Assignment 2 COL380
Image processing library for MNIST digits

Team - bratva
Pratham, Mihir, Kushagra

April 9, 2024

§1 Subtask 1: Implementing the C++ functions

§1.1 Overview

We implemented the following C++ functions - `convolve`, `activate`, `max_pooling`, `avg_pooling` and `output_probability` for matrix operations using 32-bit floating-point numbers as datatype. We also defined `vector<vector<float>>` as the data type for all the matrices.

§1.2 `convolve`

- **Function Parameters**:
  - `input` - 2D vector representing the input matrix.
  - `kernel` - 2D vector representing the convolution kernel.
  - `padding` - Integer value indicating the amount of padding to be applied to the input matrix.
- **Function Description**:

  The `convolve` function performs convolution between the input matrix and the kernel, with the option of applying padding to the input matrix.

§1.3 `activate`

- **Function Parameters**:
  - `input` - 2D vector representing the input matrix.
  - `type` - A string indicating the type of activation function to apply. It can be either "relu", "sigmoid" or "tanh".
- **Function Description**:

  The `activate` function takes an input matrix and applies the specified activation function element-wise to it. The activation function type is determined by the provided string parameter.

§1.4 `max_pooling`

- **Function Parameters**:
  - `input` - 2D vector representing the input matrix.
  - `stride` - An integer specifying the stride for pooling.

- **Function Description**:

  The `max_pooling` function performs max pooling on the input matrix with a specified stride, reducing the spatial dimensions of the matrix.

§1.5 `avg_pooling`

- **Function Parameters**:
  - `input` - 2D vector representing the input matrix.
  - `stride` - An integer specifying the stride for pooling.
- **Function Description**:

  The `avg_pooling` function performs average pooling on the input matrix with a specified stride, reducing the spatial dimensions of the matrix.

§1.6 `output_probability`

- **Function Parameters**:
  - `input_vector` - 1D vector of floats representing the input data.
  - `type` - A string indicating the type of probability transformation to apply. It can be either ”softmax” or ”sigmoid”.
- **Function Description**:

  The `output_probability` function transforms the input vector of floats into a vector of probabilities. The type of transformation is determined by the string `type`.

§2 Subtask 2: Rewriting the C++ parallelizable functions as CUDA kernels

We represent all matrices (of any dimension) as a flattened array. For each CUDA function, we obtain the set of indices of a matrix using the block and thread parameters of a thread as well as the other input parameters to the function.

§2.1 `convolve_parallel`

- **Function Parameters**:
  - `input_cuda` - Pointer to the start of flattened 2D input square matrix allocated on CUDA.
  - `kernel_cuda` - Pointer to the start of flattened 2D kernel square matrix allocated on CUDA.
  - `output_cuda` - Pointer to the start of flattened 2D output square matrix allocated on CUDA.
  - `n` - Size of input matrix.
  - `k` - Size of kernel matrix.
- **Function Description**:

  The `convolve_parallel` function performs convolution of a 2D input matrix and a 2D kernel using parallel processing on a GPU.

§2.2 `activate_parallel`

- **Function Parameters**:
  - `input_cuda` - Pointer to the start of flattened 2D input matrix allocated on CUDA.
  - `output_cuda` - Pointer to the start of flattened 2D output matrix allocated on CUDA.
  - `n` - Number of rows in input matrix.
  - `m` - Number of columns in input matrix.
  - `type` - A value that determines the type of activation.
- **Function Description**:

  The `activate_parallel` function applies ReLU or tanh activation function to a 2D input matrix using parallel processing on a GPU.

§2.3 `maxpool_parallel`

- **Function Parameters**:
  - `input_cuda` - Pointer to the start of flattened 2D input square matrix allocated on CUDA.
  - `output_cuda` - Pointer to the start of flattened 2D output square matrix allocated on CUDA.
  - `n` - Size of input matrix.
  - `pool_size` - Size of the square region from which we pool a value.
- **Function Description**:

  The `maxpool_parallel` function performs max pooling on a 2D input matrix using parallel processing on a GPU.

§2.4 `avgpool_parallel`

- **Function Parameters**:
  - `input_cuda` - Pointer to the start of flattened 2D input square matrix allocated on CUDA.
  - `output_cuda` - Pointer to the start of flattened 2D output square matrix allocated on CUDA.
  - `n` - Size of input matrix.
  - `pool_size` - Size of the square region from which we pool a value.
- **Function Description**:

  The `avgpool_parallel` function performs average pooling on a 2D input matrix using parallel processing on a GPU.

