ECE408 Final Report

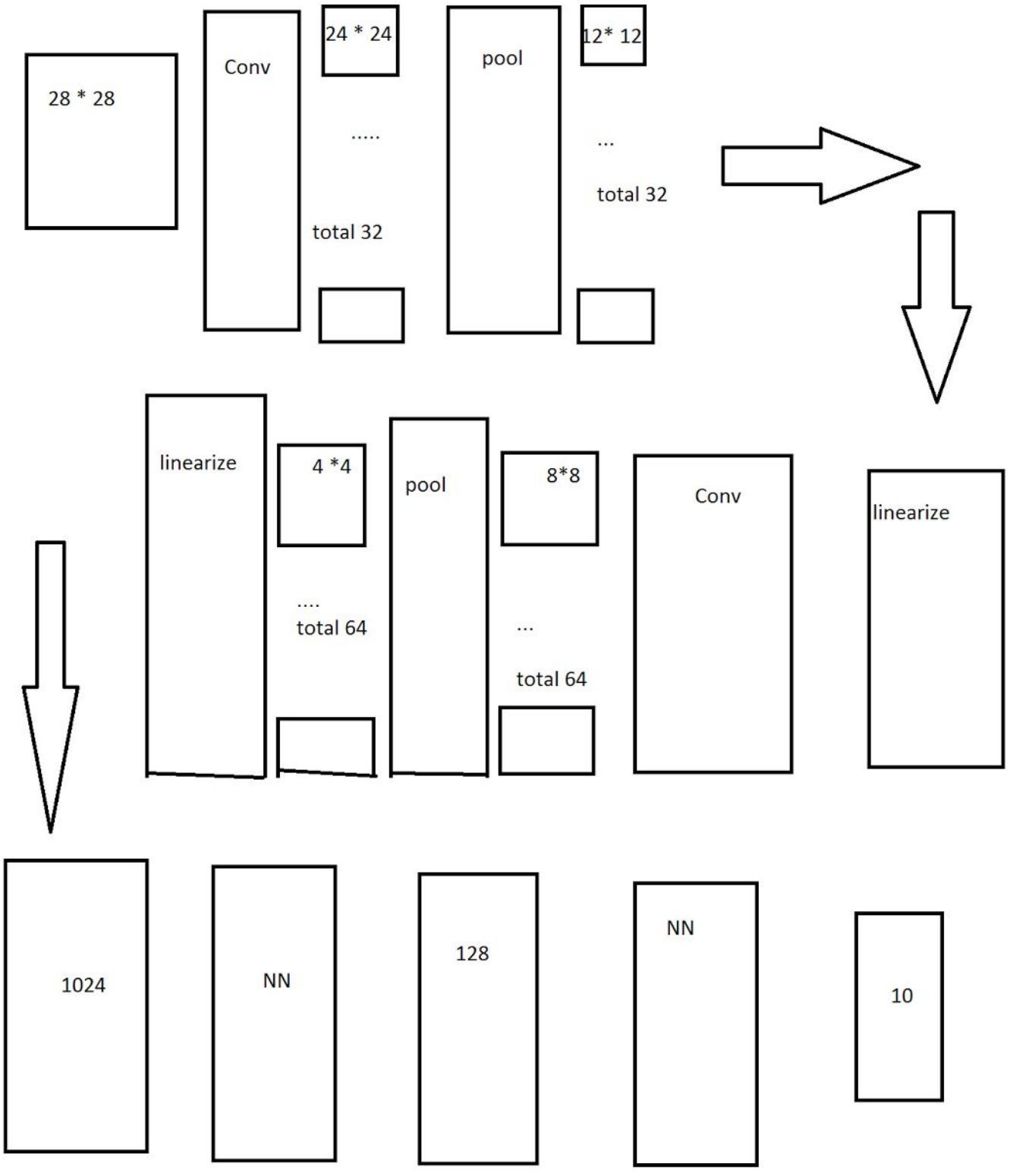
CUCUDA

Xiaohao Wang, Zhao Weng, Xiaosheng Wu

xwang165 & zweng4 & xwu38

Introduction

For our final project, we improved the performance of a convolution neural network by utilizing various parallel computation technique. We optimized the forward propagation process of convolutional neural network. The workflow of the forward propagation process was illustrated in the diagram below. There were two convolutions to go through before the neural network. At first there was only one channel for the input features. After the first convolution, there were 32 input channels and after the second convolution, there were 64 channels. After two convolutions and averages pools, the features maps went through neural networks with one hidden layer to output 10 channels for each input.


