector C(%u)\n",
49             vector_size->len_A,
50             vector_size->len_B,
51             vector_size->len_C);
52
53     if (vector_size->len_A != vector_size->len_B ||
54         vector_size->len_B != vector_size->len_C ||
55         vector_size->len_C != vector_size->len_A) {
56         fprintf(stderr, "ERROR: Vector lengths do not match!\n");
57         exit(-1);
58     }
59 }
60
61 /**
62  * @brief - sets values of matrix size structure
63  *
64  * @param matrixSize - reference to matrix size struct
65  * @param widthA - width of matrix A
66  * @param heightA - height of matrix A
67  * @param widthB - width of matrix B
68  * @param heightB - height of matrix B
```

```

69 * @param widthC – width of matrix C
70 * @param heightC – height of matrix C
71 */
72 void SetMatrixSize(MatrixSize *matrixSize,
73                    unsigned int widthA, unsigned int heightA,
74                    unsigned int widthB, unsigned int heightB,
75                    unsigned int widthC, unsigned int heightC) {
76     matrixSize->A_height = heightA;
77     matrixSize->A_width = widthA;
78     matrixSize->B_height = heightB;
79     matrixSize->B_width = widthB;
80     matrixSize->C_height = heightC;
81     matrixSize->C_width = widthC;
82
83     if (LOGGING == 1)
84         fprintf(stdout, "Matrix A(%u x %u), Matrix B(%u x %u), Matrix C(%u x %u)\n",
85                 matrixSize->A_width,
86                 matrixSize->A_height,
87                 matrixSize->B_width,
88                 matrixSize->B_height,
89                 matrixSize->C_width,
90                 matrixSize->C_height);
91 }
92
93
94 //=====
95 //=== GPU memory initialization functions
96 //=====
97
98 /**
99 * @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
100 *
101 *
102 * @param argc – from compiler
103 * @param argv – from compiler
104 * @param devID – device ID number
105 * @param vector_size – reference to vector size structure
106 * @param host_vA – pointer to host vector A (with values)
107 * @param host_vB – pointer to host vector B (with values)
108 * @param dev_A – pointer to vector A device memory reference
109 * @param dev_B – pointer to vector B device memory reference
110 * @param dev_C – pointer to vector C device memory reference
111 */
112 void VectorInitCUDA(int argc, char **argv, int devID, VectorSize *vector_size, float
    *host_vA, float *host_vB,
113                    float *&dev_A, float *&dev_B, float *&dev_C) {
114     // Assign CUDA variables
115     cudaError_t err;
116
117     // Assign size variables
118     size_t size_A = vector_size->len_A * sizeof(float);
119     size_t size_B = vector_size->len_B * sizeof(float);
120     size_t size_C = vector_size->len_C * sizeof(float);
121
122     // Allocate memory on GPU
123     err = cudaMalloc((void **) &dev_A, size_A);
124     if (err != cudaSuccess) fprintf(stderr, "ERROR allocating vector A: %s\n",
        cudaGetErrorString(err));
125     err = cudaMalloc((void **) &dev_B, size_B);
126     if (err != cudaSuccess) fprintf(stderr, "ERROR allocating vector B: %s\n",
        cudaGetErrorString(err));
127     err = cudaMalloc((void **) &dev_C, size_C);
128     if (err != cudaSuccess) fprintf(stderr, "ERROR allocating vector C: %s\n",
        cudaGetErrorString(err));
129
130     // Copy data from host PC to GPU
131     err = cudaMemcpy(dev_A, host_vA, size_A, cudaMemcpyHostToDevice);
132     if (err != cudaSuccess) fprintf(stderr, "ERROR copying vector A to GPU: %s\n",
        cudaGetErrorString(err));

```

```

133     err = cudaMemcpy(dev_B, host_vB, size_B, cudaMemcpyHostToDevice);
134     if (err != cudaSuccess) fprintf(stderr, "ERROR copying vector B to GPU: %s\n",
    cudaMemcpyErrorString(err));
135 }
136 }
137
138 /**
139  * @brief - allocates memory on GPU for matrices A, B, and C then copies the values
    for matrices A, B and C
140  *         from host PC onto the device
141  *
142  * @param argc - from compiler
143  * @param argv - from compiler
144  * @param devID - device ID number
145  * @param matrixSize - reference to vector size structure
146  * @param host_matrixA - pointer to host matrix A (with values)
147  * @param host_matrixB - pointer to host matrix B (with values)
148  * @param host_matrixC - pointer to host matrix C (with values)
149  * @param dev_matrixA - pointer to matrix A device memory reference
150  * @param dev_matrixB - pointer to matrix B device memory reference
151  * @param dev_matrixC - pointer to matrix C device memory reference
152  */
153 void MatrixInitCUDA(int argc, char **argv, int &devID, MatrixSize *matrixSize,
154                     float *host_matrixA, float *host_matrixB, float *host_matrixC,
155                     float *&dev_matrixA, float *&dev_matrixB, float *&dev_matrixC) {
156     // Assign CUDA variables
157     cudaError_t err;
158
159     // Assign size variables
160     size_t matrixA_size = matrixSize->A_height * matrixSize->A_width * sizeof(float)
    ;
161     size_t matrixB_size = matrixSize->B_height * matrixSize->B_width * sizeof(float)
    ;
162     size_t matrixC_size = matrixSize->C_height * matrixSize->C_width * sizeof(float)
    ;
163
164     // Allocate memory on GPU
165     err = cudaMalloc((void **) &dev_matrixA, matrixA_size);
166     if (err != cudaSuccess) fprintf(stderr, "ERROR allocating matrix A: %s\n",
    cudaMemcpyErrorString(err));
167     err = cudaMalloc((void **) &dev_matrixB, matrixB_size);
168     if (err != cudaSuccess) fprintf(stderr, "ERROR allocating matrix B: %s\n",
    cudaMemcpyErrorString(err));
169     err = cudaMalloc((void **) &dev_matrixC, matrixC_size);
170     if (err != cudaSuccess) fprintf(stderr, "ERROR allocating matrix C: %s\n",
    cudaMemcpyErrorString(err));
171
172     // Copy data from host PC to GPU
173     err = cudaMemcpy(dev_matrixA, host_matrixA, matrixA_size, cudaMemcpyHostToDevice)
    ;
174     if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix A to GPU: %s\n",
    cudaMemcpyErrorString(err));
175     err = cudaMemcpy(dev_matrixB, host_matrixB, matrixB_size, cudaMemcpyHostToDevice)
    ;
176     if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix B to GPU: %s\n",
    cudaMemcpyErrorString(err));
177     err = cudaMemcpy(dev_matrixC, host_matrixC, matrixC_size, cudaMemcpyHostToDevice)
    ;
178     if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix C to GPU: %s\n",
    cudaMemcpyErrorString(err));
179 }
180
181 //=====
182 //=== CUDA Vector Kernels
183 //=====
184 /**
185  * @required ALL VECTORS MUST BE THE SAME LENGTH
186  * @brief - kernel for GPU computation of a vector addition
187  * @param dev_vecA - pointer to device memory for vector A
188  * @param dev_vecB - pointer to device memory for vector B

```

