COL380 Assignment 2 Report

Aastha Rajani (2021CS10093), Disha Shiraskar (2021CS10578),
Priyadarshini Radhakrishnan (2021CS50614), Srushti Junghare(2021CS50613)

Below is our report providing insights into each optimization applied in the CUDA kernels, covering memory management, kernel design, memory access patterns, and error handling. These optimizations collectively enhance the performance and efficiency of the CUDA applications, leveraging the parallel processing capabilities of NVIDIA GPUs to their fullest extent.

1 Subtask2

1 Convolution Kernel (normalconvolution.cu):

a. Memory Allocation and Data Transfer:

Dynamic Memory Allocation : Instead of allocating memory for each row individually, the code dynamically allocates memory for arrays of pointers to rows (tempinput, tempkernel, tempoutput) on the host side. This approach reduces the number of memory allocation calls, enhancing memory management efficiency. Memory for the arrays of pointers is allocated using cudaMalloc on the GPU, ensuring efficient memory utilization.

Minimized Data Transfer: Rather than transferring entire matrices between the host and device, the code only transfers pointers to rows of matrices (dinput, dkernel, doutput). This optimization reduces data transfer overhead. Each pointer to a row is copied from the host to the device using cudaMemcpy, resulting in more efficient data transfer compared to transferring entire matrices.

Optimized Memory Access Patterns: The use of pointers to arrays of rows facilitates coalesced memory access within CUDA kernels (convolutionWithoutPaddingKernel). Coalesced memory access ensures that threads access adjacent memory locations, maximizing memory bandwidth utilization and enhancing GPU performance.

b. Kernel Optimization:
Enhanced Parallelism: The kernel function convolutionWithoutPaddingKernel partitions the input matrix into thread blocks to maximize parallelism and GPU resource utilization. CUDA threads perform convolution operations concurrently on subsets of the input matrix, leveraging the GPU's parallel processing capabilities effectively.

Reduced Unnecessary Computation: The convolution operation is implemented without padding, eliminating unnecessary computations and improving computational efficiency. By skipping iterations that would involve accessing out-of-bounds elements, the kernel focuses only on valid convolution operations, leading to faster execution.

2 Max Pooling and Average Pooling Kernels (pooling.cu):

a. Memory Allocation and Data Transfer:
Dynamic Memory Allocation: Memory for arrays of pointers to rows (tempinput, tempoutput) is dynamically allocated on the host using cudaMalloc. This approach reduces the number of memory allocation calls, enhancing memory management efficiency. Pointers to rows are then copied from the host to the device using cudaMemcpy, facilitating efficient data transfer between the host and device.

Minimized Data Transfer: Instead of transferring entire matrices, the code only transfers pointers to rows of matrices (dinput, doutput). This optimization reduces data transfer overhead between the host and device. Each pointer to a row is copied individually from the host to the device, optimizing data transfer and improving overall performance.

Optimized Memory Access Patterns: The use of pointers to arrays of rows enables coalesced memory access within CUDA kernels (maxPoolinghelper, averagePoolinghelper). Coalesced memory access ensures that threads access adjacent memory locations, maximizing memory bandwidth utilization and enhancing GPU performance.

b. Kernel Optimization:
Enhanced Parallelism: Both the maxPoolinghelper and averagePoolinghelper kernels partition the input matrix into thread blocks to maximize parallelism and GPU resource utilization. CUDA threads concurrently perform pooling operations on subsets of the input matrix, leveraging the GPU's parallel processing capabilities effectively.

Reduced Unnecessary Computation: The pooling operations are implemented efficiently, with nested loops iterating only over the elements within the pooling window. This approach eliminates unnecessary computations and focuses only on the elements relevant to the pooling operation, improving computational efficiency.

3 ReLU and Tanh Activation Kernels (relucuda.cu):

a. Memory Allocation and Data Transfer:
Dynamic Memory Allocation: Memory for arrays of pointers to rows (tempinput, tempoutput) is dynamically allocated on the host using cudaMalloc. This approach reduces the number of memory allocation calls, enhancing memory management efficiency. Pointers to rows are then copied from the host to the device using cudaMemcpy, facilitating efficient data transfer between the host and device.

Minimized Data Transfer: Instead of transferring entire matrices, the code only transfers pointers to rows of matrices (dinput, doutput). This optimization reduces data transfer overhead between the host and device. Each pointer to a row is copied individually from the host to the device, optimizing data transfer and improving overall performance.

b. Kernel Optimization:
Enhanced Parallelism: Both applyReLUkernel and applyTanhkernel kernels partition the input matrix into thread blocks to maximize parallelism and GPU resource utilization. CUDA threads concurrently perform activation function operations on subsets of the input matrix, leveraging the GPU's parallel processing capabilities effectively.

