Department of Computer Science & Engineering
COL380 - Intro to Parallel and Distributed Programming

Assignment 2

Prepared by: Anumula Venkata Jaya Chaitanya,
Mythravarun Bojanapally, Guttikonda Veera Teja
Entry No: 2021CS10088, 2021CS10573, 2021CS10107
Instructor: Rijurekha Sen
Date: April 9, 2024

Contents

1 Sequential Algorithm

The main aim of this assignment is to create a small library to recognize MNIST digits using Convolutional neural networks in C++ with the integration of OpenCV(used to process images and convert them into a 28×28 matrix). The CNN architecture consists of two convolutional layers followed by max-pooling layers, and two fully connected layers.

CNN Architecture:

- **Input Layer:** Accepts images with dimensions of 28×28 pixels.
- **Convolutional Layer 1:** performs convolutions of an image with 20 kernels of size 5×5.
- **Max Pooling Layer 1:** implements 2×2 max pooling.
- **Convolutional Layer 2:** performs convolutions of an image with 50 kernels of size 5×5.
- **Max Pooling Layer 2:** implements 2×2 max pooling.
- **Fully Connected Layer 1:** Consists of 500 neurons.
- **Fully Connected Layer 2:** Outputs to 10 neurons, serving as the output layer for digit classification.

1.1 Implementation

First we load the trained weights

Next we get the filenames of all the images using getFilenames function.

- **getFilenames() :**This function opens the directory specified by folderPath, iterates through its entries, and adds the filenames to a vector. It then sorts the filenames in ascending order and returns the vector.

We also load the trained weights into an array dynamically using loadWeights function

- **loadWeights() :** This function loads pre-trained weights and biases from a specified file and stores them in arrays sequentially. It takes the file location, weight array, bias array, and their respective sizes as parameters.

Now we start iterating through the vector returned by getFilenames function.

For each image in the dataset, a 28×28 matrix is generated and subsequently passed as input to the convolution function. Convolution function takes 28×28 image condensed to an array in row major order, 5×5 kernel condensed into an array in row major order as input and produces a 24×24 matrix condensed into an array in row major order.

- **convolution() :** This function performs convolution on input data using specified weights and biases. It computes the output dimensions based on input dimensions and kernel size. Then, it iterates through each kernel and calculates the convolution output for each position in the output.

We carry out convolution on individual images using 20 different kernels. This convolution operation produces a 24 $\times$ 24 matrix for each image, with 20 separate channels for respective kernels.

After convolution, the next step is max pooling. In this case, we perform 2 $\times$ 2 pooling, meaning that within each 2 $\times$ 2 region of the matrix, we retain only the maximum value. So, here for each image we are left with 12 $\times$ 12 matrix with 20 channels.

- **maxPooling() :** For each channel, the function traverses the input matrix using nested loops, selecting a window of size kernel_dim $\times$ kernel_dim. Within each window, it identifies the maximum value and stores it in the corresponding position of the output matrix. The output matrix has reduced dimensions (output_dim), calculated as input_dim / kernel_dim.

Now for each image, we have 12 $\times$ 12 matrix with 20 channels. After this, as per our CNN architecture, we perform convolution layer 2 and maxPooling layer 2. After these two layers, we are left with 4 $\times$ 4 matrix with 50 channels for each image.

The next layer in our architecture is a Fully Connected layer, functioning as a convolutional layer with kernels having the same dimensions as the input. Because of this, each channel returns a single output. In this layer we perform the convolution using 500 different kernels resulting in an output containing 500 channels for each image. This can be visualised as an array of length 500 for each image containing float_32 datatype. Additionally in this layer,output undergoes a rectified linear unit (ReLU) activation function.

- **ReLU() :** The ReLU activation function is a simple non-linear function that replaces any negative input values with zeros, leaving positive values unchanged

The second fully connected layer (FC2) similarly connects all units from FC1 to the output layer, which consists of 10 units corresponding to the digit classes (0-9). We apply a softmax activation function to the output of FC2 to obtain probabilities representing the likelihood of each digit class.

- **softmax() :** Softmax function exponentiates each input value, making them positive. It then calculates a normalization constant by summing all exponentiated values. Each exponentiated value is divided by this constant, producing a probability distribution.

The output of the second fully connected layer consists of an array containing 10 probabilities, each representing the likelihood of the image being 0, 1, 2, ..., or 9.

2 Parallelizing sequential code with CUDA kernels

To parallelize the sequential code with CUDA, we transform functions like convolution, maxPooling, and ReLU into parallel CUDA kernels. This requires copying necessary input data from the host to the device, setting up kernels with specific block and thread configurations, and performing operations within individual threads. Each thread typically handles a unit operation, such as computing a single output entry in convolution and maxPooling. Finally, the output is copied back from the device to the host.