§3 Subtask3: Implementing the LENET-5 neural network stitching together the implemented C++ and CUDA functions

§3.1 Overview

In the `main` function, we firstly call the functions `conv1_init()`, `conv2_init()`, `fc1_init()`, and `fc2_init()` which load the weights for the `conv` and `fc` layers from the files in the `/weights/` folder, and firstly allocates them on the CPU using `new` and the transferred to the GPU memory that is accessed using globally defined pointers, using `cudaMalloc` and `cudaMemcpy`.

Note that we use global arrays such as `conv1_values`, `conv1_filter`, `conv1_bias`, `conv1_filter_cuda`, `conv1_bias_cuda` (for `conv1`, and so on) to reutilize the device (gpu) and cpu memory over all images.

After this, the `neural_net_structure` function is called for each of the 10000 images, which takes the output of the **previous layer as its input** and the process is commonly known as **feedforward propagation** in neural networks. It uses the memory allocated in the `init` functions, for each of the images, and its main logic is summarised in the below code:

```
float* conv1_output = conv1(input);
float* pool1_output = pool1(conv1_output);
float* conv2_output = conv2(pool1_output);
float* pool2_output = pool2(conv2_output);
float* fc1_output = fc1(pool2_output);
float* final_output_probabilities = fc2(fc1_output);
```

Finally `conv1_finish()`, `conv2_finish()`, `fc1_finish()` and `fc2_finish()` functions are called to `free` the CPU and `cudaFree` the GPU memory.

§3.2 Convolution Layer 1

§3.2.1 Initialization: `conv1_init()`

Initializes the filter weights and biases for Convolution Layer 1 using pre-trained weights from the file `conv1.txt`.

§3.2.2 Operation: `conv1()`

Performs convolution on the input image using the initialized filters and biases. It calls the CUDA kernel function `convolve_cuda` with the following parameters, and updates the result in the global `output_cuda` shared array:

```
dim3 num_blocks(1,1,20);
dim3 threads_per_block(24,24,1);
convolve_cuda<<<num_blocks,threads_per_block>>> (input_cuda, conv1_filter_cuda,
↪  output_cuda, image_size, k, num_filters, conv1_bias_cuda);
```

§3.3 Pooling Layer 1

§3.3.1 Operation: `pool1()`

Performs max pooling on the input feature maps. It calls the CUDA kernel function `maxPool_cuda` with the following parameters, and updates the result in the `output_pool1_cuda` shared array:

```
dim3 num_blocks(1,1,20);
dim3 threads_per_block(12,12,1);
maxPool_cuda<<<num_blocks,threads_per_block>>> (input_pool1_cuda, output_pool1_cuda,
↪  input_size, pool_size, num_filters);
```

§3.4 Convolution Layer 2

§3.4.1 Initialization: `conv2_init()`

Initializes the filter weights and biases for Convolution Layer 2 using pre-trained weights from the file `conv2.txt`.

§3.4.2 Operation: `conv2()`

Performs convolution on the input feature maps using the initialized filters and biases. It calls the CUDA kernel function `convolve_cuda_3d_channel` with the following parameters, and updates the result in the `output_cuda` array:

```
dim3 num_blocks(1,1,1000);
dim3 threads_per_block(8,8,1);
convolve_cuda_3d_channel<<<num_blocks,threads_per_block>>> (input_cuda,
↪  conv2_filter_cuda, output_cuda, input_size, kernel_size, num_filters_input,
↪  num_filters_output);
```

§3.5 Pooling Layer 2

§3.5.1 Operation: `pool2()`

Performs max pooling on the input feature maps. It calls the CUDA kernel function `maxPool_cuda` with the following parameters, and updates the result in the `output_pool2_cuda` array:

```
dim3 number_blocks(1,1,50);
dim3 threads_per_block(4,4,1);
maxPool_cuda<<<number_blocks, threads_per_block>>> (input_cuda, output_cuda,
↪  input_size, pool_size, num_filters_output);
```

§3.6 Fully Connected Layer 1

§3.6.1 Initialization: `fc1_init()`

Initializes the filter weights and biases for Fully Connected Layer 1 using pre-trained weights from the file `fc1.txt`.

