ECE408 Applied Parallel Programming Project/Competition

Due Date: December 19, 2016

Wutong Gao (wutongg2)
Sheng Cheng (scheng41)
Chengrui Zhu (cz8)

Introduction

Deep-learning is the advanced machine learning developed high abstraction in data, which allows the machine-learning system to automatically discover the complex features. Comparing to conventional machine-learning system, deep-learning system requires enough data to allow system to automatically discover an adequate number of relevant patterns, which is always insufficient. There is one category of deep learning procedures that are easier to train and that can be generalized much better than others. These deep-learning procedures are based on a particular type of feed forward network called the convolutional neural network (CNN). However, the neural network training methods including CNN required high computer power and large dataset operation.Parallelization turned to play a key role in boosting up operation time and computer performance. In the project work, we particularly studied to accelerate and improve the forward propagation step algorithm based on the CUDA technology we have learned this semester, including shared memory, matrix multiplication, tiled 2D convolution, convolutional layer unrolling, and stream asynchronous.

Purpose

The goal of this project is to accelerate the forward propagation step of the Convolutional Neural Network (CNN) algorithm using GPUs. The sequential implementation provided follows the basic algorithm 16.4 and 16.5 described in book chapter 16. The dataset and model are from the MNIST database.

Implementation

There are many functions that have many sequential for-loops. Initially, we unrolled the entire matrix and assigned it into different streams, and launch the optimized kernels in order. So we first parallelized these sequential functions and implemented them in kernel. Then we optimized each function to get a better performance.

The optimizations we did in our parallel code:

1. Forward convolution function optimization

   For the forward convolutional function, we transformed the convolution into matrix multiplication by first unrolling input, then performing tiled matrix multiplication and finally re-rolling output. After unrolling, we got two matrices, which are convolution filters and input features so that it becomes to be possible for us to use shared memory to implement tiled matrix-matrix multiplication kernel from MP3 because we reduced the convolutional layers. This algorithm idea is efficient because it has a high ratio of floating-point operations per byte of global memory data access.

2. Matrix multiplication with relu functions

   In the serial code, each image dataset called two different functions to do the matrix multiplication process and rectified linear unit function. Now, for reducing the time complexity, we combined the

relu functions into the matrix multiplication, so it saves time to transform the data. For the previous version, relu function and matrix multiplication are separate. It takes time to transfer data from relu function to matrix multiplication. Thus, we just combine the relu into the matrix multiplication so that we can cut down transformation time to process in one time.

3. Coalesced memory

   In each of our kernel, we tried to make the memory access coalesced in order to obtain better memory throughput. We know that memory accessed coalesced can be read faster by each thread. By implement each thread with consecutive memory, the time used to access memory can be cut down.

4. All kernels in one host function

   Usually, when we launched one kernel function every time, we need to copy from Host to Device, which might also affect the running time. Therefore, in this case, we need to repeat the memory copy for at least 5 times. We combined all kernels in one host function. In this way, we just need one Memory copy from Host to Device and from Device to Host.

5. Adding loop over samples in the batch

   Since for each test case, the batch size is at least more than 1, we add loop over samples in the batch to improve the overall test performance. During each iteration, we convert the simple input feature map from its original form into the expanded matrix. Instead of using iteration for sampling, we regard the batch size as a dimension and add it to each kernel.

6. Asynchronized cuda stream

   We tried cuda stream for convolution part with better overlap. There are 5 functions in convolution part, including two memory copies and three kernels. So, at least 5 streams are needed for a reasonable suitable overlap. Meanwhile, we use 8 streams in convolution part. When we use asynchronized cuda streams, we can transfer and compute adjacent segment at the same time. Thus plenty of time will be saved compared to serialized data transfer and GPU computation.

7. Multi-GPU

   To solve the problem of a time consuming kernel, multi-GPU improves the performance a lot. In one of our history code, it has 2x speedup when we used the multi-GPU in one of the kernel.

8. cuBlas with stream

   To get a faster matrix multiplication, we use cuBlas library instead of using more advanced algorithm. Besides, using stream and small batch to accelerate the code. The advanced algorithm is faster in theory but the cuda library code is faster because of more optimization

for GPU architecture.

