COL 380: ASSIGNMENT 1

Development of a GPU-Accelerated Image Processing Library for MNIST Digit Recognition

Vatsal Jingar 2020CS50449

Stitiprajna Sahoo 2020CS10394

Piya Bundela 2021CS10118

June 2, 2024

Contents

1 Introduction

The aim of this project was to develop a small image processing library focused on recognizing digits from the MNIST dataset. MNIST, being a widely-used dataset in the field of machine learning, provides a suitable platform for testing and evaluating image processing algorithms.
The assignment was divided into four subtasks, each focusing on different aspects of the implementation, optimization, and application of the image processing library:-

- Implement basic image processing functions in C++
- Rewrite parallelizable functions as CUDA kernels
- Implement the LeNet-5 neural network architecture using the implemented functions
- Optimize throughput using CUDA streams

This report covers the implementation details and robust analysis of each subtask

2 Subtask 1: Implementing C++ Functions:

In this subtask, the goal was to implement several fundamental image processing functions using C++. These functions form the core components of the image processing library and are essential for building more complex algorithms such as convolutional neural networks (CNNs). The implemented functions include:

- Convolution: This operation is fundamental in image processing, especially in tasks like edge detection and feature extraction. The convolution function was implemented to perform convolutions between input matrices and kernels, with and without padding.
- Non-linear activations: ReLU (Rectified Linear Unit) and tanh (Hyperbolic Tangent) activation functions were implemented to introduce non-linearity into the neural network. These activation functions are crucial for enabling the network to learn complex patterns and relationships in the data.
- Subsampling: Max pooling and average pooling functions were implemented for downsampling input matrices. These functions help reduce the spatial dimensions of the input while retaining important features, making the network more computationally efficient and robust to variations in input data.
- Probability Conversion: Softmax and sigmoid functions were implemented to convert raw scores or logits into probabilities. Softmax is commonly used in multi-class classification tasks like digit recognition, while sigmoid is used in binary classification tasks.

The implementation of these functions in C++ provides a solid foundation for building more complex image processing algorithms and neural network architectures. In the next section, we will discuss the details of the implementation and provide code snippets for each function.

2.1 Workflow of the Code:

The workflow of the code is structured to implement each image processing function efficiently and modularly. Here's a detailed breakdown of the workflow:

- **Include Libraries**: The code begins by including necessary libraries such as `<iostream>` and `<vector>` to facilitate input-output operations and manage dynamic arrays, respectively.
- **Define Data Structures**: Data structures such as two-dimensional arrays (matrices) are defined to represent images, kernels, and output matrices. Vectors may also be used for storing probabilities, scores, or intermediate results during computations.
- **Implement Functions**:
  - **Convolution Functions:**
    - **convolveWithPadding:** This function performs convolution with padding to maintain the size of the input matrix.
    - **convolveWithoutPadding:** This function performs convolution without padding, resulting in an output matrix with reduced dimensions.

  Implementation Details:

    - **Convolution with Padding:**
      1. Calculate the padding size based on the kernel size.
      2. Allocate memory for the padded input matrix.

3. Copy the input matrix to the padded input matrix, leaving the padded regions with zeros.
4. Perform convolution by sliding the kernel over the padded input matrix.
5. Accumulate the convolution result in the output matrix.
6. Deallocate memory for the padded input matrix.

* **Convolution without Padding:**
1. Determine the output matrix size based on the input matrix and kernel size.
2. Iterate over the input matrix, applying the convolution operation for each position.
3. Accumulate the convolution result in the output matrix.

– **Activation Functions:**

* **applyReLU:** This function applies the ReLU activation function to each element of the input matrix.
* **applyTanh:** This function applies the tanh activation function to each element of the input matrix.

Implementation Details:

* **ReLU Activation:**
1. Iterate over each element of the input matrix.
2. If the element is negative, set it to zero; otherwise, leave it unchanged.

