Tyler Rose

Professor Annexstein

CS6068

5 December 2018

Analysis of Beer Classification in Parallel

Introduction:

The purpose of this project is to test the conversion of a classification of a beer's style based on tags that were predetermined and pulled from the beer's description to parallel. The original concept of parsing a beer's description for certain predetermined tags, separating it to a data set, then training a neural network over the features is a non-trivial problem that would be ideal for parallel operations. This project was worth investigating due to the high usage of neural networks in tech companies today and due to the high performance gains that were achievable in the data parsing part in particular. To get full use of this project, a computer with a Nvidia Graphics Processing Unit (GPU) must be used to leverage Cuda.

The first part of the program utilizing parallelism is the parsing of data which is easily separable by beer entry and can use a simple Pipe-and-Filter pattern. The actual algorithm used to parse the description uses one thread per tag/description combination then checks if the tag is in the description. Since this project encompassed both running sequential and in parallel to find comparative performance, the algorithm was modified to only use thread 0 in sequential mode. This means that the total computations is the number of beer entries times the length of the descriptions times the length of the tag. Since it was predetermined to make descriptions a maximum of 2000 characters and tags a maximum of 20 characters there is a max of 40000 computations. This means that we have a work complexity of O(n). When this is done in parallel there is a thread for each tag/description combination so it has a step complexity of O(1). This method uses 2000 characters per entry, 20 characters per tag (85), and a copy of all the tags per block in shared memory. This means that we are using 2000n + 1700 + 1700*(n/1024) bytes which is O(n).

The second part is the machine learning on the data that was parsed. Since neural networks do a lot of matrix operations to find the weights for each layer and each input this also can benefit greatly by using parallel computing[1]. Upon initial research I found that there are already libraries out there that take advantage of the GPU for these operations such as Tensorflow-GPU and Caffe2. Due to a plan to use Tensorflow for my group's senior design project, I chose to go with Tensorflow-GPU to get familiar with the technology. By using Python for the neural network, Tensorflow's "matmul" function was called which uses the GPU automatically when you have PyPi's "tensorflow-gpu" package installed and uses the CPU if PyPi's "tensorflow" is installed. Since this project is not focused on the accuracy of the prediction model but rather on the performance gains that can be achieved through parallelism, A simple neural network was formed using forward propagation.

Design and Optimization Approach:

This program works by first downloading and caching the beers from BreweryDB, parsing the data, splitting the data into a training and testing set, training the neural network, then testing for accuracy as can be seen by Figure 1.

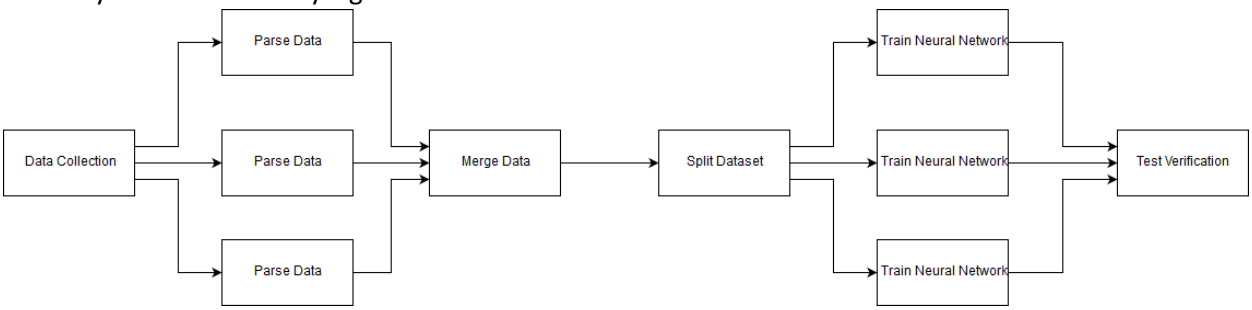

Figure 1: Program Flow Chart

The two largest parts of this are the data parsing and the neural network training. Most of the work for this project went into the data parsing as that was the focus and where there was the most room for improvement.

For the data parsing, the main concern is the amount of memory on the GPU and the memory required for storing all the descriptions, tags, and result. To handle this issue, I assumed based on a quick overview of the raw data that the descriptions would not be more than 2000 characters. If they were, they became truncated at 2000 characters. The tags were predetermined by me and were limited to 20 characters and there were 85 tags. To save space on the result, the result was stored as an 11-byte bitmask to store a binary representation of if each tag was present for the given beer. Since all this was predetermined, it was then known that there would be 1700 bytes required for the tags and 2011 bytes per beer for the results and the description. This made it easy to calculate how many beer's the GPU could parse at a time. To ensure that we did not end up with more memory on device as calculated, the beer descriptions and tags were both copied over as char** instead of thrust vectors as they use less space. For this same reason, a bitmask was used for the results as all 85 tags could be marked as present or not in only 11 bytes vs an array of Booleans which would require 85 bytes. As mentioned in the Introduction, the data parsing works by each thread taking a tag/description combination and checking if the description contains the tag. The first thing the method does is pull the tags into shared memory since many threads will be reading from there at the same time. Once this is done, the thread scans through each character in the description and checks if character matches with the tag (case insensitive). If the tag is found, the corresponding bit in the bitmask is set to 1. Consideration was given to putting the tags in constant memory but issues occurred while copying from host to device so that portion of the code was put on hold due to time constraints.