```

189 * @param dev_vecC – pointer to device memory for vector C
190 * @param alpha – multiplier for values in vector A
191 * @param beta – multiplier for values in vector B
192 * @param vecLen – length of all vectors
193 */
194 __global__ void VectorAdditionKernel(float *dev_vecA, float *dev_vecB, float *
    dev_vecC,
195                                     float alpha, float beta, int vecLen) {
196     int i = blockDim.x * blockIdx.x + threadIdx.x;
197     if (i < vecLen) {
198         dev_vecC[i] = alpha * dev_vecA[i] + beta * dev_vecB[i];
199     }
200 }
201
202 /**
203 * @required ALL VECTORS MUST BE THE SAME LENGTH
204 * @brief – kernel for GPU computation of a vector hadamard product
205 * @param dev_vecA – pointer to device memory for vector A
206 * @param dev_vecB – pointer to device memory for vector B
207 * @param dev_vecC – pointer to device memory for vector C
208 * @param alpha – multiplier for values in vector A
209 * @param beta – multiplier for values in vector B
210 * @param vecLen – length of all vectors
211 */
212 __global__ void VectorHadamardKernel(float *dev_vecA, float *dev_vecB, float *
    dev_vecC,
213                                     float alpha, float beta, int vecLen) {
214     int i = blockDim.x * blockIdx.x + threadIdx.x;
215     if (i < vecLen) {
216         dev_vecC[i] = alpha * dev_vecA[i] * beta * dev_vecB[i];
217     }
218 }
219
220 /**
221 * @required ALL VECTORS MUST BE THE SAME LENGTH
222 * REMEMBER: Call kernel using: <<<grid, threads, vecLen>>>
223 * @brief – kernel for GPU computation of a vector dot product
224 * @param dev_vecA – pointer to device memory for vector A
225 * @param dev_vecB – pointer to device memory for vector B
226 * @param result – pointer to a single float value where the result will be returned
227 * @param alpha – multiplier for values in vector A
228 * @param beta – multiplier for values in vector B
229 * @param vecLen – length of all vectors
230 */
231 __global__ void VectorDotProduct(float *dev_vecA, float *dev_vecB, float *result,
    float alpha, float beta, int vecLen) {
232     extern __shared__ float temp[];
233     int i = blockDim.x * blockIdx.x + threadIdx.x;
234     if (i < vecLen) {
235         temp[i] = alpha * dev_vecA[i] * beta * dev_vecB[i];
236     }
237     __syncthreads();
238     if (threadIdx.x == 0) {
239         float sum = 0.0;
240         for (int j = 0; j < vecLen; j++) {
241             sum += temp[j];
242         }
243         result[0] = sum;
244     }
245 }
246
247
248 /**
249 * @required INPUT AND OUTPUT VECTORS MUST BE THE SAME LENGTH
250 * @brief – kernel for GPU computation of the vector sigmoid function
251 * @param dev_matrixA – pointer to device memory for vector A
252 * @param dev_matrixC – pointer to device memory for vector C
253 * @param vecLen – length of all vectors
254 */
255 __global__ void VectorSigmoid(float *dev_vecA, float *dev_vecC, int vecLen) {
256     int index = blockDim.x * blockIdx.x + threadIdx.x;

```



```

257     if (index < vecLen) {
258         float exp = 1 + expf(-dev_vecA[index]);
259         dev_vecC[index] = 1 / exp;
260     }
261 }
262
263 /**
264  * @required INPUT AND OUTPUT VECTORS MUST BE THE SAME LENGTH
265  * @brief - kernel for GPU computation of the vector sigmoid derivative function
266  * @param dev_matrixA - pointer to device memory for vector A
267  * @param dev_matrixC - pointer to device memory for vector C
268  * @param vecLen - length of all vectors
269  */
270 __global__ void VectorSigmoidDerivative(float *dev_vecA, float *dev_vecC, int vecLen)
271 {
272     int index = blockDim.x * blockIdx.x + threadIdx.x;
273     if (index < vecLen) {
274         float exp = 1 + expf(-dev_vecA[index]);
275         float sig = 1 / exp;
276         dev_vecC[index] = sig * (1 - sig);
277     }
278 }
279
280 //==== CUDA Vector Kernel Drivers
281 //=====
282
283 /**
284  * @brief driver function for computing vector operations
285  * @param argc - from compiler
286  * @param argv - from compiler
287  * @param devID - device ID number
288  * @param vectorSize - reference to vector size structure
289  * @param operation - switch-case value for which matrix operation to perform
290  *                  1: Vector addition
291  *                  2: Vector Hadamard product
292  *                  3: Vector dot product
293  *                  4: Vector sigmoid function
294  *                  5: Vector sigmoid derivative
295  * @param host_vectorA - pointer to host vector A (with values)
296  * @param host_vectorB - pointer to host vector B (with values)
297  * @param host_vectorC - pointer to host vector C (with values)
298  * @param alpha - multiplier for values in vector A
299  * @param beta - multiplier for values in vector B
300  */
301 void RunVectorKernel(int argc, char **argv, int &devID, VectorSize *vectorSize, int
302 operation,
303                     float *host_vectorA, float *host_vectorB, float *host_vectorC,
304 float alpha, float beta) {
305     // Assign CUDA variables
306     cudaError_t err;
307     dim3 threads(NUM_THREADS, NUM_THREADS);
308     int gridX = (int) ceil((float) vectorSize->len_C / (float) threads.x);
309     int gridY = (int) ceil((float) vectorSize->len_C / (float) threads.y);
310     dim3 grid((unsigned int) gridX, (unsigned int) gridY);
311
312     // Assign computation variables
313     float *dev_vectorA = NULL;
314     float *dev_vectorB = NULL;
315     float *dev_vectorC = NULL;
316
317     size_t vectorC_size = vectorSize->len_C * sizeof(float);
318
319     // Initialize memory on GPU
320     VectorInitCUDA(argc, argv, devID, vectorSize, host_vectorA, host_vectorB,
321 dev_vectorA, dev_vectorB, dev_vectorC);
322
323     switch (operation) {
324         case 1: {
325             // Compute vector addition

```