28 * 28
Conv
24 * 24
.....
total 32
pool
12* 12
...
total 32
linearize
4 *4
....
total 64
pool
8*8
...
total 64
Conv
linearize
1024
NN
128
NN
10

Optimizations

There are 4 stages of our optimizations

1. Parallelize the convolution computation

We parallelized the computation for each output. Each thread was responsible for calculating an output element. Therefore, each thread block was responsible for a thread block size of output elements, which was defined as TILE_WIDTH * TILE_WIDTH. For each blockIdx.y index, a output matrix is calculated. For each blockIdx.x index, a sequence of output features maps corresponding to a sequence of input features was calculated.

- Memory Coalescing: Memory accesses were not coalesced because adjacent threads in the thread block did not access adjacent elements in the memory. Memory bursting was not fully utilized
- Control Divergence: For the first convolution, there were 8 + 4 = 12 control divergences in each output feature map calculation because block dimension was 16 * 16, and output dimension was 24 * 24. For the second convolution, there was 4 control divergences in each thread block and there were input_num * 64 outputs. So there were input_num * 64 * 4 total control divergences. There were 4 control divergences in each thread block because each output feature map is 8 * 8 but TILE_WIDTH is 16 * 16. so one thread

block is assigned to calculate a features map and there were 4 control divergences in calculating each output feature map.

2. Parallelize the computation of linearization and average_pool

- Control divergence: Since later, we incorporated linearization into computation of convolution, the control divergence for linearization was the same as convolution. For average pool, control divergence was 6 for each output feature map because each output feature map was of size 12 * 12 and each tile was of size 16 * 16. Also, since there were output_num * 32 output feature maps for average pool, there were in total output_num * 32 * 6 control divergences for the first average pool. For the second average pool, control divergence was 2 for each output feature map because each output feature is of size 4 * 4 and each tile is of size 16 * 16. In addition, since there are output_num * 64 output feature maps for the second average pool, there are in total output_num * 64 * 2 control divergences.
- Memory coalescing: memory access were not coalesced because adjacent threads in the threads blocks didn't access adjacent elements in the memory. So memory bursting was not fully utilized.

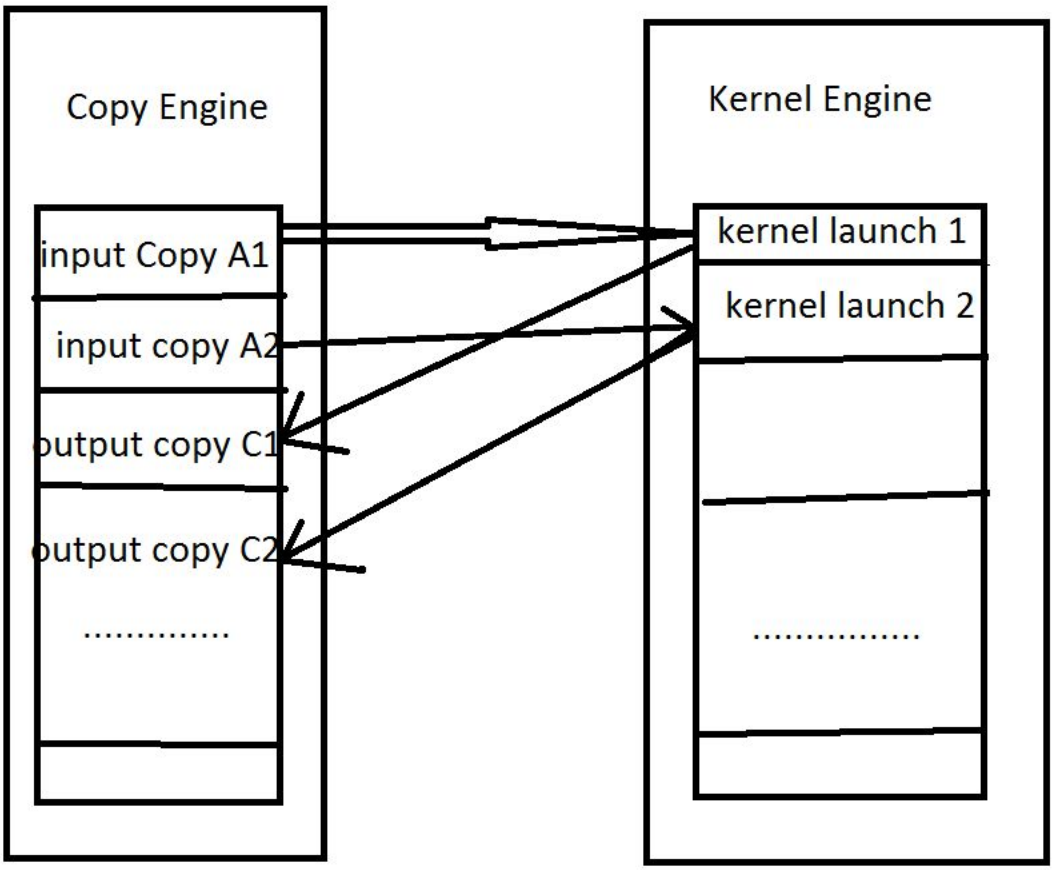
To compute linearize, we assigned each thread to a single output element. In addition, to compute average_pool, each thread was responsible for calculating an output element. The mechanism to parallelize the average_pool was the same as convolution computation except that no filters is involved to calculate each output element.