§3.6.2 Operation: `fc1()`

Performs convolution on the input feature maps using the initialized filters and biases. It calls the CUDA kernel function `convolve_cuda_3d_channel` with the following parameters, and updates the result in the `output_cuda` array:

```
dim3 num_blocks(1,1,25000);
dim3 threads_per_block(1,1,1);
convolve_cuda_3d_channel<<<num_blocks,threads_per_block>>> (input_cuda,
↪  fc1_filter_cuda, output_cuda, input_size, kernel_size, num_filters_input,
↪  num_filters_output);
```

§3.7 Fully Connected Layer 2

§3.7.1 Initialization: `fc2_init()`

Initializes the filter weights and biases for Fully Connected Layer 2 using pre-trained weights from the file `fc2.txt`.

§3.7.2 Operation: `fc2()`

Performs convolution on the input feature maps using the initialized filters and biases. It calls the CUDA kernel function `convolve_cuda_3d_channel` with the following parameters, and updates the result in the global `output_cuda` shared array:

```
dim3 num_blocks(1,1, 5000);
dim3 threads_per_block(1,1,1);
convolve_cuda_3d_channel<<<num_blocks,threads_per_block>>> (input_cuda,
↪  fc2_filter_cuda, output_cuda, input_size, kernel_size, num_filters_input,
↪  num_filters_output);
```

§3.8 Helper functions - `convolve_cuda` and `convolve_cuda_3d_channel`

The `convolve_cuda` function performs convolution operation on the input feature maps using the provided kernel filters. It flattens the 2D matrix into a 1D array and transforms the (x, y) coordinates to n * x + y. The function is invoked using pointers for CUDA arrays and takes the following parameters:

- `input_cuda`: Pointer to the input feature maps stored in device memory.
- `kernel_cuda`: Pointer to the kernel filters stored in device memory.
- `output_cuda`: Pointer to the output feature maps stored in device memory.
- `input_size`: Size of the input feature maps.
- `kernel_size`: Size of the kernel filters.
- `depth`: Depth of the input feature maps.
- `bias`: Pointer to the bias terms applied after convolution in device memory.

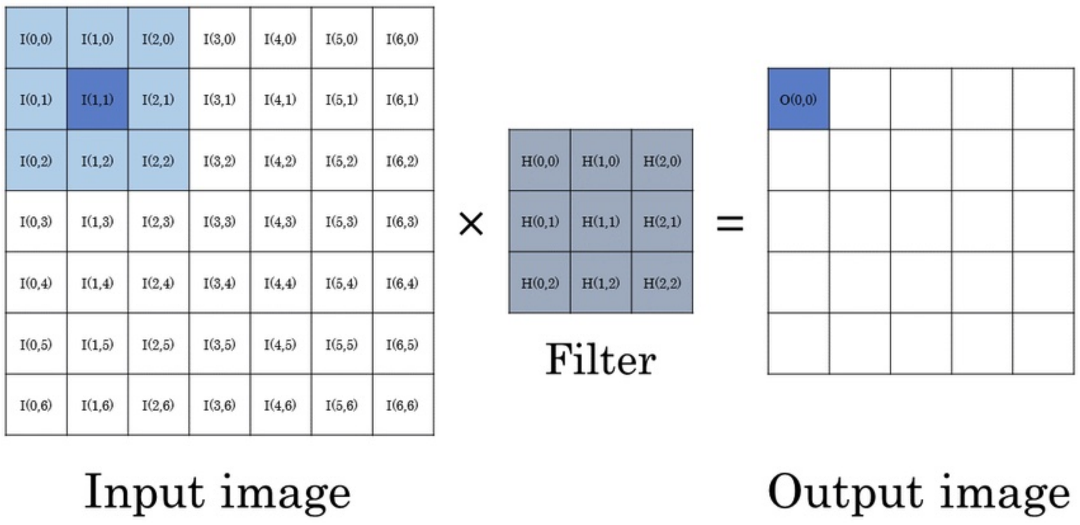

I(0,0) I(1,0) I(2,0) I(3,0) I(4,0) I(5,0) I(6,0)
I(0,1) I(1,1) I(2,1) I(3,1) I(4,1) I(5,1) I(6,1)
I(0,2) I(1,2) I(2,2) I(3,2) I(4,2) I(5,2) I(6,2)
I(0,3) I(1,3) I(2,3) I(3,3) I(4,3) I(5,3) I(6,3)
I(0,4) I(1,4) I(2,4) I(3,4) I(4,4) I(5,4) I(6,4)
I(0,5) I(1,5) I(2,5) I(3,5) I(4,5) I(5,5) I(6,5)
I(0,6) I(1,6) I(2,6) I(3,6) I(4,6) I(5,6) I(6,6)
×
H(0,0) H(1,0) H(2,0)
H(0,1) H(1,1) H(2,1)
H(0,2) H(1,2) H(2,2)
Filter
=
O(0,0)
Input image
Output image

