COL380 A2

An image processing library implemented with C++ and CUDA to recognize hand-written digits

Authors:
Yatharth Kumar, 2020CS10413
Danish Javed, 2020CS10339
Geetansh Juneja, 2020CS50649

April 5, 2024

Contents

1 Subtask 1

In subtask 1, we discuss the implementation of fundamental operations that will be required in the subsequent subtasks. We have implemented the following functions:

- convolution of a square input matrix and a square kernel, both matrices of any size and the kernel smaller than the input with and without input padding to maintain or reduce the size of the input matrix.
- non-linear activations of an input matrix of any size with relu and tanh functions on individual matrix elements.
- subsampling of square input matrices of any size with max pooling and average pooling functions
- converting a vector of random floats to a vector of probabilities with softmax and sigmoid functions

We have used 32-bit float as the default data type in all our implementations. Note that, our implementations are generic and templatized, so if you want to use 64-bit datatypes, you can use that too. Nevertheless, we sticked with 32-bit floats for all our simulations.

All the functions that we are going to discuss below are defined in `convolutions.hpp` file defined in `Subtask 1` folder.

1.1 Convolution

In this part we implement a 2D convolution operation between an input matrix and a kernel matrix, with an optional padding parameter.

The convolution operation is performed by sliding the kernel matrix over the padded input matrix. For each position in the output matrix, the convolution operation involves multiplying the corresponding elements of the kernel matrix and the overlapped portion of the padded input matrix, and then summing these products. Note that the kernel matrix is applied in a flipped manner before performing the element-wise multiplication with the input matrix. We later change it in kernel implementations to maintain consistency with LeNet Architecture. We then finally return the resultant convolved matrix.

The code snippet for convolution is as follows:

```
1  template <typename T = float>
2  std::vector<std::vector<T>> conv2D(const
       std::vector<std::vector<T>> &input, const
       std::vector<std::vector<T>> &kernel, int pad) {
3    int f = kernel.size();
4    int n = input.size();
5    std::vector<std::vector<T>> ninput(n + 2 * pad, std::vector<T>
       (n + 2 * pad, 0));
6
7    // Build padded matrix
8    for (int i = 0; i < n; i += 1) {
9      for (int j = 0; j < n; j += 1) {
```

```
10          ninput[i + pad][j + pad] = input[i][j];
11        }
12      }
13
14      // do convolution with new matrix here
15      n += 2 * pad;
16      std::vector<std::vector<T>> conv(n - f + 1, std::vector<T> (n -
         f + 1, 0));
17      for (int i = 0; i < n - f + 1; i += 1) {
18        for (int j = 0; j < n - f + 1; j += 1) {
19          for (int k = 0; k < f; k += 1) {
20            for (int l = 0; l < f; l += 1) {
21                    conv[i][j] += kernel[f - k - 1][f - l - 1] *
         ninput[k + i][l + j];
22                }
23          }
24        }
25      }
26
27      return conv;
28  }
```

1.2 Activation functions

In this section, we have defined the non-linear activation functions necessary for Fully Connected (FC) Layers. We've utilized the inline keyword before each activation function to enhance performance. This instructs the compiler to replace function calls with the actual function definition wherever applicable, reducing function call overhead.

1.2.1 relu

The following code implements the Rectified Linear Unit (ReLU) activation function for a single input value. It returns the input value if it's positive, otherwise returns 0.

```
1  template <typename T = float>
2  inline T relu(T input) {
3    return (input > 0 ? input : 0);
4  }
```

1.2.2 tanh

The following code implements the tanh activation function for a single input value. It returns the the hyperbolic tangent of the input value.

```
1  template<typename T = float>
2  inline T tanh(T input) {
3    T z = std::exp(2 * input);
4    return (z - 1.0) / (z + 1.0);
5  }
```

1.3 Subsampling

In this section, we defined the sub-sampling functions. We have implemented the max-pooling and average-pooling sub-sampling functions.