- ## Results

There are three kernels for convolution part and one kernel for average pool. Also, we have one kernel for fully forward function (actually is matrix multiplication). Each kernel are used twice. We test the execution time separately for each part. (Tested on full data set with batch size of 10,000)

1) Data transfer
   5.16ms for copy all the data from host device, including the input images and all filters.The data transferred from device to host needs 0.11ms.
2) Unrolling kernel
   The unrolling kernel for input in first convolution takes 28.85ms and reunroll takes 89.56ms. The second input unroll takes 139.48ms, while reunroll takes 17.42ms.
3) Matrix multiplication kernel takes 57.3ms and 57.7ms. Before optimization the matrix multiplication, it takes longest time in each convolution part. However, after using the cuBlas Library, we improve a lot in this part.
4) Average pool kernel
   The first average pool uses 51.20ms. The second average pool takes 9.53ms.
5) Fully forward kernel
   The fully connected layer take 61.27ms and 2.03ms.
6) Relu kernel
   The relu kernel takes 0.12ms

The overall running time for our code on the full data set is

- ## Analysis

We analyze the time from 3 parts. The reference code contains three part. The first two parts are the combination of convolutional layer and average pool. The third part is combination of fully connected layer. The reference time is 2.6s.

1. When we combine all parts together. The time decreases about 100ms, nearly 2.5s. This time is saved from the transferring between host and device.
2. When we add stream first two layer. It's nearly no improvement for time. The reason might be that the memory copy part takes much less time compared to time consuming kernel in convolution layer. So the stream is meaningless because all the functions should wait the longest one to execute. It can clearly shows in NVVP results.
3. When we didn't use the batch size as a dimension for kernel, instead we use loop for each samples. The time increases about 6s. It means paralleling the batch size has great improvement for timing.

According to several test for our code, the main problem limits the time coming from a large kernel in convolution part. It will be analyzed in

next part. We found different ways to solve the problem. The only one make sense by using the multi-GPUs.

● Challenges and Solution

The challenge of our project comes from 3 parts.

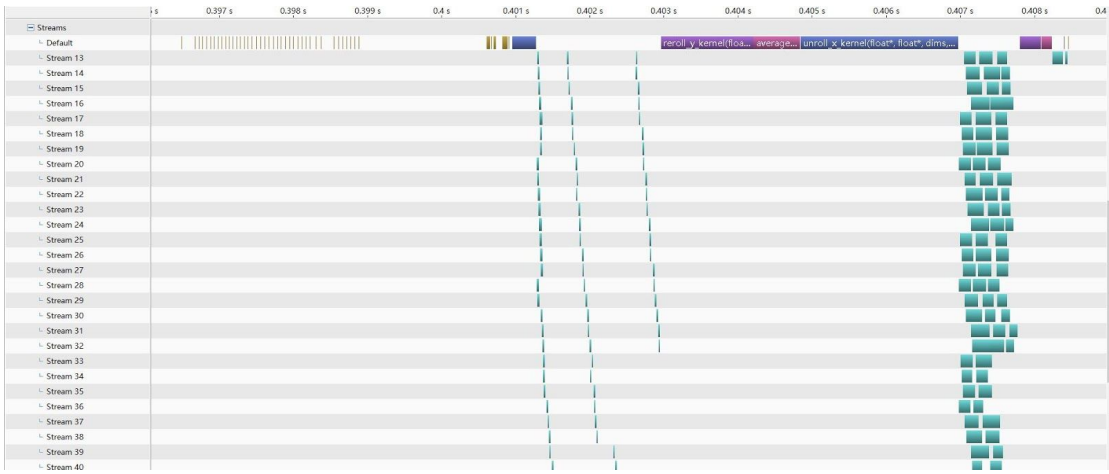
1. The cuda stream doesn't work in the way as we think. It's known that the execution time for each stage is determined by the most time consuming part of the code. In our convolution part, the matrix multiplication takes 10 times running time than other functions do. In other word, in the pipeline of cuda stream, it seems a sequential execution of matrix multiplication. So there is less improvement using stream in convolution part. For the future work, we will try to optimize the matrix multiplication and apparelize it based on some future research study.
2. From point 1, it's easy to get a goal to optimize the matrix multiplication. However, that is the biggest challenge we meet. The main problem is the input for multiplication is so big. For first convolution part, the input size is (batch size * 25 * 24 * 24). The second convolution part, the input size is (batch size * 25* 32 * 8 * 8). Even if we use share memory, it's hard to guarantee the memory could contain the samples.
3. To avoid the large matrix multiplication, we tried several more efficient ways. One solution is Winograd kernel for convolution. That means we don't need unroll. But it has a problem of memory uncollapsed, which is the same as the traditional convolution. And it also difficult to implement in code. Another solution is Strassen algorithm. According to mathematics, it has better performance than traditional matrix multiplication because it has less multiplication operation. However, when analyzing it in computer, it's an other story. This method needs to allocate more space to store the temporary value. We will study more about these two novel algorithms and see if we can hybridize and take the advantage for both way to get a better time performance.
4. The most efficient way we use is cuBlas library. It shows greater performance owing to its better optimization for Nvidia GPU architecture. It has 3x speedup for whole code and

● Conclusion

In this project/competition, our group does series of optimization after parallelizing the serial code, including forward convolution function optimization, combining relu function into matrix multiplication, making the memory access coalesced, all kernels in one host function, adding loop over samples in the batch and asynchronized cuda stream. After these series of optimization, the performance improved a lot. Future study is needed to make a further improvement.