Figure 1: 2D Input, 2D kernel (Only 1 input channel shown out of 20 for conv1), 2D Output

The `convolve_cuda_3d_channel` function extends the convolution operation to 3D feature maps and multiple channels. It takes into account the depth and number of channels when computing the output feature maps. Similar to `convolve_cuda`, it utilizes pointers for CUDA arrays and accepts the following parameters:

- `input_cuda`: Pointer to the input feature maps stored in device memory.
- `kernel_cuda`: Pointer to the kernel filters stored in device memory.
- `output_cuda`: Pointer to the output feature maps stored in device memory.

- `input_size`: Size of the input feature maps.
- `kernel_size`: Size of the kernel filters.
- `depth`: Depth of the input feature maps.
- `channels`: Number of channels in the output feature maps.

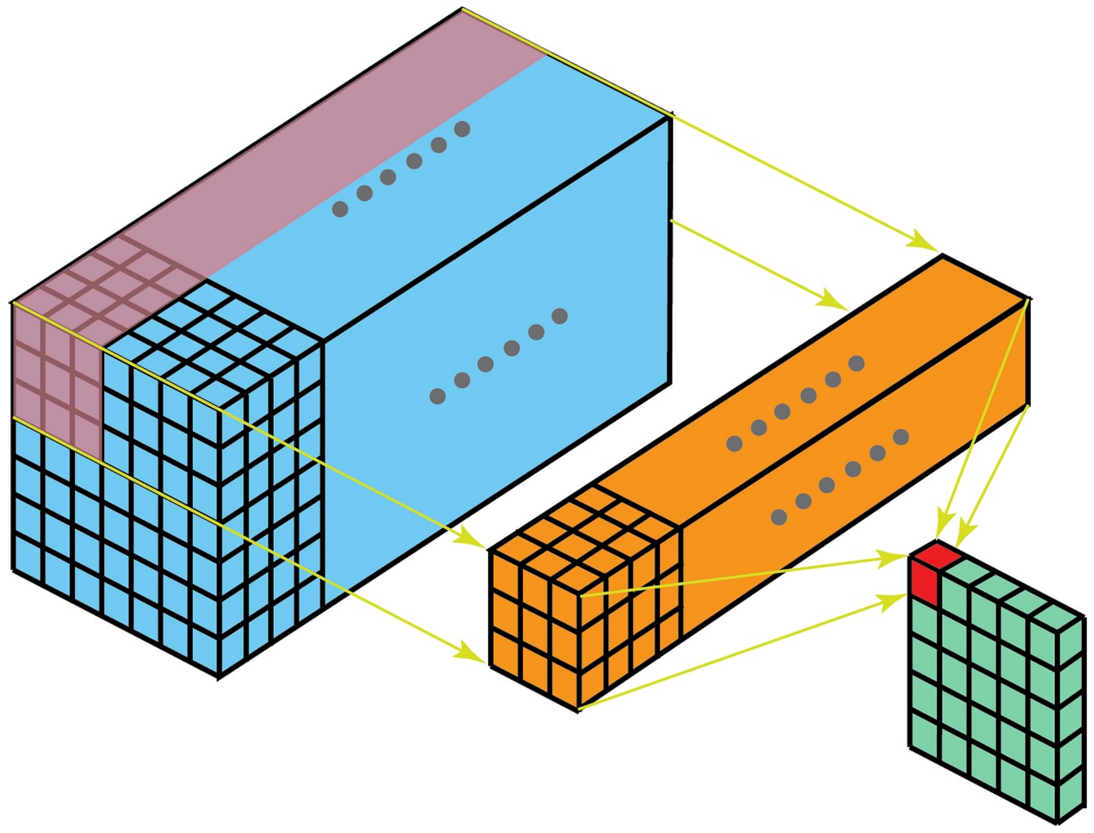


Figure 2: 3D Input, 3D Kernel, 2D Output (Only one output channel is shown out of 50 for conv2)

Note that: Atomic addition (`atomicAdd()`) is used in the convolution kernel to safely update output values in parallel, avoiding race conditions, which ensures correctness in concurrent updates to the output feature maps.