1.3.1 Max Pooling

The following function efficiently computes max pooling by sliding a filter window over the input matrix and selecting the maximum value within each window. The resulting matrix has reduced dimensions based on the filter size.

```
template <typename T = float>
std::vector<std::vector<T>> max_pool(const
    std::vector<std::vector<T>> &input, int filter_size) {

  int n = input.size();
  int f = filter_size;
  std::vector<std::vector<T>> output(n / f, std::vector<T> (n / f,
   std::numeric_limits<T>::lowest()));

  for (int i = 0, i_i = 0; i < n; i += f, i_i += 1) {
    for (int j = 0, j_j = 0; j < n; j += f, j_j += 1) {
        T mx = output[i_i][j_j];
        for (int k = 0; k < f; k += 1) {
            for (int l = 0; l < f; l += 1) {
                mx = std::max(mx, input[i + k][j + l]);
           }
        }
        output[i_i][j_j] = mx;
    }
  }

  return output;
}
```

1.3.2 Average Pooling

The following function efficiently computes average pooling by sliding a filter window over the input matrix and calculating the average value within each window. The resulting matrix has reduced dimensions based on the filter size.

```
template <typename T = float>
std::vector<std::vector<T>> avg_pool(const
    std::vector<std::vector<T>> &input, int filter_size) {

  int n = input.size();
  int f = filter_size;
```

```
 6    std::vector<std::vector<T>> output(n / f, std::vector<T> (n / f,
       0));
 7
 8    for (int i = 0, i_i = 0; i < n; i += f, i_i += 1) {
 9      for (int j = 0, j_j = 0; j < n; j += f, j_j += 1) {
10          T sum{};
11          for (int k = 0; k < f; k += 1) {
12             for (int l = 0; l < f; l += 1) {
13              sum += input[i + k][j + l];
14             }
15          }
16          output[i_i][j_j] = 1.0 * sum / (f * f);
17      }
18    }
19
20    return output;
21  }
```

1.4 Sigmoid & Softmax

The sigmoid function is implemented as given below. It takes a vector input as input and computes the sigmoid activation function for each element of the vector. The sigmoid function is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, where x is the input. The output vector contains the sigmoid-transformed values of the input vector elements.

```
 1  template <typename T = float>
 2  std::vector<T> sigmoid(const std::vector<T> &input) {
 3    int n = input.size();
 4    std::vector<T> output(n);
 5    for (int i = 0; i < n; i += 1) {
 6      output[i] = 1.0 / (1.0 + std::exp(-input[i]));
 7    }
 8
 9    return output;
10  }
```

The softmax function takes a vector input as input and computes the softmax activation function for each element of the vector. The softmax function is defined as $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$ where x_i is the input element at index i and n is the total number of elements in the input vector. The function first computes the exponential of each element in the input vector and accumulates the sum of these exponentials. Then, it divides each exponential by the sum to obtain the softmax-transformed values. The output vector contains the softmax-transformed values of the input vector elements.

```
 1
 2  __constant__ float2D kernel_2D[500];
 3  template <typename T = float>
 4  std::vector<T> softmax(const std::vector<T> &input) {
 5    int n = input.size();
```

```
 6    std::vector<T> output(n);
 7
 8    T den = 0;
 9    for (int i = 0; i < n; i += 1) {
10      output[i] = std::exp(input[i]);
11      den += output[i];
12    }
13
14    for (int i = 0; i < n; i += 1) {
15      output[i] /= den;
16    }
17
18    return output;
19  }
```

2 Subtask 2

In this subtask, we've optimized certain C++ functions into CUDA kernels, leveraging the parallel computing power of GPUs where applicable. Some of these kernels are further refined for integration within the LeNet Architecture in subsequent subtasks.

Specifically, we've implemented CUDA kernels for 2D convolution (with and without shared memory), 3D convolution (with and without shared memory), and max-pooling operations. However, we've chosen not to develop a kernel for average pooling, as it wasn't utilized in the LeNet architecture.

Note

In the subsequent code snippets, we'll encounter float1D, float2D, float3D, and float4D types. These types are introduced for clarity, indicating that the pointers represent 1D, 2D, 3D, or 4D arrays, respectively. However, it's important to note that internally, these arrays are implemented as flattened 1D arrays. This declaration is made at the beginning of our source code as follows:

```
typedef float float1D;
typedef float float2D;
typedef float float3D;
typedef float float4D;
```

This naming convention enhances readability for the reader, clarifying the dimensionality of the arrays despite their internal representation as 1D arrays.

