NUPUR UNAVEKAR
Image Processing Library

Files Submitted

```
kerbno1_kerbno2_kerbno3/
├── README.md
├── img/
├── weights/
├── pre-proc-img/
├── preprocessing.py/
├── report.pdf
├── output/
└── src/
    ├── assignment2_subtask1.cu
    ├── assignment2_subtask2.cu
    ├── assignment2_subtask3.cu
    └── assignment2_subtask4.cu
```

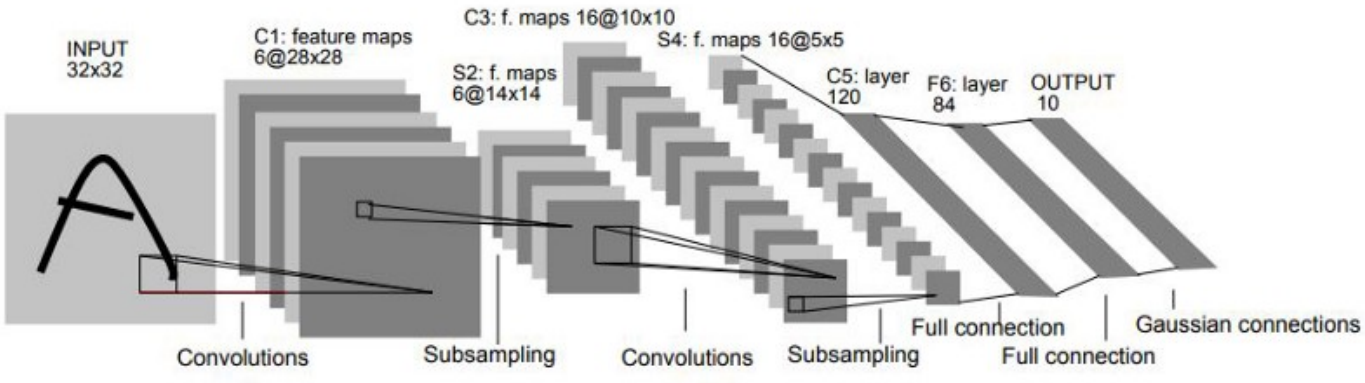

Figure 1: LaNeT-5 Architecture

Commands to run

make all : Run the Makefile to compile all the subtasks

Subtask 1:

1. **Convolution:** ./subtask1 1 <N> <M> <P> <Matrix of size N*N> <Kernel of size M*M> <Padding value P>
2. **Non-linear-activations:**
   (a) **ReLU:** ./subtask1 2 0 <N> <M> <Matrix of size N*M>
   (b) **tanH:** ./subtask1 2 1 <N> <M> <Matrix of size N*M>
3. **Subsampling:**
   (a) **Max Pooling:** ./subtask1 3 0 <M> <N> <Matrix of size N*N>
   (b) **Avg Pooling:** ./subtask1 3 1 <M> <N> <Matrix of size N*N>
4. **Converting a vector:**
   (a) **Sigmoid:** ./subtask1 4 0 <Vector of numbers>
   (b) **Softmax:** ./subtask1 4 1 <Vector of numbers>

Subtask 2:

1. **Convolution:** ./subtask2 1 <N> <M> <P> <Matrix of size N*N> <Kernel of size M*M> <Padding value P>
2. **Non-linear-activations:**
   (a) **ReLU:** ./subtask2 2 0 <N> <M> <Matrix of size N*M>
   (b) **tanH:** ./subtask2 2 1 <N> <M> <Matrix of size N*M>
3. **Subsampling:**
   (a) **Max Pooling:** ./subtask2 3 0 <M> <N> <Matrix of size N*N>
   (b) **Avg Pooling:** ./subtask2 3 1 <M> <N> <Matrix of size N*N>
4. **Converting a vector:**
   (a) **Sigmoid:** ./subtask2 4 0 <Vector of numbers>
   (b) **Softmax:** ./subtask2 4 1 <Vector of numbers>