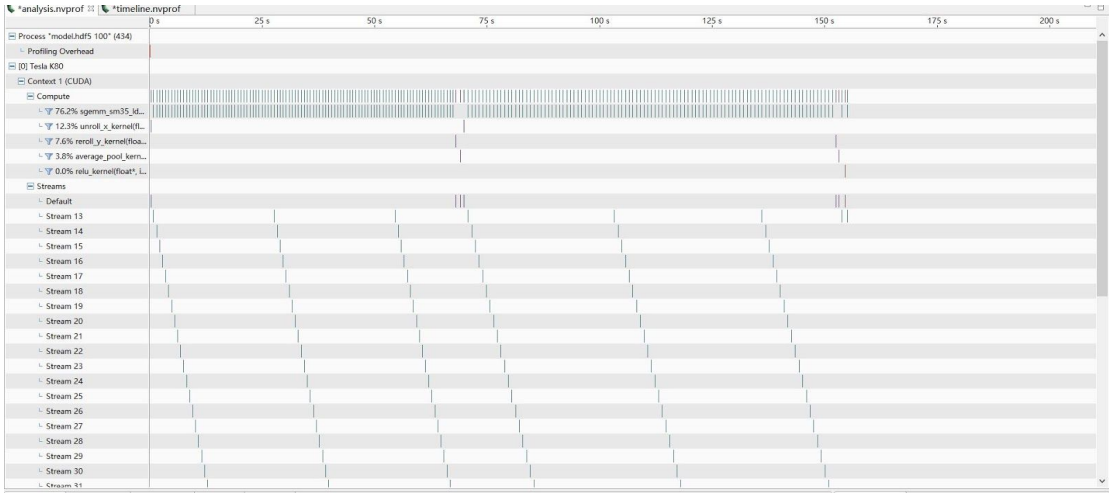
- **NVProf and NVVP Results**

The whole timeline figure



The matrix stream figure

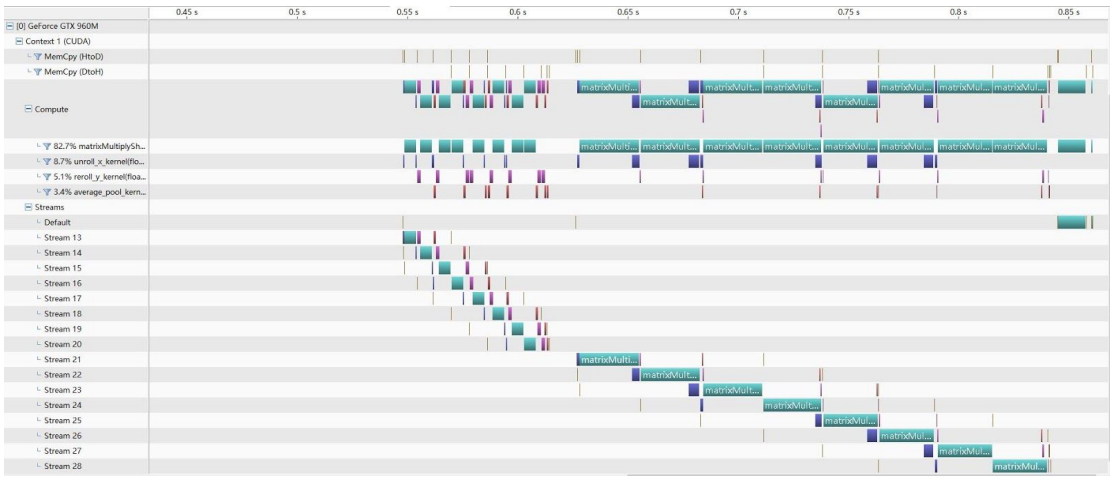
More detail on matrix multiplication stream figure



The analysis figure for code

Timeline without matrix multiplication optimization

Analysis without matrix multiplication optimization



Timeline without matrix multiplication optimization with stream

- **References**

Kirk, D. B., & Wen-mei, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.

CUDA C/C++ Streams and Concurrency. Steven Rennich

CUDA Streams Justin Luitjens

CUDA Toolkit Documentation.

Wikipedia - Strassen algorithm, Winograd kernel