2.1 conv2DKernel

The `conv2DKernel` performs 2D convolution operation on input matrices using GPU parallelization. Each thread in the GPU grid computes one element of the output matrix, applying the convolution operation between the input matrix and the kernel matrix. The kernel utilizes the GPU's parallel processing capability to efficiently compute convolutions

across multiple elements of the output matrix simultaneously. It incorporates boundary checking to ensure that only valid output matrix elements are computed, avoiding out-of-bounds memory access. Additionally, a bias term is added to the convolution result before storing it in the output matrix.

```
__global__ void conv2DKernel(float2D *input, float2D *kernel,
    float2D *conv, int n, int f, float bias) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  if (row < n - f + 1 and col < n - f + 1) {
    float temp = 0;
    for (int k = 0; k < f; k += 1) {
      for (int l = 0; l < f; l += 1) {
        temp += kernel[k * f + l] * input[(k + row) * n + (l +
    col)];
      }
    }
    conv[row * (n - f + 1) + col] = temp + bias;
  }

}
```

2.2 conv2DKernel_shared

This is similar to `conv2DKernel` discussed above but this implementation uses the shared memory (for input matrix) and constant memory (for weights) to enhance the throughput of the kernel.

```
__global__ void conv2DKernel_shared(float2D *input, float2D *conv,
    int n, int f, float bias, int filter_num){
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  __shared__ float N_ds[28][28];
  if(row < n and col<n){
    N_ds[row][col] = input[row * n + col];
  }
  __syncthreads();

  if (row < n - f + 1 and col < n - f + 1) {
    float temp = bias;
    for (int k = 0; k < f; k += 1) {
      for (int l = 0; l < f; l += 1) {
        assert(filter_num * 25 + k * f + l < 500);
        temp += kernel_2D[filter_num * 25 + k * f + l] * N_ds[k +
    row][l + col];
      }
    }
    conv[row * (n - f + 1) + col] = temp;
```

```
20    }
21  }
```

2.3 conv3DKernel

The provided CUDA kernel conv3DKernel performs 3D convolution operation on input 3D matrices using GPU parallelization. Each thread in the GPU grid computes one element of the output matrix, applying the convolution operation between the input matrix and the kernel matrix. The kernel utilizes the GPU's parallel processing capability to efficiently compute convolutions across multiple elements of the output matrix simultaneously.

```
1  __global__ void conv3DKernel(float3D *input, float3D *kernel,
       float2D *conv, int n, int f, int channels, float bias) {
2    int row = blockIdx.y * blockDim.y + threadIdx.y;
3    int col = blockIdx.x * blockDim.x + threadIdx.x;
4
5    if (row < n - f + 1 and col < n - f + 1) {
6      float temp = 0;
7      for (int k = 0; k < f; k += 1) {
8        for (int l = 0; l < f; l += 1) {
9          for (int channel = 0; channel < channels; channel += 1) {
10           temp += kernel[channel * f * f + k * f + l] *
       input[channel * n * n + (k + row) * n + (l + col)];
11         }
12       }
13     }
14     conv[row * (n - f + 1) + col] = temp + bias;
15   }
16 }
```

2.4 conv3DKernel_shared

This is similar to `conv3DKernel` discussed above but this implementation uses the shared memory to enhance the throughput of the corresponding kernel.

```
1  __global__ void conv3DKernel_shared(float3D *input, float3D
       *kernel, float2D *conv, int n, int f, int channels, float
       bias) {
2    int row = blockIdx.y * blockDim.y + threadIdx.y;
3    int col = blockIdx.x * blockDim.x + threadIdx.x;
4
5    __shared__ float N_ds[12][12][20];
6
7    if(row < n and col < n){
8      for (int channel = 0; channel < channels; channel += 1) {
9        N_ds[row][col][channel] = input[channel * n * n +  row * n +
       col];
10     }
11   }
```

```
12    __syncthreads();
13
14    if (row < n - f + 1 and col < n - f + 1) {
15      float temp = 0;
16      for (int k = 0; k < f; k += 1) {
17        for (int l = 0; l < f; l += 1) {
18          for (int channel = 0; channel < channels; channel += 1) {
19            temp += kernel[channel * f * f + k * f + l] * N_ds[k +
       row][l + col][channel];
20          }
21        }
22      }
23      conv[row * (n - f + 1) + col] = temp + bias;
24    }
25  }
```