3. Streaming

We thought besides parallelization of computation of each output, we can also parallelize the copy and kernel launch tasks. So we added streaming into the project. At first, only two streams existed. When the first stream finished copying the first chunk of input features maps and launches kernel, the second stream can copy the second chunk of input features maps into device. So the processing of copying and kernel launching became parallel.

We used pinned memory, cudaHostAlloc so that OS can't page out the pinned memory pages. Then better data transfer rate was guaranteed by using DMA. For elements which can be shared by different streams, normal Memcpy was used and for elements which can't, cudaMemcpyAsync was used. During convolution computation, we separated the computations into different segments. Each segments contained 100 outputs. Then two streams alternated. When one stream

finished copying elements and launched kernel, another stream started copying elements. This way, tasks of copying and kernel launch overlapped. The following diagram illustrated the mechanism of streaming. However, our code always gives Docker execution: exit status: 139 which means seg fault in Linux, although we have cudaFreed variables and CudaFreeHost pinned memory.

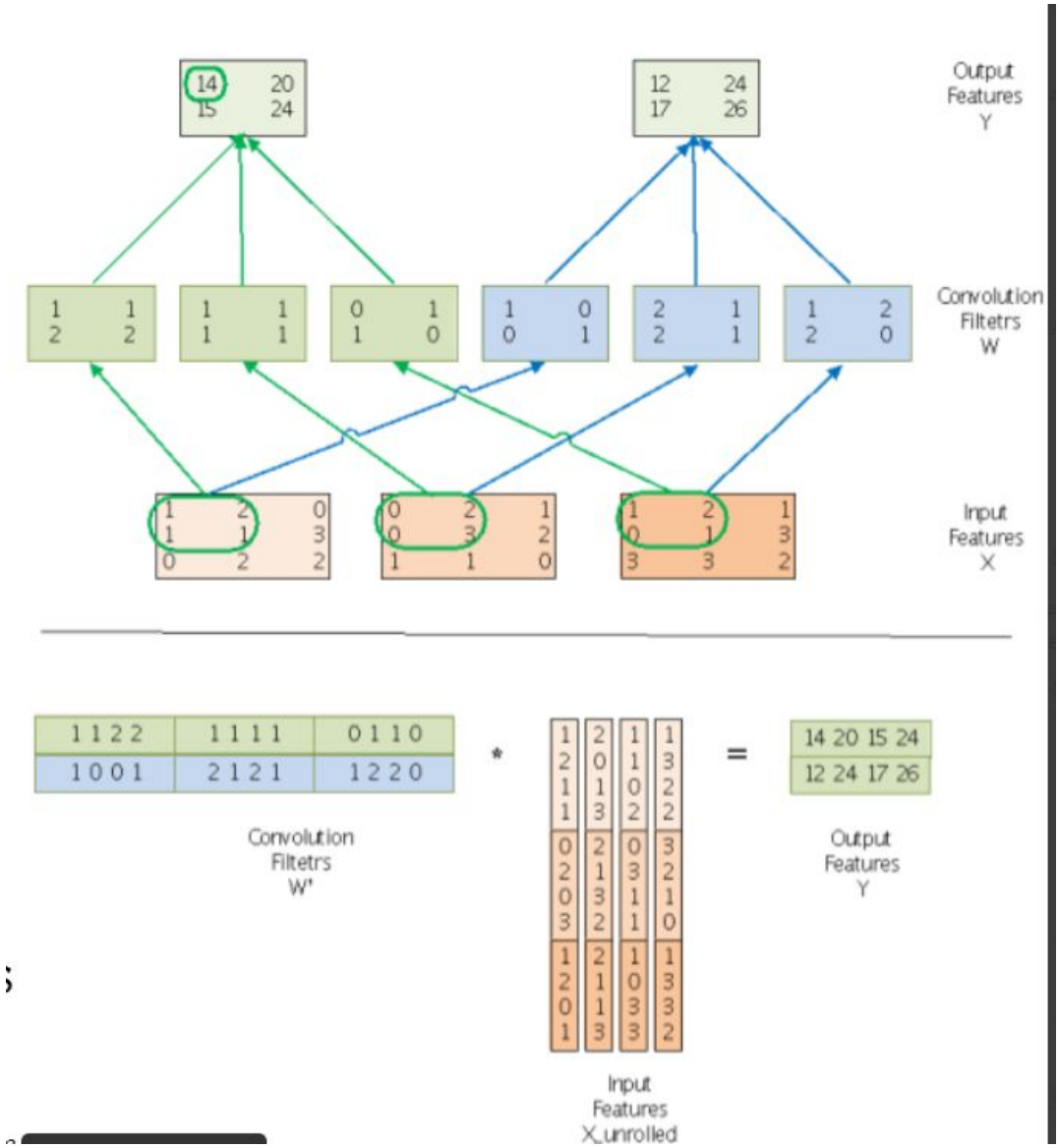


4 Build Matrix Multiplication on the fly

For matrix multiplication part of convolution, use the same matrix format as book. We designed it to load data to shared memory on the fly.

The mechanism was illustrated in the diagram below. Instead of first building a matrix from convolution inputs and filters and storing that in the global memory, we built the matrix tiling computation on the fly. That way, there was smaller overhead before computation of matrix multiplication than that when a concrete matrix needed to exist in global memory.

In addition, for simplicity, we chose to use 25 by 25 blocks so each thread was responsible for loading one element from mask and one from the input. This sizing is convenient because the size of each filter map is 25.



Performance Analysis

After parallelization of convolution, linearization, and average_pool. Our computation's running time was around 40 seconds. After integrating parallelization of neural network, the running time became around 9 seconds. After