* **Tanh Activation:**
1. Iterate over each element of the input matrix.
2. Apply the tanh function to each element.

– **Subsampling Function:**

* **subsampling:** This function performs max pooling or average pooling based on the input parameters.

Implementation Details:

* **Max Pooling:**
1. Slide a pooling window over the input matrix.
2. Compute the maximum value within each window.
3. Assign the maximum value to the corresponding position in the output matrix.

* **Average Pooling:**
1. Slide a pooling window over the input matrix.
2. Compute the average value within each window.
3. Assign the average value to the corresponding position in the output matrix.

– **Probability Conversion Functions:**

* **softmax:** This function converts raw scores into probabilities using the softmax function.
* **sigmoid_function:** This function applies the sigmoid function to convert raw scores into probabilities for binary classification tasks.

Implementation Details:

* **Softmax Conversion:**
1. Compute the sum of exponentiated scores.

2. Divide each exponentiated score by the sum to obtain normalized probabilities.

* **Sigmoid Conversion:**
    1. Apply the sigmoid function to each raw score to obtain probabilities between 0 and 1.

- **Main Function**: A main function may be provided to demonstrate the functionality of the implemented functions. Sample input data (e.g., input matrices, kernels) are defined, and the implemented functions are called to perform desired operations.
- **Memory Management**: Memory allocation and deallocation are performed as needed, especially in functions that require dynamic memory allocation for temporary matrices (e.g., convolution with padding).

2.2 Data Layout Choice:

The data layout used in the code primarily consists of matrices represented as two-dimensional arrays or vectors. Here's a detailed description of the data layout:

- **Matrices:** Input matrices, kernels, and output matrices are represented as two-dimensional arrays. Each element of the array corresponds to a pixel value (for images) or a weight value (for kernels). The dimensions of the arrays determine the size of the matrices, with rows representing height and columns representing width.
- **Vectors:** Vectors may be used to store probabilities, scores, or intermediate results during computations. These vectors are one-dimensional arrays that can dynamically resize to accommodate varying numbers of elements.
- **Data Types:** Floating-point numbers (e.g., float) are commonly used to represent pixel values, weights, probabilities, and scores. The choice of data type depends on the precision required for the specific application.
- **Memory Layout:** Matrices and vectors are stored in contiguous memory locations, allowing for efficient access and manipulation. Row-major or column-major ordering may be used depending on the programming language and conventions.
- **Dynamic Memory Allocation:** Dynamic memory allocation is utilized when dealing with matrices or vectors of variable sizes. Memory is allocated at runtime using functions like malloc() or new, and deallocated using free() or delete to prevent memory leaks.

2.3 Bottlenecks of the code:

Several potential bottlenecks may arise in the implemented image processing functions:

1. **Nested Loops:** Many image processing operations, such as convolution and subsampling, involve nested loops to iterate over elements of input matrices or windows. Nested loops can result in high computational complexity, especially for large input matrices or kernels. The time complexity of these operations is typically $O(n^2 \times k^2)$, where n is the size of the input matrix and k is the size of the kernel. As a result, the execution time can increase significantly with larger input sizes or kernel sizes.

2. **Memory Access Patterns:** Image processing operations often require random access to memory locations, which can lead to inefficient memory access patterns. For example, accessing non-contiguous memory locations in nested loops can result in cache misses and increased memory latency. This can significantly impact performance, especially on architectures with limited cache size or memory bandwidth.

3. **Sequential Execution:** In CPU-based implementations, image processing functions are typically executed sequentially, limiting opportunities for parallelism and concurrency. Operations that could be parallelized across multiple cores or threads may not fully utilize available resources, leading to suboptimal performance.

3 Parallelization of Image Processing Functions Using GPUs

To make image processing functions more efficient, especially for large datasets and computationally intensive tasks, leveraging the parallel processing power of GPUs (Graphics Processing Units) can be immensely beneficial. Let's discuss how each of the implemented functions can be parallelized using GPUs:

3.1 Convolution Functions:

Convolution involves applying a filter/kernel to an input image, which can be highly parallelized. Each pixel in the output image can be computed independently, making convolution an ideal candidate for parallelization on GPUs.