Subtask 3:

1. ./subtask3

Subtask 4:

1. With streams: ./subtask4 1
2. Without streams: ./subtask4 0

Subtask 1

1. **Convolution:** The function `ConvolutionWithPadding()` takes three arguments: the input matrix, the kernel matrix, and the padding value. It returns the result after multiplying the padded input with the filter/kernel. We're using the same padding rule.
2. **ReLU/Tanh Activations:** The functions `applyRelu()` and `applyTanh()` are simple functions for applying non-linear activations. They take a 2D array (matrix) as input and return the maximum of each element with 0 or the tanh function applied to each element, respectively.
3. **Max Pooling:** The function `maxPooling()` takes two arguments: the input matrix and the pool size. It returns a 2D array of maximum values from pools across the input matrix, with a suitable output size.
4. **Average Pooling:** The function `averagePooling()` also takes the input matrix and pool size as arguments. It returns a 2D array of average values from pools across the input matrix, with a suitable output size.
5. **Softmax & Sigmoid Functions:** The functions `softmax()` and `sigmoid()` are straightforward. They take a 1D vector as input and return the output vector resulting from applying the softmax or sigmoid functions, respectively.

Note: Stride is assumed to be 1.

Subtask 2

1. The input and output format for all the functions is same as Subtask1.
2. Kernels were created for convolution, maxPooling and averagePooling functions using CUDA. The stride as specified for convolution and maxPooling is 1 (number of filters is also just 1).
3. Now for convolution if the input matrix size is N*N and kernel size is k*k, output matrix size will be (N-k+1)*(N-k+1). So our time complexity would be $O((N-k+1)^2 * k^2)$.
4. Similarly for maxPooling and averagePooling each, input matrix size N*N, and pool window of p*p, the time complexity would be $O((N-k+1)^2 * k^2)$.
5. Since these 2 functions require more computation than other functions, CUDA is used to improve their throughput using GPU parallelization.

Subtask 3

We implemented the LENET-5 neural network, using convolution, maxPooling, fullyConnected, reLU, Softmax functions. First we used our python script to convert each image in the MNIST test dataset to a txt file (With the same name as the png image) containing 28*28 floats. We made a .cpp file which implemented the neural network in accordance with the LENET architecture and the dimensions mentioned in details.txt file. We have the following layers: convolution1, maxPooling1, convolution2, maxPooling2, FullyConnected1 (with reLU function), FullyConnected2 and finally the softmax function. The code on testing on the dataset of 10,000 images had an accuracy of 99.09 % (comparing the digit with the highest probability given by our network and the actual digit).

After the implementation of cpp code, it was optimized using the CUDA kernels for a better efficiency and so the convolution function and maxPooling functions were converted to kernels. For convolution the blockSize is used as 16*16, and the grid size is used as $(\frac{(inputSize+blockSize.x-1)}{blockSize.x}, \frac{(inputSize+blockSize.y-1)}{blockSize.y})$. For maxPooing the blockSize is (poolKernelSize, poolKernelSize) and grid size is $(\frac{(pooledSize+poolBlockSize.x-1)}{poolBlockSize.x}, \frac{(pooledSize+poolBlockSize.y-1)}{poolBlockSize.y})$. At each such layer the host copied the required matrices to the device. It performed the calculations and copied back the matrix output of that layer to the host. To improve the efficiency further the output matrices were not copied back to the host at every layer rather they were directly used by the next layer on the device. This reduced the cost to copy back at every layer.

```
__global__ void convolution(const float* inputMatrix, const float* kernel, const float* bias,
                                float* outputMatrix, int inputChannel, int outputChannel,
                                int inputSize, int outputSize, int kernelSize) {
    // Calculate the global thread index
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int d = blockIdx.z;

    if (i < outputSize && j < outputSize) {
        float sum = 0.0f;
        for (int c = 0; c < inputChannel; c++) {
            for (int ki = 0; ki < kernelSize; ki++) {
                for (int kj = 0; kj < kernelSize; kj++) {
                    int inputIdx =(c*inputSize*inputSize)+ ((i + ki)* inputSize) + (j + kj);
                    int kernelIdx=(d * inputChannel * kernelSize * kernelSize) + (c * kernelSize *
    kernelSize) + (ki * kernelSize) + kj;
                    sum += inputMatrix[inputIdx] * kernel[kernelIdx];
                }
            }
        }
        int outIdx= (d* outputSize*outputSize) + (i*outputSize) +j;
        outputMatrix[outIdx] = sum + bias[d];
    }
}
```

