

# Assignment 2: Image Processing Library to Recognize MNIST Digits

April 9, 2024

Name	Student ID
Rishi Chandra	2021CS10584
Sharad Kumar	2021CS10099
Chinmay Salunkhe	2021CS10590

## 1 Subtask 1: Implement of some C++ functions

### 1.1 Convolution of square matrix

```
void convolution(int mRow, int mCol, int knRow, int knCol, float** kernel, float** matrix, float** ans, float bias ){
    int ansRow = mRow - knRow + 1;
    int ansCol = mCol - knCol + 1;

    for(int i = 0 ; i<ansRow ; i++){
        for(int j = 0 ; j<ansCol ; j++){
            float sum = 0;
            for(int p = 0 ; p<knRow ; p++){
                for(int k = 0 ; k<knCol ; k++){
                    sum += kernel[p][k] * matrix[i+p][j+k];
                }
            }
            ans[i][j] = sum + bias;
        }
    }
}
```

Figure 1: Function to initialise matrix

This function convolution computes a 2D convolution between a kernel and a matrix, storing the result in an output matrix ans. It iterates through each element of ans, calculates the convolution by multiplying corresponding elements of the kernel and input matrix, adds a bias term, and stores the result in ans. The function assumes valid input dimensions for the matrices and kernel.

## 1.2 Relu Function

The ReLU function sets all negative values in the input matrix to zero, leaving positive values unchanged. It is commonly used as an activation function in neural networks to introduce non-linearity. For each element in the matrix, if the value is less than zero, it is replaced with zero; otherwise, it remains unchanged.

```
void ReLU(float** matrix, int row, int col){
    for(int i = 0 ; i<row ; i++){
        for(int j = 0 ; j<col ; j++){
            if(matrix[i][j]<0) matrix[i][j] = 0;
        }
    }
}
```

Figure 2: Function to initialise matrix

## 1.3 Tanh Function

The tanH function calculates the hyperbolic tangent of each element in the input matrix. It maps input values from the range  $(-\infty, \infty)$  to the range  $(-1, 1)$ , providing a smooth non-linear transformation. The formula  $(\exp(\text{temp}) - \exp(-\text{temp})) / (\exp(\text{temp}) + \exp(-\text{temp}))$  computes the tanH value for each element in the matrix. It is often used in neural networks for its properties such as zero-centered output and saturation prevention.

```
void tanH(float** matrix, int row, int col){
    for(int i = 0 ; i<row ; i++){
        for(int j = 0 ; j<col ; j++){
            float temp = matrix[i][j];
            temp = (exp(temp)-exp(-temp))/(exp(temp)+exp(-temp));
            matrix[i][j] = temp;
        }
    }
}
```

Figure 3: Function to initialise matrix

## 1.4 Max pooling function

The maxPooling function downsamples a matrix by dividing it into pooling windows of size pooling\_size. Within each window, it finds the maximum value and assigns it to the corresponding position in the output matrix (ans). This operation reduces the dimensions of the matrix while retaining the most significant values, commonly used in convolutional neural networks for feature extraction.

```
// C++ code for max pooling
void maxPooling(int mRow, int mCol, int pooling_size, float** matrix, float** ans){
    for(int i = 0 ; i<mRow ; i+=pooling_size){
        for(int j = 0 ; j<mCol ; j+=pooling_size){
            float maxi = matrix[i][j];
            for(int p = i ; p<i+pooling_size ; p++){
                for(int k = j ; k<j+pooling_size ; k++){
                    if(maxi < matrix[p][k]) maxi = matrix[p][k];
                }
            }
            ans[i/pooling_size][j/pooling_size] = maxi;
        }
    }
}
```

Figure 4: Function to initialise matrix

## 1.5 Avg pooling function

The avgPooling function performs average pooling on a matrix, dividing it into non-overlapping windows of size pooling\_size. Within each window, it calculates the average value and assigns it to the corresponding position in the output matrix (ans). This pooling operation helps reduce spatial dimensions while preserving average feature information.

```
// C++ code for avg pooling
void avgPooling(int mRow, int mCol, int pooling_size, float** matrix, float** ans){

    for(int i = 0 ; i<mRow ; i+=pooling_size){
        for(int j = 0 ; j<mCol ; j+=pooling_size){
            float sum = 0;
            for(int p = i ; p<i+pooling_size ; p++){
                for(int k = j ; k<j+pooling_size ; k++){
                    sum += matrix[p][k];
                }
            }
            ans[i/pooling_size][j/pooling_size] = sum/(pooling_size*pooling_size);
        }
    }
}
```

Figure 5: Function to initialise matrix

## 1.6 Softmax and Sigmoid function

### **sigmoid:**

This function computes the sigmoid activation function for a given array of floats. The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

where  $x$  is the input value. It iterates through each element of the array, applies the sigmoid function, and updates the element with the result.

