# ECE 408

## Final Project Report

Team: 308 E National Lab
Xiaocong Chen
Xinzhou Zhao
Tianyi Shan

## Table of Contents

## 1. Introduction

In this project, we accelerated the CNN using GPUs with CUDA. The application was accelerated with several steps including convolution, relu, average pool. The most important and challenging part is convolution and we will mainly give the analysis and evaluation about this part in the report. We will also give the analysis and methods of optimization of each part.

## 2. Convolution optimization

### 2.1 Forward path of convolutional layer

All experiment results are under the circumstance of test data with batch size 10000, whose width and length of the input image for first convolution is 28*28, the width and length of the input image for second convolution is 12*12, and the size of mask is 5*5. In the following content, the input width and length are represented by *tile_width*. The *tile_width* of the two convolutions are 24 and 8 respectively. The width and length of mask is represented by k.

### 2.1.1 Basic kernel with forward path of convolutional layer

First, we used a basic CUDA kernel to implement the convolutional layer. The block size for the first and second convolution are both 32*32*1. This kernel has a high level of parallelism but do not use any shared memory or constant memory. The parameters of the two kernels are shown as in Table. 1.

| | Convolution kernel1 | Convolution kernel2 |
|---|---|---|
| Run time | 2.8s | 17.2s |
| Data reuse factor | 1 | 1 |
| Control divergence | Yes | Yes |
| Memory coalesce | No | No |
| Block_size | 32*32 | 32*32 |
| Grid_size | num_batch*num_mask | num_batch*num_mask |

Table 1 Analysis of Basic kernel with forward path of convolutional layer.

### 2.1.2 Forward path of convolutional layer using less threads

Next, we found that there are many useless threads in each block, since in first convolution, there are 24*24 output elements, and in second convolution, there are 8*8 output elements. So we tried to reduce the number of useless threads, the block sizes for two convolutions are 24*24 and 8*8 respectively.

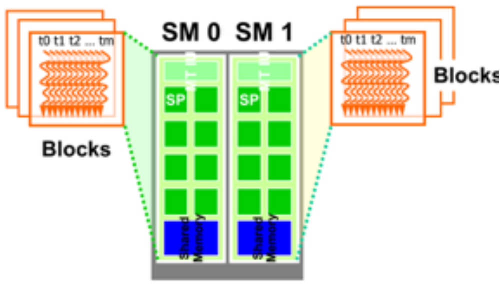


Fig. 1 The Streaming multiprocessors (from slides).

As shown in Fig. 1, since the computation capacity of multiprocessor is limited by number of threads and blocks, useless threads may cause unnecessary latency due to control divergence, and waste the computation resource. So we can improve the performance by reducing the number of threads in each block, and the analysis is listed in Table 2.

| | Convolution kernel1 | Convolution kernel2 |
|---|---|---|
| Run time | 2.0s | 5.4s |
| Data reuse factor | 1 | 1 |
| Control divergence | Yes | Yes |
| Memory coalesce | No | No |
| Block_size | 24*24 | 8*8 |
| Grid_size | num_batch*num_mask | num_batch*num_mask |

Table 2 Analysis of implementation with forward path in convolutional layer.

### 2.1.3 Basic kernel of convolutional layer using shared and constant memory

In next step, we tried to use the shared memory and constant memory to reduce the latency due to memory bandwidth. In kernel one, we used constant memory to save the elements of mask: *__constant__ float M1[5*5*32]*. we cannot use the constant memory for the second convolution because of the limited size of constant memory space.

Both two kernels can use shared memory. The input size is (*tile_width*+k-1) * (*tile_width*+k-1), while the output size is (*tile_width*) * (*tile_width*). Each block will load an input image and it can achieve data reuse since we do not have to access elements in the global memory every time. The Table. 3 is the analysis of two kernels using shared and constant memory. The analysis is shown in Table. 3.

### 2.1.4 convolutional layer with new design of block

As Fig. 2 shows, every block has 12*12 threads and we use *num_batch*num_mask* blocks in total. However, the multiprocessor can deal with limited number of blocks every time. So if the number of threads in each block is small. The multiprocessor cannot deal with the maximum number of threads which may have negative influence on the performance.

| | Convolution kernel1 | Convolution kernel2 |
|---|---|---|
| Run time | 0.45s | 1.7s |
| Data reuse | $(k^2*tile\_width^2)/(tile\_width+k-1)^2$ | $(k^2*tile\_width^2)/(tile\_width+k-1)^2$ |
| Control divergence | Yes | Yes |
| Memory coalesce | No | No |
| Block_size | 28*28 | 12*12 |
| Grid_size | num_batch*num_mask | num_batch*num_mask |

Table 3 Analysis of Basic kernel of convolutional layer using shared and constant memory.

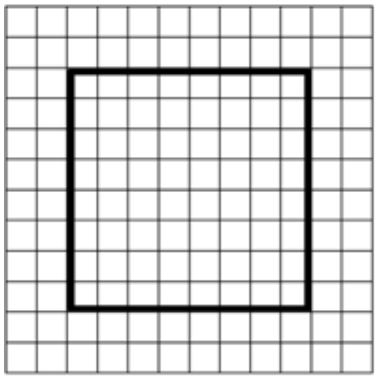


Fig. 2 The block for the kernel of the second convolution.

Another problem is that 12*12 are loaded in shared memory for each block, but the output tile size is only 8*8, which means some threads are only loading elements in shared memory without writing jobs. We can improve the performance by making each thread contribute to the output and reduce the number of threads in total. In addition, this method also leads to control divergence which will cause control divergence and lose performance for GPUs.