Listing 1: Convolution Kernel

```
__global__ void maxPoolingKernel(const float* inputVolume, float* outputVolume,
                                 int inputChannels, int inputSize,
                                 int outputSize, int kernelSize) {
    // Calculate the global thread index
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int c = blockIdx.z;

    if (i < outputSize && j < outputSize) {
        float maxVal = -FLT_MAX;
        for (int ki = 0; ki < kernelSize; ++ki) {
            for (int kj = 0; kj < kernelSize; ++kj) {
                int inputIdx = c * inputSize * inputSize + (i * kernelSize + ki) * inputSize + (j *
    kernelSize + kj);
                float val = inputVolume[inputIdx];
                maxVal = fmaxf(maxVal, val);
            }
        }

        outputVolume[c * outputSize * outputSize + i * outputSize + j] = maxVal;
    }
}
```

Listing 2: Max Pooling Kernel

The CUDA implementation begins by defining parameters such as the size of the flattened input array (`inputSizeImage`), input size (`inputSize1`), input channel (`inputChannel1`), output channel (`outputChannel1`), and kernel size (`kernelSize1`). The input data is read from a file into the `inputArray` using `readInputArrayFromFile()` function. Memory allocation and data transfer between host and device are performed using `cudaMalloc()` and `cudaMemcpy()` functions. CUDA kernel configurations, including block size and grid size (`blockSize` and `gridSize`), are set for the convolution operation. The convolution operation is executed using `convolution` kernel. Memory is freed using `cudaFree()` after each operation. Max pooling operation is performed using `maxPoolingKernel` with appropriate configurations. This process is repeated for subsequent convolution and max pooling operations. ReLU activation is applied using `applyReLU()` function. Fully connected layer operation (`fullyConnected()`) and softmax activation (`applySoftmax()`) are performed subsequently.