```

323     VectorAdditionKernel<<<grid, threads>>>(dev_vectorA, dev_vectorB,
dev_vectorC, alpha, beta,
324         vectorSize->len_C);
325     err = cudaGetLastError();
326     if (err != cudaSuccess) fprintf(stderr, "ERROR in Vector Add Computation
: %s\n", cudaGetErrorString(err));
327     break;
328 }
329 case 2: {
330     // Compute vector Hadamard Product
331     VectorHadamardKernel<<<grid, threads>>>(dev_vectorA, dev_vectorB,
dev_vectorC, alpha, beta,
332         vectorSize->len_C);
333     err = cudaGetLastError();
334     if (err != cudaSuccess)
335         fprintf(stderr, "ERROR in Vector Hadamard Computation: %s\n",
cudaGetErrorString(err));
336     break;
337 }
338 case 3: {
339     // Compute vector dot product
340     VectorDotProduct<<<grid, threads, vectorSize->len_C>>>
341         (dev_vectorA, dev_vectorB, dev_vectorC
, alpha, beta, vectorSize->len_C);
342     err = cudaGetLastError();
343     if (err != cudaSuccess)
344         fprintf(stderr, "ERROR in Vector Dot product Computation: %s\n",
cudaGetErrorString(err));
345     break;
346 }
347 case 4: {
348     // Compute sigmoid function
349     VectorSigmoid<<<grid, threads>>>(dev_vectorA, dev_vectorC, vectorSize->
len_C);
350     err = cudaGetLastError();
351     if (err != cudaSuccess)
352         fprintf(stderr, "ERROR in Vector Sigmoid Computation: %s\n",
cudaGetErrorString(err));
353     break;
354 }
355 case 5: {
356     // Compute sigmoid derivative
357     VectorSigmoidDerivative<<<grid, threads>>>(dev_vectorA, dev_vectorC,
vectorSize->len_C);
358     err = cudaGetLastError();
359     if (err != cudaSuccess)
360         fprintf(stderr, "ERROR in Vector Sigmoid Derivative Computation: %s\
n", cudaGetErrorString(err));
361     break;
362 }
363 default: {
364     fprintf(stderr, "ERROR: No vector kernel selected. Operation Aborted");
365     break;
366 }
367 }
368
369 // Make sure device is finished
370 err = cudaDeviceSynchronize();
371 if (err != cudaSuccess)
372     fprintf(stderr, "ERROR synchronizing Vector Kernel calculation: %s\n",
cudaGetErrorString(err));
373
374 // Copy data from GPU to host PC
375 err = cudaMemcpy(host_vectorC, dev_vectorC, vectorC_size, cudaMemcpyDeviceToHost
);
376 if (err != cudaSuccess)
377     fprintf(stderr, "ERROR copying vector C to Host: %s\n", cudaGetErrorString(
err));
378
379 // Free GPU memory

```

```

380     err = cudaFree(dev_vectorA);
381     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing vector A on GPU: %s\n",
cudaGetErrorString(err));
382     err = cudaFree(dev_vectorB);
383     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing vector B on GPU: %s\n",
cudaGetErrorString(err));
384     err = cudaFree(dev_vectorC);
385     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing vector C on GPU: %s\n",
cudaGetErrorString(err));
386     err = cudaDeviceSynchronize();
387     if (err != cudaSuccess) fprintf(stderr, "ERROR synchronizing Vector Kernel End:
%s\n", cudaGetErrorString(err));
388     if (LOGGING == 1) fprintf(stdout, "Vector Kernel finished.\n");
389 }
390
391 //=====
392 //== CUDA Matrix Kernels
393 //=====
394
395 /**
396  * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
397  * @brief - kernel for GPU computation of matrix additions
398  * @param dev_matrixA - pointer to device memory for matrix A
399  * @param dev_matrixB - pointer to device memory for matrix B
400  * @param dev_matrixC - pointer to device memory for matrix C
401  * @param alpha - multiplier for values in matrix A
402  * @param beta - multiplier for values in matrix B
403  * @param matrix_width - width of all matrices
404  * @param matrix_height - height of all matrices
405  */
406 __global__ void MatrixAddKernel(float *dev_matrixA, float *dev_matrixB, float *
dev_matrixC,
407                                float alpha, float beta, int matrix_width, int
matrix_height) {
408     int row = blockIdx.x * blockDim.x + threadIdx.x;
409     int col = blockIdx.y * blockDim.y + threadIdx.y;
410     int index = col + row * matrix_height;
411     if (col < matrix_width && row < matrix_height) {
412         dev_matrixC[index] = alpha * dev_matrixA[index] + beta * dev_matrixB[index];
413     }
414 }
415
416 /**
417  * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
418  * @brief - kernel for actual GPU computation for the matrix Hadamard product
419  * @param dev_matrixA - pointer to device memory for matrix A
420  * @param dev_matrixB - pointer to device memory for matrix B
421  * @param dev_matrixC - pointer to device memory for matrix C
422  * @param alpha - multiplier for values in matrix A
423  * @param beta - multiplier for values in matrix B
424  * @param matrix_width - width of all matrices
425  * @param matrix_height - height of all matrices
426  */
427 __global__ void MatrixHadamardKernel(float *dev_matrixA, float *dev_matrixB, float *
dev_matrixC,
428                                       float alpha, float beta, int matrix_width, int
matrix_height) {
429     int row = blockIdx.x * blockDim.x + threadIdx.x;
430     int col = blockIdx.y * blockDim.y + threadIdx.y;
431     int index = col + row * matrix_height;
432     if (col < matrix_width && row < matrix_height) {
433         dev_matrixC[index] = alpha * dev_matrixA[index] * beta * dev_matrixB[index];
434     }
435 }
436
437 /**
438  * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
439  * @brief - kernel for GPU computation of matrix sigmoid function
440  * @param dev_matrixA - pointer to device memory for matrix A
441  * @param dev_matrixC - pointer to device memory for matrix C

```