### **softmax:**

This function computes the softmax activation function for a given array of floats. Softmax function is commonly used in machine learning for multi-class classification problems. It computes the probabilities corresponding to each class. The softmax function for an element  $x_i$  in the array is defined as:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

where  $n$  is the number of elements in the array. It first calculates the sum of exponentials of all elements in the array (denominator), then divides each element's exponential by this sum to obtain normalized probabilities.

```
void sigmoid(int n, float* vec){
    for(int i = 0 ; i<n ; i++){
        vec[i] = 1/(1+exp(-vec[i]));
    }
}

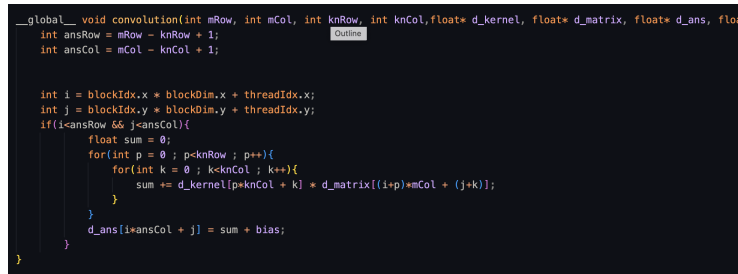
//called in main.c
void softmax(int n, float* vec){
    float dr = 0;
    for(int i = 0 ; i<n ; i++){
        dr += (float)exp(vec[i]);
        vec[i] = (float)exp(vec[i]);
    }
    for(int i = 0 ; i<n ; i++){
        vec[i] = vec[i]/dr;
    }
}
```

Figure 6: Function to initialise matrix

## 2 Subtask 2: Rewrite parallelizable functions from C++ as CUDA kernels

### 2.1 Convolution of square matrix

This CUDA kernel function, named `convolution`, performs a 2D convolution operation on matrices. Given input matrices (`d_kernel` and `d_matrix`), it computes the convolution and stores the result in `d_ans`. The kernel operates on blocks and threads to parallelize the computation efficiently. It takes parameters such as the dimensions of input matrices (`mRow`, `mCol`, `knRow`, `knCol`), the input matrices themselves (`d_kernel`, `d_matrix`), the output matrix (`d_ans`), and a bias term (`bias`). The function loops through the elements of the output matrix, computing the convolution by multiplying corresponding elements of the kernel and the input matrix and summing them up. Finally, it adds the bias term to the computed sum before storing it in the output matrix.

A screenshot of a code editor with a dark background and light-colored text. The code is a CUDA kernel function named 'convolution'. It takes several parameters: 'mRow', 'mCol', 'knRow', 'knCol', 'd\_kernel', 'd\_matrix', 'd\_ans', and 'bias'. The function calculates the dimensions of the output matrix 'd\_ans' and then iterates over each element in 'd\_ans' using a nested loop. For each element, it calculates a sum by multiplying corresponding elements from 'd\_kernel' and 'd\_matrix' and then adds the 'bias' term. The final result is stored in 'd\_ans'.

```
__global__ void convolution(int mRow, int mCol, int knRow, int knCol, float* d_kernel, float* d_matrix, float* d_ans, float bias)
{
    int ansRow = mRow - knRow + 1;
    int ansCol = mCol - knCol + 1;

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i < ansRow && j < ansCol){
        float sum = 0;
        for(int p = 0; p < knRow; p++){
            for(int k = 0; k < knCol; k++){
                sum += d_kernel[p*knCol + k] * d_matrix[(i+p)*mCol + (j+k)];
            }
        }
        d_ans[i*ansCol + j] = sum + bias;
    }
}
```

Figure 7: Function to initialise matrix

### 2.2 Relu Function

This CUDA kernel function, named `ReLU`, implements the Rectified Linear Unit (ReLU) activation function element-wise on a matrix `d_matrix`. The function operates in parallel using blocks and threads.

**Parameters:**

- `row`, `col`: Dimensions of the input matrix `d_matrix`.
- `d_matrix`: Pointer to the memory location of the input matrix.

The function calculates the indices `i` and `j` of the matrix element within the block grid. Within each thread, it checks if the value of the element at position `(i, j)` in the matrix is less than zero. If it is, the element is replaced by zero, implementing the ReLU activation function.