§3.8.1 Max and Avg Pooling: `maxPool_cuda` and `avgPool_parallel`

The `maxPool_cuda` and `avgPool_parallel` function performs max and average pooling on the input feature maps using a specified pool size. It takes advantage of CUDA parallelization for efficient computation. The function is invoked with pointers to CUDA arrays and the following parameters:

- `input_cuda`: Pointer to the input feature maps stored in device memory.
- `output_cuda`: Pointer to the output feature maps stored in device memory.
- `input_size`: Size of the input feature maps.
- `pool_size`: Size of the pooling window.
- `num_filters`: Number of filters (channels) in the input feature maps. (for maxPool)

The functions traverses the input feature maps, calculates the maximum/average value within each pooling window, and updates the corresponding element in the output feature maps with the maximum/average value.

§3.9 Performance of Subtask 3

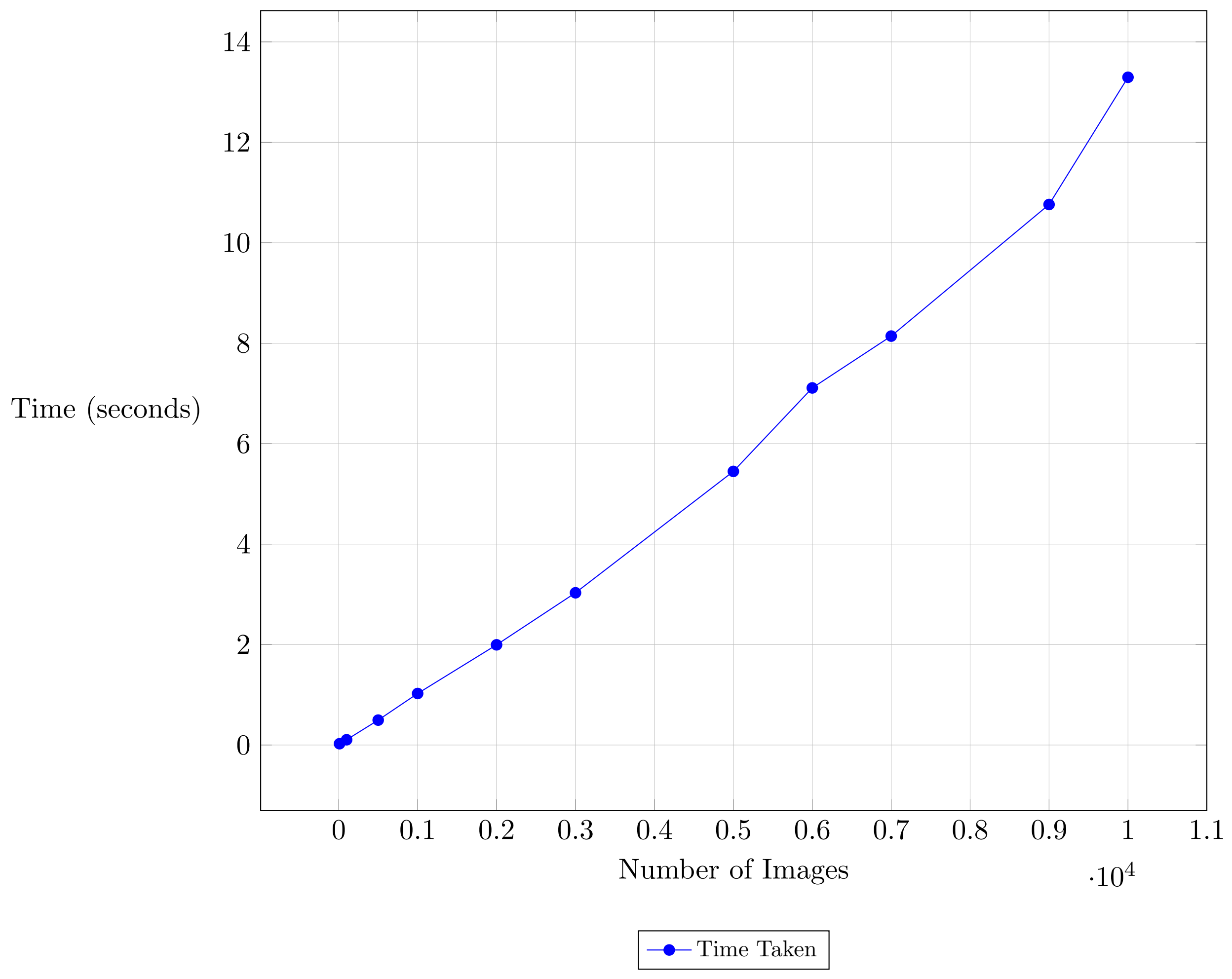


Figure 3: Subtask 3, Image classification time vs. Number of Images

§4 Subtask 4: Optimizing throughput with CUDA streams

CUDA streams enable concurrent execution of multiple kernels and memory transfers on NVIDIA GPUs, enhancing performance through efficient resource utilization.

