Exploration of the CUDA-based performance improvement of a convolutional neural network

Huang Zhengjie - 1308971
Tuo Jintao - 1324918

I. Introduction

A. Goal

This task is to utilize CUDA parallel programming to port the calculations of the convolutional neural network to GPU. As the neural network involves a lot of calculations that are not heavily dependent on each other, it would be more efficient to calculate them in parallel.

B. CUDA parallel programming

NVIDIA, a graphics card manufacturer, introduced CUDA (Compute Unified Device Architecture) to enable GPU to solve complex computing problems.Compared with cpu, GPU has more ALU units. Therefore, if the ALUs of GPU can be used for parallel calculations, the running time of the program will be greatly reduced.
The principle of robot finding kitten is convolution neural network, which classifies and recognizes objects in images by convolution operation and processing of input pictures.Through the observation of the original program, we find that the computation is mainly concentrated in the $BLOB * convolution$ function, which requires a lot of multiplication in the cpu. And this function takes most of the running time of the program. Therefore, our main optimization idea is to port the calculation of this function to GPU through CUDA, and let GPU do the calculation to obtain shorter running time.

C. Test Environment

In order to repeat the experiment and get the same results, it is very important to ensure the same test environment. Especially the maximum number of threads per block allowed by GPU. The test environment for this task is a TU/e server with three GeForce GTX 570 GPUs. The server uses Ubuntu 16.04.5 LTS system, and the maximum number of threads per block allowed by GPU is 1024.The connection to this server was made by MobaXterm.

D. Starting point

In the original program, the codes are written in `C` except the preprocessing part. When the program is executed, it takes about 3.45 seconds[1].

[1]This is only the average and the actual case may vary due to various issues.

TABLE I
Number of different layers in the neural network

Layer	Number	Time taken
CONVOLUTION	54	3.502890s
POOLING	1	0.000134s
ELTWISE	10	0.001173s

II. Analysis

To speed up the program, we first need to analysis what is happening in the program and figure out which parts can be speeded up.
There are three kinds of layers in the neural network which are convolution, pooling, and eltwise separately. And the times that they are executed in one running are shown in Table I. By modifying the timer in the `network.c` file, we can get their total execution time. The time taken by each layer is also shown in the same table.
It is rather clear what we should focus on as the time taken by the convolution layers is significantly larger than the other two layers. Even if we optimize the other two layers to the best, no time required, the improvement is still subtle.
It is not hard to find out that the multiple for loop in `convolution.c` takes the most time than other parts. The general idea is to code the loop into threads and execute them in parallel. However, there are 54 convolution layers. We still need to have a deep understanding of the convolution layers. We can see from Table II that different layers have different loop sizes and more importantly, the size of each loop varies a lot. For instance, the outermost is initially looped once while it will be iterated 960 times at most later which makes it hard for us to schedule the threads in a static way.
Therefore, we need to schedule the threads dynamically depending on the loop sizes.
According to NVIDA's technical specification [1], only 1024 threads are allowed in one block but the limitations on the grid size are relatively looser. So, our initial idea was to put as many threads as possible into a block and fit these blocks into grids to maximize performance.

III. Experiments

In this section, we introduce the experiment that were carried out during our exploration. Some are not working very well, but still provide a foundation for us to improve.

TABLE II
LOOP SIZE OF SOME CONVOLUTION LAYERS

1st	2nd	3rd	4th	5th	6th&7th	total
1	32	3	112	112	9	1204224
1	24	96	56	56	1	7225344
1	144	24	56	56	1	10838016
144	1	1	28	28	9	112896
1	192	32	28	28	1	4816896
384	1	1	14	14	9	75264
1	96	384	14	14	1	7225344
1	160	576	7	7	1	4515840
960	1	1	7	7	9	47040
1	1000	1280	1	1	1	1280000

if $loop4 * loop5 > 1024$ **then**
    grid = dim3(4,4);
    block = dim3(loop4/4, loop5/4);
else
    grid = dim3(1,1);
    block = dim3(loop4, loop5);
end

