Implementing Lenet5 by using CUDA extension

2014190150

Jung Woo Lee (mgp10067)

1. Introduction

The critical problem of initial multilayered neural network was less flexibility toward topology of data. Even the date set with little change, the model consider data as all different data to train. Since the raw fully connected multi layered neural network did not consider the topology but only consider raw data, it requires extreme numbers of training data and high time cost to train and test.

LeNet is a model that first introduced Convolutional Neural Network(CNN), which solved above problem of fully connected neural network. The model includes convolution process that use kernel filter to extract feature maps. Also, by using max-pooling to subsample feature maps for greatest correlation. As a result, by adding convolution and pooling process before training fully connected network, model can be trained with smaller data set with more flexibility. Since the model uses features of data to train model, it can consider topology without new data set to test another data set.

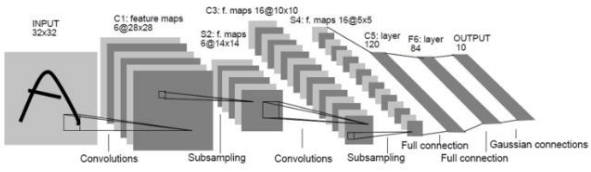

Figure 1. LeNet5 Architecture

Although LeNet5 uses smaller size data set, the training time and testing time still costs high when the data is more complex. To reduce training time of model, general purposed GPU technique is applied. Not only for LeNet, but also for every machine learning models, gpGPU skill greatly reduced time to both train and testing.

The GPU has pretty simple architecture, containing a lot of ALU (or cores) that can run operations in parallel. Therefore, by calculating training process in optimized parallel operations, the time cost to perform machine learning will be greatly decreased.

In this report, I will introduce the characteristics of GPU programming (CUDA), implementation of LeNet5 by using CUDA, and optimization to speed up testing performance.

2. Implementation of CUDA

To run GPU as a general purpose, user must allocate certain memory space of GPU and copy data from main memory to the GPU memory. Since CPU and GPU do not share their memory, data must be copied from host to device and then, after all calculation, result must be copied back to host from device. The data transfer is one of the greatest bottle neck of gpGPU processing and thus, user should not frequently call data from CPU in the Cuda kernel function but copy before calculation.

```
cudaMalloc((void**)&d_output, sizeof(double) * batch * output_size);

// Copy Parameters

cudaMemcpy(d_conv1_weight, conv1_weight,
           sizeof(double) * conv1_in_channel * conv1_out_channel *
               conv1_kernel_size * conv1_kernel_size,
           cudaMemcpyHostToDevice);
```

Figure 2. Prepare device memory

A. *Normalization*

Normalization of input data increases final accuracy after the training. I implemented normalize function with global and applied same as cpu version with mean 0.5 and var 0.5. The difference is an use of dim3 variables to allocate block size and grid size to GPU. I used 2d grid with input channel and batch, and 2d block with input size and input size to fit input data. By running the function, each thread will normalize input values in parallel instead of using naïve iteration.

B. *Convolution*

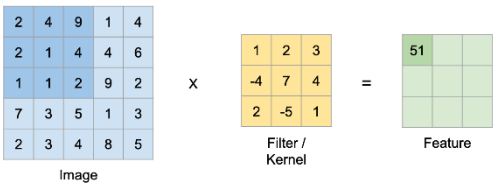

Figure 3. Convolution technique example

In the convolution process, I applied grid dimension as conv1 output channel x batch number and block dimension as input size-(kernel size -1). Operations in the function is same as CPU version, for instance in convolution 1, 5x5 size kernel iterate the 32x32 input to extract 28x28 feature maps. First 4 loops were replaced by each parallel thread while, other 3 to calculate feature map still remains.

The key difference is that in CPU version code, the calculated value with kernel, continually added to output channel data directly while in CUDA version, I used temporal variable to sum up all calculation and applied to output channel at last to minimize direct

memory contact.

C. *Relu*

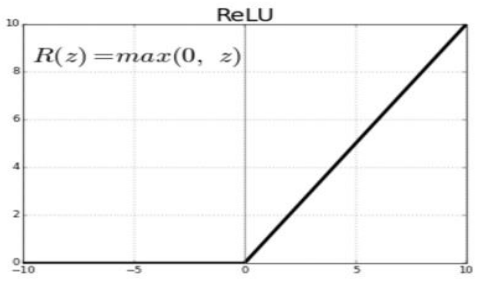

Figure 4. Relu Function

Relu Function is an activation function to emphasize the features in feature map. Since after kernel calculation, many features are extracted. To Increase accuracy, convert all the negative value to 0 will allow model to only consider features with positive correlations. Also, by replacing negative floating point to zero, the calculation speed will also increase. Since the size of data does not change, I used original batch variable and previous layer channel size as a grid dimension and size of each feature map data as block dimension.