- By distributing the computation of convolution across multiple threads or CUDA cores on the GPU, significant speedup can be achieved compared to sequential CPU-based implementations.
- Techniques such as tiling or block processing can be used to efficiently partition the input image and distribute the workload across GPU threads.

3.2 Activation Functions:

Activation functions like ReLU and tanh operate element-wise on matrices, making them inherently parallelizable.

- Each element of the input matrix can be processed independently by different GPU threads, allowing for parallel computation of activation function outputs.
- GPU kernels can be designed to efficiently apply activation functions to large matrices in parallel, leveraging the massive parallelism offered by GPU architectures.

3.3 Subsampling Function:

Subsampling, or pooling, involves aggregating information from neighboring pixels, which can also be parallelized.

- Both max pooling and average pooling operations can be parallelized by processing multiple elements concurrently using GPU threads.
- By partitioning the input matrix into smaller blocks and assigning each block to different GPU threads, pooling operations can be performed in parallel across the entire input matrix.

3.4 Probability Conversion Functions:

Softmax and sigmoid functions, like activation functions, operate element-wise on vectors, making them suitable for parallelization on GPUs.

- Each element of the input vector can be processed independently by different GPU threads, allowing for parallel computation of softmax or sigmoid outputs.
- GPU kernels can be designed to efficiently apply softmax or sigmoid functions to large vectors in parallel, exploiting GPU parallelism to accelerate computation.

Using GPUs for parallel processing can significantly accelerate image processing tasks, enabling faster computation and analysis of large datasets.

4 Subtask 2: Rewriting Functions as CUDA Kernels

In this subtask, we aimed to leverage the parallel processing power of GPUs by rewriting selected image processing functions as CUDA kernels. CUDA (Compute Unified Device Architecture) allows us to harness the massive parallelism of GPUs to accelerate computationally intensive tasks such as convolution, pooling, and fully connected layers. By optimizing memory access patterns and exploiting parallelism, we aimed to achieve significant speedup compared to CPU-based implementations.

4.1 CUDA Functions Detail

Input data (input image arrays, convolutional kernels, bias if applicable) is first transferred from the CPU to the GPU memory. They are stored as **1-Dimensional float arrays in C**

4.1.1 1. `conv_kernel_p1`:

This function does the first step of convolution.

- The CUDA kernel `conv_kernel_p1` is launched with appropriate thread blocks to process the convolution operation in parallel. Each block is responsible to compute the 2D convolution of one channel of input image (which is a 2D layer) by one kernel channel (2D layer). The dimension of blocks = *(output_channels, input_channels)*. The assignment of convolution operations to blocks shown in figure 1