As said before, we came up with an idea to solve this problem, which increases number of threads in each block and avoid waste of threads by reducing the total number of blocks

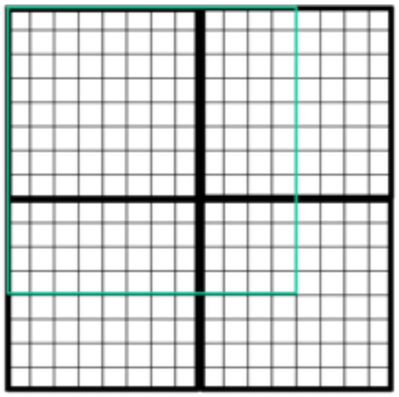


Fig. 3 The revised kernel of the second convolution.

We designed a new kind of block (Figure 3). We used each block to load "four images", the block size is 16*16 and four images can use the same "image" shared memory load from global memory. Each thread is used for one output elements which avoid waste of threads.

This design also avoids the control divergence, since the row and column have 16 threads and every warp has 32 threads. The number of blocks also decreased as each thread can now process more "images", so the number of blocks decreased to 1/4 of previous design. This design also increased the data reuse factor of kernel.

Table 4 lists the analysis of this new design.

| | Convolution kernel1 | Convolution kernel2 |
|---|---|---|
| run time | 0.15s | 0.67s |
| data reuse | $(k^2\text{*tile\_width}^2)/(\text{tile\_width}+k-1)^2$ | $(k^2\text{*tile\_width}^2)/4\text{*}(\text{tile\_width}+k-1)^2$ |
| control divergence | No | No |
| memory coalesce | No | No |
| block_size | 28*28 | 16*16 |
| grid_size | num_batch*num_mask | (1/4)*num_batch*num_mask |

Table 4 Analysis of convolutional layer with new design of block.

## 2.2 Convolutional layer with unroll

Another implementation of convolution is by using unrolled matrix, which is mentioned in chapter 16.14. This method outperforms the previous one as, first, it executes the convolution by matrix multiplication, and there are many matrix multiplication libraries to use, like cuBLAS, which is a further speed up; second, the memory reuse factor is related to tile width, while that of normal convolution is related to mask size.

### 2.2.1 Normal Optimization

A normal optimization is to have each thread to be responsible for each output pixel. This is mainly used for debugging the function logic. There is no memory reuse involved, nor does memory coalesce.

### 2.2.2 Tile Implementation

To increase memory reuse, one optimization is to use tile implementation. All threads in the same tile will load the input data together into shared memory, including the convolution mask and input feature map. Then is the matrix multiplication, and moving the tile position on feature map forward. This process is illustrated in Fig. 4.

Besides using shared memory, there are some small optimization in this kernel function.

First, virtually enlarge *tile_width* for better data reuse. when the block size is [32*32], size of convolution filter *W'* will be [64, 5*5*32], and input features matrix for each test image *X_unrolled* is [5*5*32, 8*8]. As can be seen, each output feature map is [8*8] = 64, and the height of *W'* is 64. Both of them are twice of *tile_width*. However, max *tile_width* for a square tile is 32, so in each iteration, each block will be loading 2 convolution sub-masks in shared memory, which are *Mask1*[][] and *Mask2*[][], and 2 input feature sub-blocks, which are inputImg1 and inputImg2. Now each thread will cover 2x2=4 output pixels in output feature map. The memory reuse factor is 2 times as before.

Second, transposed when loading *W'*. The dimension of convolution mask is [5x5x32x64], where the distance between neighboring pixels in the same mask is 32x64. Without transposed, threads with contiguous *threadIdx.x* will be far away from each other, then there will not be memory coalescing. However, mask values on same column in *W'* are contiguous in memory, which are loaded by threads with contiguous *threadIdx.y*. To have better memory coalescing, we transpose loading of *W'*, where thread [x][y] will be loading [y][x] in shared memory instead.

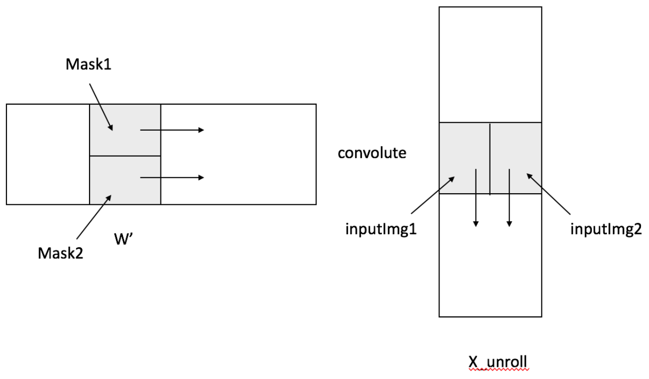


Fig. 4 Convolution with unrolled matrix in tile implementation.

Here is the table of the analysis of convolutional layer with unroll.

| | Convolution kernel1 | Convolution kernel2 |
|---|---|---|
| Run time | 51.9ms | 260.2ms |
| Data reuse | 32 | 32 |
| Control divergence | No | No |
| Memory coalesce | Yes | Yes |
| Block_size | [32, 32] | [32, 32] |
| Grid_size | Num_batch | Num_batch |

Table 5 Analysis of convolutional layer with unroll using shared memory.

### 2.2.3 Using Constant Memory

Next, our group tried to use constant memory to reduce the latency. The convolution mask is not modified during convolution. One optimization is to load mask into constant memory instead of global memory, so to save memory read time. We have implemented this in first convolution, whose mask size is [5*5*32]. However, the mask for second convolution is too large to be stored in constant memory, which is [5*5*32*64].

For the insufficient memory space problem, one solution is to load the mask in rolling manner, where first partition is loaded, then read, then released, then next partition. But the code will be much more complicated, and there are overhead of loading and releasing constant memory, as well as launching kernel function. Therefore, constant memory is not used in second convolution.