Utilization of Device Functions: Device functions relu and tanhactivation are used within the kernels to compute ReLU and tanh activation functions, respectively. Utilizing device functions helps in code organization and reuse, enhancing maintainability and modularity.

4 Softmax and Sigmoid Kernels (softmaxcuda.cu):

a. Memory Allocation and Data Transfer:
Dynamic Memory Allocation: Memory for arrays of pointers to rows (tempinput, tempexponents, tempoutput) is dynamically allocated

on the host using cudaMalloc. This approach reduces the number of memory allocation calls, enhancing memory management efficiency. Pointers to rows are then copied from the host to the device using cudaMemcpy, facilitating efficient data transfer between the host and device.

Minimized Data Transfer: Instead of transferring entire matrices, the code only transfers pointers to rows of matrices (dinput, dexponents, doutput). This optimization reduces data transfer overhead between the host and device. Each pointer to a row is copied individually from the host to the device, optimizing data transfer and improving overall performance.

b. Kernel Optimization:

Enhanced Parallelism: Kernels like computeExponentials, computeSum, and softmax partition the input matrix into thread blocks to maximize parallelism and GPU resource utilization. CUDA threads concurrently perform computation operations on subsets of the input matrix, leveraging the GPU's parallel processing capabilities effectively.

Utilization of Shared Memory: The computeSum kernel attempts to utilize shared memory (sdata) to optimize memory access patterns and improve computational efficiency. However, this part is currently commented out in the code.

2 Subtask 3

In this section, we were asked to write code to implement LeNet architecture, using the functions defined in subtask2, that takes as input a 28x28 image, reads the pre-trained weights from the attached files and uses them to output the top 5 softmax probabilities. We started with a most basic implementation of subtask3 and then optimised it to reduce the time required for processing. Here, we describe all the 3 implementations first one being the most basic and last one being the most optimized.

2.1 Implementation 1

Note that all the functions defined in subtask2 involved processing a 2D input i.e. functions in subtask2 were like 2D-Convolution, 2D-Pooling etc. In this implementation, we used these 2D functions as it is without any changes. Below is a description of the structure of the code :

- Header Inclusions: The code includes necessary header files for I/O operations, vector handling, timing, and CUDA kernels for various operations like adding biases, matrix addition, convolution, pooling, ReLU activation, and softmax calculation.
- LoadWeights Function: This function loads the weights and biases for each layer from external files. It reads the weights and biases from the files and stores them in appropriate data structures.
- LeNet5 Function: This function represents the LeNet-5 model. It takes the input image matrix, applies convolutional layers, pooling layers, and fully connected layers, and finally calculates softmax probabilities. These functions were called such that first a CPU call to these functions was made which then made a GPU kernel call.
- Main Function: In the main function, an input image matrix is loaded from a file. The LeNet5 function is called to process the input image and calculate output probabilities. Finally, the elapsed time is printed.

Reasons that delayed processing in this implementation :

- Since we had used 2D functions for convolution, pooling etc we had to iterate everytime across the number of input channels and output channels using a for loop.
- Since 2D functions defined where CPU functions which then called a GPU kernel within them , all these calls in the for loop were blocking calls and hence took place sequentially, leading to limited parallelism.
- Moreover, since we were continuously switching between CPU calls and GPU kernel calls, a lot of time was used up in doing memory copying from GPU to CPU and vice versa.

2.2 Implementation 2

In order to reduce time in image processing, we tried to work upon the above mentioned reasons in this implementation. The overall structure of our code remains the same, however there are slight modifications in the Lenet5 function. They are as follows:

- We modified our subtask2 functions to directly work upon 3D input and 4D kernels, this saved time used in iterating in the sequential for loops in the previous implementation.
- As the kernels in subtask2 functions now directly worked upon multidimensional arguments, it allowed us to exploit the parallelism provided by the GPU as much as possible.
- We completely erased out calls made to the CPU functions of convolution, pooling, relu, and softmax. Instead, we directly made calls to the GPU kernels of the corresponding function in Lenet5 function.
- Since now there were no CPU calls in the Lenet5 function, only 2 times was switching within CPU and GPU was required – first switch involved copying input, kernels and biases to GPU and second switch involved copying the output of softmax from GPU to CPU.
- As Lenet5 only involved kernel calls, now output of one layer could directly be fed as input to the next layer without copying to CPU in between. This avoided unnecessary calls for memory copying.