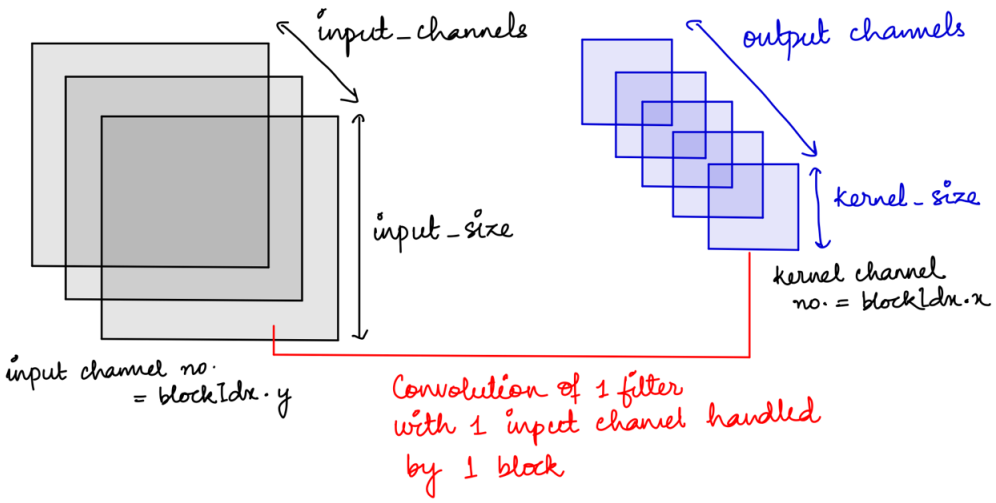

Figure 1: The operations of convolution assigned to each block

- Each thread within the CUDA threadblock computes a specific element of the channel of the output, thus *a block produces one channel of output.*. Domension of threads in the blocks = *(output_size, output_size)*, where output_size = input_size - kernel_size + 1. *Note that, the size we are talking about here is the size of 2 dimensional input and kernels.* The assignment of convolution operations among threads shown in figure 2

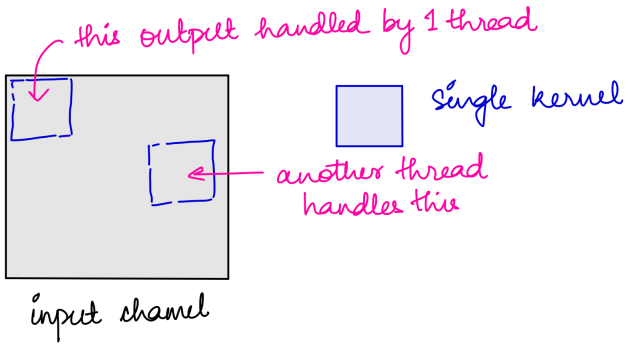

Figure 2: The thread-wise operations of the 2-D convolutions

- The resulting 2-D outputs are stored in the global memory of the GPU. We have maintained a flag which is used as:
  - *flag = 0*: Number of input channels > 1, which means the 2-D convolution outputs produced by different blocks have to be added to form the final convolution output channels. So the convolution outputs of different blocks are stored separately, without adding bias here.
  - *flag = 1*: Number of input channels $= 1$, which means 2-D convolution output channels are ready to be processed for the next layer of computation. Thus, bias is added to the outputs and stored in GPU memory.

4.1.2 2. `conv_kernel_p2`:

This function processes the outputs of `conv_kernel_p1` to form the output ready for next layer of computation.

- Each block handles the output accummulation for one channel of output kernels. Dimension of blocks = *(Number of output kernel channels)*.
- Each thread in the block adds any specific row, column of all the outputs that are a result of convolution of the kernel with different input channels. Dimension of threads = *(output_size, output_size)*. Finally bias is also added to the specific row, column of output layers. The final convolution outputs are stored in the Global GPU memory as 1-Dimensional arrays for further layers.

4.1.3 3. `maxpool_kernel`:

This CUDA kernel implements the max pooling operation, aiming to aggregate maximum values within pooling windows efficiently.

- Threads are organized into blocks, with each block processing the maxpool outputs for a single input channel. Dimension of blocks = *(Number of input channels)*
- Within each block, each thread computes the max value of one maxpool filter, dimension of threads = *(output_size, output_size)*, where output_size = input_size - filter_size + 1
- The resulting output array is stored in the global memory.

4.1.4 4. `fc_kernel`:

This CUDA kernel performs the fully connected operation, aiming to compute the weighted sum of inputs efficiently on the GPU.

- Here we have only 1 block to do this kernel computation.
- Each thread within the block produces one output neuron, so dimension of threads = *(Size of output layer)*
- The resulting output tensor representing the output layer of the fully connected operation is stored in the global memory for inference

Overall, these rewritten CUDA kernels aim to efficiently parallelize convolution, pooling, and fully connected operations on the GPU, leveraging the massive parallelism of GPU architectures to achieve significant performance improvements compared to CPU-based implementations. Careful consideration is given to memory access patterns, parallelization strategies, and optimization techniques to maximize the efficiency and scalability of the CUDA kernels.