### 2.2.4 Layout Transformation

Besides, out group changed layout transformation to realize the memory coalesce. As stated before, the original dimension does not fit for memory coalescing. We transform the layout in input array and when writing to output array.

Take the second convolution as an example. The original input feature map is [k*12*12*32], and output feature map is [k*8*8*64], where k is the batch size. After transformation, input is [k*32*12*12], and output is [k*64*16*4]. The following table shows the analysis of this method.

### 2.2.5 Comparison

Table 7 shows the analysis of convolution layer-1 in different optimization stages. Before transposing convolution mask, some warp will experience control divergence, whose *threadIdx.y* is near 25, as the mask unrolled matrix size is [32*25]. The optimization among these 4 kernels is mainly on memory bandwidth, so there is a continuous speed up.

| | Convolution kernel1 | Convolution kernel2 |
|---|---|---|
| Run time | 34.9ms | 188.2ms |
| Data reuse Factor | 32 | 32 |
| Control divergence | No | No |
| Memory coalesce | Yes | Yes |
| Block_size | Num_batch | Num_batch |
| Grid_size | [32, 32] | [32, 32] |

Table 6 Analysis of optimized convolutional layer with unroll after layout transformation.

| | Normal Optimization | Tile Implementation | Using Constant Memory | Layout Transformation |
|---|---|---|---|---|
| Run time (ms) | 387.468 | 305.16 | 51.9436 | 34.8897 |
| Data Reuse Factor | 1 | 32 | 32 | 32 |
| Control Divergence | Yes | No | No | No |
| Grid_size | Num_batch | Num_batch | Num_batch | Num_batch |
| Block_size | [32, 32] | [32, 32] | [32, 32] | [32, 32] |

Table 7 Analysis of the first convolutional layer using unroll matrix.

### 2.2.6 cuBLAS

There are many matrix multiplications in our code, including convolution with unroll and fully_forward(). We found that there is cuBLAS library that can speed up the matrix multiplication, which utilizes a mature linear algebra library. The support from cuBLAS is also one of the reasons that many deep learning research use unroll method in convolution layer, i.e. GEMM. It is said that normally cuBLAS can achieve 2x speed up.

Using cuBLAS, first, we need to change the matrix storage from row major to column major, which is to transpose the matrix when storing it. Second, we will call cublasSetMatrix() to store matrix into cuBLAS format. Third, cublasSgemm_v2() can calculate the result matrix from matrix multiplication. Fourth, cublasGetMatrix() to obtain result matrix back from cuBLAS format, then cublasDestroy_v2() to free cuBLAS matrix memory.

However, we failed to implement this as we received the error message. It seems the compiler does not support the cuBLAS. Maybe the cmake file should be modified to support cuBLAS, which will be the future optimization.

## 2.3 Comparison between convolutional layer forward path and unroll

Table 8 lists the highest speeds we achieved using forward path and unroll to implement convolutional layer. Compared with normal tile convolution, whose data reuse factor is $\frac{(tile\ width)^2*(mask\ width)^2}{(tile\ width-mask\ width+1)^2} \approx (mask\ width)^2 = 25$, convolution with unrolled matrix achieves reuse factor of $tile\ width$ =32, which is higher than previous one.

| | Convolution kernel1 (foward) | Convolution kernel2 (foward) | Convolution kernel1 (unroll) | Convolution kernel1 (unroll) |
|---|---|---|---|---|
| Run time | 0.15s | 0.67s | 0.034s | 0.18s |
| Data reuse factor | $\frac{(k^2 * \text{tile\_width}^2)}{(\text{tile\_width} + k - 1)^2}$ | $\frac{(k^2 * \text{tile\_width}^2)}{(\text{tile\_width} + k - 1)^2}$ | 32 | 32 |
| Control divergence | No | No | No | No |
| Memory coalesce | No | No | Yes | Yes |
| Block_size | 28*28 | 16*16 | 32*32 | 32*32 |
| Grid_size | num_batch*num_mask | (1/4)*num_batch*num_mask | num_batch | num_batch |

Table 8 Comparison in convolutional layer between forward path and unroll implementation.

As the comparison result shows, we can achieve highest data reuse and shortest run time using unroll method. So we adopted the unroll method to for this project.

## 2.4 Fast Fourier Transformation (FFT) for convolutional layer

Fast Fourier Transformation (FFT) is a highly parallel "divide and conquer" algorithm for the calculation of Discrete Fourier Transformation of single, or multidimensional signals and it is good at large, power of two sized data processing. It can be efficiently implemented using the CUDA programming model and the CUDA distribution package includes CUFFT, a CUDA-based FFT library, whose API is modeled after the widely-used CPU-based "FFTW" library.

The basic outline of Fourier-based convolution is:

- Apply direct FFT to the convolution kernel,
- Apply direct FFT to the input data array (or image),
- Perform the point-wise multiplication of the two preceding results,
- Apply inverse FFT to the result of the multiplication.

However, even though we write a bug free code utilizing FFT to do convolution, but we failed. Because the version of the CUDA on the server do not support this library.

## 3. Optimization of other functions

### 3.1 Relu4 and Relu2

The relu4() and relu2() functions cast the array value to be non-negative. The serial code version iterates through the array, reads each value, changes it if needed, then write back to it. As relu4() and relu2() are similar, only relu4() is studied here.

One thing to point out is that, relu() function differs from other parts, i.e. fully_forward(), in that, most of the other part can be represented by linear algebra, but relu() cannot be simply represented by that. For instance, the average_pool() function can be regarded as convolution with a mask of all 0.25. Two contiguous linear algebra operations can be collaborated into one, thereby reducing the computing effort. Unfortunately, we need to separate functions before and after relu(), and execute the relu() logic in between.