We have the functions similar to subtask 3 such as `conv1_init()`, `conv1_finish()` (for conv1) and the helper functions such as `convolve_cuda` and `convolve_cuda_3d_channel`.
In the `int main` function, the program initializes convolutional and fully connected layers using the functions `conv1_init`, `conv2_init`, `fc1_init`, and `fc2_init`. After initialization, the program allocates memory for input and output data on the GPU using CUDA memory allocation functions (`cudaMalloc`).

Additionally, it creates a large array of CUDA streams (`cudaStream_t`) for concurrent execution of CUDA kernels and memory transfers, calling `cudaStreamCreateWithFlags(&streams[i],`

`cudaStreamNonBlocking);`

The following arrays are allocated on the GPU using `cudaMalloc()`: `input_cuda_conv1`, `input_cuda_pool1`, `input_cuda_conv2`, `input_cuda_pool2`, `input_cuda_fc1`, `input_cuda_fc2`, `input_cuda_fc2_after`, `output_cuda`, and `output_cuda_after`.

Then inference is performed on a neural network using CUDA. It includes the allocation of memory for various arrays on the GPU and then executes a series of CUDA kernels for each image in the input dataset. The number of images is represented by `number_images`.

For each image, we iterate through the **inference pipeline**, executing CUDA kernels for **convolution, max pooling, bias addition, activation function application, and output probability calculation**. The use of `streams[i % numstreams]` ensures that each image's computation is scheduled on a separate CUDA stream, allowing for concurrent execution of multiple images.

After processing all images, `cudaDeviceSynchronize()` is called to ensure that all CUDA operations are completed before proceeding, after which we call `cudaStreamDestroy(streams[i])`, write to the output file and call `cudaFree` and `conv1_finish`. etc.
e one stream executes convolutional or fully connected layers, another stream can initiate memory transfers, effectively hiding communication latency and maximizing GPU utilization.

§4.1 Stream Optimization Strategies

- Utilizing a large number of CUDA streams created using `cudaStreamCreateWithFlags()` function, enabling non-blocking behavior (`numstreams`) for parallelizing inference tasks across functions such as `convolve_cuda`, `maxPool_cuda`, and `activate_parallel`.
- Asynchronously launching CUDA kernels and memory transfers to overlap computation and communication within functions like `convolve_cuda` and `maxPool_cuda`.
- Concurrently executing CUDA kernels in different streams for functions such as `convolve_cuda_3d_channel` and `output_probability` to maximize GPU utilization.
- Employing dynamic stream management techniques to adaptively assign tasks based on workload and GPU resources, particularly in functions like `convolve_cuda` and `output_probability`.
- Organizing memory transfers and kernel launches within CUDA streams to optimize memory bandwidth utilization, as seen in functions such as `cudaMemcpy` and `cudaMalloc`.
- Kernel launches (`<<< >>>`) specify the associated CUDA stream, ensuring kernels are executed asynchronously within their respective streams, which enables pipelining of computation and enhances parallelism.
- After launching kernels in all streams, the code synchronizes each stream using `cudaStreamSynchronize()` to ensure completion of all tasks before proceeding and the streams are destroyed using `cudaStreamDestroy()` to release resources after their use.

