CS 217 Project Report

Parallelizing Serial C Code With CUDA

Team Member

Huy Tran
- SID: 862465460
- NetID: qtran050

Tsung Wei Liu
- SID: 862393596
- NetID: tliu205

Yin Yu Chang
- SID: 862396036
- NetID: ychan240

Project Overview

Purpose of This Project

The idea of this project is to parallelize the following program (Source: https://courses.cs.washington.edu/courses/cse599/01wi/admin/Assignments/bpn.html). The program is an implementation of a Backpropagation Neural Network in serial C code. This network is designed for time-series forecasting, specifically to predict the annual number of sunspots. The goal is to speedup this program as much as we can using the parallelization knowledge and techniques learned in class.

What Is A Backpropagation Network?

Backpropagation network is a multilayer perceptron network that uses the backpropagation algorithm for learning. There are three layers in the backpropagation network: input layer, hidden layer, and output layer.

Training a backpropagation network includes a forward pass and a backward pass in each iteration. In the forward pass, input data is fed into the input layer and propagated through the hidden layers by neurons. Finally, the final output is generated at the output layer.

In the backward pass, first we calculate the error. The error is calculated by a loss function like Mean Square Error by comparing the output we get in the forward pass with the true value.

Then, the error is propagated back through the network from the output layer to the input layer. Secondly, we calculate the gradient (partial derivatives of weight) using chain rule. The gradient shows how to adjust the weights to minimize the error. Lastly, we update the weights.

How Is The GPU Used To Accelerate The Application?

In this project, the GPU is used to concurrently execute the program. For example, if we would like to do matrix multiplication, it can be executed concurrently and simultaneously to compute the value. Therefore, the program would benefit from that.

The parallel algorithms used in this project are matrix addition, matrix multiplication, and parallel for loop. The parallelization is done in CUDA and OpenMP. The source code was divided into smaller pieces of code and put in separate files. We then attempted to parallelize the functions that we thought could be parallelized.

The libraries that we used for this project were stdlib.h, stdio.h, math.h, sys/time.h and omp.h

- The first of three comes from the original code.
- sys/time.h is used to record the execution time.
- omp.h is used for OpenMP, to gain speedup on the CPU.

This section is just an overview of the implementation. More details on this is included below in the “Implementation Details” section.

Implementation Details

First, we divided the program into smaller files, and each file contains a module from the program to make it easier to parallelize and divide up the work between team members. The list of files are:

- support.h - declarations of libraries, variables, and structs used in the program
- random.cu - randoms drawn from distributions module
- app.cu - application-specific code module
- init.cu - initialization module
- weight.cu - support for stopped training module
- propagate.cu - propagating signals module
- back.cu - backpropagating errors module
- sim.cu - simulating the net module
- main.cu - main function to run the program

For parallelization, we used a couple of different algorithms, including matrix addition, matrix multiplication, and parallel for loop. For each function that could be parallelized, we first wrote

the code for the CUDA kernel, then modified that function, turning it into the host code to invoke the kernel. In the host code functions, we followed the same pattern.

First, we initialized the host variables, and allocated memory for the device variables on the GPU using cudaMalloc(). We then copied the host variables to device using cudaMemcpy() and cudaMemset(). After that, we set the grid dimension and block dimension to an appropriate number of threads and launched the kernel. Then, we copied the device variables back to host using cudaMemcpy(). Finally, we freed the memory using cudaFree(). We also added error handling for these CUDA API function calls by using a cudaError_t type variable. In the main() function, we added a timer to record the execution time of the program.

We attempted to parallelize the following functions:

- SetInput() - parallel for loop
- GetOutput() - parallel for loop
- ComputeOutputError() - parallel for loop. Also used a modified version of atomicAdd() to handle double variables
- PropagateLayer() - matrix multiplication
- BackpropagateLayer() - matrix multiplication
- AdjustWeights() - matrix multiplication

For this program, we chose to use 256 as our block size. This means that there are 256 threads in a block. The reason why we chose 256 is because 256 is a multiple of 32 and each warp has 32 threads. This ensures that there would be no idle threads during execution. Also, we chose 256 instead of another multiple of 32 (such as 128, 512, 1024, etc.) because we did some research and found that 256 threads is usually a good balance between maximizing utilization and avoiding resource contention.

During our parallelization with CUDA, some parts worked, and some parts did not. The functions that we were able to parallelize with CUDA are: SetInput(), GetOutput(), and ComputeOutputError(). The ones that did not work include PropagateLayer(), BackpropagateLayer(), and AdjustWeights(). We were not completely sure why the parallelization did not work for those functions. One reason might have been that those functions have complicated indexing. For example, PropagateLayer() has a nested for loop, and each array indexing is different. For instance, upper[i][j] * lower[i]. Moreover, in the original code the initial value of i is 1 and the initial value of j is 0. Maybe we did not handle the indexing properly and hence the parallelization did not work.

When we tried to run the program with the functions that parallelization did not work, the program kept running indefinitely, because weight in the output could not be changed, and the program could not finish. Below are a few screenshots of the output when the program failed.

```
NMSE is 0.424 on Training Set and 0.438 on Test Set
NMSE is 0.446 on Training Set and 0.435 on Test Set
NMSE is 0.424 on Training Set and 0.440 on Test Set
NMSE is 0.424 on Training Set and 0.441 on Test Set
NMSE is 0.425 on Training Set and 0.435 on Test Set
NMSE is 0.425 on Training Set and 0.436 on Test Set
NMSE is 0.424 on Training Set and 0.439 on Test Set
```

```
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
NMSE is 1.005 on Training Set and 0.961 on Test Set
```

Since some of the functions we attempted to parallelize with CUDA did not work, we decided to use OpenMP. OpenMP is a library that supports parallelizing a program. Basically, it utilizes the functionality of threads to speed up the execution time. However, it only parallelizes the program executed on the CPU. We added the OpenMP functionality by adding the omp.h library in the support.h file. To parallelize with OpenMP, we simply add “#pragma omp parallel for” before each for loop. If there is a second nested for loop, then we add “#pragma omp parallel” before that nested for loop.

