Accelerating Forward Propagation of Convolutional Neural Networks by PyCUDA

E4750_2018Fall_ZFFP_report

Wenqi Jiang wj2285 Hanzhou Gu hg2498
Columbia University

Abstract——We build parallel algorithms of forward propagation for general convolutional neural networks and implemented ZF-Net for performance testing. Several methods include tiling, diminishing control divergence and carefully setting block size are used to optimize the algorithm speed. Finally, the most optimized algorithm is 107.35% and 1287 times faster than naive parallel method and serial method respectively.

Keywords——Parallel Computing, GPU, CUDA, Convolutional Neural Network

1. Overview

1.1 Problem in a Nutshell

Convolutional neural network is among one of the most popular deep learning structures today. Doing large scale convolution on GPU is almost a default choice for deep learning researchers, because convolution is a highly parallelizable computation where GPU can achieve overwhelmingly better speed than CPU.

Some tutorials on parallelizing convolution have been given [2]. However, these tutorials mainly focus on small scale convolution, which may not be able to fulfill the need of implementing large scale convolutional neural networks. Dealing with large scale convolution is much more difficult than smaller ones, because constant and shared memory may not able to store the large inputs and filters. A high-performance library, cuDNN [5], has been released. But it optimizes convolution at assembly level, not directly using CUDA.

In our project, we use PyCUDA to build a general model implementing forward propagation for almost any convolutional neural network structures. We first build every single building block: convolutional layers, fully-connected layers, softmax function. Then, we optimize them by using faster memories, setting better block setting and diminish control divergence. Finally, we experiment our algorithms of ZF-Net [4] and compare their performance.

1.2 Prior Work

1.2.1 ZF-net

In the paper Visualizing and Understanding Convolutional Networks, a typical convolutional neural network called ZF-net is introduced. Like other typical models in CNN, ZF-net consists of convolutional layers, pooling layer, fully connected layer and Softmax classification.

Below is Architecture of our 8-layer CNN model. A 224 by 224 crop of an image (with 3 color planes) is presented as the input. This is convolved with 96 different 1st layer filters (red), each of size 7 by 7, using a stride of 2 in both x and y. The resulting feature maps are then: passed through a rectified linear function, pooled (max within 3x3 regions, using stride 2) and contrast normalized across feature maps to give 96 different 55 by 55 element feature maps. Similar operations are repeated in layers 2,3,4,5. The last two layers are fully connected, taking features from the top convolutional layer as input in vector form (6 · 6 · 256 = 9216 dimensions). The final layer is a C-way softmax function, C being the number of classes. All filters and feature maps are square in shape.

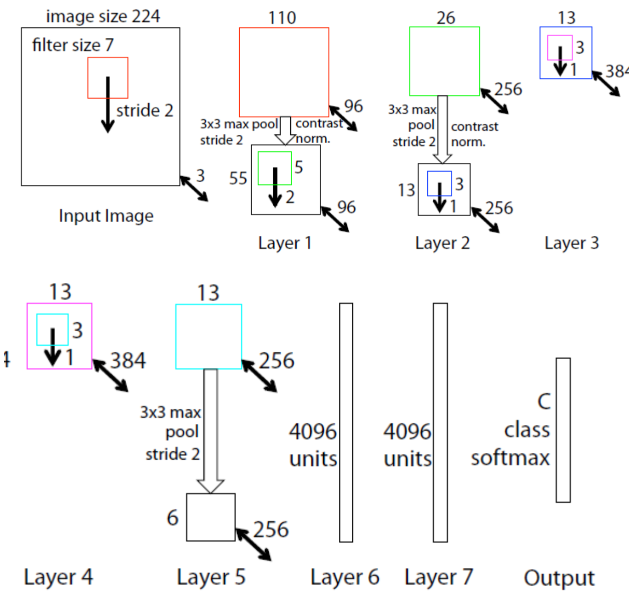


Figure 1 Structure of ZF-Net

1.2.2 cuDNN

NVIDIA cuDNN is a GPU-accelerated library of primitives for DNNs. With the help of GPU, it provides tuned implementations of routines that arise frequently in DNN applications, such as: convolution, pooling, softmax. This also gave us insight on what to parallel in CNN propagation process.