Below is the pseudocode representing the provided C++ implementation:

```
const int inputSizeImage = 28 * 28 * 1;
const int inputSize1 = 28;
const int inputChannel1 = 1;
const int outputChannel1 = 20;
const int kernelSize1 = 5;
float inputArray[inputSizeImage];
const std::string filename2 = filenaame;
// Define the readInputArrayFromFile function
readInputArrayFromFile(inputArray, filename2, inputSizeImage);

// Define the CUDA memory allocation and data transfer functions
cudaMalloc(), cudaMemcpy();
dim3 blockSize(16, 16);
dim3 gridSize((inputSize1 + blockSize.x - 1) / blockSize.x,
              (inputSize1 + blockSize.y - 1) / blockSize.y, outputChannel1);

// Define the convolution kernel invocation (con1)
convolution<<<gridSize, blockSize, 0>>>(d_inputMatrix, d_convMatrix,
                                        d_bias, d_outputMatrix,
                                        inputChannel1, outputChannel1,
                                        inputSize1, inputSize1 - kernelSize1 + 1,
                                        kernelSize1);
cudaFree();
. .
dim3 poolBlockSize(poolKernelSize1, poolKernelSize1);
dim3 poolGridSize();
// max pooling (mp1)
maxPoolingKernel<<<poolGridSize, poolBlockSize>>>(d_outputMatrix, d_pooledOutput,
                                                  outputChannel1, outputSize1,
                                                  pooledSize1, poolKernelSize1);
. .
//convolution2 (con2)
convolution<<<gridSize2, blockSize2, 0>>>(d_inputMatrix2, d_convMatrix2,
                                        d_bias2, d_outputMatrix2,
                                        inputChannel2, outputChannel2,
                                        inputSize2, inputSize2 - kernelSize2 + 1,
                                        kernelSize2);
..
// max pooling2 (mp2)
maxPoolingKernel<<<poolGridSize2, poolBlockSize2,0>>>(d_outputMatrix2, d_pooledOutput2,
                                                  outputChannel2, outputSize2,
                                                  pooledSize2, poolKernelSize2);
. .
// convolution3 (fc1)
convolution<<<gridSize3, blockSize3, 0>>>(d_pooledOutput2, d_convMatrix3,
                                        d_bias3, d_outputMatrix3,
                                        inputChannel3, outputChannel3,
                                        inputSize3, inputSize3 - kernelSize3 + 1,
                                        kernelSize3);
. .
//relu
applyReLU(outputMatrix3, outputSize3, outputChannel3);  //500*1*1
..
// last convolution (fc2)
fullyConnected(outputMatrix3, convMatrix4, bias4, outputMatrix4, inputChannel4, outputChannel4,
                                       inputSize4, outputSize4, kernelSize4);
..
applySoftmax(outputMatrix4, outputMatrix5, outputSize4, outputChannel4);//(finally probablities)
```

Listing 3: C++ Pseudocode

Subtask 4

In the following code snippet, we demonstrate how CUDA streams are created and utilized in a multi-stream execution environment. Following was the architecture we used for cuda streams

```
#include <cuda_runtime.h>
#include <iostream>

int main() {
    // Number of streams
    int number_of_streams = 10000;

    // Creating array of CUDA streams
    cudaStream_t streams[number_of_streams];
    for(int i = 0; i < number_of_streams; i++) {
        cudaStreamCreate(&streams[i]);
    }

    //Assume that the constants that used in multiplication,etc
    //are already defined

    float * inputArray;
    cudaMallocHost(&inputArray, inputSizeImage * sizeof(float));
    const std::string filename2 = filenaame;
    readInputArrayFromFile(inputArray, filename2, inputSizeImage);

    float *d_inputMatrix, *d_convMatrix, *d_bias, *d_outputMatrix;
    cudaMalloc(&d_inputMatrix, 784 * sizeof(float));
    cudaMalloc(&d_convMatrix, outputChannel1 * inputChannel1 * kernelSize1 * kernelSize1 * sizeof(
    float));
    cudaMalloc(&d_bias, outputChannel1 * sizeof(float));
    cudaMalloc(&d_outputMatrix, outputChannel1 * outputSize1 * outputSize1 * sizeof(float));

    cudaMemcpyAsync(d_inputMatrix, inputArray, inputSizeImage * sizeof(float), cudaMemcpyHostToDevice,
     stream);
    cudaMemcpyAsync(d_convMatrix, convMatrix, outputChannel1 * inputChannel1 * kernelSize1 *
    kernelSize1 * sizeof(float), cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync(d_bias, bias1, outputChannel1 * sizeof(float), cudaMemcpyHostToDevice, stream);
    dim3 blockSize(16, 16);
    dim3 gridSize((inputSize1 + blockSize.x - 1) / blockSize.x, (inputSize1 + blockSize.y - 1) /
    blockSize.y, outputChannel1);
    convolution<<<gridSize, blockSize, 0, stream>>>(d_inputMatrix, d_convMatrix, d_bias,
    d_outputMatrix, inputChannel1, outputChannel1, inputSize1, inputSize1 - kernelSize1 + 1,
    kernelSize1);
    cudaFree(d_inputMatrix);
    cudaFree(d_convMatrix);
    cudaFree(d_bias);
    .
    .
    .

    //Finally freeing the host memory
    cudaFreeHost(inputArray);
    cudaFreeHost(outputMatrix3);
    cudaFreeHost(outputMatrix4);
    cudaFreeHost(outputMatrix5);

    // Synchronizing streams
    for (int i = 0; i < number_of_streams; i++) {
        cudaStreamSynchronize(streams[i]);
    }

    // Destroying streams
    for(int i = 0; i < number_of_streams; i++) {
        cudaStreamDestroy(streams[i]);
    }

    return 0;
}
```