```

442 * @param matrix_width - width of all matrices
443 * @param matrix_height - height of all matrices
444 */
445 --global-- void MatrixSigmoid(float *dev_matrixA, float *dev_matrixC,
446                               int matrix_width, int matrix_height) {
447     int row = blockIdx.x * blockDim.x + threadIdx.x;
448     int col = blockIdx.y * blockDim.y + threadIdx.y;
449     int index = col + row * matrix_height;
450     if (col < matrix_width && row < matrix_height) {
451         float exp = 1 + expf(-dev_matrixA[index]);
452         dev_matrixC[index] = 1 / exp;
453     }
454 }
455
456 /**
457 * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
458 * @brief - kernel for GPU computation of the matrix sigmoid derivative function
459 * @param dev_matrixA - pointer to device memory for matrix A
460 * @param dev_matrixC - pointer to device memory for matrix C
461 * @param matrix_width - width of all matrices
462 * @param matrix_height - height of all matrices
463 */
464 --global-- void MatrixSigmoidDerivative(float *dev_matrixA, float *dev_matrixC,
465                                          int matrix_width, int matrix_height) {
466     int row = blockIdx.x * blockDim.x + threadIdx.x;
467     int col = blockIdx.y * blockDim.y + threadIdx.y;
468     int index = col + row * matrix_height;
469     if (col < matrix_width && row < matrix_height) {
470         float exp = 1 + expf(-dev_matrixA[index]);
471         float sig = 1 / exp;
472         dev_matrixC[index] = sig * (1 - sig);
473     }
474 }
475
476 //=====
477 //=== CUDA Matrix Kernel Drivers
478 //=====
479
480 /**
481 * @brief - Uses CUBLAS library to perform alpha(A x B) + beta(C) matrix
482 *           multiplication and addition
483 * @param argc - from compiler
484 * @param argv - from compiler
485 * @param devID - device ID number
486 * @param matrixSize - reference to vector size structure
487 * @param host_matrixA - pointer to host matrix A (with values)
488 * @param host_matrixB - pointer to host matrix B (with values)
489 * @param host_matrixC - pointer to host matrix C (with values)
490 * @param alpha - value for alpha in CUBLAS function
491 * @param beta - value for beta in CUBLAS function
492 * @param transposeA - true if A should be transposed
493 * @param transposeB - true if B should be transposed
494 */
495 void MatrixMultiplyCUBLAS(int argc, char **argv, int &devID, MatrixSize *matrixSize,
496                           float *host_matrixA, float *host_matrixB, float *
497                           host_matrixC,
498                           float alpha, float beta, bool transposeA, bool transposeB)
499 {
500     // Assign CUDA variables
501     cublasHandle_t handle;
502     cudaError_t err;
503     cublasCreate(&handle);
504     cudaDeviceProp deviceProp;
505     cudaGetDeviceProperties(&deviceProp, devID);
506     dim3 threads(NUM_THREADS, NUM_THREADS);
507     dim3 grid(matrixSize->C_width / threads.x, matrixSize->C_height / threads.y);
508
509     // Assign computation variables
510     float *dev_matrixA = NULL, *dev_matrixB = NULL, *dev_matrixC = NULL;

```

