Parallelizing Gradient Descent on Different Architectures

Shashank Ojha and Kylee Santos

Link:

https://shashank-ojha.github.io/ParallelGradientDescent/

Summary:

We are going to create optimized implementations of gradient descent on both GPU and multi-core CPU platforms, and perform a detailed analysis of both systems' performance characteristics. The GPU implementation will be done using CUDA, where as the multi-core CPU implementation will be done with OpenMP.

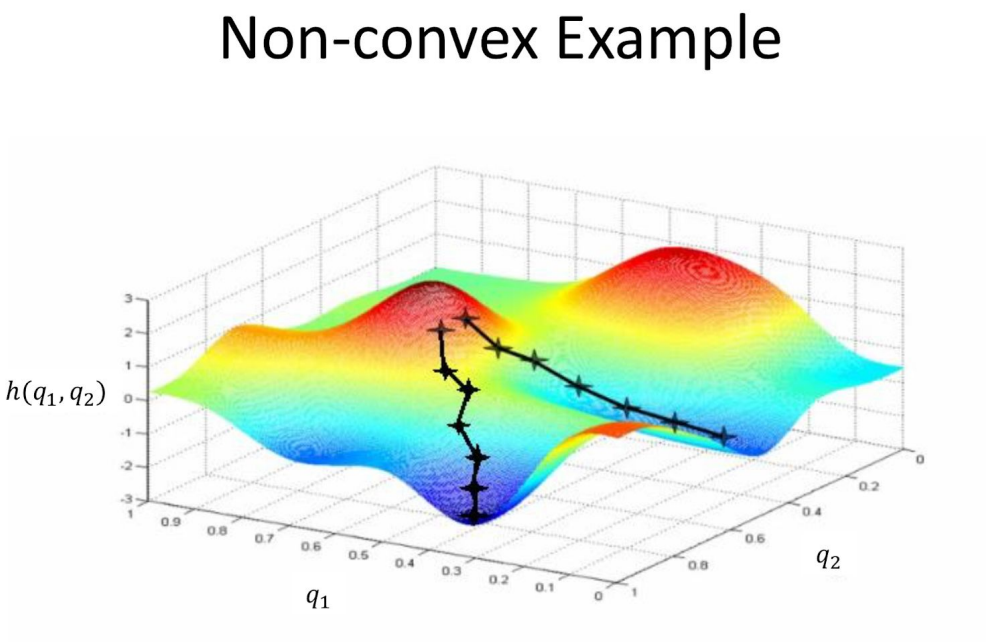
Background:

Gradient Descent is a technique used to find a minimum using numerical analysis in times where directly computing the minimum analytically is too hard or even infeasible. The idea behind the algorithm is simple. At any given point in time, you determine the effect of modifying your input variable on the cost you are trying to minimize. In the one dimensional case, if increasing the value of your input decreases your cost, then you increase the value of your input by a small step to get you closer to the minimum cost. Generalizing to higher dimensions, the gradient tells you the direction of the greatest increase on your cost function, so we move the input in the opposite direction to hopefully decrease the value of the cost function.

Building on top of this idea, we have two kinds of gradient descent methods. One is called batch gradient descent and the other is stochastic gradient descent. Let's discuss both. We can imagine we have some regression problem where we are trying to estimate some function f(x). We are given n labeled data points of the form $(x_i , f(x_i))$. Our job is to find some estimator, g(x), of the desired function such that we minimize the squared error $(f(x_i) - g(x_i))^2$. Now in order to find the true gradient of our cost function, we would need to plug in all our points. This is what's known as batch gradient descent. If n is large, however, this computation can be very expensive especially since at each time step we only make a small modification. Thus, it does not always make sense to doing such a heavy computation to make a tiny step forward. In contrast, stochastic gradient descent chooses a random data point to determine the gradient. This gradient is much noisier since it doesn't have the whole picture, but it can greatly speed-up performance.

In most cases, it is best to start with stochastic gradient descent, but then eventually switch to batch for fine-grained movement.

The last important detail to discuss is the concept of local and global minima. When the cost function we are trying to minimize is convex, any local minima will be equal to the global minima by definition. However, for non-convex functions, we might stumble across a local minima as opposed to a global minima. Consider the following graphic take from Howie Choset's Robot Kinematics and Dynamics Class.



As you might observe, the initial position start state may affect our answer when performing gradient descent. This is because the gradient is different at each point and thus starting at a different point may take us in a different direction. With respect to stochastic and batch gradient descent, stochastic gradient has the property that it might jump out of local minima because it has a random aspect to it that makes the gradient more noisy. This has a variety of applications in machine learning where we are working with large data sets and a gradient descent becomes very computationally intensive.