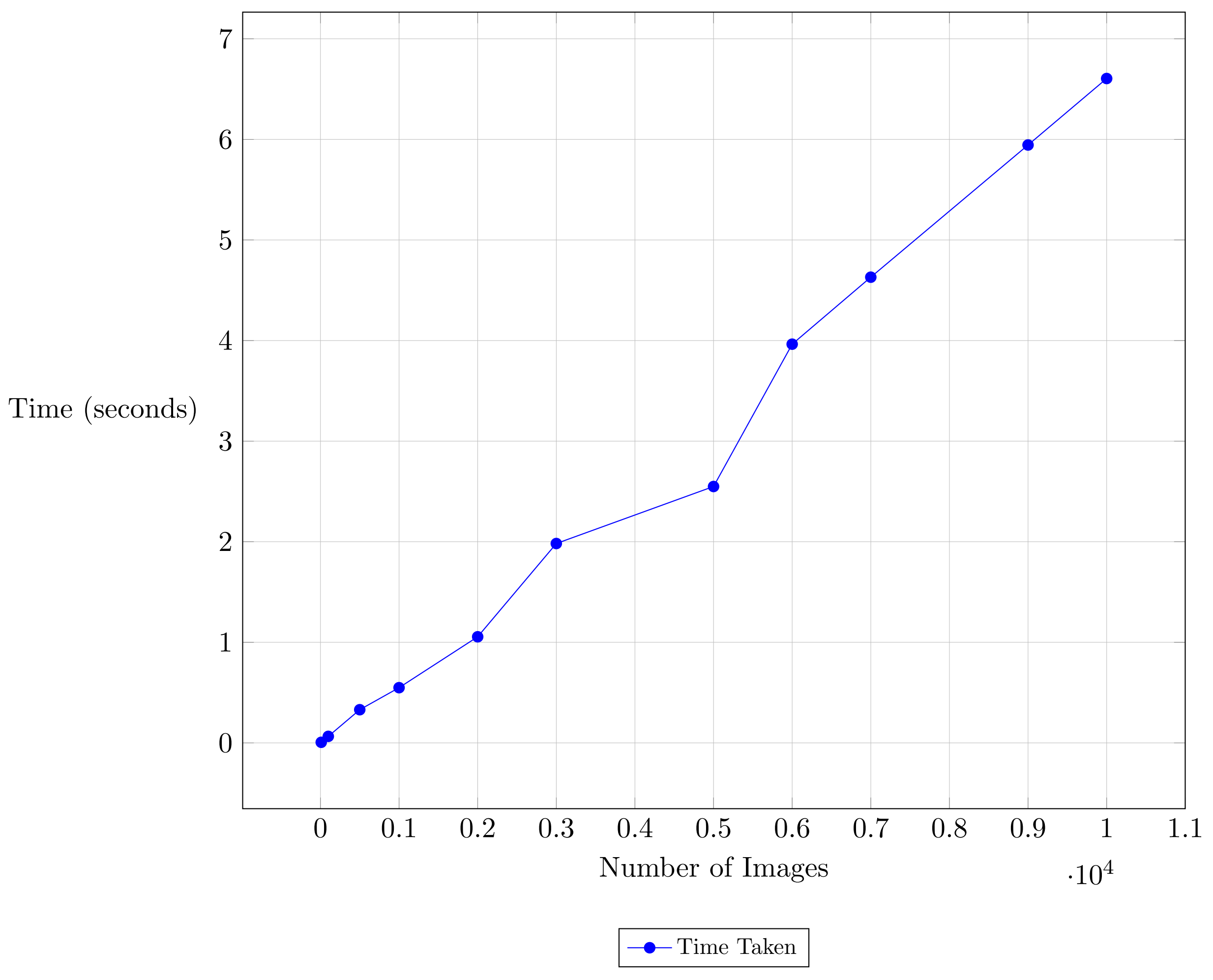
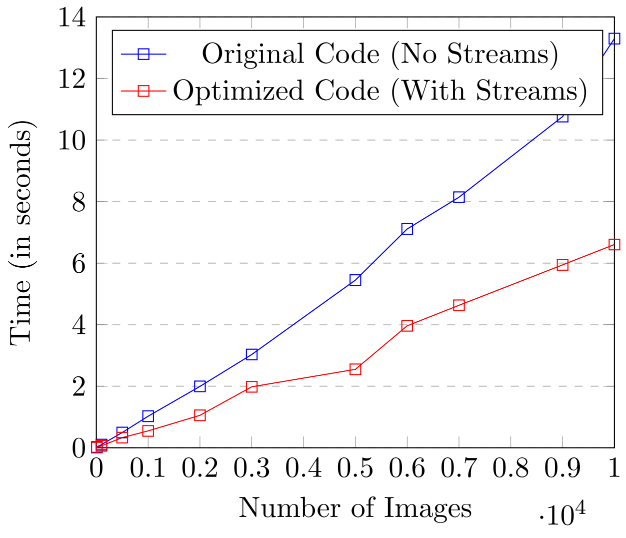


Figure 4: Subtask 4 Performance vs. Number of Images (for 5 streams)

