COL 380 - Assignment 2

Manshi Sagar 2020CS50429
Rajat Mahaur 2020CS50534
Richa Yadav 2020CS50438

March 2024

1 Subtask 1

1.1 Explanation

For Subtask-1 we implemented following functions:

- **convultion** - it takes teo 2D square arrays, an input array and a kernel and outputs an array of size $(inputSize - kernelSize + 1)$. This function slides the kernel over input array and computes the weighted sum and stores the output.
- **convutionWithPadding** - it also performs the convolution of input matrix and kernel but in earlier function after performing convolution the size of output matrix was compressed but in this function we adds extra rows and columns containing zeroes so that output matrix's size is same as the given input matrix. Both sides we add extra $kernelSize/2$ columns and similarly extra rows are added.
- **relu** - Relu of the matrix is calculated by taking max of 0 and value stored at every index, such that every negative entry of input is replaced by zero.
- **tanh** - tanh of the input matrix is calculated using the inbuilt tanh function in cmath library.
- **maxPooling** - This function reduces the spatial dimesions of feature map. It takes an input matrix(NXN) and pooling size (P) such that N is a multiple of P and divides the matrix into non-overlapping regions of size PXP. For each region it calculates maximum value and stores in the output matrix of size (N/P X N/P).
- **avgPooling** - Similar to maxPooling function but in this we take average of all values in the region instead of calculating max.
- **softmax** - this function takes a vector of real numbers in input and converts it into a probability distribution.
- **sigmoid** - this functions take a float as input and maps it to a value between 0 and 1.

2 Subtask 2

2.1 Explanation

For converting subtask 1 into subtask 2, we made 2 functions for every above function. One function is a host function and other is device function also known as kernel function marked with __global__.

- In host function we are first allocating space for input, kernel and output matrix using **cudaMalloc** function, then using **cudaMemcpy**, it transfers input and kernel matrix data from host to device. Set up the blocksize and gridsize. It calls kernel function and stores the output in d_output and then transfers the data of d_output from device to host using cudaMemcpy into output matrix and in the end it frees up the space using **cudaFree** function.

- In the kernel function, each thread accesses the input data using its specific block and thread identifiers provided by blockIdx and threadIdx. Each thread then performs computations on the portion of data assigned to it within the overall input, independently of other threads.

Every thread in a kernel is uniquely identified by blockIdx, blockDim, and threadIdx. So each thread is assigned some rows or columns of the input data according to these indices.

If blocksize is taken as 16x16, it means that one block contains 16x16 threads and grid dimension specifies no. of blocks in a grid. Grid size is calculated based on input size and block size to insure that blocks are evenly distributed throughout the grid and there are sufficient threads to process the entire amount of input data.

Threads are organized into blocks, which are then scheduled to run on the GPU's processing units. Blocks are managed by a scheduler, which distributes them to the various SMs, on the GPU. Inside each SM, the threads within the blocks are executed concurrently, allowing multiple blocks to be processed simultaneously.

3 Subtask 3

3.1 Implementation

- **Optimizing memory management :** Firstly, we preloaded and stored all weights and biases for every level, ensuring that we wouldn't need to reread them for each image. Subsequently, we transferred this data to the GPU memory upfront to avoid the time-consuming disk operations associated with data transfer. Additionally, we allocated memory for the output of each layer in advance, thus minimizing the need for memory allocation during computation, and ensuring that this allocation process occurred only once.