Listing 4: Creating and Utilizing CUDA Streams

CUDA streams are a powerful feature in CUDA programming that allow for concurrent execution of multiple tasks on the GPU. They enable overlap of kernel execution with memory transfers and other kernel executions, thus maximizing GPU utilization and improving overall performance.

In the provided code snippet, CUDA streams are utilized to execute multiple convolution operations concurrently. Here's a detailed explanation of the significance and implementation of CUDA streams in the context of the provided code:

1. **Creation of CUDA Streams (Lines 8-12):**
   - `cudaStreamCreate()` function is used to create an array of CUDA streams (`streams`) with a total of `number_of_streams` streams.
   - Each stream represents a separate execution context on the GPU. Creating multiple streams allows for concurrent execution of tasks.
2. **Memory Allocation (Lines 17-26):**
   - Host and device memory is allocated for input, convolution matrices, biases, and output matrices using `cudaMalloc()` and `cudaMallocHost()` functions.
   - Memory allocation is essential for storing input data, weights, biases, and output results on the GPU.
3. **Data Transfer and Kernel Invocation (Lines 28-33):**
   - Input data (`inputArray`), convolution matrices (`convMatrix`), and biases (`bias1`) are asynchronously copied from host to device memory using `cudaMemcpyAsync()` function with the specified stream `stream`.
   - Kernel function `convolution` is launched with the specified grid and block dimensions and associated with the stream `stream`. This enables the convolution operation to be executed asynchronously with other tasks associated with the same stream.
4. **Memory Deallocation (Lines 34-36):**
   - Device memory allocated for input, convolution matrices, and biases is freed using `cudaFree()` function. This step is important for memory management and avoids memory leaks.
5. **Synchronization (Lines 47-50):**
   - After all tasks associated with CUDA streams have been launched, `cudaStreamSynchronize()` function is called for each stream to ensure that all operations associated with the stream have completed before proceeding further.
   - Synchronization is crucial to ensure correct execution order and to avoid race conditions between CPU and GPU operations.
6. **Stream Destruction (Lines 52-55):**
   - Finally, CUDA streams are destroyed using `cudaStreamDestroy()` function to release associated resources.

Significance of CUDA Streams:

- **Concurrency:** CUDA streams enable concurrent execution of tasks on the GPU, thereby utilizing the GPU resources more efficiently and improving performance.
- **Overlap of Operations:** By using streams, operations like memory transfers and kernel executions can be overlapped, reducing overall execution time.
- **Asynchronous Execution:** Tasks associated with different streams can execute asynchronously, allowing for better utilization of available GPU resources and increased throughput.

Timings

We have tested the architecture (LaNeT5) on all the 10000 images and found out that without using CUDA streams we are able to achieve the timing of around **36-40 seconds** while with using different number of streams (non-default streams) (10, 100, 1000, 5000, 10000) we first had timing equal to that with default stream, roughly 35 seconds, but the timing improved to around **32-34 seconds** with a larger number of streams. Although the difference was not significant, the streams achieved the purpose of concurrency and showed a bit improvement. We observed that our `convolution()` and `maxpooling()` functions take roughly around 120 microseconds (which is considerably fast). Reading the weights takes around 200 milliseconds. For the same, we are storing the weights in an input array to avoid unnecessary delays.
NOTE : All the timings were noted using high frequency clock using the module `<ctime>`