COL380 Assignment 2 report
Image Processing Library using CUDA

1 Code overview

1. **SubTask 1: Implementation of basic CNN functions**
   - Functions implemented: convolveMatrix, reLU, tanH, maxPool, averagePool, sigmoid, softmax.
   - The main function parses the command line arguments and calls the necessary functions with the corresponding inputs.
   - We are using vectors for inputs as well as outputs for ease of use (could have been implemented using pointers too).
2. **SubTask 2: Parallelization using CUDA Kernels**
   - All the corresponding functions have now been converted to CUDA kernels.
   - As an extra function we also created a padding function that pads the input matrix into the required padded matrix and parallelized this using CUDA kernels as well
   - In almost all functions we have used a 4 dimensional hierarchy of threads (2 block dimensions followed by 2 thread dimensions). [All indices are accessed in the following manner: blockID * blockDim + threadID]
   - For softmax function, since the summing up part could not be easily parallelized, we first performed the exponentiation (parallely), followed by finding the sum of the entries (serially), and then normalizing the values (parallely). [On a side note: the summing up could be parallelized using reduction kernels, which we have not implemented for now due to lack of time for experimentation]
   - The issue for non-parallelizability of operations like summing/max over elements pops up in convolution as well as pooling too but we have tackled this issue by allowing a kernel thread to work on kernel dim / pool dim number of elements
   - While several experiments could be done to determine the block dimensions and thread dimension split up, we have decided to go with 256 threads per block with 8 blocks. (Optmization discussions will follow this section)
3. **SubTask3: Implementation of LENET-5 using CUDA Kernels**:
   - Most of the functions used in this part is similar to the one in SubTask2. However since there are now input and output channels, this adds an extra dimension to the previous convolution functions.
   - We have now made a convolutional layer function which is parameterized input, output dimensions and kernel size. This helps us to create different convolution layers using the same cuda function
   - Additionally helper functions were made for reading and storing input image vectors and the weights (in the weights struct)
   - Finally the forward prop functions stitches all the layers together and passes the input matrix through all layers and saves the final output in the outputVector (stored on device) which is then copied back to hostoutputVector and the top five probabilites are returned
   - In this subtask, since the dimensions for all layers were fixed, we were able to hardcode the thread hierarchy sturcture layout (threadsPerBlock and blocksPerGrid)
4. **SubTask4: Optimizing throughput with CUDA Streams**:
   - Finally the subtask3 code is optmized using CUDA Streams to pass images in different streams
   - All the cudaMalloc, cudaMemcpy and cudaFree functions have now been converted into cudaMallocAsync, cudaMemcpyAsync and cudaFreeAsync such that it runs asynchronously for each stream.
   - We are also removing all cudaDeviceSynchronize function calls to avoid waiting for all threads

2 Design Decisions

- Some of the optmizations done in subtask2 and subtask3 are in relation with the Block Dimensions within the grid and the Thread Dimensions within the Block.
- In subtask2, we decided to use 256 threads per block with them being split equally in both dimensions (i.e. a 16x16 distribution of threads) as well as 8 blocks per grid split up as 4x2.
- The primary reason for the above is that we tried to maximize gpu occupancy (i.e. warp efficiency) by having it be a multiple of 32 and distributed accordingly (keeping a limit of 2-48 threads per SM)
- Note that the exact speedup would also depend on the gpu device being run on because each device would have different specifications of its Streaming Processor (SM) which could potentially affect the number of threads per block and blocks per SM
- In subtask3 and subtask4 however, since we know the exact input and output dimensions at each layer, we had hardcoded the values for block Dimensions and thread dimensions

3 Runtime Analysis

Note that there are 2 kinds of results - One for uncached memory and another for cached memory. (This is due to multiple runs on the gpu which caches the inputs for further future use)

- In uncached memory (i.e. First run)
  - Without streams - 52s
  - With streams - 30s
- In cached memory (i.e subsequent runs)
  - Without streams - 11s
  - With streams - 7s

4 Team info

	Name	Entry Number
1	Sarthak	2020CS10379
2	Brian Sajeev Kattikat	2021CS50609
3	Aman Hassan	2021CS50607

Table 1