First as done in the sequential code, we initialise arrays for each layer to store the pre trained weights and biases. We also load the weights and biases using loadWeights function.

We also initialise the image matrix(28×28) as done in the sequential program.

As the pretrained weights should be sent as an input to the CUDA kernels, we allocate arrays on the GPU using CudaMalloc and transfer the loaded weights from the host to the device (GPU) using cudaMemcpy().

2.1 Convolution kernel

Now, to start the convolution layer, we allocate an array of size : no.of kernels $\times$ output matrix dimension $\times$ output matrix dimension $\times$ dim*sizeof(float).

```
float* conv_1_output_d;
       cudaMalloc(&conv_1_output_d,
          conv_1_num_kernels*conv_1_output_dim*conv_1_output_dim*sizeof(float));
```

Now, we invoke the kernel as follows.

```
convolutionCuda<<<conv_1_num_kernels,dim3(conv_1_output_dim,conv_1_output_dim)>>>
          (image_1d_d, conv_1_output_d, conv_1_weights_d, conv_1_bias_d,
             image_dim, kernel_dim, conv_1_kernel_depth, conv_1_num_kernels);
```

To ensure that every element in the output matrix is computed in a single thread, we initialise a block for each kernel. Each block contains the output matrix dimension $\times$ and the output matrix dimension no.of threads.

Now in the convolution kernel, we evaluate a single entry of output matrix per thread. The index of the entry can be calculated by the thread Ids and block Id of the kernel as follows.

```
// as we store the matrices in a 1D array in row major order we calculate
    only one index.
// kernel_depth is the no.of channels in the filter.
int row = threadIdx.y;
int col = threadIdx.x;
int kernel = blockIdx.x;
float sum = 0.0f;
for each channel i, each row j, each column k of kernel(filter)
    int weight_index = kernel*kernel_depth*kernel_dim*kernel_dim +
        i*kernel_dim*kernel_dim + j*kernel_dim + k;
    int input_index = i*input_dim*input_dim + (row+j)*input_dim + (col+k);
    sum += weights[weight_index]*input[input_index]

output[kernel*output_dim*output_dim + row*output_dim + col] = sum +
    bias[kernel];
```

Before writing into the output, we determine the channel we are writing into using the block ID, and calculate the row and column using the thread IDs. To minimize the number of writes into the output matrix, we first initialize a local variable called "sum" and complete the computation before writing into the output matrix. This output matrix is the input of the first maxPooling kernel.

2.2 maxPooling kernel

Now, to initialize the max pooling layer, we allocate an array of size: number of kernels $\times$ output matrix dimension $\times$ output matrix dimension $\times$ sizeof(float).

```
float* pool_1_output_d;
cudaMalloc(&pool_1_output_d,
    conv_1_num_kernels*pool_1_output_dim*pool_1_output_dim*sizeof(float));
```

We initialize the kernel as follows:

```
maxPoolingCuda<<<conv_1_num_kernels,dim3(pool_1_output_dim,pool_1_output_dim)>>>
(input_d, pool_1_output_d, conv_1_output_dim, conv_1_num_kernels, pool_dim);
```

For a single image, we initialize a block for each kernel(we use the no.of kernels variable initialised for convolution layer as both are the same), and each block contains output matrix dimension $\times$ output matrix dimension number of threads, ensuring each entry in the output matrix is computed in a single thread.

Now, in the max pooling kernel, we evaluate a single entry of the output matrix per

thread. The index of the entry can be calculated by the thread IDs and block ID of the kernel as follows:

```
int output_dim = input_dim / kernel_dim;
int row = threadIdx.y;
int col = threadIdx.x;
int kernel = blockIdx.x;
float mx = -FLT_MAX;
for(int i=0;i<kernel_dim;i++){
    for(int j=0;j<kernel_dim;j++){
        int row_index = row*kernel_dim + i;
        int col_index = col*kernel_dim + j;
        mx = fmaxf(mx, input[kernel*input_dim*input_dim + row_index*input_dim
            + col_index]);
    }
}
output[kernel*output_dim*output_dim + row*output_dim + col] = mx;
```

Here, we determine the output channel, row, and column using the block ID and thread IDs. Then, we iterate over each element in the kernel's region, finding the maximum value, and write it to the corresponding location in the output matrix.

Now the output of the first maxPooling kernel is again passed into the convolution kernel with required arguments and the output is then passed into the maxPooling kernel.