```

509     int m = matrixSize->A_height;
510     int n = matrixSize->B_width;
511     int k = matrixSize->A_width;
512     cublasOperation_t transA = CUBLAS_OP_N, transB = CUBLAS_OP_N;
513     if (transposeA) transA = CUBLAS_OP_T;
514     if (transposeB) transB = CUBLAS_OP_T;
515     size_t matrixC_size = matrixSize->C_height * matrixSize->C_width * sizeof(float)
;
516
517     // Initialize memory on GPU
518     MatrixInitCUDA(argc, argv, devID, matrixSize,
519                   host_matrixA, host_matrixB, host_matrixC,
520                   dev_matrixA, dev_matrixB, dev_matrixC);
521
522     // Perform matrix multiplication
523     // SGEMM PARAMS: (handle, transposeA, transposeB, m, n, k, alpha, matrix A, k,
matrix B, n, beta, matrix C, n)
524     cublasSgemm(handle, transA, transB, m, n, k, &alpha, dev_matrixA, m,
525                dev_matrixB, n, &beta, dev_matrixC, m);
526     err = cudaGetLastError();
527     if (err != cudaSuccess) fprintf(stderr, "ERROR in SGEMM: %s\n",
cudaGetErrorString(err));
528
529     // Make sure device is finished
530     err = cudaDeviceSynchronize();
531     if (err != cudaSuccess) fprintf(stderr, "ERROR synchronizing SGEMM calculation:
%s\n", cudaGetErrorString(err));
532
533     // Copy data from GPU to host PC
534     err = cudaMemcpy(host_matrixC, dev_matrixC, matrixC_size, cudaMemcpyDeviceToHost
);
535     if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix C to Host: %s\n",
cudaGetErrorString(err));
536
537     // Free GPU memory
538     err = cudaFree(dev_matrixA);
539     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix A on GPU: %s\n",
cudaGetErrorString(err));
540     err = cudaFree(dev_matrixB);
541     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix B on GPU: %s\n",
cudaGetErrorString(err));
542     err = cudaFree(dev_matrixC);
543     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix C on GPU: %s\n",
cudaGetErrorString(err));
544     err = cudaDeviceSynchronize();
545     if (err != cudaSuccess) fprintf(stderr, "ERROR synchronizing SGEMM end: %s\n",
cudaGetErrorString(err));
546     if (LOGGING == 1) fprintf(stdout, "Matrix Kernel finished.\n");
547 }
548
549 /**
550  * @required ALL MATRICES MUST BE THE SAME DIMENSIONS
551  * @brief driver function for computing the matrix operations
552  * @param argc - from compiler
553  * @param argv - from compiler
554  * @param devID - device ID number
555  * @param matrixSize - reference to matrix size structure
556  * @param operation - switch-case value for which matrix operation to perform
557  * 1: Matrix addition
558  * 2: Matrix Hadamard product
559  * 3: Sigmoid function
560  * 4: Sigmoid derivative
561  * @param host_matrixA - pointer to host matrix A (with values)
562  * @param host_matrixB - pointer to host matrix B (with values)
563  * @param host_matrixC - pointer to host matrix C (with values)
564  * @param alpha - multiplier for values in matrix A
565  * @param beta - multiplier for values in matrix B
566  */
567 void RunMatrixKernel(int argc, char **argv, int &devID, MatrixSize *matrixSize, int
operation,

```

```

568         float *host_matrixA , float *host_matrixB , float *host_matrixC ,
float alpha , float beta) {
569     // Assign CUDA variables
570     cudaError_t err;
571     dim3 threads(NUM.THREADS, NUM.THREADS);
572     int gridX = (int) ceil((float) matrixSize->C_width / (float) threads.x);
573     int gridY = (int) ceil((float) matrixSize->C_height / (float) threads.y);
574     dim3 grid((unsigned int) gridX, (unsigned int) gridY);
575
576     // Assign computation variables
577     float *dev_matrixA = NULL, *dev_matrixB = NULL, *dev_matrixC = NULL;
578     size_t matrixC_size = matrixSize->C_height * matrixSize->C_width * sizeof(float)
;
579
580     // Initialize memory on GPU
581     MatrixInitCUDA(argc, argv, devID, matrixSize,
582                     host_matrixA, host_matrixB, host_matrixC,
583                     dev_matrixA, dev_matrixB, dev_matrixC);
584
585     switch (operation) {
586     case 1: {
587         // Compute Matrix Addition
588         MatrixAddKernel<<<grid, threads>>>(dev_matrixA, dev_matrixB, dev_matrixC
, alpha, beta,
589                                             matrixSize->C_width, matrixSize->C_height);
590         err = cudaGetLastError();
591         if (err != cudaSuccess) fprintf(stderr, "ERROR in Matrix Add Computation
: %s\n", cudaGetErrorString(err));
592         break;
593     }
594     case 2: {
595         // Compute Hadamard Product
596         MatrixHadamardKernel<<<grid, threads>>>(dev_matrixA, dev_matrixB,
dev_matrixC, alpha, beta,
597                                             matrixSize->C_width, matrixSize->C_height);
598         err = cudaGetLastError();
599         if (err != cudaSuccess)
600             fprintf(stderr, "ERROR in Matrix Hadamard Computation: %s\n",
cudaGetErrorString(err));
601         break;
602     }
603     case 3: {
604         // Compute Sigmoid function
605         MatrixSigmoid<<<grid, threads>>>(dev_matrixA, dev_matrixC, matrixSize->
C_width, matrixSize->C_height);
606         err = cudaGetLastError();
607         if (err != cudaSuccess)
608             fprintf(stderr, "ERROR in Matrix Sigmoid Computation: %s\n",
cudaGetErrorString(err));
609         break;
610     }
611     case 4: {
612         // Compute Sigmoid derivative function
613         MatrixSigmoidDerivative<<<grid, threads>>>
614             (dev_matrixA, dev_matrixC, matrixSize->
C_width, matrixSize->C_height);
615         err = cudaGetLastError();
616         if (err != cudaSuccess)
617             fprintf(stderr, "ERROR in Matrix Sigmoid Derivative Computation: %s\
n", cudaGetErrorString(err));
618         break;
619     }
620     default: {
621         fprintf(stderr, "ERROR: No matrix kernel selected. Operation Aborted");
622         break;
623     }
624 }
625
626 // Make sure device is finished
627 err = cudaDeviceSynchronize();

```