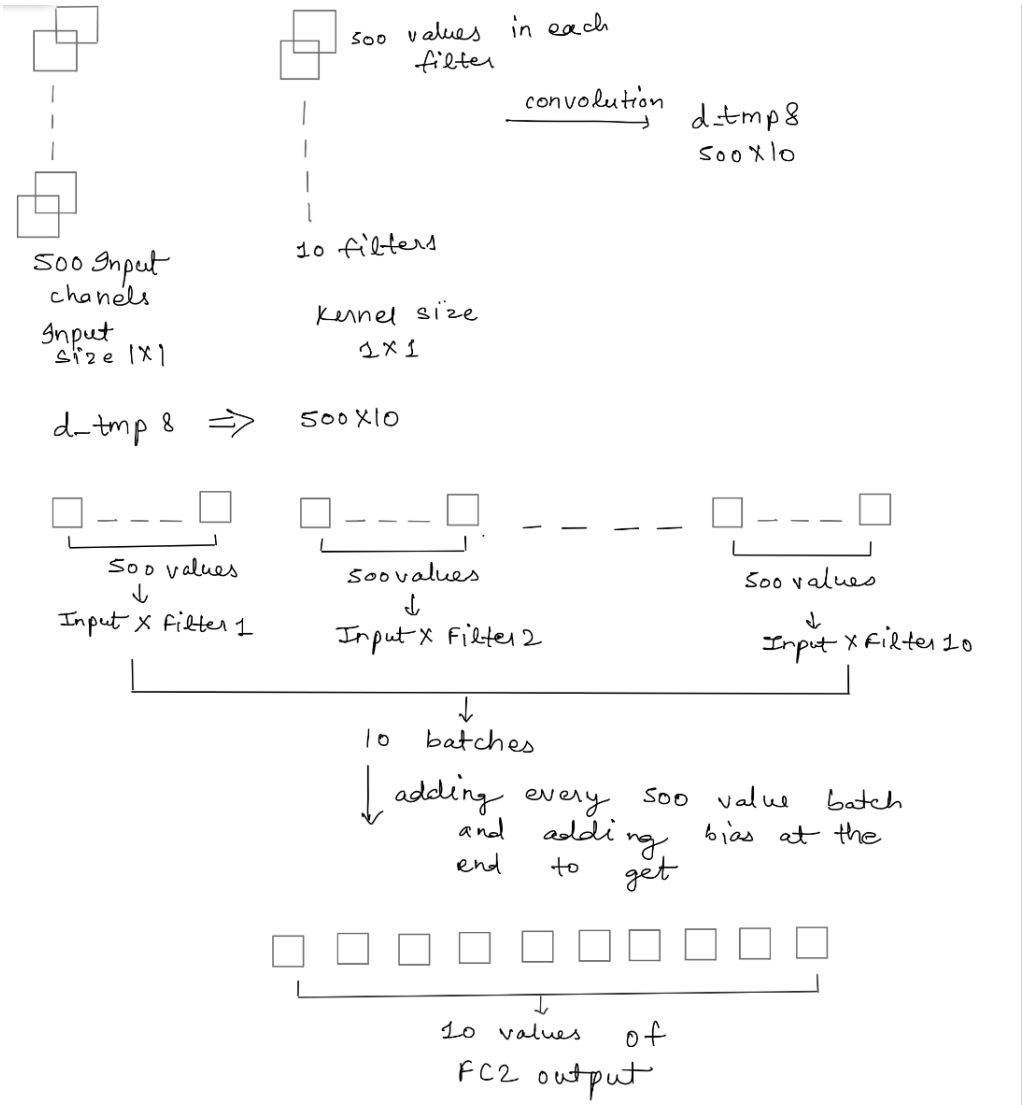
- **Optimizing Memory Transfer Operations per Image :** For each image, we retrieve the image data from a text file and subsequently transfer it to the GPU's memory. So, overall process involves only two CUDA operations: one for allocating memory during image loading and another for transferring the final softmax result back to the host.

- **Direct Kernel Invocation :** In subtask 2, the previous approach involved invoking a function for every layer. Within this function, we transferred input memory from the host to the device, executed the kernel function, and subsequently transferred the output back to the host. However, we've optimized this process by directly invoking the kernel function for each layer since the input, weights, and biases are already stored in device memory.

- **Making kernel functions for 3D convolution :** In convolution 2, we are first doing convolution which results in the output of 20X50 matrices of size 8X8. Then we are using a kernel function named computeConv2Output which sums the value of 20 matrices for each kernel and adds bias at the end and outputs 50 matrices of size 8X8. We have optimised our conv2 as every thread in a grid computes a value of 8X8 matrix and we have total 20 grids working, so total threads working concurrently are 20X8X8, so we have taken a blocksize of 8X8 and a gridsize of 20 accordingly.