Compared with limited modules of cuDNN, our project focus on building forward propagation in convolutional neural network from scratch. Going forward, we plan to focus on continually improving performance by adopting different approaches of optimization. We are also eager to measure and compare time consumed in each approach and analyze the optimization strategy of GPU memory.

1.2.3 Programming massively parallel processors

The Chapter 7 of textbook Programming Massively Parallel Processors: A Hands-on Approach introduces parallelized convolution as a important parallel computation pattern. It first introduces a basic parallel algorithm whose execution speed is limited by DRAM bandwidth for accessing both the input and mask elements. Then it introduced the modified solution with constant memory and tiled kernel for both 1D and 2D convolution. The technique is applicable to our convolution layer in CNN and reminds us carefully consider memory access and its boundary.

2. Description

In this section, we will first profile our objectives and the challenging part in our work in 2.1. Then, in 2.2, we will further explain our work in detail. Finally, all algorithm design and implementation details will be provided in 2.3.

2.1. Objectives and Technical Challenges

Our objective is to build robust and efficient parallel forward propagation algorithm for any convolutional neural networks. In our experiment, we test the correctness and efficiency of several optimization methods on ZF-Net, a classic convolutional neural network.

We divide our task into several sections:

1. Build serial and parallel forward propagation function of convolutional layers, fully-connected layers and softmax function

2. Optimize the parallel algorithms by using shared memory and other methods like setting better block size and diminish control divergence.

3. Construct the building blocks into ZF-Net, verify the correctness and compare the performance of different algorithms.

Optimizing convolutional layer may be the most challenging part in our experiment, which could be mainly attribute to the scaring large scale of convolution. Traditionally, filters in convolutions should be quite small. In this case, we can simply store them in constant memory of GPU, which can significantly reduce access to global memory and boost algorithm performance. In most convolutional neural networks however, the filter sizes are so large that constant memory and shared memory may not be possible to store filters in one shot. Therefore, we need to carefully consider our algorithm: it should efficiently use shared or constant memory while not causing memory overflow.

Intuitively, fully connected layer can be parallelized because it can be modelled as a matrix multiplication process. To design a parallel kernel for this problem, we built a matrix multiplication kernel which takes matrix of arbitrary size as input. Since the computation of matrix is straightforward, the core optimization here is to focus on memory and cache. Although using more constant memory can largely fasten the computation because its property of fast access. However, its limited size makes it inappropriate to use it in large size matrix multiplication. So, we need think of other memory schema that can be accessed by multiple threads and store reusable data in the computation. Also grid size should be adjusted to fit the arbitrary matrix size without manually setting it.

For Softmax classification, it can be easily parallelized the by applying softmax equation to each element and assign each element to separate threads, but it can only reduce small amount of time, since the computation is still tedious and wasteful when doing sum operation for each thread. Thus, we need to figure out how to better leverage parallelism when doing computation of sum and max. Another challenge is that since we only have 10 classes of objects, we may not find significant help from GPU for such small size of data.

2.2. Problem Formulation and Design

We use PyCUDA to build parallel forward propagation algorithms from scratch. As it shown in Figure 2, we split our project into several subtasks first: build general convolutional layer functions, build general fully connected layer functions and other stuffs such as softmax function and activation functions.

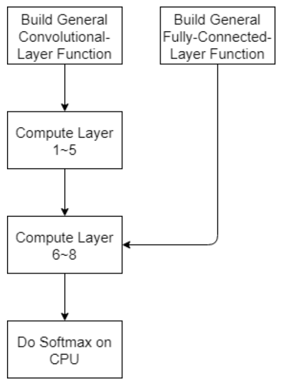

Build General Convolutional-Layer Function
Build General Fully-Connected-Layer Function
Compute Layer 1~5
Compute Layer 6~8
Do Softmax on CPU

Figure 2 the process of building ZF-Net

Then, we use these building blocks to construct ZF-Net and compare the performance of several optimized algorithms.