```

628     if (err != cudaSuccess)
629         fprintf(stderr, "ERROR synchronizing Matrix Kernel calculation: %s\n",
        cudaGetErrorString(err));
630
631     // Copy data from GPU to host PC
632     err = cudaMemcpy(host_matrixC, dev_matrixC, matrixC_size, cudaMemcpyDeviceToHost
    );
633     if (err != cudaSuccess) fprintf(stderr, "ERROR copying matrix C to Host: %s\n",
        cudaGetErrorString(err));
634
635     // Free GPU memory
636     err = cudaFree(dev_matrixA);
637     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix A on GPU: %s\n",
        cudaGetErrorString(err));
638     err = cudaFree(dev_matrixB);
639     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix B on GPU: %s\n",
        cudaGetErrorString(err));
640     err = cudaFree(dev_matrixC);
641     if (err != cudaSuccess) fprintf(stderr, "ERROR freeing matrix C on GPU: %s\n",
        cudaGetErrorString(err));
642     err = cudaDeviceSynchronize();
643     if (err != cudaSuccess) fprintf(stderr, "ERROR synchronizing Matrix Kernel end:
    %s\n", cudaGetErrorString(err));
644     if (LOGGING == 1) fprintf(stdout, "Matrix Kernel finished.\n");
645 }
646
647
648 //=====
649 //=== Test Function
650 //=====
651
652 void runTest(int argc, char **argv, int devID) {
653     int N = 10;
654     float *host_A, *host_B, *host_C, *host_D;
655     float *host_vA, *host_vB, *host_vC, *host_vD, *host_vE;
656
657     // Create matrices
658     MatrixSize *testMatrixSize = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
659     size_t calcSize = N * N * sizeof(float);
660     host_A = (float *) calloc(calcSize, 1);
661     host_B = (float *) calloc(calcSize, 1);
662     host_C = (float *) calloc(calcSize, 1);
663     host_D = (float *) calloc(calcSize, 1);
664     SetMatrixSize(testMatrixSize, N, N, N, N, N + 1, N + 1);
665
666     // Create vectors
667     VectorSize *testVectorSize = (VectorSize *) calloc(sizeof(VectorSize), 1);
668     size_t calcSize_V = N * sizeof(float);
669     host_vA = (float *) calloc(calcSize_V, 1);
670     host_vB = (float *) calloc(calcSize_V, 1);
671     host_vC = (float *) calloc(calcSize_V, 1);
672     host_vD = (float *) calloc(calcSize_V, 1);
673     host_vE = (float *) calloc(calcSize_V, 1);
674     SetVectorSize(testVectorSize, N);
675
676     // Initialize matrix values
677     for (int i = 0; i < N * N; i++) {
678         host_A[i] = (float) i;
679         host_B[i] = (float) i;
680     }
681
682     // Initialize vector values
683     for (int i = 0; i < N; i++) {
684         host_vA[i] = (float) i;
685         host_vB[i] = (float) i;
686     }
687
688     // MATRIX TESTS
689
690     if (LOGGING == 1) {

```

```

691     fprintf(stdout, "Matrix A:\n");
692     for (int i = 0; i < N + 1; i++) {
693         for (int j = 0; j < N + 1; j++) {
694             fprintf(stdout, "%6.0f ", host_A[i * j]);
695         }
696         fprintf(stdout, "\n");
697     }
698
699     fprintf(stdout, "\nMatrix B:\n");
700     for (int i = 0; i < N; i++) {
701         for (int j = 0; j < N; j++) {
702             fprintf(stdout, "%6.0f ", host_B[i * j]);
703         }
704         fprintf(stdout, "\n");
705     }
706 }
707
708 RunMatrixKernel(argc, argv, devID, testMatrixSize, 3, host_A, host_B, host_C,
709 1.0, 1.0);
710 RunMatrixKernel(argc, argv, devID, testMatrixSize, 4, host_A, host_B, host_D,
711 1.0, 1.0);
712
713 if (LOGGING == 1) {
714     fprintf(stdout, "\nMatrix C:\n");
715     for (int i = 0; i < N; i++) {
716         for (int j = 0; j < N; j++) {
717             fprintf(stdout, "%6.10f ", host_C[i * j]);
718         }
719         fprintf(stdout, "\n");
720     }
721     fprintf(stdout, "\nMatrix D:\n");
722     for (int i = 0; i < N; i++) {
723         for (int j = 0; j < N; j++) {
724             fprintf(stdout, "%6.10f ", host_D[i * j]);
725         }
726         fprintf(stdout, "\n");
727     }
728 }
729
730 // VECTOR TESTS
731
732 if (LOGGING == 1) {
733     fprintf(stdout, "Vector A:\n");
734     for (int i = 0; i < N; i++) {
735         fprintf(stdout, "%6.0f ", host_vA[i]);
736     }
737     fprintf(stdout, "\n");
738     fprintf(stdout, "\nVector B:\n");
739     for (int i = 0; i < N; i++) {
740         fprintf(stdout, "%6.0f ", host_vB[i]);
741     }
742     fprintf(stdout, "\n");
743 }
744
745 RunVectorKernel(argc, argv, devID, testVectorSize, 3, host_vA, host_vB, host_vC,
746 1.0, 1.0);
747 RunVectorKernel(argc, argv, devID, testVectorSize, 4, host_vA, host_vB, host_vD,
748 1.0, 1.0);
749 RunVectorKernel(argc, argv, devID, testVectorSize, 5, host_vA, host_vB, host_vE,
750 1.0, 1.0);
751
752 if (LOGGING == 1) {
753     fprintf(stdout, "Vector C:\n");
754     for (int i = 0; i < N; i++) {
755         fprintf(stdout, "%6.0f ", host_vC[i]);
756     }
757     fprintf(stdout, "\n");
758     fprintf(stdout, "\nVector D:\n");
759     for (int i = 0; i < N; i++) {
760         fprintf(stdout, "%6.0f ", host_vD[i]);
761     }
762     fprintf(stdout, "\n");
763 }

```



```

756     for (int i = 0; i < N; i++) {
757         fprintf(stdout, "%6.10f ", host_vD[i]);
758     }
759     fprintf(stdout, "\n");
760     fprintf(stdout, "\nVector E:\n");
761
762     for (int i = 0; i < N; i++) {
763         fprintf(stdout, "%6.10f ", host_vE[i]);
764     }
765     fprintf(stdout, "\n");
766 }
767 }
768
769 //=====
770 //=== Utility Functions
771 //=====
772
773 void ReadCSV(std::ifstream &file, int elements, float *array)
774 {
775     std::string csvData;
776     getline(file, csvData);
777
778     std::istringstream dataStream(csvData);
779
780     for (int col = 0; col < elements; col++){
781         std::string value;
782         getline(dataStream, value, ',');
783         if ( !dataStream.good() )
784             break;
785         std::istringstream convertor(value);
786         convertor >> array[col];
787     }
788 }
789
790 void InitializeWeights(float *weights, MatrixSize *dims)
791 {
792     int cols = dims->C_width;
793     int rows = dims->C_height;
794     int numEl = cols*rows;
795     for(int idx = 0; idx < numEl; idx++){
796     {
797         weights[idx] = ((float) rand() / (RAND_MAX));
798     }
799 }
800
801 //=====
802 //=== Main Function
803 //=====
804
805 /**
806  * @brief computes weight matrices for a shallow neural network
807  * @param argc - from compiler
808  * @param argv - from compiler
809  * @return 0 if success
810  */
811 int main(int argc, char **argv) {
812     // Assign CUDA variables
813     int devID = 0;
814     cudaGetDevice(&devID);
815     cudaError_t mainErr;
816     //runTest(argc, argv, devID);
817
818     // Define NN layer lengths
819     unsigned int layer_1 = 784;
820     unsigned int layer_2 = 128;
821     unsigned int layer_3 = 10;
822
823     // Allocate memory for matrices and vectors
824     float *a1, *a2, *a3;    // Activation vectors
825     float *z2, *z3;        // Pre-sigmoid intermediary vectors

```