D. *Pooling*

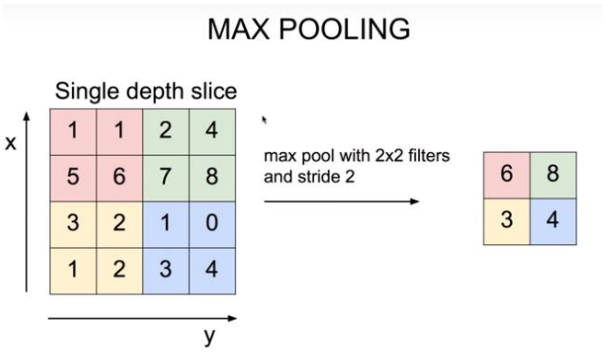

Figure 5. Pooling Example

In the pooling layer, the size of data from previous layer will be decreased into half. By reducing the size of each feature map, the calculation speed of fully connected layer calculation will be increased. Also, by emphasizing the maximum number in 2d kernel, feature to train still remains. The number of data does not change but the size of each feature data will be decreased after the pooling layer.

In CUDA implementation, I applied half size of feature map as block dimension, and thus, the thread index must be multiplied by 2 when thread calculate input base to compare max value. Other operations such as kernel loops to find maximum value is same as cpu version. Only iteration variables are changed to block index and thread index for each thread. Only maximum value will remain as output.

E. *Fully connected Layer*

At last, 3 layers of fully connected layer will be performed to train model with extracted feature maps. CPU version use 3 nested loops to calculate output with pre trained and input from previous layer.

In the CUDA implementation, the block size is output channel size with batch number of blocks. Since batch and output channel become the number to identify thread, 2 nested loops can be deleted. Also, similar to convolutional layer, I used extra variable to minimize direct calculation to output data memory.

After 3 fully connected layers and 2 activation function, relu, the result output is predicted.

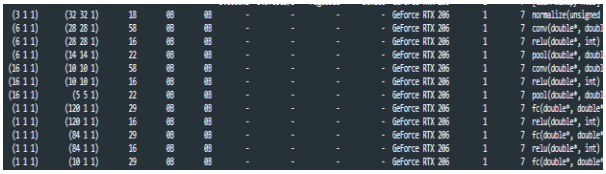

Figure 6. Grid size and Block size for each function

After each layer, CUDA must sync the threads before moving onto next layer by using cudaDeviceSynchronize()

3. Optimizations (Speed Up)

The naïve version of CUDA lenet implementation resulted about 1.6 ms to predict all the images with 128 batches. To reduce prediction time, I tried experiments and found several changes that can speed up the CUDA program.

A. *Compile Options*

Since Nvidea comertialized various versions of GPU, there are many different architectures of GPU. However, CUDA works as same for each GPU. Therefore, there is a compile option to optimize to complie cuda program for best fit to user's own model. By providing architecture code, the nvcc compiler will automatically optimize the program to GeForce RTX 206 of server.

Also, the CUDA supports fast math operations for faster calculation. This option runs cuda functions same as __function() which can increase the performance speed while accuracy of calculation might decrease.

```
$(BUILD)/LeNet5_cuda.o: $(SRC)/LeNet5_cuda.cu $(BUILD)/LeNet5.o
	$(NVCC) $(FLAGS) $(LIBS) -use_fast_math -gencode=arch=compute_75,code=sm_75 -o $@ -c $<
```

Figure 7. Added Compile Option

Implementation of both compile option reduced predict speed to about 1.55 ms.

B. *Fmaf function*

Similar to the fast math compile option, there is a functions for faster calculation. Among the functions, there is a fmaf(x,y,z) which calculate x*y +z in one operation. The function returns float type. Since there

were many calculations of multiply and addition to calculate indexs, I applied fmaf functions and converted types if int is necessary.

```
int output_index = blockIdx.y * (gridDim.x * blockDim.y * blockDim.x)
                 + blockIdx.x * (blockDim.y * blockDim.x)
                 + (int)fmaf(threadIdx.y,blockDim.x,threadIdx.x);
double tmp = bias[blockIdx.x];
```

Figure 8. Apply fmaf function in conv function

I applied the fmaf function to conv function and fully connected functions to reduce calculation times. As a result, the predict speed reduced from 1.55 to about 1.49ms.