This operation ensures that any negative values in the input matrix are replaced by zero, promoting sparsity and introducing non-linearity to the neural network model.

```

__global__ void ReLU(int row, int col, float* d_matrix){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i < row && j < col){
        if(d_matrix[i*col + j] < 0) d_matrix[i*col + j] = 0;
    }
}

```

Figure 8: Function to initialise matrix

## 2.3 Max pooling function

This CUDA kernel function, named `maxPooling`, implements max pooling operation on a 2D matrix. Max pooling reduces the dimensions of the input matrix by taking the maximum value from each sub-region.

**Parameters:**

- `mRow`, `mCol`: Dimensions of the input matrix `d_matrix`.
- `d_matrix`: Pointer to the memory location of the input matrix.
- `d_ans`: Pointer to the memory location where the result of max pooling will be stored.

Within each thread, it checks if the current element's indices ( $i$ ,  $j$ ) satisfy the condition for pooling (i.e., indices are even). If the condition is met, the function initializes `maxi` with the current element's value.

Then, it iterates through a  $2 \times 2$  sub-region starting from the current element's position ( $i$ ,  $j$ ). It compares each element within the sub-region with `maxi` and updates `maxi` if a larger value is found.

Finally, the maximum value `maxi` from the sub-region is stored in the output matrix `d_ans` at the corresponding reduced index ( $i/2$ ,  $j/2$ ).

This kernel effectively downsamples the input matrix by selecting the maximum value from non-overlapping  $2 \times 2$  sub-regions.

```

__global__ void maxPooling(int mRow, int mCol, float* d_matrix, float* d_ans){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i < mRow && j < mCol && i%2==0 && j%2==0){

        float maxi = d_matrix[i*mCol + j];
        for(int p = i ; p<i+2 ; p++){
            for(int k = j ; k<j+2 ; k++){
                if(maxi < d_matrix[p*mCol + k]) maxi = d_matrix[p*mCol + k];
            }
        }
        d_ans[(i/2)*(mCol/2) + j/2] = maxi;
    }
}

```

Figure 9: Function to initialise matrix

### 3 Subtask 3: Implement neural network LENET-5 stitching together the implemeted C++ and CUDA functions

#### 3.1 Code Description

- **Struct data:**
  - Defines a structure to hold various parameters and layers of the neural network, including convolutional kernels, biases, image data, and node layers.
- **Function initialise:**
  - Initializes memory for the data structure.
- **CUDA Kernels:**
  - **convolution:** Performs convolution operation between a kernel and a matrix.
  - **maxPooling:** Performs max-pooling operation on a matrix.
  - **addMatrix:** Adds two matrices element-wise.
  - **ReLU:** Applies the Rectified Linear Unit (ReLU) activation function.
- **Functions for GPU Operations:**
  - **cudaConvolution:** Executes convolution operation on the GPU.
  - **cudaMaxPooling:** Executes max-pooling operation on the GPU.
  - **cudaAddMatrix:** Adds two matrices on the GPU.

- `cudaReLU`: Applies ReLU activation on the GPU.
- **Other Utility Functions:**
  - `softmax`: Implements the softmax function for classification output.
  - `freeMatrix`: Frees memory allocated for matrices.
  - `createMatrix`: Dynamically allocates memory for matrices.
  - `readFiles`: Reads weights and image data from files.
  - `freeMemory`: Frees memory allocated for the `data` structure.
  - `initLayers`: Initializes node layers in the `data` structure.
- **Main Execution Functions:**
  - `conv1`, `maxpool1`, `conv2`, `maxPool2`, `conv3`, `ReLU1`, `conv4`: Execute various layers of the CNN.
  - `softmax_and_print`: Computes softmax probabilities and writes them to an output file.
- **Execution Flow:**
  - `execute`: Orchestrates the execution flow of the CNN for a single image.
  - `execute_folder`: Executes CNN for all images in a specified folder.
- **Main Function:**
  - `main`: Entry point of the program. Calls `execute_folder` to process images in a folder.

## 4 Results

```
Time taken to copy parameters(CPU -> GPU): 86.95 milliseconds
Time taken for inference: 903.98 milliseconds
[cs1210099@khas018 /scratch/pbs/pbs.63446.pbshpc.x8z/pdpfinal/src]
$
```

Figure 10: Time on 1000 images on HPC wo streams

```
Time taken to copy parameters(CPU -> GPU): 2115.20 milliseconds
Time taken for inference: 4227.73 milliseconds
[cs1210099@khas022 /scratch/pbs/pbs.63485.pbshpc.x8z/pdpfinal/src]
$
```

Figure 11: Time on 5000 images on HPC wo streams



```

Time taken to copy parameters(CPU -> GPU): 2103.68 milliseconds
Time taken for inference: 8434.79 milliseconds
[cs1210099@khas022 /scratch/pbs/pbs.63485.pbshpc.x8z/pdpfinal/src]
$