```

826 float *W1, *W2;           // Weight matrices
827 float *y;                 // One-hot result vector
828 float *del3, *del2;       // Error vectors
829 float *scratch1, *scratch2; // Error vectors
830 float *Del2, *Del1;       // Error gradients
831
832 a1 = (float *) calloc((size_t) layer_1, sizeof(float));
833 a2 = (float *) calloc((size_t) layer_2, sizeof(float));
834 a3 = (float *) calloc((size_t) layer_3, sizeof(float));
835 z2 = (float *) calloc((size_t) layer_2, sizeof(float));
836 z3 = (float *) calloc((size_t) layer_3, sizeof(float));
837 y = (float *) calloc((size_t) layer_3, sizeof(float));
838 W1 = (float *) calloc((size_t) layer_2 * layer_1, sizeof(float));
839 W2 = (float *) calloc((size_t) layer_3 * layer_2, sizeof(float));
840 del3 = (float *) calloc((size_t) layer_3, sizeof(float));
841 del2 = (float *) calloc((size_t) layer_2, sizeof(float));
842 scratch1 = (float *) calloc((size_t) layer_2, sizeof(float));
843 scratch2 = (float *) calloc((size_t) layer_2, sizeof(float));
844 Del2 = (float *) calloc((size_t) layer_3 * layer_2, sizeof(float));
845 Del1 = (float *) calloc((size_t) layer_2 * layer_1, sizeof(float));
846
847 // Initialize vector and matrix size structures for computation
848 MatrixSize *inter2 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
849 MatrixSize *inter3 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
850 MatrixSize *grad1 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
851 MatrixSize *grad2 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
852 MatrixSize *backprop1 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
853 MatrixSize *backprop2 = (MatrixSize *) calloc(sizeof(MatrixSize), 1);
854
855 VectorSize *activation2 = (VectorSize *) calloc(sizeof(VectorSize), 1);
856 VectorSize *activation3 = (VectorSize *) calloc(sizeof(VectorSize), 1);
857 VectorSize *delta2 = (VectorSize *) calloc(sizeof(VectorSize), 1);
858 VectorSize *delta3 = (VectorSize *) calloc(sizeof(VectorSize), 1);
859
860 if (LOGGING == 1) fprintf(stdout, "Intermediate 2: ");
861 SetMatrixSize(inter2, 1, layer_1, layer_2, 1, layer_2);
862 if (LOGGING == 1) fprintf(stdout, "Intermediate 3: ");
863 SetMatrixSize(inter3, 1, layer_2, layer_3, layer_2, 1, layer_3);
864 if (LOGGING == 1) fprintf(stdout, "Grad 1: ");
865 SetMatrixSize(grad1, 1, layer_2, 1, layer_1, layer_2, layer_1);
866 if (LOGGING == 1) fprintf(stdout, "Grad 2: ");
867 SetMatrixSize(grad2, 1, layer_3, 1, layer_2, layer_3, layer_2);
868 if (LOGGING == 1) fprintf(stdout, "Backprop 1: ");
869 SetMatrixSize(backprop1, layer_2, layer_1, layer_2, layer_1, layer_2, layer_1);
870 if (LOGGING == 1) fprintf(stdout, "Backprop 2: ");
871 SetMatrixSize(backprop2, layer_3, layer_2, layer_3, layer_2, layer_3, layer_2);
872
873 if (LOGGING == 1) fprintf(stdout, "Activation 2: ");
874 SetVectorSize(activation2, layer_2);
875 if (LOGGING == 1) fprintf(stdout, "Activation 3: ");
876 SetVectorSize(activation3, layer_3);
877 if (LOGGING == 1) fprintf(stdout, "Delta 2: ");
878 SetVectorSize(delta2, layer_2);
879 if (LOGGING == 1) fprintf(stdout, "Delta 3: ");
880 SetVectorSize(delta3, layer_3);
881
882 // Set number of epochs and samples
883 int epochs = 1; // Number of training epochs (iterations through data)
884 int num_train = 20000; // Number of samples;
885 int num_test = 5000;
886
887 // Initialize weights
888 InitializeWeights(W1, grad1);
889 InitializeWeights(W2, grad2);
890
891 // Perform neural network training
892 for (int epoch = 0; epoch < epochs; epoch++) {
893
894     // Open training data files
895     std::ifstream x_train_data("../data/train_img.csv");

```