2.5 Max Pool Kernel

The `max_pool` CUDA kernel implements the max-pooling operation for 2D input matrices using GPU parallelization. Each thread in the GPU grid computes one element of the output matrix, applying the max-pooling operation within a specified filter size. The kernel iterates over each output matrix element and determines the maximum value within the corresponding filter window in the input matrix. It utilizes the GPU's parallel processing capability to efficiently compute max-pooling across multiple elements of the output matrix simultaneously. The maximum value within each filter window is stored in the output matrix, resulting in a downsampled representation of the input matrix.

```
1  __global__ void max_pool_kernel(float2D *input, float2D *output,
       int n, int f){
2    int row = blockIdx.y * blockDim.y + threadIdx.y;
3    int col = blockIdx.x * blockDim.x + threadIdx.x;
4
5    if(row < n / f and col < n / f){
6      float temp = std::numeric_limits<float>::lowest();
7      int st_row = row * f, st_col = col * f;
8      for (int i = 0; i < f; i += 1) {
9          for (int j = 0; j < f; j += 1) {
10             temp = std::max(temp, input[(st_row + i) * n + (st_col
       + j)]);
11         }
12     }
13     output[row * n / f + col] = temp;
14   }
15 }
```

3 Subtask 3

In this subtask we use created gpu kernels to implement Lenet Architecture. We first load all the weights, biases of all the layers and flattened image and copy them to the device memory. Layer 1 weights which are used in first layer is copied to the gpu using cudaMemcpyToSymbol api because it is declared as constant memory and rest are copied to gpu using cudamemcpy. The following code snippet shows the same:

```
1  if (!load_weights(conv1_wts, conv1_bias, 500, 20,
       "weights/conv1.txt")) {
2      std::cout << "Unable to load conv1 weights." << std::endl;
3      return 0;
4    }
5    cudaMemcpyToSymbol(kernel_2D, conv1_wts, 500 *
       sizeof(float),0,cudaMemcpyHostToDevice);
6
7    if (!load_weights(conv2_wts, conv2_bias, 25000, 50,
       "weights/conv2.txt")) {
8      std::cout << "Unable to load conv2 weights." << std::endl;
9      return 0;
10   }
11
12   float4D *d_conv2_wts;
13   cudaMalloc(&d_conv2_wts,25000*sizeof(float));
14 cudaMemcpy(d_conv2_wts,conv2_wts,25000*sizeof(float),cudaMemcpyHostToDevice);
15
16   \\ rest all the weights are copied similarly
```

Now we finally stitch together all the kernels to form the Lenet architecture.

```
1    conv1(d_image, d_conv1_output, conv1_bias);
2    pool1(d_conv1_output, d_pool1_output);
3    conv2(d_pool1_output, d_conv2_output, d_conv2_wts, conv2_bias);
4    pool2(d_conv2_output, d_pool2_output);
5    fclayer1(d_pool2_output, d_fclayer1_output, d_fclayer1_wts,
       d_fclayer1_bias);
6    fclayer2(d_fclayer1_output, d_fclayer2_output, d_fclayer2_wts,
       d_fclayer2_bias);
```

Output of the last layer i.e 2nd fully connected layer is copied back to host. Then we apply softmax on the copied output and store it in the output file.

```
1    cudaMemcpy(fclayer2_output, d_fclayer2_output, 10*sizeof(float),
       cudaMemcpyDeviceToHost);
2    cudaDeviceSynchronize();
3
4    std::vector<float> prob= top5_prob(fclayer2_output,10);
5
6    if(!writeToFile(prob,"output/"+filename)){
```

```
7       std::cerr << "Unable to write to output/"+filename << "\n";
8     }
```