  Similarly, in FC_1, we've parallelized the convolution process for 50 matrices simultaneously. To achieve this, we've chosen a block size of (4, 4) and a grid size of (2, 2), totaling 64 threads to parallelize the computation for 50 matrices.

  Similarly in FC_2, we've parallelized the convolution process for 500 matrices simultaneously. To achieve this, we've chosen a block size of (32, 32) and a grid size of 1, totaling 1024 threads to parallelize the computation for 500 matrices.



3.2 Experiments and Optimisations

1. Initially, our approach involved directly utilizing the functions developed in subtask 2 within subtask 3. However, at each level of computation, this resulted in the execution of 3 cudaMallocs and cudaMemcpy functions. Consequently, our implementation was requiring ***10 seconds*** *per image* for computation.

2. We optimized our code by utilizing cudaMallocs and cudaMemcpy only once, rather than for every image. All weights and biases were loaded beforehand, prior to initiating computations on image files. This implementation led to a significant reduction in the time required for computation per image, achieving a processing time of ***1.2 seconds.***

3. We further improved our approach by directly calling each kernel function instead of using an intermediary function. This optimization led to a decrease in the time required for computation per image, achieving a processing time of approximately ***0.2 seconds.***

4. After implementing the aforementioned changes, we found that FC_1 posed the most significant time constraint. Originally, we parallelized the convolution process for a 4X4 matrix. However, since the convolution kernel was called sequentially 500X50 times, we refined our approach by parallelizing each entry of a 4X4 matrix and 50 such matrices concurrently. This optimization allowed for the parallelization of 50X4X4 steps, significantly reducing the processing time per image to ***0.02 seconds.***

5. Subsequently, we extended this approach to all layers involving 3D kernels, such as Conv2 and FC_2. As a result, the processing time per image was reduced to ***0.01 seconds***.

3.3 Observations

Time required per image approximately	0.01 seconds
Total time required for 10K images	90 seconds
Correct images obtained out of 10k images	9998
Accuracy obtained	99.98 %

Table 1: Observations

```
correct: 9899 total: 10000
Total Time taken: 91670milli seconds
Accuracy: 98.99%
```

4 Subtask 4

- Stream Creation overhead: Initially, CUDA streams were dynamically created within the loop for processing each image. This led to overhead due to repeated stream creation and destruction for each image. Total of 50 streams were created individually inside the image processing loop.

```
# Previous Implementation Pseudocode
for img in images:
    for i in range(n):  # Create streams dynamically for each image
        create_stream()
        kernel<<<>>>(stream, ...)  # Invoke kernel using dynamically created stream
        destroy_stream()
```

- Reusing pre-created streams for all images: All 50 CUDA streams are now pre-created outside the image processing loop. Streams are instantiated only once, reducing overhead. Reusing pre-created streams across all images within the loop improves efficiency.

```
# Optimized Implementation Pseudocode
for i in range(n):  # Pre-create streams outside the image processing loop
    create_stream()
for img in images:
    for i in range(n):  # Reuse pre-created streams for all images
        kernel<<<>>>(stream, ...)  # Invoke kernel using pre-created stream
```

- We observed a significant increase in program execution time when the number of streams was increased from 50 to 500 in a specific part of the program. The increase in execution time was attributed to several factors, including stream synchronization issues and contention for memory and compute resources. Instead of using 500 streams, each for one kernel call, we opted to use 50 streams to process 500 kernels, ie, each stream processing 10 kernel calls.

```
correct: 9899 total: 10000
Total Time taken: 66980milli seconds
Accuracy: 98.99%
```

Time and accuracy obtained using CUDA Streams.

4.1 Results

Time required per image approximately	0.008 seconds
Total time required for 10K images (with streams)	66 seconds
Correct images obtained out of 10k images	9998
Accuracy obtained	99.98 %

Table 2: Results with streams

The speedup formula is given by:

$$Speedup = \frac{ExecutionTimeWithoutStreams}{ExecutionTimeWithStreams}$$

Substituting the given values:

$$Speedup = \frac{90\,sec}{66\,sec} = \frac{9}{8} = 1.36$$

So, the speedup achieved by using streams is approximately 1.125 times.