```

896     std::ifstream y_train_data("./data/train_res.csv");
897
898     for (int sample = 0; sample < num_train; sample++) {
899         // LOAD a1 AND y VECTORS:
900         ReadCSV(x_train_data, layer_1, a1);
901         ReadCSV(y_train_data, layer_3, y);
902
903         // FORWARD PROPOGATION:
904         MatrixMultiplyCUBLAS(argc, argv, devID, inter2, a1, W1, z2, 1.0, 1.0,
false, true); // Compute z2
905         mainErr = cudaGetLastError();
906         if (mainErr != cudaSuccess) fprintf(stderr, "z2 Computation: %s\n",
cudaGetErrorString(mainErr));
907         RunVectorKernel(argc, argv, devID, activation2, 4, z2, z2, a2, 1.0, 1.0)
;
908         // Compute a2
909         mainErr = cudaGetLastError();
910         if (mainErr != cudaSuccess) fprintf(stderr, "a2 Computation: %s\n",
cudaGetErrorString(mainErr));
911         MatrixMultiplyCUBLAS(argc, argv, devID, inter3, a2, W2, z3, 1.0, 1.0,
false, true); // Compute z3
912         mainErr = cudaGetLastError();
913         if (mainErr != cudaSuccess) fprintf(stderr, "z3 Computation: %s\n",
cudaGetErrorString(mainErr));
914         RunVectorKernel(argc, argv, devID, activation3, 4, z3, z3, a3, 1.0, 1.0)
;
915         // Compute a3
916         mainErr = cudaGetLastError();
917         if (mainErr != cudaSuccess) fprintf(stderr, "a3 Computation: %s\n",
cudaGetErrorString(mainErr));
918
919         // BACKWARD PROPOGATION:
920         RunVectorKernel(argc, argv, devID, delta3, 1, z3, y, del3, 1.0, (float)
-1.0); // Compute del3
921         mainErr = cudaGetLastError();
922         if (mainErr != cudaSuccess) fprintf(stderr, "del3 Computation: %s\n",
cudaGetErrorString(mainErr));
923
924         MatrixMultiplyCUBLAS(argc, argv, devID, inter3, del3, W2, scratch1, 1.0,
1.0, false, false); // Compute del2 lhs
925         mainErr = cudaGetLastError();
926         if (mainErr != cudaSuccess) fprintf(stderr, "pre-del2 lhs Computation: %
s\n", cudaGetErrorString(mainErr));
927
928         RunVectorKernel(argc, argv, devID, delta2, 5, z2, y, scratch2, 1.0, (
float) -1.0); // Compute del2 rhs
929         mainErr = cudaGetLastError();
930         if (mainErr != cudaSuccess) fprintf(stderr, "pre-del2 rhs Computation: %
s\n", cudaGetErrorString(mainErr));
931
932         RunVectorKernel(argc, argv, devID, delta2, 2, scratch1, scratch2, del2,
1.0, (float) -1.0); // Compute del2
933         mainErr = cudaGetLastError();
934         if (mainErr != cudaSuccess) fprintf(stderr, "del2 Computation: %s\n",
cudaGetErrorString(mainErr));
935
936         MatrixMultiplyCUBLAS(argc, argv, devID, grad1, del2, a1, Del1, 1.0, 1.0,
true, false); // Compute Del1
937         mainErr = cudaGetLastError();
938         if (mainErr != cudaSuccess) fprintf(stderr, "Del1 Computation: %s\n",
cudaGetErrorString(mainErr));
939
940         MatrixMultiplyCUBLAS(argc, argv, devID, grad2, del3, a2, Del2, 1.0, 1.0,
true, false); // Compute Del2
941         mainErr = cudaGetLastError();
942         if (mainErr != cudaSuccess) fprintf(stderr, "Del2 Computation: %s\n",
cudaGetErrorString(mainErr));
943
944         // Gradient descent
945         RunMatrixKernel(argc, argv, devID, backprop1, 1, W1, Del1, W1, 1.0,
(float) -1.0 / (float) num_train); // Compute new W1
946         RunMatrixKernel(argc, argv, devID, backprop2, 1, W2, Del2, W2, 1.0,

```

```

946         (float) -1.0 / (float) num_train); // Compute new W2
947     cudaDeviceSynchronize();
948     if ( (sample % 1000) == 0) printf("Iteration: %d\n", sample);
949     if ( (sample % 5000) == 0) cudaDeviceReset();
950 }
951 // Close training data files
952 x_train_data.close();
953 y_train_data.close();
954
955 }
956
957 // Open verification data files
958 std::ifstream x_test_data("./data/tests_img.csv");
959 std::ifstream y_test_data("./data/tests_res.csv");
960
961 int correct = 0;
962
963 for(int test_sample = 0; test_sample < num_test; test_sample++)
964 {
965     // LOAD a1 AND y VECTORS:
966     ReadCSV(x_test_data, layer_1, a1);
967     ReadCSV(y_test_data, layer_3, y);
968
969     // FORWARD PROPOGATION:
970     MatrixMultiplyCUBLAS(argc, argv, devID, inter2, a1, W1, z2, 1.0, 1.0, false,
true); // Compute z2
971     mainErr = cudaGetLastError();
972     if (mainErr != cudaSuccess) fprintf(stderr, "z2 Computation: %s\n",
cudaGetErrorString(mainErr));
973     RunVectorKernel(argc, argv, devID, activation2, 4, z2, z2, a2, 1.0, 1.0);
974     // Compute a2
975     mainErr = cudaGetLastError();
976     if (mainErr != cudaSuccess) fprintf(stderr, "a2 Computation: %s\n",
cudaGetErrorString(mainErr));
977     MatrixMultiplyCUBLAS(argc, argv, devID, inter3, a2, W2, z3, 1.0, 1.0, false,
true); // Compute z3
978     mainErr = cudaGetLastError();
979     if (mainErr != cudaSuccess) fprintf(stderr, "z3 Computation: %s\n",
cudaGetErrorString(mainErr));
980     RunVectorKernel(argc, argv, devID, activation3, 4, z3, z3, a3, 1.0, 1.0);
981     // Compute a3
982     mainErr = cudaGetLastError();
983     if (mainErr != cudaSuccess) fprintf(stderr, "a3 Computation: %s\n",
cudaGetErrorString(mainErr));
984
985     float a3max = 0.0;
986     int a3max_idx = 0;
987     int ymax_idx = 0;
988
989     for(int i = 0; i < layer_3; i++)
990     {
991         if(a3[i] > a3max)
992         {
993             a3max = a3[i];
994             a3max_idx = i;
995         }
996         if(y[i] == 1) ymax_idx = i;
997     }
998     if(ymax_idx == a3max_idx) correct++;
999 }
1000 x_test_data.close();
1001 y_test_data.close();
1002 printf("The network correctly identified %d of %d samples\n", correct, num_test)
;
1003 return 0;
1004 }

```