```

Figure 12: Time on 10000 images on HPC wo streams

```

Time taken to copy parameters(CPU -> GPU): 2162.99 milliseconds
Time taken for inference: 6842.94 milliseconds
[cs1210584@khas014 /scratch/pbs/pbs.63554.pbshpc.x8z/pdpfinal/src]
$

```

Figure 13: Time on 10000 images on HPC with streams

```

Time taken to copy parameters(CPU -> GPU): 2175.67 milliseconds
Time taken for inference: 3457.10 milliseconds
[cs1210584@khas014 /scratch/pbs/pbs.63554.pbshpc.x8z/pdpfinal/src]
$

```

Figure 14: Time on 5000 images on HPC with streams

```

Time taken to copy parameters(CPU -> GPU): 4193.99 milliseconds
Time taken for inference: 717.26 milliseconds
[cs1210584@khas014 /scratch/pbs/pbs.63554.pbshpc.x8z/pdpfinal/src]
$

```

Figure 15: Time on 1000 images on HPC with streams

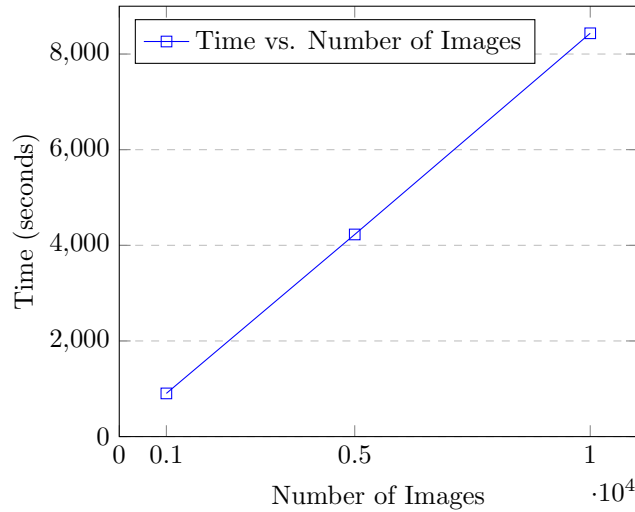


Figure 16: Line graph showing time vs. number of images

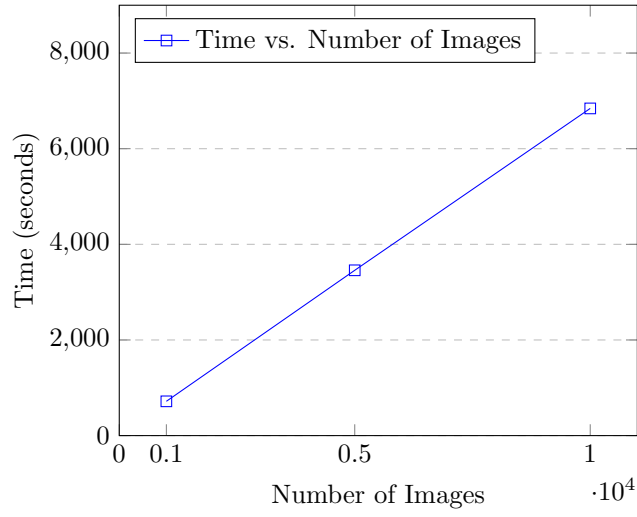


Figure 17: Line graph showing time vs. number of images

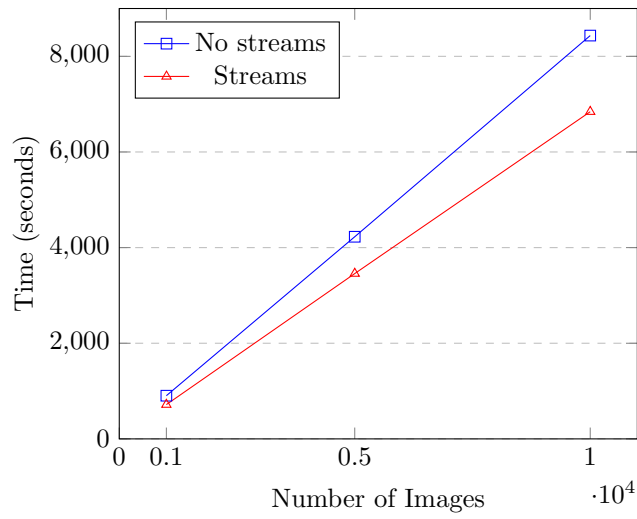


Figure 18: Comparison of two data sets: Line graph showing time vs. number of images