C. Unroll internal loops

One of the bottle neck of CUDA version is internal loops in the cuda functions. In the conv, pool, and fc function, there are internal loop to calculate output from each layer. Although each thread in the block runs in parallel, the internal loop runs as sequential way. To minimize the bottle neck due to internal sequential loop, I applied #pragma unroll above the for loop. The command will unroll the loop with branch prediction.

```
#pragma unroll
for(int ic = 0; ic < IC; ic++){
  tmp += weight[(int)fmaf(threadIdx.x, IC, ic)] * input[(int)fmaf(blockIdx.x,IC,ic)];
}
```

Figure 9. #pragma unroll in fc function

As a result, I was able to reduce the speed of prediction from 1.49ms to about 1.47ms.

4. Further Analysis and Failures

By using gpu trace shell command, I was able to check the runtime of each kernel functions and the size of grids and blocks.

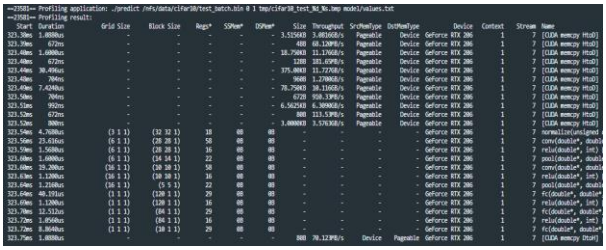

Figure 10. GPU trace of lenet5_cuda.cu with 1 batch

According to the figure 10, conv, fc layer functions and memory copy cause most delays. First and second convolution took 23.6us and 19.2 us respectively. Also, the first fully connected layer ran for 40.2us which is the largest time consuming. The reason of fc function latency is an internal 1d loops that calculate 120 feature maps in 1 grid.

Therefore, I tried to change grid or block size to optimize wrap size and to perform parallelism. However, change of grid and block dimension caused errors in prediction because of different index calculation.

The next try was using shared memory to save weight value so only one load can be performed instead of each thread load specific weight data. However, shared memory required the constant number to allocate size, and I tried to split fc function to fc1, fc2, fc3. But another problem occurred. Since the fc1 weight has 400 size in 1d, too much shared memory was necessary.

After the failure of shared memory, I imagined constructing device function that can run concurrently to replace internal loop. However, providing other thread resources from each thread was not possible and this attempt caused error.

Another bottle neck is due to memcpy from host to device. The latency is due to low speed of communication between host memory and device memory. Therefore, I approached to use memcpy in asynchronous manner by using different streams.

To prevent errors from prediction, the last memcpy is occurred to synchronize others. In my first guess, concurrent running of memcpy functions will reduce the total time of running prepare function. However, time duration of each memcpy increased as figure 11 shows and thus could not solve the memcpy bottle neck.

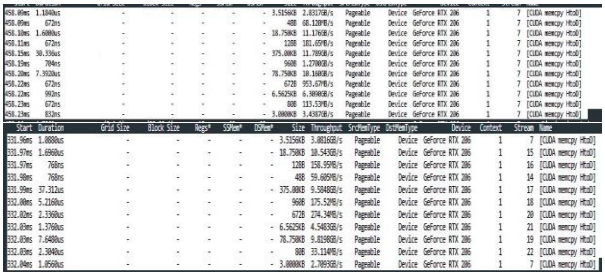

Figure 11. Comparison between memcpy and MemcpyAsync()

According to reference [3], calculation of converted number from 32-bit floating point to 16-bit fixed point has only small change. Thus, if there was no limitation in conversion of precision, data conversion of input can increase the speed performance.

Furthermore, GPU is capable of calculating with float data type than that of double. Therefore, if we convert all the double data after copied from CPU and perform lenet5 model with float type data, it will also result performance increase. However, when I applied type casting in each function, it resulted overhead due to conversion from double to float and then float to double for all thread operations.

5. Reference

[1] *CUDA Toolkit Documentation,2007-2020 Nvidia Corporation. https://docs.nvidia.com/cuda/*

[2] *Yann LeCun, Leon Bottou, Gradient-Based Learning Applied to Document Recognition*

[3] *Tianshi Chen, Zidong Du, DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning.*