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

In Subtask 3, our objective was to assemble the previously developed C++ and CUDA functions into a coherent implementation of the LeNet-5 convolutional neural network (CNN). LeNet-5, a classic architecture renowned for its effectiveness in handwritten digit recognition, serves as the cornerstone of our implementation. This architecture, consisting of two convolutional layers, two pooling layers, two fully connected (FC) layers, a ReLU activation layer, and a softmax layer, provides a robust framework for accurately classifying handwritten digits.

By integrating our custom CUDA kernels for convolution and pooling operations, we aim to construct a powerful CNN capable of accurately mapping input images to their corresponding digit labels. This subtask represents a pivotal moment where our framework's disparate components converge, enabling us to harness the power of deep learning and GPU acceleration to achieve state-of-the-art performance in digit recognition.

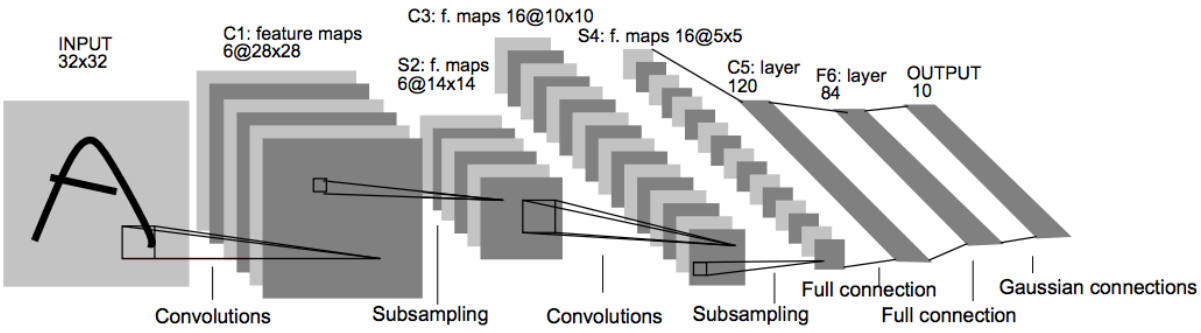

Figure 3: LeNet-5 Architecture

5.1 Optimisations done to improve Latency:

1. **One-time memory initialisations**
   - We have initialised all the device as well as host memory at the starting of the main function, and reused the arrays for every image processing. This reduces latency as well as memory utilisation.
2. **Memory Layout as Arrays:**
   - All data is aligned as 1-Dimensional arrays (pointer to arrays) and required memory is allocated to the arrays before the start of processing the files. We choose array over vectors as arrays being contiguous blocks of memory tend to exhibit better cache locality compared to vectors, which are dynamically allocated data structures.
3. **Indexing of Input elements and Kernels:**
   - **Inputs for `conv_kernel_p1`:** The elements of a single input channel (2D matrix) are aligned contiguously, and the elements of single kernel channel is also aligned contiguously. This ensures that cache misses are reduced, as the convolution operation does read operation from these channels.

- **Inputs for `conv_kernel_p2`:** The outputs of the `conv_kernel_p1` are stored in the way where the output element of convolution correspoding to a specific row and column of different input channels are stored contiguously. This is because in `conv_kernel_p2`, different threads accumulate different elements of the specific output channel by adding the output elements of corresponding row and column of different input channels. So to ensure minimal cache misses, the data is aligned as shown in Figure 4