The neural network part of this project was mostly just an install procedure of tensorflow-gpu instead of tensorflow. Since tensorflow has built in all of its cuda interface, there was no additional coding that needed done between the sequential and parallel implementations. There were multiple difficulties concerning software required and versioning to get it all working, however. Originally, the project was setup using Cuda 10 but that quickly fell through as tensorflow-gpu does not support cuda

10 yet. A downgrade to Cuda 9 was required first as well as CudNN needing installed. CudNN is Nvidia's in house Neural Network framework for Cuda.

Following all the changes made to parallelize this scenario, it was expected that the data parsing would make the largest impact. Since this is making all 1109 descriptions with all 85 tags parse simultaneously rather than sequentially, an improvement of around twenty times was expected. The neural network, on the other hand, was not expected to have such a drastic effect. Since the data set is rather small for training examples and ran in 5000 iterations, the code was not optimized to keep data stored between each iteration. A speedup of around two times was expected for this case.

Application Performance Analysis and Project Results:

Through the course of this project, a computer with a six core Intel Core i5-8600K was used at 3.60 GHz with 16 GB of RAM. This system also had a Nvidia GeForce GTX 1080 Ti with 11 GB of device RAM. Cuda 9 with CudNN and tensorflow-gpu were used for all GPU code. The results when ran are shown in Table 1.

	Sequentially	Parallel
Parsing	16.471 s	0.113 s
Training	48.268 s	68.258 s
Total	64.739 s	68.398 s

Table 1: Results

These results bring a lot of insight to how tensorflow works as well as the amount of improvement can really be gained by using parallel computing.

For the parsing, these results exceeded all expectations. This shows an improvement of just under 150 times. This result makes sense as we are essentially computing all 94,000 results at the same time rather than one at a time. The point where this will become more interesting is when you have data that meets or exceeds the available memory on the graphics card. If it is under double what is available on the device it should only take around .2 seconds but having to copy all the data over to the graphics card could take a considerable amount of time. Another interesting case for future testing would be a more optimized sequential approach for finding a substring and using all 6 cores on the CPU that are available. This may make the difference much less drastic.

More interesting, however, is the neural network training results. The sequential approach was around 40% faster. This can be explained based on the approach used and the data used. First off, the data set is rather small for matrix multiplication to make too large of a timing difference. Second, and more important is the fact that the data is not kept on the device between iterations of training. This means that before each iteration the data was copied from the host to the device then at the end of the iteration it was copied back to the host then repeated. This is a very wasteful process and would account for the slowdown.

Further work could be done in quite a few area's to further optimize the code. The first change to be made is the optimization of the neural network training code. A custom kernel should be written to write the data to a new area on the device and load as much of the data samples to the device at once rather than one by one. The next change that could be made is optimizing the data parsing on the

CPU for a more accurate comparison. The final change that could be made is using a more effective neural network model. A more effective model would require more computations and levels which may make the parallel code much more effective. Perhaps another interesting comparison could be made using Caffe 2 rather than Tensorflow for the Neural Network. According to Baige Liu and Xiaoxue Zang of Stanford University, Caffe 2 seems to have better training speeds than Tensorflow with the GPU but has larger file size which are of no concern for this project[2].

Division of Work:

Seeing as this was a single person project, I have done all the work on this myself and all code was developed from scratch. That being said, my contributions to this project included the development of the API call to get the data from BreweryDB. I also wrote the code to cache the results so the API does not need to be hit every single time since that is not relevant to the effects of parallelizing the code. The data parsing, the kernel, and all other data processing code and all the python code to split the data, train the network, and test the accuracy was written by me as well.

Bibliography:

1. Why are GPUs necessary for training Deep Learning models? [Internet]. Analytic Vidhya.; [cited 2018 December 5]. Available from: https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/
2. Caffe2 vs. TensorFlow: Which is a Better Deep Learning Framework? [Internet]. Baige Liu and Xiaoxue Zang.; [cited 2018 December 5]. Available from: http://cs242.stanford.edu/assets/projects/2017/liubaige-xzang.pdf

Code Appendix:

https://github.uc.edu/roset3/BeerClassification