### 3.1.1 Normal Optimization

An intuitive optimization is to let each thread take charge of each output pixel. Based on its position on GPU thread cluster, the thread finds out its corresponding pixel index, and applies similar logic as in serial code.

### 3.1.2 Optimized by Embedded into Other Function

We tried to optimize the intuitive optimization by embedding into other functions. As stated, there is a read-then-write process for each pixel. In fact, the write back to same array is unnecessary, as we only need its value for next step. Therefore, the second optimization is to embed the relu logic into next function. For example, the relu4() is followed by average_pool(). Then, when adding the pixel value for averaging, 0 will be taken for its pixel value if it is negative, thereby skipping the writing back to same array. This optimization will also be mentioned in following part.

### 3.1.3 Analysis

Table 9 lists the analysis of serial and parallel kernels. The parallel kernel achieves a speed up of around 300 times.

| | Serial layer-1 | Serial layer-2 | Parallel layer-1 | Parallel layer-2 |
|---|---|---|---|---|
| Run time (ms) | 3013.68 | 665.508 | 12.4404 | 2.47833 |
| Data Reuse Factor | NA | NA | 1 | 1 |
| Control Divergence | NA | NA | No | No |
| Grid_size | NA | NA | num_batch*$\frac{(24*24*32)}{1024}$ | num_batch*$\frac{(8*8*64)}{1024}$ |
| Block_size | NA | NA | 1024*1 | 1024*1 |

Table 9 The comparison between serial and parallel relu4 layers.

## 3.2 Average_Pool

This function subsamples the feature map by 2 in both height and width, by taking the average of each [2x2] sub-block for each output pixel. The serial code iterates through each output pixel, and takes the average of its corresponding [2x2] sub-block.

### 3.2.1 Normal Optimization

An intuitive optimization is to let each thread take charge of each output pixel. Based on its position on GPU thread cluster, the thread takes average of the sub-block in input array.

### 3.2.2 Including relu4

We tried to optimize this function by including relu4(). As stated before, the relu4() logic can be embedded into its following function. An optimization is by eliminating the extra read-and-write in relu4(). When reading value from input feature map, if it is negative, tmpSum will add 0; otherwise, tmpSum will add that value as usual.

### 3.2.3 Tile Optimization

We can observe that the pixels in each sub-block for output pixel are not contiguous. For example, output pixel [i, j] will need to read value of [2*i, 2*j], [2*i+1, 2*j], [2*i, 2*j+1] and [2*i+1, 2*j+1] in input map. To achieve memory coalescing, contiguous threads can load input pixel values into shared memory, with tile implementation, then read value from shared memory later.

However, the disadvantage of this method is that there is an extra read-and-write into shared memory. Although reading value from shared memory does not cost much, layout transformation would be better without arousing much overhead. So we use layout transformation to solve this problem.

### 3.2.4 Layout Transformation

We solved the memory coalescing problem using layout transformation. This optimization is widely used in our project, for example, convolution layer and average_pool layer. Take this function as an example, there are two layout transformation implemented.

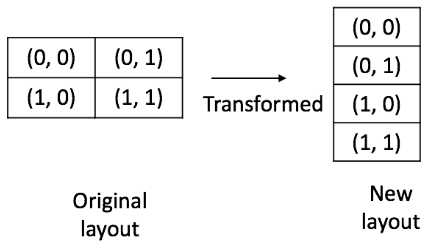


Fig. 5 Layout transformation for average pool.

As shown in above Fig. 5, the first transformation is by transforming each [2x2] sub-block into [4x1] sub-block. Therefore, pixel values read by threads with contiguous *threadIdx.x* will be coalesced. Also, the dimension of feature map will be changed, i.e., from [8x8] to [16x4].

The second transformation is to move the last dimension forward to second dimension, which is the feature map index for each tested image. For instance, the original dimension for input feature map set of second average_pool layer is [k, 8, 8, 64], where k is the batch size, or number of test images; after transformation, it becomes [k, 64, 8, 8] (without involving first transformation). Reason for this is, in original layout, neighboring pixels in same feature map have distance of number of feature map, which is the last dimension, with each other. When we manipulate on same feature map, we only care about pixel values from same feature map, instead of on different feature map values of same pixel position. So, moving the last dimension forward can achieve better memory coalescing. In short, after these two transformation, the corresponding dimensions are listed in Table 10.

The relu4() has been embedded in average_pool() function. The last 3 layers should not be transformed as the original layout is consistent with fully_forward() mask.

Although the layout is transformed, it does not invoke overhead. The output data in these 4 layers are transformed when written into output, so next layer does not need to explicitly transform the input data.

| Layout | Input before transformation | Output before transformation | Input after transformation | Output after transformation |
|---|---|---|---|---|
| convolution | [k,28,28,1] | [k,24,24,32] | [k,28,28,1] | [k,32,48,12] |
| average_pool | [k,24,24,32] | [k,12,12,32] | [k,32,48,12] | [k,32,12,12] |
| convolution | [k,12,12,32] | [k,8,8,64] | [k,32,12,12] | [k,64,16,4] |
| average_pool | [k,8,8,64] | [k,4,4,64] | [k,64,16,4] | [k,4,4,64] |

Table 10 The corresponding dimensions of transformation.