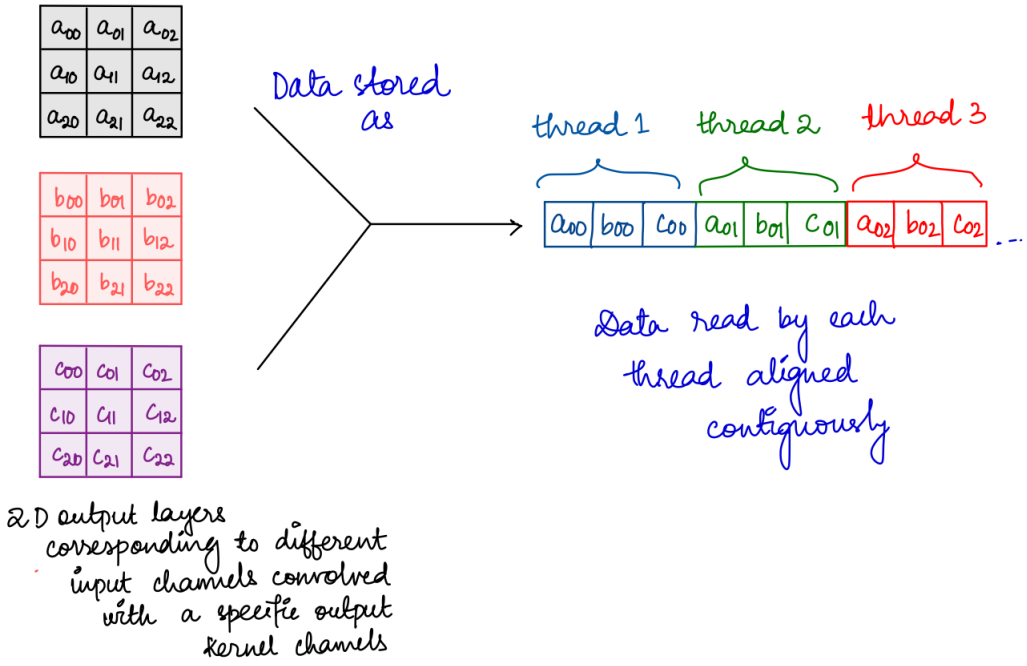

Figure 4: Alignment of the elements of different 2-D output layers corresponding to different input channels convolved with a specific kernel channel as array to ensure minimal cache misses

- **Inputs and Outputs for maxpool:** The input and output layers of maxpool operation are stored in a similar fashion as to the inputs of `conv_kernel_p1`, where the elements of a single input channel (2D matrix) are aligned contiguously, and the elements of single maxpool filter is also aligned contiguously. This ensures that cache misses are reduced, as the maxpool operation does read operations from these channels.
- **Inputs and Weights of `fc_layer`:** The inputs and weights corresponding to a specific output neuron are stored contiguously in an array for minimum cache misses.

4. **Shared Memory:**

- Shared Memory to store the inputs in the Fully-Connected 2 layer, as the inputs were used by multiple threads of the block. This reduced memory access times and overall latency by few milliseconds.
- We try to store kernels in the shared memory per-block in convolution operations, as the same kernel filter is used by all the threads in the block, so loading it into shared memory would reduce latency. But in the implementation of shared memory, we had to introduce `__syncthreads()`,

which caused an overhead, thus adjusting the latency and making it almost similar to the implementation without shared memory.

6 Subtask 4: Optimize throughput with CUDA streams

In subtask 4, our objective was to make the processing of multiple images concurrent by the use of streams. We made the following optimisations using streams in the code.

1. We create 100 streams, for concurrently processing 100 images. In each stream, 100 images were processed sequentially.