2.3 Software Design

In this section, we will break the problem into convolutional layers, fully-connected layers and softmax function. In each part we will discuss in detail about algorithm, block setting, anticipated result, etc.

2.3.1 convolutional layers

Serial method: The first method we implemented is serial method: using CPU and for loop in python to compute the convolution. The input of convolutional layers in ZF-Net are three-dimensional, so we use 3 for loops in python to compute every output points. Notice that convolution itself is three-dimensional, so we need 6 for loops in total.

```
for i in output row:
    for j in output col:
        for k in output layer:
            do convolution
```

Serial method is mainly used for validating convolution correctness of parallel methods and we will compare the performance comparison between serial and parallel methods later.

Naive parallel method: A naive version of parallel method was implemented without using any shared or constant memory. We will compare optimized parallel algorithms to the naive one later.

```
In each thread:
    do convolution
```

Load input into shared memory: This is a natural thought because only inputs are small enough to fit in shared memory. We leave filter in global memory, which is significantly larger than inputs.

```
In each block:
    load inputs into shared memory
    In each thread:
        do convolution
```

Because the GPU still needs to grab filter from global memory every time when doing convolution, and the size of filter is much larger than inputs, we do not expect this optimization leads to a performance boost.

Load input and iteratively load filter into shared memory: An advanced method is to iteratively load filters as well into shared memory: the GPU load a small fraction of filter each time, do convolution; then load another portion and do convolution. However, for program simplicity, we only used one layer of threads grabbing data to shared memory, which leads to control divergence.

This method should be better than the previous algorithm, especially when the filter can be reused for many times. However, due to the time GPU spend on computing index, its performance can be even worse than naive parallel method when filter reuse time is low.

```
In each block:
    load inputs into shared memory
    for fractions in filter:
        Threads in layer 1:
            load fraction to shared memory
        In each thread:
            do convolution
```

Advanced block setting & diminish control divergence: Finally, we implemented an advanced block setting, which allows filters loaded in shared memory being reused as much time as possible.

```
In each block:
    for fractions in filter:
        In each thread:
            load some input
            load some filter
            do convolution
```

In this setting, the z dimension of a block is always 0, while x and y dimension should be as large as possible. Using this setting, one block only loads one filter instead of, e.g., 384 filters. Thus, the shared memory may be sufficient to load store one whole filter. Most importantly, once a block loads a filter, the filter can be reused many times.

Also, we diminish the control divergence in this algorithm: every thread is responsible to load some fraction of input and filter, which can lead to a speedup.

Automatic block & grid setting function: we designed a function that automatically compute the block and grid size for any input and filter size. This function will mainly take memory size into account, assures that the program will not suffer from memory overflow. It can be used on almost all NVIDIA GPU because the function will take memory size of different devices as input.

2.3.2 fully-connected layers

As we have analyzed in part 2.1, the challenge of this layer is to better leverage shared memory and we should carefully design grid size. By using shared memory, we can do tiled matrix multiplication. We process the computation in several steps. For each step, load several elements in to shared memory which is the tiled memory, and reuse it to compute multiple product in the same step. In this way we can leverage the faster access of shared memory and don’t need to worry about excess size boundary of relatively large shared memory in the whole GPU memory architecture. After we compute the result of matrix multiplication we should pass all data to a ReLu function.

For serial version, use three for loops to compute the multiplication and add cumulatively. Below is the pseudo code for serial computation.

```
Serial:
for i in matrix1.length:
    for j in matrix2.length:
        for k in matrix1.width:
            do multiplication

    pass all element to ReLU function
```

As for 4 parallel method, we conduct optimization leveraging shared memory to do tiled matrix multiplication which take advantage of its faster access.