### 3.2.5 Analysis of using layout transformation

Table 11 shows the analysis of different kernels of average_pool() layer-1. Run times of layer-2 are 2501.65ms (for serial), 27.6193ms (for normal optimization) and 10.0552ms (for layout transformation). The normal optimization achieves around 60 speed up, while the layout transformation further achieves 10 times speed up.

| | Serial | Normal Optimization | Layout Transformation |
|---|---|---|---|
| Run time (ms) | 10690.9 | 182 | 14.5701 |
| Data Reuse Factor | NA | 1 | 1 |
| Control Divergence | NA | No | No |
| Grid_size | NA | Num_batch | Num_batch |
| Block_size | NA | [12, 12] | [12, 12] |

Table 11 The comparison between serial and parallel relu4 layers.

Ideally, the memory bandwidth is increased by 32 times for layer-1, and 64 for layer-2. Reasons for the lower speed up with memory coalescing should be the kernel is composed of other parts than memory reading.

## 3.3 Argmax

This function finds out the highest score among 10 labels for each test image. The serial code iterates through all the scores, with outer loop among different test image, inner loop among those 10 labels of a specific test image.

### 3.3.1 Normal Optimization

An intuitive optimization is to let each thread take charge of each test image. Based on its position on GPU thread cluster, the thread determines the image index.

### 3.3.2 List Reduction

Similar with optimization of calculating sum of an array with list reduction, finding the max value can also be implemented with list reduction. For example, we can have 5 threads for each test image, then in each iteration each thread finds the max of the neighboring 2 values, and reduce them to 1 value. In this way, each thread needs to run at most 5 iteration, which is ceil(log(10)), instead of 10 iteration.

However, we don't implement it in this way for following reasons. First, __syncthreads() is needed among these 5 threads, which is expensive. The overhead of running 4 __syncthreads() is not worthwhile. Second, more threads will be needed, so the block dimension may not fit well in SM as block size is larger and more inflexible. Third, there is more control divergence among same warp.

Table 12 shows the analysis of argMax(). Normal optimization achieves a speed up of around 10 times, which is not so high compared with other kernels. This may be due to the overhead of calling device function.

| | Serial | Normal Optimization |
|---|---|---|
| Run time (ms) | 3.15529 | 0.326171 |
| Data Reuse Factor | NA | 1 |
| Control Divergence | NA | No |
| Grid_size | NA | Ceil(Num_batch/1024) |
| Block_size | NA | 1024 |

Table 12 Comparison between serial and parallel argMax().

## 3.4 Memory Transform

In naïve implementation, host code will memcpy output data of last function from device memory to host, then memcpy same data from host to device to get ready for next function. This memory transfer is unnecessary, as we can leave that output data in device, and let next function directly use the output data as current input, which is similar with overlapping the memory transfer.

One possible problem is that the global memory may be overloaded as the whole process goes. The solution is to cudaFree() input data of last function before moving to next function, as the input data is not needed in following process. By implementing in this way, global memory occupation percentage is at same level as before memory transfer overlapping.

### 3.4.1 Block Level Overlapping

The memory transfer just discussed is in global level, which is on global memory between host and device. We also consider the overlapping of memory transfer on block level. Blocks in some functions share the same position on feature map. For instance, the block manipulating specific block in convolution layer may also manipulate same block of pixels in average_pool layer. Therefore, we can merge these two functions together, and let the block execute those two function. Then these two function will 'share' the shared memory, without needing to write back to global memory. It can save the time of writing data to global memory and reading from that, instead caching them in shared memory.

However, we don't implement this. First, it will be much harder to maintain the code as the logic is more complicated. Second, the thread number needed in different function is different, so there will be more control divergence in this implementation. Third, the shared memory will be overloaded.

## 3.5 Stream

A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently.

When we decided to used stream, we expected a great performance progression. But the results turned out to be a slight improvement. The reason is that the GPU is already work in full work-load. If the GPU is dealing with a comparatively small dataset, it will have enough space to load the data that will be used in next operation. However, when the GPU is almost full, extra data will not be loaded into the GPU until the GPU finished with the previous operation.

### 3.6 Acceleration with appropriate floating number

The original data type used is single precision float32. We tried on float16, or half, which is 2 bytes long. CUDA supports float16 since CUDA 7.5. Using float16 can speed up matrix multiplication, but may lose some precision.

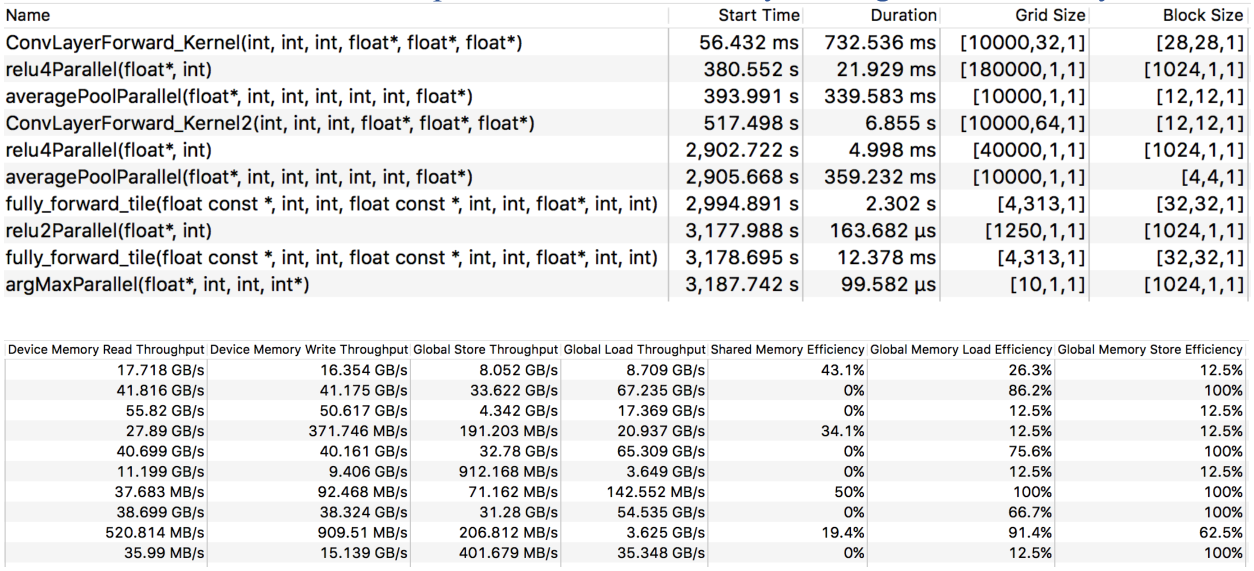
To use float16, first we need to cast float type data to half by calling __float2half(). Second, we can call cublasSgemmEx() to perform mixed-precision matrix multiplication. Third, we cast result float matrix back to half by calling __half2float().