4 Subtask 4

We added cuda streams here so that multiple images can be processed in parallel and also to increase throughput and get proper gpu utilization. We first intialized 32 cuda streams which means 32 images will be processed in parallel. The following snippet shows initialization of these streams:

```
1  // Number of Streams
2  #define NUM_STREAMS 32
3  cudaStream_t stream[NUM_STREAMS];
4  // Initialize cuda streams
5  for (int i = 0; i < NUM_STREAMS; ++i)
6      cudaStreamCreate(&stream[i]);
```

To run cuda streams we require all the operations i.e memory copy from device to host and host to device and gpu kernels to be asynchronous. Memcpy api of cuda is a synchronous memory copy instruction so instead we use memcpyasync api of cuda which requires the memory allocated on the host to be pinned (i.e it cannot be paged out by os) because copying data from host pinned memory to the device doesn't require cpu action and hence can be asynchronously copied by DMA engine. The following code snipped shows the allocation and deallocation of pinned memory:

```
1    // Allocate Pinned Memory
2    float2D *image;
3    cudaMallocHost((void**)&image, 784*NUM_STREAMS*sizeof(float));
4
5    float3D *fclayer2_output;
6    cudaMallocHost((void**)&fclayer2_output,
       10*NUM_STREAMS*sizeof(float));
7
8    // Free pinned memory
9    cudaFreeHost(image);
10   cudaFreeHost(fclayer2_output);
```

We only use declare image memory and the final output memory (i.e final probabilities recieved from gpu) as pinned. Memory for Weights and biases don't need to be allocated as pinned because we load it once to the gpu at the start of the program.

Now we distribute the set of 32 images from the 10000 images to these 32 cuda streams and allow all of them to process these in parallel. An image with index i goes to cuda stream $i\%NUM\_STREAMS$. The following code snippet shows how images are distributed to streams:

```
1    int num_itr = (images.size()+NUM_STREAMS-1)/NUM_STREAMS;
```

```
 2    for(int i=0; i<num_itr; i++){
 3      int st = i*NUM_STREAMS, en =
      std::min((i+1)*NUM_STREAMS,(int)images.size());
 4      // Distributing images to streams
 5      for(int j=st; j<en; j++){
 6          int idx = j%NUM_STREAMS;
 7          if (!load_image(image+idx*784, 784,images[j])) {
 8              std::cout << "Unable to load image " << images[j] <<
      std::endl;
 9              return 0;
10          }
11
      cudaMemcpyAsync(d_image+idx*784,image+idx*784,784*sizeof(float),
      cudaMemcpyHostToDevice,
12          stream[idx]);
13          conv1(d_image+idx*784, d_conv1_output+idx*11520,
      conv1_bias, idx);
14          pool1(d_conv1_output+idx*11520, d_pool1_output+idx*2880,
      idx);
15          conv2(d_pool1_output+idx*2880, d_conv2_output+idx*3200,
      d_conv2_wts, conv2_bias,idx);
16          pool2(d_conv2_output+idx*3200, d_pool2_output+idx*800,
      idx);
17          fclayer1(d_pool2_output+idx*800,
      d_fclayer1_output+idx*500, d_fclayer1_wts, d_fclayer1_bias,
      idx);
18          fclayer2(d_fclayer1_output+idx*500,
      d_fclayer2_output+idx*10, d_fclayer2_wts, d_fclayer2_bias,
      idx);
19          cudaMemcpyAsync(fclayer2_output+idx*10,
      d_fclayer2_output+idx*10, 10*sizeof(float),
      cudaMemcpyDeviceToHost, stream[idx]);
20      }
21    }
```

Time taken to process 10000 images with 1 stream - 58345 ms
Time taken to process 10000 images with 2 stream - 37942 ms
Time taken to process 10000 images with 4 stream - 26666 ms
Time taken to process 10000 images with 8 stream - 20026 ms
Time taken to process 10000 images with 16 stream - 17496 ms
Time taken to process 10000 images with 32 stream - 16066 ms