The first paralleled method we use is a naive method which simply uses each thread to do some computation. The grid size is (matrix1.length, matrix2. width)

```
Naive Parallel:
    for each thread(i,j):
        for k in matrix 1.width:
            do matrix multiplication

    pass sum to ReLU function
```

The naïve parallel multiplication has a weakness in its kernel because each thread has to read a line of matrix1 and a column of matrix 2 from the global memory, where the read and write are slower compared to shared memory. Thus, we introduce optimization1 to load elements from one of matrix to shared memory in each iteration. The tile size is (32,32).

```
Optimization 1:
    for each block:
        for each iteration:
            copy one of matrix to shared memory
            do tiled multiplication

    for each thread:

    pass sum to ReLU function
```

In the previous method we can see that tiles of first matrix have been move to the shared memory while tiles in second matrix remain in the global memory. If we copy both of them two the shared memory, more access to the global memory will be replaced by that to the shared memory, hence reduce the reading latency further, which is the essence of optimization 2. The tile size is (32,32).

```
Optimization 2:
    for each block:
        for each iteration:
            copy two matrices to shared memory
            do tiled multiplication

    for each thread:

    pass sum to ReLu function
```

The optimization 3 is combining tiled multiplication with prefetching strategy. For this method, we adopt prefetching strategy that we always read data for the next round. We first prefect for the first iteration. In the following iteration we do tiled multiplication and prefect data for the next round.

By leveraging prefetching, we aim at reducing time wasted in synchronization which is a fine-grid optimization. This may result in little reduction of time compared to previous method. But it tends to help if the input size is very large. The tile size is (32,32).

```
Optimization 3:
    for each block:
        for each iteration:
            prefetch the tile into register
            copy input to shared memory
            do tiled multiplication

    for each thread:

    pass sum to ReLU function
```

2.3.3 Softmax Function

The softmax equation takes as input a set of float values and outputs a new set of values between 0 and 1, and where the values add up to one. Inputs can be any real values of any range. Output from softmax represent probabilities and we will set the output with maximum probability to be the prediction result.

For parallel softmax function, the challenge is to better parallelize sum and max function to free each thread from accessing many parts of global memory in the input of softmax. We also need to smooth the connection between sum and max and store as little temporary data in global memory as possible. Before applying softmax equation, we subtract maximum number in each row for all elements in a row as preprocessing to avoid overflow in GPU since CUDA only takes 32bit float and exponential value in softmax can be very large. Also, since the number of types of image which is size of input can be small for one piece of image, thus we do batch softmax to make good use of the parallel property in GPU.

```
Serial Softmax:
    for each row:
        calculate sum for each row
    for each element:
        divide by sum in its row
    for each row:
        calculate index of largest probability
```

```
Naïve Parallel Softmax:

        for each thread:
            calculate max in its row
            subtract max from each element
        halving valid thread in each iteration:
            do reduction to compute Sum for each row
            store in private memory
```

```
    for each thread:
        divide by Sum for each element
        compute maxidx in its row
    compute max in its row
```

The naive version is to compute max and max index for each thread itself and do sum using reduction, while the optimized version is to do Max, Sum and FindMax all in reduction with memory. The naïve version stores partial result into private memory or registers and access global memory when needed. The optimized version only accesses global memory once and store intermediate result all in shared memory.

```
Shared Parallel Softmax:
    halving valid thread in each iteration:
        do reduction to calculate Max for each row
    store Max in shared memory
    for each thread:
        subtract max from each element
    halving valid thread in each iteration:
        do reduction to compute Sum for each row
    for each thread:
        divide by Sum for each element
    halving valid thread in each iteration:
        do reduction to compute Maxidx for each row
    return Maxindex & map to prediction
```

We consider doing reduction without shared memory at first. However, because we need to do reduction many times for Max and Sum, the intermediate result must be stored in memory, thus we choose to store those results all in shared memory to accelerate computation.

Finally, we do a parallel searching to find index of max element by leveraging reduction and map biggest probability to the prediction of image. For both naïve and optimized version, the block size is number of image types and grid size is the same as batch size. In our case, block size is (10,1) block number is (128,1) for a batch size of 128 and 10 types of image.