2.3 Final Implementation

The changes made in this implementation are not necessary, but were exclusively done to ensure that subtask3 continues to work efficiently even on multiple images as input. They are listed below :

- If previous implementation was used to run across multiple images, it invovlved iterating across the number of images and in each iteration making call to Lenet5 CPU function. This did not exploit GPU parallelism to its fullest. So, we completely removed the CPU Lenet5 function and instead made all kernels calls within the main function itself.
- Moreover if previous implementation was used to run across multiple images, it involved loading the weights and kernels for every image again and again everytime Lenet5 function was called for that image, even though the values of weights and kernels was same for all images. Now, in the main function, we first loaded all the weights, biases and kernels - once for all images and then ran a for loop to call the corresponding kernels for each image.

3 Subtask 4

3.1 Implementation details and logic

In subtask 4 we have utilized CUDA streams to optimize the processing of multiple images concurrently, thereby improving performance . CUDA streams enable concurrent execution of multiple kernels or memory transfers on the GPU. By dividing the workload into smaller tasks and executing them concurrently within different streams, the overall processing time is significantly reduced. We have used streams to process batches of images in parallel, thereby maximizing GPU utilization and throughput.

- We have first initialized *num_streams* CUDA streams for concurrent processing of batches, and allocated host memory for weights, biases, input images, and output probabilities, with weights and biases loaded from external files and input images loaded from separate text files.
- The main loop then iterates over batches of input images. We have kept batch size to be equal to *num_streams*. In each batch iteration, the main loop organizes tasks across streams as follows:
- First each stream is assigned one image for convolution. The convolution kernels for all images in the batch are pushed into the streams' execution queues with one image per stream.
- After convolution, the same set of streams executes pooling operations on the convolved images. Pooling kernels for each image are pushed into the respective streams' execution queues.
- Following, the convolution and pooling kernels are again pushed into respective stream's execution queues.
- Following second pooling, the streams process fully connected and relu layers for each image in the batch. Similarly, the corresponding kernels are enqueued for execution in the streams.
- Finally, softmax calculation is performed using the outputs of the fully connected layers. This operation is also scheduled within the same set of streams.

This organization ensures that each stream handles the complete pipeline for one image at a time, facilitating efficient processing and resource utilization across the GPU. Once the processing of all batches is complete, the output probabilities are copied back to host memory using asynchronous memory copy operations. The computed probabilities are then printed as required.

3.2 Kernel optimisations

- **Custom CUDA Kernels:** Defined custom CUDA kernels tailored for convolution, max pooling, ReLU activation, fully connected layers and softmax operations.
- **Efficient Memory Access:** Within the convolutionKernel, memory access patterns are optimized to ensure coalesced memory accesses. For example, when accessing input image data, adjacent threads access consecutive memory locations to minimize memory latency.
- **Thread Utilization:** The grid dimensions and block dimensions are configured to fully utilize the GPU's computational resources. For instance, if the input image size is width x height, the grid and block dimensions are set appropriately to distribute the workload across multiple CUDA cores effectively.
- **Compute Intensive Operations:**Inside the convolutionKernel, unnecessary memory accesses and arithmetic operations are minimized to focus on compute-intensive tasks such as convolutional filtering and element-wise multiplication, thus reducing execution time.
- **Shared Memory Usage:** Shared memory is utilized within the convolutionKernel to cache frequently accessed input data and intermediate results. For example, the kernel may load a tile of the input image into shared memory to reduce the need for costly global memory accesses during convolution operations.

3.3 Stream optimisations

- **Concurrent Execution:** CUDA streams are used to execute concurrent operations on the GPU. For instance, while one stream is executing convolution operations on a image, another stream can simultaneously perform convolution operations on other images of same batch, effectively utilizing the GPU's parallel processing capabilities.
- **Parallel Processing:** The workload is divided into smaller tasks such as convolution and pooling and one complete sequence of operation for different images is given to different streams. For instance, each stream may handle convolution, pooling, and other operations for a specific batch of images concurrently.
- **Memory Transfer Overlap:** Memory copy operations, such as loading input images into device memory and transferring intermediate results back to host memory, are overlapped with kernel execution using asynchronous memory copy functions. This overlap hides memory transfer latency and maximizes GPU utilization.

- **Batch Processing:** Batch processing in Round Robin Manner is implemented to process multiple images concurrently within each stream. For example, each stream may process a batch of images in parallel, exploiting data parallelism and maximizing GPU occupancy.
- **Synchronization Optimization:** Synchronization points between streams are minimized to allow independent tasks to execute asynchronously. We have divided the tasks between streams in such a way that their execution are independent and hence, no synchronization calls are required.
- **Resource Utilization:** Stream utilization is optimized by dynamically balancing workload distribution across available GPU resources. For instance, the number of streams and the size of each batch are adjusted to ensure efficient resource utilization and avoid underutilization or contention for computational resources.