However, this implementation also failed as the system does not support the half type, which is the similar problem as in cuBLAS.

## 4. NVProf and NVVP Results

In this part, we use three typical optimal results of NVVP to analyze the optimization of our project. The results of NVVP give us some hint on how to achieve the better performance of our project.

### 4.1 NVVP Results of Forward path of convolutional layer using shared memory



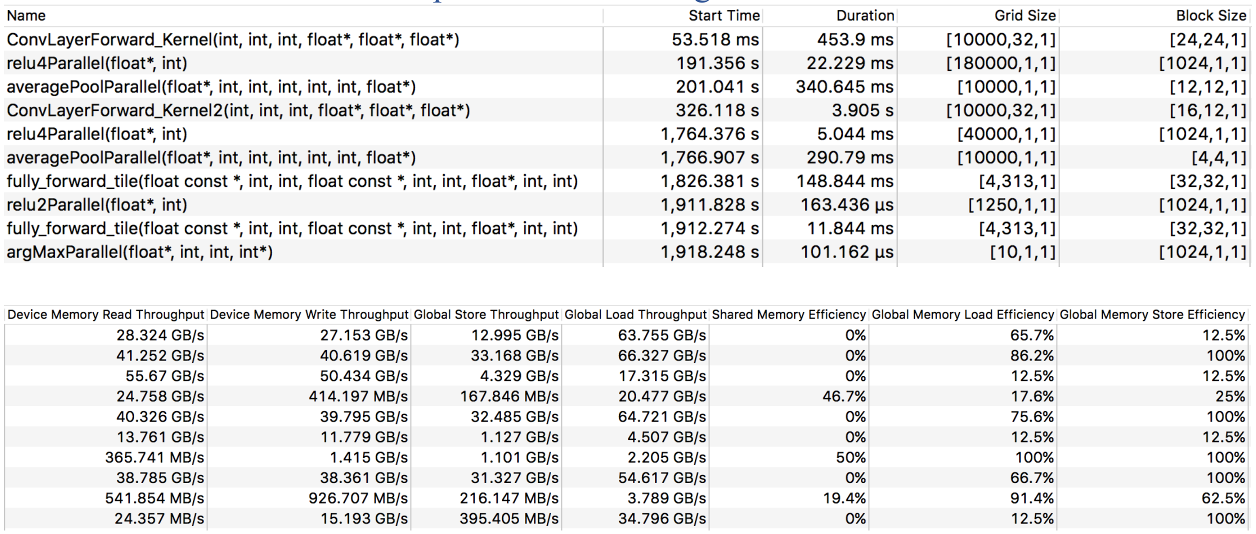
| Name | Start Time | Duration | Grid Size | Block Size |
|---|---|---|---|---|
| ConvLayerForward_Kernel(int, int, int, float*, float*, float*) | 56.432 ms | 732.536 ms | [10000,32,1] | [28,28,1] |
| relu4Parallel(float*, int) | 380.552 s | 21.929 ms | [180000,1,1] | [1024,1,1] |
| averagePoolParallel(float*, int, int, int, int, int, float*) | 393.991 s | 339.583 ms | [10000,1,1] | [12,12,1] |
| ConvLayerForward_Kernel2(int, int, int, float*, float*, float*) | 517.498 s | 6.855 s | [10000,64,1] | [12,12,1] |
| relu4Parallel(float*, int) | 2,902.722 s | 4.998 ms | [40000,1,1] | [1024,1,1] |
| averagePoolParallel(float*, int, int, int, int, int, float*) | 2,905.668 s | 359.232 ms | [10000,1,1] | [4,4,1] |
| fully_forward_tile(float const *, int, int, float const *, int, int, float*, int, int) | 2,994.891 s | 2.302 s | [4,313,1] | [32,32,1] |
| relu2Parallel(float*, int) | 3,177.988 s | 163.682 µs | [1250,1,1] | [1024,1,1] |
| fully_forward_tile(float const *, int, int, float const *, int, int, float*, int, int) | 3,178.695 s | 12.378 ms | [4,313,1] | [32,32,1] |
| argMaxParallel(float*, int, int, int*) | 3,187.742 s | 99.582 µs | [10,1,1] | [1024,1,1] |