3. Results

In this section, we will compare the performance of several optimization methods in convolutional layers, fully connected layers and softmax function.

3.1 Convolutional Layers

In table 1 and table 2, we compared the speed of serial method and 4 parallel methods. There are several properties worth to mention.

First, all parallel methods are overwhelmingly better than serial methods, which proves the value of using GPU in our experiment.

Second, even if storing inputs to shared memory, it consumes almost same time as naive parallel methods. This is because input size is much smaller than filter size in our experiment.

Third, storing input and iteratively storing filter is a good choice in some layers, but may not have well performance in others. As shown in Figure 3, this method is obviously better than the naive parallel method. The reason is the input shape: some layers have a input shape with large height and width and very few layers, while input shape of other layers have small height and width and many layers.

	serial	naive	input in shared memory	input & filter in shared memory	advanced block setting & diminish control divergence
Layer 1	41473.43	18.30	17.23	12.20	6.98
Layer 2	9043.57	27.23	29.64	24.30	8.69
Layer 3	3307.84	15.19	14.75	17.24	8.35
Layer 4	3715.06	19.89	18.91	23.08	11.85
Layer 5	2495.45	13.24	13.43	13.66	9.32
Total	60035.35	93.85	93.96	90.48	45.19

Table 1 Time consumed comparison (in milliseconds) of 4 parallel methods and serial method. In this table, parallel time includes GPU executing time and memory copy time.

	naive	input in shared memory	input & filter in shared memory	advanced block setting & diminish control divergence
speed comparing to serial method	639.70	638.94	663.52	1328.51
speed comparing to naive parallel method	/	1.00	1.04	2.08

Table 2 Parallel methods speed comparing to serial and naive parallel method. We consider memory copy time and GPU execution time together in this table. For example, speed of advanced block setting method is 1328.51 times of serial version and 2.08 times of naive parallel version.

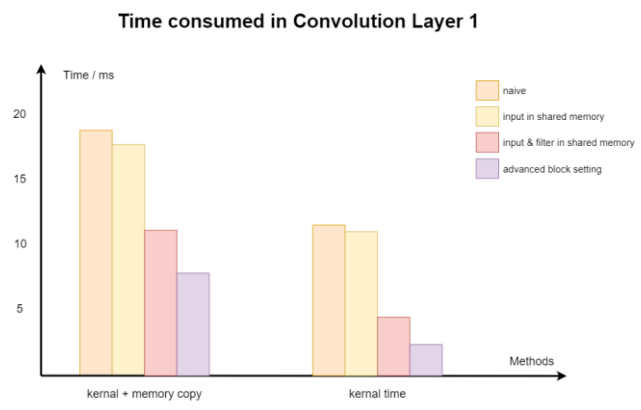


Figure 3 Comparing efficiency of 4 parallel methods in Layer 1

For the former type of input, load filter iteratively into shared memory is worthwhile, because it would be used many times due to large height and width of inputs. An example of this is Layer 1 of ZF-Net, which is shown in figure 3. In this case, the filter can be reused for a decent amount of time, which leads to a significant speedup comparing to naive parallel method.

For the latter type of input, iteratively loading filter into shared memory may not be a good choice: we can only reuse the filter for a couple of times (even just 1 time as in Layer 4), while GPU would spend some time on indexing. As we can see in table 1, iteratively loading filter can be slower than naive parallel method in some layers whose input sizes are 'narrow but thick', e.g. Layer 4.

Fourth, use advanced block setting and diminish control divergence is the best method in our experiment, no matter in which layer. There are three main advantages of this method: the shared memory is enough to store both inputs and filters; we can reuse data in shared memory for many times; almost no thread is idle. Thus, its performance is much better than the other 3 parallel methods.