4 Analysis of Subtask3

Number of Images	Execution Time (sec)
1	0.28
100	0.41
1000	1.5
2000	2.74
4000	5.11
6000	8.09
8000	10.62
10000	12.98

Table 1: Observation

5 Analysis of subtask 4

Number of streams	Number of Images	Execution Time (secs)
2	1	0.210
	100	0.270
	1000	1.470
	2000	2.510
	4000	4.720
	6000	6.790
	8000	8.750
	10000	13.430
4	1	0.210
	100	0.290
	1000	1.220
	2000	2.140
	4000	4.060
	6000	6.010
	8000	7.920
	10000	9.910
8	1	0.210
	100	0.290
	1000	1.160
	2000	2.030
	4000	3.900
	6000	5.740
	8000	7.780
	10000	9.690
16	1	0.210
	100	0.290
	1000	1.110
	2000	2.010
	4000	3.830
	6000	5.610
	8000	7.400
	10000	8.830
25	1	0.210
	100	0.280
	1000	1.130
	2000	1.980
	4000	3.810
	6000	5.600
	8000	7.310
	10000	8.680

Table 2: Observation

6 Graphical analysis of subtask 3 and subtask 4

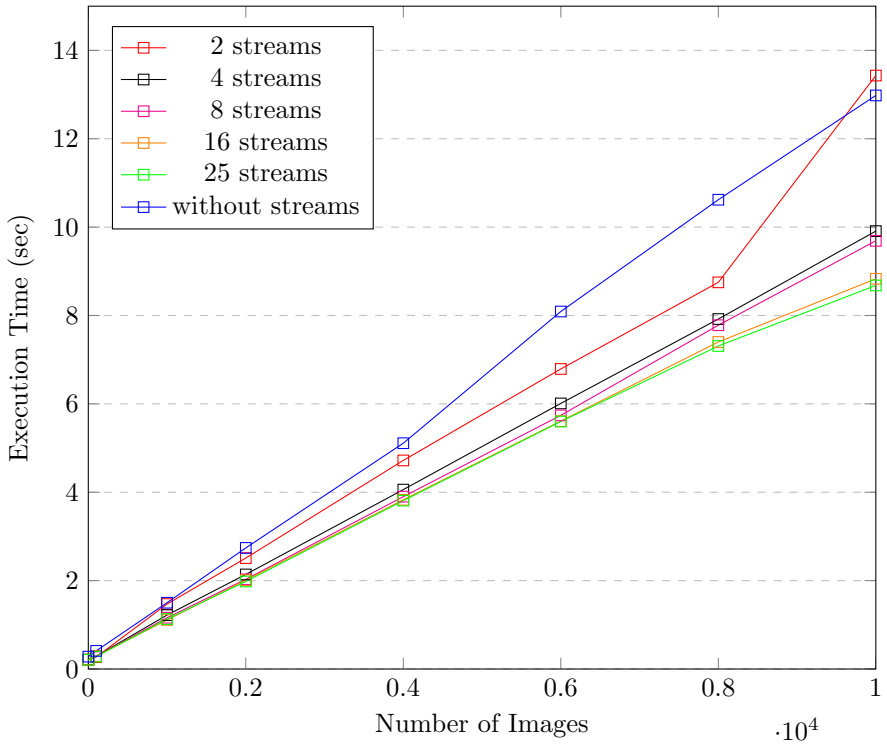

Figure 1: Execution Time vs Number of Images

The graph depicts the execution time of the provided CUDA implementation for varying numbers of images processed concurrently, with different numbers of streams utilized. Here's a brief analysis:

- Without Streams: The execution time increases linearly with the number of images due to sequential processing, resulting in the longest execution times among all configurations.
- 2 Streams: Introducing two streams leads to a noticeable reduction in execution time compared to the sequential approach. The performance improvement is moderate.
- 4 Streams: With four streams, the execution time decreases further, indicating better parallelization and resource utilization on the GPU.
- 8 Streams: Utilizing eight streams results in a significant reduction in execution time, suggesting improved concurrency and GPU utilization.

- 16 Streams: Further increasing the number of streams to sixteen continues to decrease execution time, albeit with diminishing returns compared to lower stream counts.
- 25 Streams: The performance improvement becomes marginal as the number of streams reaches twenty-five, suggesting diminishing returns due to potential overheads associated with managing a larger number of streams.

Overall, the graph illustrates the benefits of utilizing CUDA streams for parallelizing CNN computations, with a notable decrease in execution time as the number of streams increases up to a certain point. Beyond a certain threshold, the performance gains diminish, highlighting the importance of optimizing stream usage based on the specific characteristics of the GPU and workload.