making use of matrix multiplication and building matrix on the fly, the running time was reduced to around 4 seconds. We also experimented with streaming. After making use of streaming, the running time should be reduced to around 2 seconds.

Conclusion

We successfully utilized techniques learned in ECE408 to this final project and speeds up the processing while maintaining the accuracy the same as that of sequential code. During experiments with different techniques, we followed the guideline

1. parallelize linearization, convolution, average pool without any other optimizations
2. Parallelize neural networks computation.
3. Convert convolution computations into matrix multiplication and further optimize matrix multiplications using shared memory
4. Combine linearization and convolution so during calculations of the output of convolution so that a kernel launch was saved.
5. Combine streaming to parallelize tasks

6. Move filters into the constant memory. However, since we were not sure whether the constant memory was large enough to fit all filters for each convolution computation, we didn't include that in the final project design

Reflection

1. During design of this final project, we did meet some bugs. The most typical one was that we used wrong index to get the element in the input features and filters during computations of convolution.
2. For streaming, cudaFree was missing which causes seg faults.
3. In addition, we first developed the final project locally. On our machine, we had a relatively powerful GPU, GeForce GTX 960M. The constant memory was not enough to fit all filters for each kernel launch of convolution computations. For scalability of our implementation, we wanted to make sure our implementation can be run on most machines, we didn't include that part of design into the project, that is, filter elements were still accessed from global memory. Without scalability consideration, we thought our running time can be further reduced.
4. Furthermore, we thought there was way to optimize indexing so that memory coalescing could be better for our convolution computation.