Table 3 and Table 4 shows the GPU execution time without counting memory copy time. The kernel time shows that advanced block setting method is much better than other parallel methods, while all other 3 methods consume almost the same time. Unfortunately, there may not be a good way to diminishing the memory copy time in our experiment.

	serial	naive	input in shared memory	input & filter in shared memory	advanced block setting & diminish control divergence
Layer 1	41473.43	12.05	11.59	6.76	1.74
Layer 2	9043.57	23.30	25.95	20.64	5.09
Layer 3	3307.84	12.15	11.82	14.30	5.43
Layer 4	3715.06	15.91	15.01	19.20	7.89
Layer 5	2495.45	9.73	10.09	10.35	6.04
Total	60035.35	73.14	74.46	71.25	26.19

Table 3 GPU execution time versus serial time (in milliseconds), without counting memory copy time.

	naive	input in shared memory	input & filter in shared memory	advanced block setting & diminish control divergence
speed comparing to serial method	820.82	806.28	842.60	2292.30
speed comparing to naive parallel method	/	0.98	1.02	2.79

Table 4 Parallel methods speed comparing to serial and naive parallel method. We consider memory copy time and GPU execution time together in this table. For example, speed of advanced block setting method is 1328.51 times of serial version and 2.08 times of naive parallel version.

3.2 Fully Connected Layers & Softmax

First, we analyze the result of experiment of fully connected layers.

Table 5 and Table 6 shows how parallel computation reduce time for 3 fully connected layers. In Table 5 time of parallel method ignores memory copy time while in Table 6 that include memory copy time.

	serial	naive	part of in shared memory	all input in shared memory	shared memory with prefetch
FC1	2875.13	28.93	20.64	16.88	16.72
FC2	1202.82	20.23	13.69	7.61	7.49
FC3	48.36	0.78	0.64	0.53	0.52

Table 5 Time consumed comparison (in milliseconds) of serial method and 4 parallel methods without memory copy time.

	serial	naive	part of in shared memory	all input in shared memory	shared memory with prefetch
FC1	2875.13	220.67	146.35	119.72	110.58
FC2	1202.82	146.22	96.19	87.93	62.33
FC3	48.36	5.83	4.32	4.05	3.91

Table 6 Time consumed comparison (in milliseconds) of serial method and 4 parallel methods including memory copy time.

Intuitively, the time consumed by serial method for fully connected layer is much longer than parallel method, since the input matrix is quite big. The result verifies this assumption and shows a significant reduction even when we include memory copy time in total time consumed by the parallel method. This prove the effectiveness of parallelism in fully connected layer.

For comparison of 4 parallel methods, we can also refer the result shown in Table 7 to see the comparison between result from different optimization. This table gives an average value of ratio between parallel and serial method for 3 FC layers. By comparing half shared method and full shared method we can conclude that we can put as much data as possible into shared memory to reduce execution time. However, since some threads who quickly finish the computing has to wait for others, so the time is wasted and there is no overlap. If we managed to break this synchronization, we can let the threads who finishes computing faster to fetch date earlier without waiting for those threads who are still computing. Thus, a prefetching tiled algorithm can be applied to solve this problem.

We can also find the superiority of last method using shared memory with prefetching strategy which does a more than 12 times quicker computation than serial methods. We ensure the effectiveness and correctness at the same time.

	naive	partial shared memory	input all in shared memory	shared memory with prefetching
speed comparing to serial method	9.18	9.30	11.87	12.88
speed comparing to naive parallel method	/	1.01	1.40	1.29

Table 7 Parallel methods speed comparing to serial and naive parallel method. We consider memory copy time and GPU execution time together in this table. For example, speed of prefetching method is 12.88 times of serial version and 1.29 times of naive parallel version.

Secondly, we analyze the result of experiment of final softmax layer.

	serial	naive	input in shared memory
Softmax	7.05	0.96	0.36

Table 8 Time consumed comparison (in milliseconds) of serial method and 2 parallel methods without memory copy time.