2. We allocate memory (of input images, kernel-weights, and outputs of all layers) for the 100 streams initially. These memory are assigned to the image in the stream, and on completion of one image in the stream, the memory are reassigned to a different image in the stream. The memory assignment is done using `CudaMemCpyAsync`, and these memory reuse helps in time as well as memory optimisation.

```
for(int j = 0; j < sub_batch_size; j++){
    for(int i = 0; i < num_sub_batches; i++){
        cudaMemcpyAsync(d_inp[i], inp[j * num_sub_batches + i], 28 * 28 * sizeof(float),
        cudaMemcpyHostToDevice, streams[i]);
    }
    for(int i = 0; i < num_sub_batches; i++){
        conv_kernel_p1<<<blocks1, threads1, 0, streams[i]>>>(d_inp[i], d_out1_p1[i],
        insize1, d_conv1_kernel, ksize1, inchannels1, kchannels1, d_conv1_bias, flag1);
    }
    for(int i = 0; i < num_sub_batches; i++){
        maxpool_kernel<<<blocks2, threads2, 0, streams[i]>>>(d_out1_p1[i], d_out2[i],
        insize2, ksize2, stride2, inchannels2);
    }
    for(int i = 0; i < num_sub_batches; i++){
        conv_kernel_p1<<<blocks3, threads3, 0, streams[i]>>>(d_out2[i],
        d_out3_p1[i],
        insize3, d_conv2_kernel, ksize3, inchannels3, kchannels3, d_conv2_bias, 0);
    }
    for(int i = 0; i < num_sub_batches; i++){
        conv_kernel_p2<<<50, threads3_2, 0, streams[i]>>>(d_out3_p1[i], d_out3_p2[i],
        kchannels3, inchannels3, insize3 - ksize3 + 1, d_conv2_bias);
    }
    for (int i = 0; i < num_sub_batches; i++)
        maxpool_kernel<<<blocks4, threads4, 0 ,streams[i]>>>(d_out3_p2[i], d_out4[i],
        insize4, ksize4, stride4, inchannels4);
    for (int i = 0; i < num_sub_batches; i++)
        conv_kernel_p1<<<blocks5, threads5, 0, streams[i]>>>(d_out4[i], d_out5_p1[i],
        insize5, d_conv3_kernel, ksize5, inchannels5, kchannels5, d_conv3_bias, 0);
    for (int i = 0; i < num_sub_batches; i++)
        conv_kernel_p2<<<500, threads5_2, 0, streams[i]>>>(d_out5_p1[i], d_out5_p2[i],
```

```
        kchannels5, inchannels5, insize5 - ksize5 + 1, d_conv3_bias);
    // FC2
    for (int i = 0; i < num_sub_batches; i++)
        fc_kernel<<<1, threads6, 0, streams[i]>>>(d_out5_p2[i], d_out6[i], d_fc2_weight,
        d_fc2_bias, insize6, outsize6);
    cudaDeviceSynchronize();
}
```

3. We call each kernel and memcpy function in a different for loop, this will add the kernel functions and the memcpy functions to their queues, so that there would be no blocking of the kernel, and we achieve maximum concurrency in kernel scheduling for different images. The following code snippet shows a part of our execution, where each kernel are assigned to different streams in a for loop.

```
for(int i = 0; i < num_sub_batches; i++){
    cudaMemcpyAsync(d_inp[i], inp[j * num_sub_batches + i], 28 * 28 * sizeof(float),
    cudaMemcpyHostToDevice, streams[i]);
}
for(int i = 0; i < num_sub_batches; i++){
    conv_kernel_p1<<<blocks1, threads1, 0, streams[i]>>>(d_inp[i], d_out1_p1[i],
    insize1, d_conv1_kernel, ksize1, inchannels1, kchannels1, d_conv1_bias, flag1);
}
for(int i = 0; i < num_sub_batches; i++){
    maxpool_kernel<<<blocks2, threads2, 0, streams[i]>>>(d_out1_p1[i], d_out2[i],
    insize2, ksize2, stride2, inchannels2);
}
```

7 Observations:

Number of Images	Time taken w/o streams (in ms)	Time taken with streams (in ms)
1	1	1
1000	977	783
10000	9770	7825

Table 1: Time taken to process the image files together (in milliseconds) with and without streams in CUDA on HPC of IITD

Here we have considered the time taken for the processing of data and producing the output labels by inferencing through the LeNet Architecture.