§4.2 Performance of Subtask 4



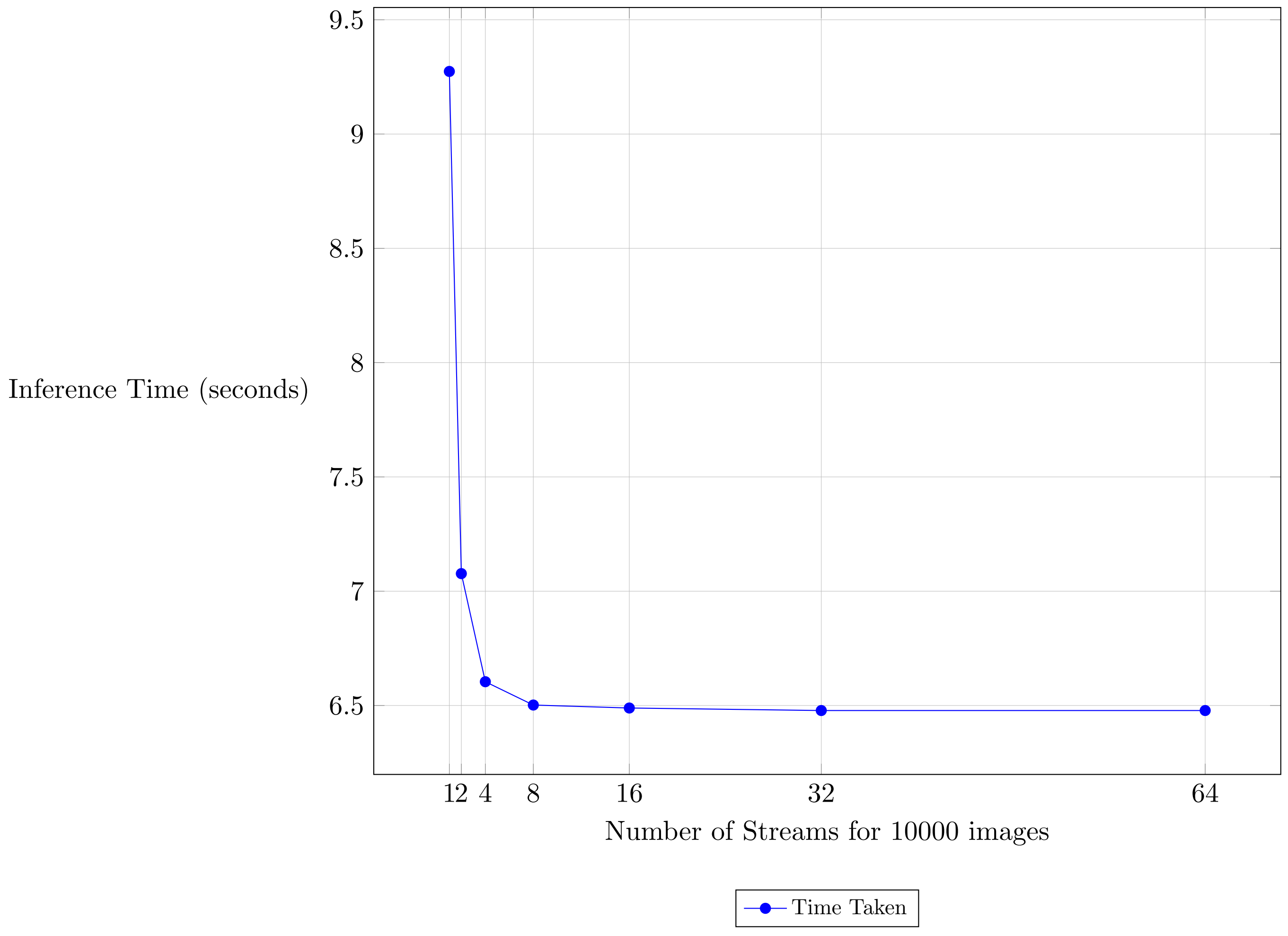


Figure 5: Performance vs. Number of Streams for 10000 images

§4.3 Performance Improvements

- **Enhanced Throughput:** Simultaneous execution of kernels and memory operations across multiple streams maximizes GPU utilization, resulting in increased throughput.
- **Decreased Latency:** Asynchronous execution hides memory access latency by overlapping computation with data transfers, leading to expedited overall execution.
- **Enhanced Scalability:** Leveraging CUDA streams enables the code to scale effectively with larger datasets or more intricate models while maintaining performance.
- **Optimized Resource Utilization:** Efficient memory management and concurrent execution mitigate idle time, optimizing the utilization of available GPU resources.

§4.4 Conclusion

The integration of CUDA streams in our code shows efficient optimization methods for enhancing the speed of digit prediction tasks on NVIDIA GPUs. Through the utilization of parallelism and the synchronization of computation with communication, our code achieves notable enhancements in performance and scalability. These optimizations are pivotal for fully leveraging the computational capabilities of contemporary GPUs and are indispensable for accelerating a wide range of parallel computing tasks, including deep learning