| Device Memory Read Throughput | Device Memory Write Throughput | Global Store Throughput | Global Load Throughput | Shared Memory Efficiency | Global Memory Load Efficiency | Global Memory Store Efficiency |
|---|---|---|---|---|---|---|
| 17.718 GB/s | 16.354 GB/s | 8.052 GB/s | 8.709 GB/s | 43.1% | 26.3% | 12.5% |
| 41.816 GB/s | 41.175 GB/s | 33.622 GB/s | 67.235 GB/s | 0% | 86.2% | 100% |
| 55.82 GB/s | 50.617 GB/s | 4.342 GB/s | 17.369 GB/s | 0% | 12.5% | 12.5% |
| 27.89 GB/s | 371.746 MB/s | 191.203 MB/s | 20.937 GB/s | 34.1% | 12.5% | 12.5% |
| 40.699 GB/s | 40.161 GB/s | 32.78 GB/s | 65.309 GB/s | 0% | 75.6% | 100% |
| 11.199 GB/s | 9.406 GB/s | 912.168 MB/s | 3.649 GB/s | 0% | 12.5% | 12.5% |
| 37.683 MB/s | 92.468 MB/s | 71.162 MB/s | 142.552 MB/s | 50% | 100% | 100% |
| 38.699 GB/s | 38.324 GB/s | 31.28 GB/s | 54.535 GB/s | 0% | 66.7% | 100% |
| 520.814 MB/s | 909.51 MB/s | 206.812 MB/s | 3.625 GB/s | 19.4% | 91.4% | 62.5% |
| 35.99 MB/s | 15.139 GB/s | 401.679 MB/s | 35.348 GB/s | 0% | 12.5% | 100% |

Fig. 4 The NVVP Results of Forward path of convolutional layer

As shown in Fig.4, for the second convolutional layer, the shared memory efficiency is 34.1%. The global memory store efficiency is 12.5% since we do not have memory coalescing. The global memory store efficiency is also 12.5%. Which means we can improve the shared and global memory efficiency to get better results. Improving the memory efficiency and achieving the memory coalescing are our tasks to improve the performance. Since each block only loads 12*12 elements, so we change the design of the block to achieve better performance.

## 4.2 NVVP Results of Forward path with new design of block



| Name | Start Time | Duration | Grid Size | Block Size |
|---|---|---|---|---|
| ConvLayerForward_Kernel(int, int, int, float*, float*, float*) | 53.518 ms | 453.9 ms | [10000,32,1] | [24,24,1] |
| relu4Parallel(float*, int) | 191.356 s | 22.229 ms | [180000,1,1] | [1024,1,1] |
| averagePoolParallel(float*, int, int, int, int, int, float*) | 201.041 s | 340.645 ms | [10000,1,1] | [12,12,1] |
| ConvLayerForward_Kernel2(int, int, int, float*, float*, float*) | 326.118 s | 3.905 s | [10000,32,1] | [16,12,1] |
| relu4Parallel(float*, int) | 1,764.376 s | 5.044 ms | [40000,1,1] | [1024,1,1] |
| averagePoolParallel(float*, int, int, int, int, int, float*) | 1,766.907 s | 290.79 ms | [10000,1,1] | [4,4,1] |
| fully_forward_tile(float const *, int, int, float const *, int, int, float*, int, int) | 1,826.381 s | 148.844 ms | [4,313,1] | [32,32,1] |
| relu2Parallel(float*, int) | 1,911.828 s | 163.436 µs | [1250,1,1] | [1024,1,1] |
| fully_forward_tile(float const *, int, int, float const *, int, int, float*, int, int) | 1,912.274 s | 11.844 ms | [4,313,1] | [32,32,1] |
| argMaxParallel(float*, int, int, int*) | 1,918.248 s | 101.162 µs | [10,1,1] | [1024,1,1] |

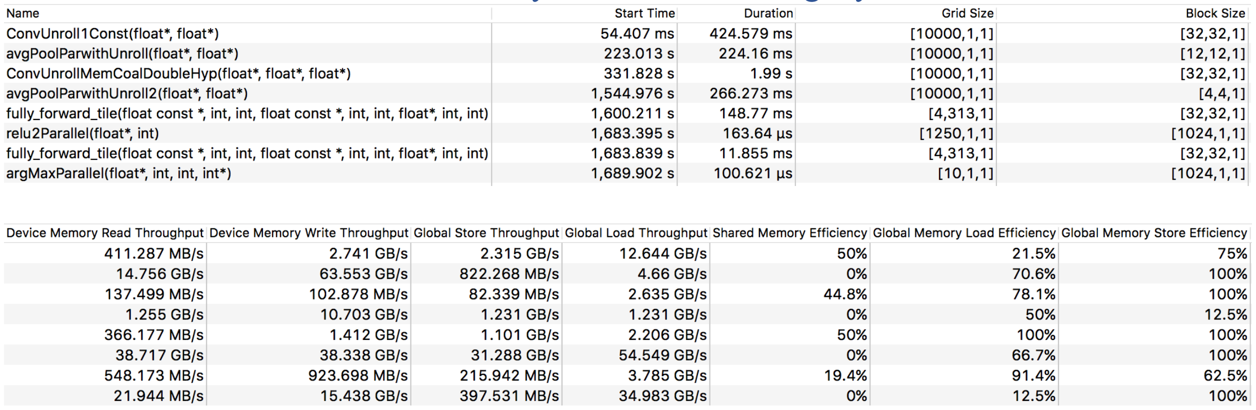
| Device Memory Read Throughput | Device Memory Write Throughput | Global Store Throughput | Global Load Throughput | Shared Memory Efficiency | Global Memory Load Efficiency | Global Memory Store Efficiency |
|---|---|---|---|---|---|---|
| 28.324 GB/s | 27.153 GB/s | 12.995 GB/s | 63.755 GB/s | 0% | 65.7% | 12.5% |
| 41.252 GB/s | 40.619 GB/s | 33.168 GB/s | 66.327 GB/s | 0% | 86.2% | 100% |
| 55.67 GB/s | 50.434 GB/s | 4.329 GB/s | 17.315 GB/s | 0% | 12.5% | 12.5% |
| 24.758 GB/s | 414.197 MB/s | 167.846 MB/s | 20.477 GB/s | 46.7% | 17.6% | 25% |
| 40.326 GB/s | 39.795 GB/s | 32.485 GB/s | 64.721 GB/s | 0% | 75.6% | 100% |
| 13.761 GB/s | 11.779 GB/s | 1.127 GB/s | 4.507 GB/s | 0% | 12.5% | 12.5% |
| 365.741 MB/s | 1.415 GB/s | 1.101 GB/s | 2.205 GB/s | 50% | 100% | 100% |
| 38.785 GB/s | 38.361 GB/s | 31.327 GB/s | 54.617 GB/s | 0% | 66.7% | 100% |
| 541.854 MB/s | 926.707 MB/s | 216.147 MB/s | 3.789 GB/s | 19.4% | 91.4% | 62.5% |
| 24.357 MB/s | 15.193 GB/s | 395.405 MB/s | 34.796 GB/s | 0% | 12.5% | 100% |