Table 8 and Table 9 show the result of serial method, naïve method and shared memory optimization. Considering only kernel time, the naïve method and optimized method is much quicker than serial time. When counting in memory copy time, we can also see a significant reduction in time. Although the batch size is not quite big, that is 128 sample images each time. If we enlarge the batch size, say 1280, we will see an even better performance of paralyzed softmax versus serial version.

	serial	naive	input in shared memory
Softmax	7.05	2.70	2.28

Table 9 Time consumed comparison (in milliseconds) of serial method and 2 parallel methods including memory copy time.

Figure 4 and Table10 show more clear comparison between different methods. Since the naïve version of parallel softmax also use part of reduction, this method still obtains good result. The Table10 and Figure4 still shows an ideal result although we worried about relatively small size of input for softmax layer compared to previous layer which takes in a large volume of input data.

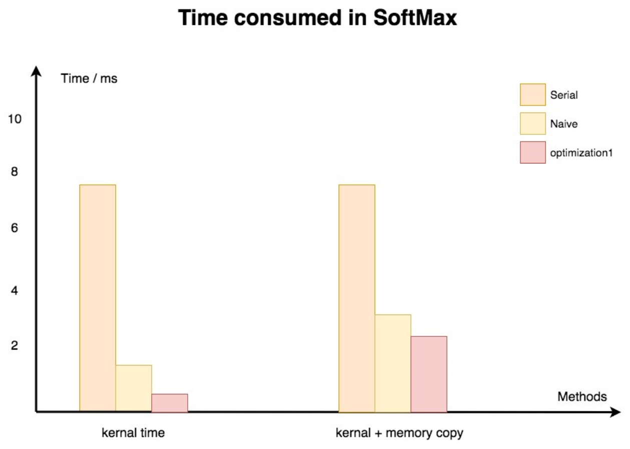


Figure 4 Time consumed in softmax layer

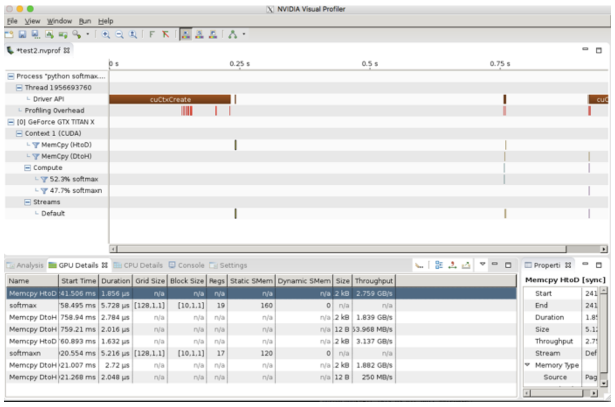
	naive	input in shared memory
speed comparing to serial method	2.10	3.68
speed comparing to naive parallel method	/	1.75

Table 10 Parallel methods speed comparing to serial and naive parallel method. We consider memory copy time and GPU execution time together in this table. For example, speed of advanced block setting method is 3.68 times of serial version and 1.75 times of naive parallel version.

As we can see from Figure 5 profiling result, memory copy process is synchronized.

For native version softmax, throughput and memory copy time of host to device 2.759 GB/s and 1.504us. The throughput and memory copy time of device to host 1.856 GB/s and 2.784us. The block size is (128,1,1) and thread number per block is (10,1,1).

For optimized version softmax, throughput and memory copy time of host to device 3.137 GB/s and 1.822us. The throughput and memory copy time of device to host 1.632 GB/s and 2.72us. The block size is (128,1,1) and thread number per block is (10,1,1). Throughput is affected by of duration and thread number, which means that the more duration, larger the thread number, the larger throughput.



Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD	41.506 ms	1.856 µs	n/a	n/a	n/a	n/a	n/a	2 kB	2.759 GB/s
softmax	58.495 ms	5.728 µs	[128,1,1]	[10,1,1]	19	160	0	n/a	n/a
Memcpy DtoH	758.94 ms	2.784 µs	n/a	n/a	n/a	n/a	n/a	2 kB	1.839 GB/s
Memcpy DtoH	759.21 ms	2.016 µs	n/a	n/a	n/a	n/a	n/a	12 B	53.968 MB/s
Memcpy HtoD	60.893 ms	1.632 µs	n/a	n/a	n/a	n/a	n/a	2 kB	3.137 GB/s
softmaxn	20.554 ms	5.216 µs	[128,1,1]	[10,1,1]	17	120	0	n/a	n/a
Memcpy DtoH	21.007 ms	2.72 µs	n/a	n/a	n/a	n/a	n/a	2 kB	1.882 GB/s
Memcpy DtoH	21.268 ms	2.048 µs	n/a	n/a	n/a	n/a	n/a	12 B	250 MB/s

Memcpy HtoD [sync]

Start	241
End	241
Duration	1.8
Size	5.1
Throughput	2.7
Stream	Def
Memory Type	
Source	Pag

Figure 5 profiling result of softmax layer

By Table10 and Figure 4, Although naïve version utilizes private memory to reduce access time, it still needs to access global memory many times which wastes time. The result shows optimized method is much better than naïve method Therefore, optimized version proves doing multiple reduction and using shared memory can largely reduce time for computation even for a small size of input. You can see that even if we include memory copy time the shared softmax 3.68 times quicker than serial and 1.75 times quicker than naïve version.

3.3 Overall result

	naive parallel	best optimized parallel
serial	620.93	1287.47
naive parallel	/	2.07

Table 11 overall speed comparison with a batch size of 128, both memory copy time and GPU execution time are considered.

We compare the overall performance of our best optimized method, naive parallel method and serial method in this section.

A performance test of whole ZF-Net is given in Table 11: the best optimized version is 107% faster than naive parallel method and 1286 times faster than serial method.

4. Discussion and Further Work

We successfully build the parallel version of forward propagation for general convolutional neural networks. The best optimized version is significantly better than naive parallel version.

There still exists space for improvement. In convolution part, the advanced block setting method improves block setting and diminishes control divergence simultaneously, which boost the algorithm performance. But we can further discuss which factor contributes more to the improvement.

Also, we only implement batch forward propagation on fully-connected layers and softmax function. Adding batch computation for the convolution part might lead to a better performance. To further optimize matrix multiplication in fully connected layer. We can adopt Strassen algorithm which leverages divide and conquer algorithm to reduce the time complexity of this computation. And we can also model iterative version of Strassen algorithm as a reduction problem for thought of divide and conquer.

5. Conclusion

In this project, we finish building parallel forward propagation of general convolutional neural networks. The best optimized version is 107% faster than naive parallel version and 1286 times faster than serial algorithm. In our project, we found that wisely using shared memory can lead a boost to the performance. Also, setting optimized block and grid size and diminishing control divergence are also essential techniques for high performance parallel algorithms. One can further set more control groups to see which factors contribute more to the performance improvement.

6. Acknowledgements

We sincerely appreciate the help from Professor Zoran Kostic, who give us many high-level suggestions to better planning and presenting our project. We also thank to

Teaching Assistants Jeongmin Oh and Tianyao Hua, who provide us useful details to better implement our project.

7. References

[1] Bitbucket: https://jwQAQ@bitbucket.org/4750_zffp/zffp.git

[2] Kirk, David B., and W. Hwu Wen-Mei. Programming massively parallel processors: a hands-on approach. Morgan kaufmann, 2016.

[3] Convolution in Parallel & Matrix Multiplication in Parallel Goodfellow, Ian, et al. Deep learning. Vol. 1. Cambridge: MIT press, 2016.

[4] Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding convolutional networks." European conference on computer vision. Springer, Cham, 2014.

[5] Larry Brown, Accelerate Machine Learning with the cuDNN Deep Neural Network Library, September 7th, https://devblogs.nvidia.com/accelerate-machine-learning-cudnn-deep-neural-network-library/

Individual Percentage Contributions

Task	**Wenqi Jiang**	**Hanzhou Gu**
Overall	**55%**	**45%**
Convolutional layers	√	
Fully-connected layers & softmax		√
Presentation preparation	**60%**	**40%**
Report writing	**50%**	**50%**