Fig. 1. Primitive thread scheduling

A. Inner loop parallelism

At first, when we were not quite familiar with CUDA, we decided to start from the easiest part.
We can see that the 4th and 5th layers are relatively small and the product of their loop size is fairly close to 1024 from Table II. And the maximum is $112 * 112$ which can be scheduled if we have more blocks. The detailed way of scheduling is shown in Fig. 1.
However, this didn't go well and the program even became slower. After a deep analysis into the result, we found out that this was not totally useless as the processing time of the first several layers were decreased significantly. And the reason for this is that **the first several layers have larger inner loops** while the other layers have larger outer loops. If we use CUDA to execute one thread at one time, that is not worth is since setting up the environment for CUDA to execute also takes some time.
The issue now is **how to deal with the cases that have larger outer loops**.

if $loop4 * loop5 > 1024$ **then**
    grid = dim3(4,4);
    block = dim3(loop4/4, loop5/4);
else
    grid = dim3(loop3,loop2, loop1);
    block = dim3(loop4, loop5);
end

Fig. 2. Improved thread scheduling

if $loop4 * loop5 > 1024$ **then**
    grid = dim3($4 * Kx^a$,$4 * Ky$);
    block = dim3(loop4/4, loop5/4);
else
    grid = dim3(loop3,loop2, loop1);
    block = dim3(loop4, loop5);
end

[a]Kx refers to the kernel size, so is Ky

Fig. 3. Extend kernel

TABLE III
PROFILING OF TWO KERNELS WITH DIFFERENT ALGORITHMS

Techniques	Conv_1	Conv_2
Entire loop parallelism	49.9ms	49.0ms
Extend kernel	54.3ms	46.8ms
Unroll	38.2ms	20.7ms

B. Entire loop parallelism

To optimize the processing time of the rest of the layers, we have to make sure that the outer loops can also execute in parallel. The pseudo code of the new algorithm is shown in Fig 2. It not only keeps the part that speeds up the processing of the first several layers, but also makes it possible for the layers with larger outer loops to execute in parallel. With this thread scheduling, the innermost 2 loops are executed in the kernel which is reasonable as they are of small sizes like 1 or 3.

C. Kernel extending

As is mentioned in the previous section, the kernel contains a double for loop now whose size is relatively small. It is also possible to extend the kernel to multiple threads. The pseudo code is shown in Fig. 3. With this little change, the indexes of the loop inside the kernel also need to be changed.

D. Unroll

The unroll technique can significantly reduce the number of branches by extending the size of each loop. Since the loop is not large, we can unroll all of it according to the kernel size(1 or 3).

The performance of these three techniques are compared in Table III. The total running time of the program fluctuates due to memory related issues and is not taken into consideration here. But just for reference, the total running time of the unroll technique is around 0.65 to 0.85 seconds. Of the total running time, the convolution part only takes up a small part which is 58.0ms as is shown in Table III.

TABLE IV
GPU PROFILING OF ONE RANDOM RUN

Name	Calls	Avg	Time	Time(%)
Conv_1	16112	2.3370us	37.666ms	46.64%
Conv_2	43	473.46us	20.359ms	25.19%
HtoD	163	96.010us	15.650ms	19.36%
DtoH	55	129.33us	7.1129ms	8.00%
preprocessing	1	27.495us	27.495us	0.03%

IV. FURTHER ANALYSIS

We looked to improve the unroll technique and therefore analyzed the performance of the program with unroll technique. The complete profiling result is shown in Table. IV. It is for sure that there are still some room for improvement but considering the small running time, the improvement can not be significant.

The HtoD(Memcpy Host to Device) and DtoH(Memcpy Device to Host) are inevitable as the inputs for every layer are newly generated. However, there is a possiblity that the output of one layer on the GPU can be reused by its next layer as the input. Due to limited time, we did not try this.

V. CONCLUSION

With the help of GPU, we speed up the running of a convolutional neural network from around 3.5 seconds to 0.7 seconds. This is almost the ceiling as the time taken by the core calculation module is already decreased to a milliseconds level. However, there are indeed room to do some further improvements including utilizing the shared space, use specific thread scheduling depending on the specific layer, etc.

REFERENCES

[1] Programming Guide :: CUDA Toolkit documentation https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlfeatures-and-technical-specifications__technical-specifications-per-compute-capability