Challenges:

The challenges are more related to finding the best suited programming model given an architecture. This is because each architecture calls for a different implementation. With the high number of threads available to GPUs, we can potentially calculate the gradient with a larger

number of points and make a more accurate step towards the optimal, but each update may be slower than on multi-core CPUs. In contrast, multi-core CPUs will have the advantage of making updates faster because there is no offload overhead involved. This tradeoff between correctness and speed presents an interesting challenge of finding the programming model that optimizes performance in each case.

Another challenge of the project comes from the fact that we are working with large datasets, so how we organize our memory to hold all the data will be another area of focus for this project. For example, previous papers have stated that it may be beneficial to partition the data into disjoint subsets so each core or block on the GPU doesn't have to share data. Of course, this may lead to less accurate updates since each core or block only has a part of the whole picture. However, the speedup gained from less sharing might still allow us to arrive closer to the optimal sooner.

Resources:

We will start off by implementing our gradient descent using CUDA on NVIDIA GeForce GTX 1080 GPUs and using OpenMP on Xeon Phi Machines. We will start the code from scratch since the actual implementation of the gradient descent isn't too complex and we may make several modifications to it based on the architecture. This assignment is an exploration of different system designs and architectures, therefore it does not make sense to build off someone else's code. We want full control of everything.

There are several online papers about this topic. For now, we will use the following two papers for reference.

1. http://martin.zinkevich.org/publications/nips2010.pdf
2. https://arxiv.org/pdf/1802.08800.pdf

The one thing we still have to figure out is how to obtain a good dataset. We want a data set that is large and minimizes a non-convex function, so we can observe cases where our gradient descent might fall into a local minimum as opposed to a global one. If time permits, we might run our program on other machines and measure performance on those as well.

Goals and Deliverables:

Our overall goal that we plan to achieve is to recreate the results of the papers mentioned above. The desired output of our program is a $\alpha = \{1.5, 1.1, 1.01\}$ approximation of the optimal solution and we want to examine the runtime it takes to get to those levels of correctness. We aim for a runtime that is around 8x faster than the serial version. We chose this benchmark after

reading the papers linked above. According to their results, they achieved approximately a 12x speedup for their data set, so we expect to see similar results.

Over the course of the project, we plan to implement a parallelized version of SGD on both GPUs using CUDA and multi-core CPUs using OpenMP and/or OpenMPI. On both architectures, we want to try a couple of designs. Listed are some of those designs:

1. Load all the data on each core, compute the updated value in parallel based on a random data point on each core, and average their results to achieve better correctness.
2. Partition the inputs, compute the updated value for each of the subsets in parallel, and average their results to achieve better correctness.
3. The same as method 1, but compute k updates together on each core and then average their results.
4. The same as method 2, but compute k updates together on each core and then average their results.

For the poster presentation, we expect to have lots of graphs of our results. Specifically, we want to at least have the following graphs:

1. Speedup vs Number of Threads for each Architecture
2. Speedup vs Approximation Factor for each Architecture (Speedup of Max threads available to the machine with respect to 1 thread)
3. Speedup vs Dataset size for each Architecture (Speedup of Max threads available to the machine with respect to 1 thread)

If time permits, we also have some additional ideas that we could explore further. We can implement a mixture of batch and stochastic gradient descent to achieve a performance as good or better than the reference papers for some approximation of the optimal. In addition, we can look into OpenCL in order to run SGD on heterogeneous machines that have access to both GPUs and CPUs and measure performance there.

Platform Choice:

The goal of this project is to determine the performance tradeoffs of paralleling gradient descent on GPUs and CPUs. The reason we choose to work with NVIDIA GeForce GTX 1080 GPUs and Xeon Phi Machines is mainly because we are most familiar with the machines.

Schedule:

Week of 11/5 - 11/9:

Find a couple data sets to test on

Get a sequential version of the code working in C/C++

Get measurements on how long it takes to get within $\alpha = \{1.5, 1.1, 1.01\}$ of the optimal.

Week of 11/12 - 11/16:

Implement Parallel SGD using CUDA

Implement all four algorithmic designs

Get measurements on how long it takes to get within $\alpha = \{1.5, 1.1, 1.01\}$ of the optimal.

Week of 11/19 - 11/23 (Week of Thanksgiving Break):

Implement Parallel SGD using OpenMP and/or OpenMPI with just one of the four algorithmic designs discussed

Week of 11/26 - 11/30:

Implement all four algorithmic designs for the OpenMP version

Get measurements on how long it takes to get within $\alpha = \{1.5, 1.1, 1.01\}$ of the optimal.

Week of 12/3 - 12/7:

Look into additional optimizations that can be done in each architecture in terms of how the data is organized/cached and make those optimizations

Week of 12/10 - 12/14:

Look into OpenCL if time permits or continue working on any of the previous tasks if behind.