How To Run The Code

- Ideally, after it finishes the execution of the program, it will generate the BPN.txt file. And, this file contains the process and result of the backpropagation network.
- For this project, we do not need to type input by ourselves. It is already in the “app.cu” file. Therefore, we can run the program directly.
- To clean the execution file and object file use “make clean” to do that.
- To compile the code use “make” to compile it.
- To run the program use “./nn” to execute it.
- To get the profiling data, we can use “gprof nn >> report.txt”
- These commands are also documented on README.md.

Evaluation/Results

- Execution time with C code: 4.81s

- Execution time with CUDA (with GetOutput(), SetInput(), ComputeOutputError() parallelized): 1736s
- Execution time with OpenMP (with BackpropagateLayer(), PropagateLayer(), AdjustWeight() parallelized): 1.63s
- Execution time of both CUDA and OpenMP combined: 860.9s

When we leveraged the OpenMP functionalities to speed up the execution, it did make the program run faster. When comparing the program in C code and OpenMP, the execution time decreased by 66% (4.81s vs 1.63 s). When comparing the program with parallelization in CUDA and OpenMP, the execution time decreased by 50% (1736 s vs 860.9 s). The parallelization in CUDA should have made the program run faster than C code. However, our program was run on the Bender server. There are only 4 GPUs in the server and there are a lot of students trying to run their program at the same time as well. Therefore, the server was overloaded and slowed down our program's execution time.

Profiling of The Program

The screenshot of the original code

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 41.94      0.83     0.83                             AdjustWeights(NET*)
 17.18      1.17     0.34   222680     1.53     1.53  SetInput(NET*, double*)
 16.68      1.50     0.33   222680     1.48     1.48  ComputeOutputError(NET*, double*)
 13.64      1.77     0.27   222680     1.21     1.21  GetOutput(NET*, double*)
  9.60      1.96     0.19                             TestNet(NET*)
  1.01      1.98     0.02                             RandomWeights(NET*)
  0.00      1.98     0.00        1     0.00     0.00  NormalizeSunspots()
  0.00      1.98     0.00        1     0.00     0.00  __sti____cudaRegisterAll()
```

In the flat profile of the original code, we can see that the most time consuming part is the AdjustWeights(Net*). It accounts for 41.94% of the total execution time. SetInput(NET*, double*), ComputeOutputError(Net*, double*), and GetOutput(NET*, double*) also consumed a significant amount of time 17.18, 16.68, and 13.64 respectively.

The screenshot of the parallelized program

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 47.83      0.76     0.76                             AdjustWeights(NET*)
 15.73      1.01     0.25   168170     1.49     1.49  GetOutput(NET*, double*)
 15.11      1.25     0.24   168170     1.43     1.43  ComputeOutputError(NET*, double*)
 12.59      1.45     0.20                             TestNet(NET*)
  7.55      1.57     0.12   168170     0.71     0.71  SetInput(NET*, double*)
  1.26      1.59     0.02                             RandomWeights(NET*)
  0.00      1.59     0.00        1     0.00     0.00  __sti____cudaRegisterAll()
```

Since we parallelized SetInput(NET*, double*), ComputeOutputError(Net*, double*), and GetOutput(NET*, double*) these three functions, we can see that the portion of time consumed by these functions decrease in the flat profile of parallelized programs. Moreover, all the function calls of these three functions decrease from 222, 680 to 168, 170 times. The GetOutput() function execution time decreased to 0.25 from 0.27 originally. And the ComputeOutputError() function execution time decreased to 0.24 from 0.33 originally. Lastly, the SetInput() function decreased to 0.12 from 0.34 originally.

Status of The project

In this project, we parallelized successfully 3 functions, “ComputeOutputError”, “SetInput” and “GetOutput”, and failed to do so for 3 functions, “PropagateLayer”, “BackpropagateLayer” and “AdjustWeight”. We then used OpenMP to help parallelize those functions.

The major technical challenges we encountered were that the server was really slow. Also, it leads to incorrect results of backpropagation networks. For example, when the server is slow for multiple jobs, it will dramatically affect the performance and also the correctness. The program just kept running indefinitely, when it should have finished in a short amount of time.

Task	Breakdown
Project Setup	Huy Tran - 20%, Tsung Wei Liu - 50%, Yin Yu Chang - 30%
Implementation using CUDA	Huy Tran - 33.3%, Tsung Wei Liu - 33.3%, Yin Yu Chang - 33.3%
Implementation using OpenMP	Huy Tran - 20%, Tsung Wei Liu - 40%, Yin Yu Chang - 40%
Debugging	Huy Tran - 40%, Tsung Wei Liu - 30%, Yin Yu Chang - 30%
Result Evaluation/Profiling	Huy Tran - 30%, Tsung Wei Liu - 30%, Yin Yu Chang - 40%
Project Report	Huy Tran - 33.3%, Tsung Wei Liu - 33.3%, Yin Yu Chang - 33.3%
Presentation	Huy Tran - 33.3%, Tsung Wei Liu - 33.3%, Yin Yu Chang - 33.3%