GPU Project: SGD implementation on GPUS

Prithvi Krishna Gattamaneni
Adithya Parthasarathy
pkg238@nyu.edu
ap4608@nyu.edu

1 Gradient descent intro

Gradient descent intro
The structure of several machine learning algorithms is to have a data matrix of dimensions n*d and a vector y of size n, where n is the number of data points, and d is the dimensionality. Y is a vector of predictions for each data point. This setting fits a wide range of tasks such as regression and classification.

In each of these settings it is common to have a loss function $L(H; X, y)$. This function usually returns a scalar greater than zero, that is indicative of how well the hypothesis H is able to map the data X to y. The smaller the loss function is, the better it is able to model the data. For very trivial formulations of the loss function, it is possible to find H that minimizes L in closed form, but in most cases this is not true.

In cases where no closed form exists, it is very preferable to pick a L that is differentiable. Gradient descent is a technique that allows us to use this property to minimize $L(H; X, y)$ wrt H. The idea here is to find the differential of $L(H; X, y)$ wrt H which we denote as $\nabla_H(L(H; X, y))$.

After initializing H, an update is made to H by the following

$$H \leftarrow H - \eta \nabla_H(L(H; X, y))$$

where η is some step size (some small value, typically 0.3). By doing the above repeatedly, H eventually converges to the solution.

2 Why SGD

The drawback of the above solution is that calculation of $\nabla_H(L(H; X, y))$ requires us to look at all the datapoints of X. This obviously does not scale very well for large datasets, as we have to visit each datapoint for every step that is taken towards optimizing H. In order to workaround this another approach is to take a subset of X for the calculation of $\nabla_H(L(H; X, y))$ at each step. In order to truly represent the distribution it is common practice to take a much smaller and random subset of size say 32, during the calculation of $\nabla_H(L(H; X, y))$. This is called minibatch gradient descent. However when this subset size is decreased to one, we call this stochastic gradient descent. In every epoch, all the datapoints are visited in some random order. The advantage of SGD is that even though the differential we compute at each step is not completely accurate, it represents the correct direction of the step that H needs to take while consuming far less computing power, as a result of which SGD is known to converge much faster than traditional gradient descent, and minibatch descent.
A summary of the SGD approach is as follows.

Algorithm 1 SGD

procedure SGD
    initialize H
    for i=1 to T **do**
        for each datapoint X_j, y_j **do**
            $H \leftarrow H - \eta \nabla_H (L(H; X_j, y_j))$

3 Scope for parallelism

Looking at the implementation of SGD, we straight away see two embedded for loops, which is an indication of SIMD parallelism. However it is clear that we can leverage this only if the hypothesis H can be averaged out across multiple threads. In many machine learning settings this is indeed the case. The idea here is to use multiple threads, say N threads, and to have each thread then perform SGD independently of each other, and to do so on a subset of the total Dataset, that has been divided randomly amongst the N threads. After a fixed number of iterations, the idea is then to take an average of each H_i for all the threads from i=1 to N.

4 Explanation of first implementation

For the first attempt we assume a regression setting. The data is generated randomly using a python script and noise is added to the predictions to simulate a real dataset. We chose this approach as it easier to test our implementations with datasets of different sizes. we pick random solution vector w, which is then used to generate the solution y. The data matrix x and y is then given to the implementation.

The approach here is to divide the dataset amongst the threads. Each thread then performs SGD on it's dataset, and after a certain number of iterations, we take the average of each threads solution to get the final result. Formally this can be expressed as shown below.

Algorithm 2 Parallel SGD with samples $X = x_0 ... x_m$, iterations T, step size η, number of threads n, and states w

Require: $H = \frac{m}{n}, \epsilon > 0, n > 1$

procedure PARALLEL SGD
    Partition X giving H samples to each thread
    for i=1 to n in parallel do **do**
        initialise $w^i = 0$
        for all t=0...T **do**
            pick a datapoint (X_j, y_j)
            $w^i \leftarrow w^i - \eta \nabla_{w^i} (L(w^i; X_j, y_j))$

The code for this implementation was implemented in both main.cu and in seq.cpp, i.e a gpu and a cpu implementation of the same parallel algorithm. We now compare the performance of both the GPU and the parallel CPU implementations.

5 Comparison to CPU parallel implementation

6 CPU vs GPU results for this

7 Nvprof results

8 Optimisations such as using other w to

9 Solution

A

10 Solutions

10 (a) Solutions

10 (a) [i] Solution

10 (a) [ii] Solution

10 (a) [iii] Solution

10 (b)

11 Solution