Fig. 5 The NVVP Results of Forward path of convolutional layer

As shown in Fig.5, we change the block for a new design, which means the elements in the shared memory of each block will be used more often than the first one, so the efficiency of the shared memory improve from 34.1% to 46.7%. The run time changes from 6.8s to 3.9s. The global memory is needed to be loaded in shared memory, the efficiency of the global memory store efficiency is improved because of the new design of blocks.

However, the global memory load efficiency and global memory store efficiency are relevant low, and we should improve their efficiency if we want to get better performance.

## 4.3 NVVP Results of convolutional layer with unroll using layout



| Name | Start Time | Duration | Grid Size | Block Size |
|---|---|---|---|---|
| ConvUnroll1Const(float*, float*) | 54.407 ms | 424.579 ms | [10000,1,1] | [32,32,1] |
| avgPoolParwithUnroll(float*, float*) | 223.013 s | 224.16 ms | [10000,1,1] | [12,12,1] |
| ConvUnrollMemCoalDoubleHyp(float*, float*, float*) | 331.828 s | 1.99 s | [10000,1,1] | [32,32,1] |
| avgPoolParwithUnroll2(float*, float*) | 1,544.976 s | 266.273 ms | [10000,1,1] | [4,4,1] |
| fully_forward_tile(float const *, int, int, float const *, int, int, float*, int, int) | 1,600.211 s | 148.77 ms | [4,313,1] | [32,32,1] |
| relu2Parallel(float*, int) | 1,683.395 s | 163.64 µs | [1250,1,1] | [1024,1,1] |
| fully_forward_tile(float const *, int, int, float const *, int, int, float*, int, int) | 1,683.839 s | 11.855 ms | [4,313,1] | [32,32,1] |
| argMaxParallel(float*, int, int, int*) | 1,689.902 s | 100.621 µs | [10,1,1] | [1024,1,1] |

| Device Memory Read Throughput | Device Memory Write Throughput | Global Store Throughput | Global Load Throughput | Shared Memory Efficiency | Global Memory Load Efficiency | Global Memory Store Efficiency |
|---|---|---|---|---|---|---|
| 411.287 MB/s | 2.741 GB/s | 2.315 GB/s | 12.644 GB/s | 50% | 21.5% | 75% |
| 14.756 GB/s | 63.553 GB/s | 822.268 MB/s | 4.66 GB/s | 0% | 70.6% | 100% |
| 137.499 MB/s | 102.878 MB/s | 82.339 MB/s | 2.635 GB/s | 44.8% | 78.1% | 100% |
| 1.255 GB/s | 10.703 GB/s | 1.231 GB/s | 1.231 GB/s | 0% | 50% | 12.5% |
| 366.177 MB/s | 1.412 GB/s | 1.101 GB/s | 2.206 GB/s | 50% | 100% | 100% |
| 38.717 GB/s | 38.338 GB/s | 31.288 GB/s | 54.549 GB/s | 0% | 66.7% | 100% |
| 548.173 MB/s | 923.698 MB/s | 215.942 MB/s | 3.785 GB/s | 19.4% | 91.4% | 62.5% |
| 21.944 MB/s | 15.438 GB/s | 397.531 MB/s | 34.983 GB/s | 0% | 12.5% | 100% |

Fig. 6 The NVVP Results of convolutional layer with unroll using layout

As the Fig.6 shows, we use the unroll method to realize the convolutional layer, the global memory store efficiency changes from 25% to 100%. The run time changes from 6.8s to 1.99s. Since we use the layout (shown in 3.2.5) to realize the memory coalescing, we increase our memory load efficiency from 17.6% to 78.1%. As stated before, we get better performance. Using unroll matrix multiplication with the layout transformation can realize the memory coalescing, which improves the efficiency of shared and global memory. It can also eliminate the control divergence to achieve better performance.

## 5. Result

As the above said, we use many methods to achieve the better performance, Table 13 lists the result run time of each function compared with serial code.

| | relu4(1) | relu4(2) | average_pool(1) | average_pool(2) | argmax | convolution(1) | convolution(2) |
|---|---|---|---|---|---|---|---|
| serial | 3013.6ms | 665.50ms | 10690ms | 2501ms | 3.15ms | >10000ms | >10000ms |
| parallel | 12.4404 | 2.47833 | 14.5701ms | 10.0552ms | 0.326171 | 34.8897 | 188.695 |

Table 13 The result run time of each function compared with serial code.

Compared with the serial algorithm, the GPUs improve the performance greatly. In a word, we accelerate the machine learning algorithm using GPUs with cuda language using only 270ms!

## 6. Contributions of each team member

Everyone in our group has contribute a lot to this project. Tianyi Shan implemented the forward path of convolution and relu4; Xinzhou Zhao is responsible for the attempts of FFT, stream, fully_forward and argMax; Xiaocong Chen designed the unroll method of convolution, average_pool and layout transformation. We really enjoy this cooperation!