As fully connected layer is also a convolutional layer we again use the convolution kernel and then pass the output into ReLU activation function.

2.3 ReLU kernel

Now, for the ReLU activation function, we simply assign all the output elements in one block, with the number of threads equal to the number of elements in the output. Each thread performs the ReLU operation on one element of the output.

```
reluCuda<<<1,fc_1_num_neurons>>>(fc_1_output_d, fc_1_num_neurons);
```

The ReLU kernel is defined as follows:

```
__global__ void reluCuda(float* input, int size){
    int index = threadIdx.x;
    input[index] = max(0.0f, input[index]);
}
```

In this kernel, each thread determines its index and applies the ReLU function to the corresponding element of the input array.

The output of the ReLU kernel is again passed into the Fully connected layer 2(convolution) and then we perform softmax for the output of FC2 layer.

We also ensured to release the memory allocated on the GPU and Host after completing the computations to prevent memory leaks.

In this way we were able to parallelize the sequential code using CUDA kernels. Next we also implement CudaStreams on this program to increase throughput, so that multiple images can be concurrently processed.

3 Cuda Streams

3.1 CUDA Streams Implementation

To incresase the throughput using CUDA streams, we process multiple images concurrently. So we create individual stream for each image as follows.

```
cudaStream_t streams[filenames.size()];
for(int i=0;i<filenames.size();i++){
    cudaStreamCreate(&streams[i]);
}
```

Listing 1: CUDA Streams Initialization

Now to load weights and biases using loadWeights function, we allocate memory on both host and device. We allocate memory on device using `cudaMallocHost` because we are using `cudaMemcpyAsync` which requires pinned memory. As usually we allocate memory on host using `cudaMalloc`.

We also pre-allocate memory on the device for image data and intermediate layer outputs using `cudaMalloc`. We pre-allocate the memory because if we allocate in for-loop it causes synchronisation of streams and defeats our purpose. We load image data into device memory asynchronously using `cudaMemcpyAsync` within each stream.

```
// Memory allocation for image data and intermediate layer outputs
float** images_1d = new float*[filenames.size()];
for(int i=0;i<filenames.size();i++){
    cudaMallocHost(&images_1d[i], image_dim*image_dim*sizeof(float));
}

float** images_1d_d = new float*[filenames.size()];
for(int i=0;i<filenames.size();i++){
    cudaMalloc(&images_1d_d[i], image_dim*image_dim*sizeof(float));
}

// Asynchronously transfer image data to device memory
for(int i=0;i<filenames.size();i++){
    cudaMemcpyAsync(images_1d_d[i], images_1d[i],
```

```
        image_dim*image_dim*sizeof(float), cudaMemcpyHostToDevice, streams[i]);
}
```

Listing 2: Memory Allocation and Async Data Transfer

Next, we launch CUDA kernels for each layer within their respective streams. For example, convolution is launched within each stream as follows.

```
// Convolution 1
convolutionCuda<<<conv_1_num_kernels,dim3(conv_1_output_dim,conv_1_output_dim),0,streams[i]>
    (images_1d_d[i], conv_1_outputs_d[i], conv_1_weights_d, conv_1_bias_d,
        image_dim, kernel_dim, conv_1_kernel_depth, conv_1_num_kernels);
```

Listing 3: Launching CUDA Kernels within Streams

After completion of launching all kernels, we exit the loop. Now we synchronize the streams.

```
// Synchronize all streams
for(int i=0;i<filenames.size();i++){
    cudaStreamSynchronize(streams[i]);
    cudaStreamDestroy(streams[i]);
}
```

Listing 4: Synchronizing CUDA Streams

Finally, we perform cleanup by freeing allocated memory and destroying streams.

```
// Free allocated memory
for(int i=0;i<filenames.size();i++){
    cudaFreeHost(images_1d[i]);
    cudaFree(images_1d_d[i]);
    // Free memory for other intermediate outputs
}
// Free device memory for weights, biases, and intermediate outputs
cudaFree(conv_1_weights_d);
cudaFree(conv_2_weights_d);
// Free memory for other weights, biases, and intermediate outputs
```

Listing 5: Memory Cleanup

This CUDA streams implementation enables concurrent processing of multiple images, improving overall throughput.

4 Performance

We ran both sequential cuda code and cuda code with streams on image datasets of various sizes. Here are the times taken

Table 1: Time taken for cuda code **without** streams

Image Dataset Size	**Time taken(s)**
1000	7.16
5000	18.09
10000	30.54

Table 2: Time taken for cuda code **with** streams

Image Dataset Size	**Time taken(s)**
1000	6.19
5000	9.21
10000	18.09