CS 217: GPU Architecture and Parallel Programming

FINAL PROJECT REPORT

Backpropagation

By:

1. Sachin Shinde (861243251)
2. Tanaya Vadgave (861242264)

Introduction

In this project we are parallelizing sequential version multi-layer neural network using CUDA. The neural network has 3 layers (input, hidden and output). Input layer size is variable (16777216 default), hidden layer size is 16 and output layer size is 1. The network is trained using backpropagation learning method.

We are not only parallelizing sequential version but optimizing GPU kernel execution to achieve speed.

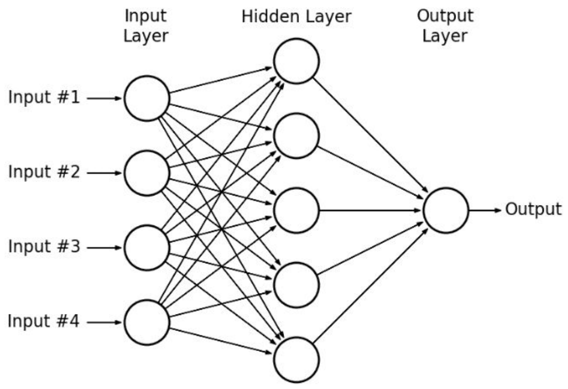

Fig. 1: A Simple Neural Network

Note:

*The results in this report were obtained by running the code on **Bender** server provided by UCR with configuration: 2 Six Core Intel E5-2620 v3 CPUS with 64GB RAM; GPU: NVIDIA Corporation GF110GL [Tesla C2050 / C2075].*

Sequential Version Results:

Input Size	Time Taken for learning
12000000	4.1862s
16777216	5.8591s
18000000	6.2920s

Table 1: Training Results; Sequential

Optimizations:

Optimization 1:

Intuition:

To have one kernel implementation which performs all the computations.

Changes in the code:

We started by making just one kernel which performed all the operation that is layer forward and the error computation and backward propagation.

Difficulties:

It became clear to us that this implementation won't be efficient and might actually increase the total training time since, very different computations are involved in the steps for training. The method was abandoned for a much modular approach.

Total time spent:

4 hours (approx.)

Optimization 2:

Intuition:

To have 4 kernels which perform the 6 needed steps for our training.

Changes in the code:

We started this by making 4 kernels for the 6 functions, since first two function and last two functions were performing the same computation, this seemed like the best way to go.

Following functions/kernels were added to the code:

1. ***void launch_layerforward(float *l1, float *l2, float *conn, int n1, int n2)*** *:* This function was used to define the grid and block size and call the ***layerforward*** kernel.
2. ***__global__ void layerforward(float *l1, float *l2, float *conn, int n1, int n2)*** *:* This kernel was added as a replacement for the layerforward function.
3. ***void launch_output_error(float *delta, float *target, float *output, int nj, float *err)*** *:* This function was used to define the grid and block size and call the ***output_error*** kernel.
4. ***__global__ void output_error(float *delta, float *target, float *output, int nj, float *err)*** *:* This kernel performs the same computations as output_error function.
5. ***void launch_hidden_error(float *delta_h, int nh, float *delta_o, int no, float *who, float *hidden, float *err)*** *:* This function was used to define the grid and block size and call the ***hidden_error*** kernel.
6. ***__global__ void hidden_error(float *delta_h, int nh, float *delta_o, int no, float *who, float *hidden, float *err)*** *:* This kernel performs the same computations as hidden_error function.

7. ***void launch_adjust_weights(float *delta, int ndelta, float *ly, int nly,float *w, float *oldw)*** *:* This function was used to define the grid and block size and call the ***adjust_error*** kernel.
8. ***__global__ void adjust_weights(float *delta, int ndelta, float *ly, int nly,float *w, float *oldw)*** *:* This kernel performs the same computations as adjust_weights function.

Difficulties:

Main difficulties involved in development of this process was converting the code to use the row major format. After the optimization of code, to use the row major format and using 1D arrays instead of 2D, the challenge was to understand exactly how the computations are performed so as to use the GPU most efficiently. This also involved using the ***atmoicAdd*** for performing the addition in ***output_error*** and ***hidden_error*** kernels. This method took a lot of time, since we had to reconfigure and make changes to the project so we can run both the sequential version of the code and GPU version and compare the results.

This code also involved the most thinking as we had to move towards the GPU implementation one step at a time which involved testing the code for errors after each step and since, the device variables and host variables cannot be used interchangeably, it did involve a lot of ***memCpy*** operations after every kernel implementation.

Total time spent:

10 hours (approx.)

Comments:

This method was successfully implemented and showed good results, this code was optimized as well as sped up the performance.

Results:

Input Size	Time Taken for learning	Accuracy
12000000	3.4039s	Same as sequential version
16777216	4.7592s	Same as sequential version
18000000	5.1062s	Same as sequential version

Table 2: Training Results; GPU

The above changes were made by keeping the **BLOCK_SIZE** as **8**. We started to wonder how the block size would affect the results and played around with them a bit, the following table shows the results we got by varying the block size.

Block Size	Time Taken for learning	Accuracy
1	4.1290s	Same as sequential version
2	4.0983s	Same as sequential version
4	4.4002	Same as sequential version
8	4.7592s	Same as sequential version
16	5.0041s	Same as sequential version
32	5.2552s	Same as sequential version
64	5.2513s	Same as sequential version
128	5.2516s	Same as sequential version
256	5.2501s	Same as sequential version

Table 3: Training Results by varying Block Size; Input Size: 16777216; GPU

Optimization 3:

Intuition:

We compared the method of convolution and forward layer and found them to be similar.

Changes in the code:

We noticed that the convolution algorithm works in a very similar way in which the forward propagation works. The weights can be considered as masks and applied on the input units.

We used the kernel implementation taught in the class, in ***Module 8, Stencil***. The layer_forward kernel was modified to use the convolution method. We tried to use this to reduce the number of burst calls and increase the memory coalescing.

Difficulties:

A lot of time was spent on understanding the process by which we can modulate the convolution and use in the current code. We tried modifying the forward layer code but we mostly got different values, we couldn't figure out why the results were the same, my assumption is we were messing up the indexes somewhere which resulted in wrong memory space access and thus garbage values were changing the results. We couldn't spend any more time on this and decided to stay put with the results we had.

Total time spent:

2-3 hours (approx.)

Summary:

The 4 kernel optimization was used, which performed 6 steps required to perform the training. Few functions were added to both Sequential and GPU code for measuring the time. Also, the sequential code was modified to use Row Major Format as well for arrays, so as to have the fair comparison between the two implementations.

We were successfully able to optimize the code and make it run faster than a sequential code.

Running instructions:

We have written python script to run both sequential and CUDA implementation. It compares results of out.txt as well.

To Run type in command line when in root folder of the project:

./run