Parallelizing Incremental Batch Gradient Descent with Hogwild!

Guilherme Albertini, Mayukh Ghosh, Zicheng Ren

Final Report for High-Performance Computing (CSCI-GA 2945) - Spring 2022

Professor Peherstorfer

05/09/2022

Contents

1. Abstract

Stochastic gradient descent (SGD) was initially thought to be an inherently serial algorithm: each thread must wait for another to update before moving on to the next iteration of the steepest descent computation. To make the algorithm parallel, an obvious approach would be to lock each update step (i.e. each thread locks the current estimate of the solution before updating the parameter, and unlocks once the update is done). One quickly sees that the overhead from numerous lock allocations and scheduling supersedes the actual times spent on updates — this is where the Hogwild! algorithm shines: by removing all thread locks from parallel SGD code, the asynchronous parallelization has shown to be mathematically efficient and resulted in magnitudes of speedup over the vanilla version. Here we study the speedup and accuracy of the original Hogwild! algorithm as applied to a batch SGD by varying configurations (i.e. batch sizes and design matrix dimensions) using a serial implementation in C++ as a baseline compared against a CUDA-parallized version leveraging two GeForce RTX 2080 Ti GPUs (11 GB memory each) on the cuda2

CIMS server. For fixed matrix size, smaller batch sizes led to greater speedups than those for larger batch sizes. On the other hand, the test loss for the parallel version is greater than that of the serial version.

2. Introduction

Much study has been done on lock-free algorithms since the inception of the original Hogwild! version. Here we specifically aim to study the algorithm's effects on batch gradient descent's error rate by modulating the input space and comparing the speedups for both the parallel and serial versions. We also aim to investigate the impacts of the lock-free approach on the mean squared error loss in regards to the training set and testing set for differently sized matrices.

Namely, we focus on matrices sized with 10 predictor variables and 3 sets of training instances: 10,000, 50,000 and 100,000 configurations. The learning rate is set to 0.05 and subsequently halved after each update to the parameter to quickly ascertain the next update weights. Note that hyperparameter tuning was not the focus of our investigation here though we see the convergence of the algorithm after 100 epochs for every set of tests.

We compare different types of setups in our project: the speedup of a fixed size design matrix (50,000 by 10) for different batch sizes of 8, 16, 32, 64, 128, 256; the test error (really, the average mean squared error) for a fixed size design matrix given a batch size; and finally the speedup of a fixed batch size of 256 along with the test error after learning weights for the 10,000, 50,000 and 100,000 test set instances. All these tests are run for both the serial and parallel implementation. We aggregate and compare results for each setup.

3. Methodology and Challenges

3.1 Serial Implementation (C++), Data Generation

The serial implementation of the batch SGD was implemented in C++ using standard libraries and a custom suite of timing tools provided by the instructor. The source code provides details for the latter (see utils.h). For each test configuration, the loss over 100 epochs is computed using mean squared error. The number of predictors, training size, batch size, and number of epochs can also be set by the user whereas the learning rate is initialized to 0.05. We generated the data for the test and train set using a Python script that also plotted results from the csv files generated (found in generator.ipynb). Note that the data-generating distribution is produced from a polynomial function. The average loss is computed per configuration and results are plotted for the setups described previously.

3.2 Parallel Implementation (CUDA)

The parallel implementation, namely hogwild, was also implemented in C++ using standard libraries and measurements of the timing of the program was measured using the CUDA library. Just like the serial implementation, the loss is computed over 100 iterations for all experimental configurations. The gradient calculation for each batch is computed by different cores in the GPU. In other words each thread computes the prediction, gradient and updates the weight from the corresponding gradients and saves these values in their own part of global memory for next iteration. We choose to use global memory instead of shared memory to save these values in CUDA because the limit size of shared memory cannot allocate enough space when the size of configuration is very large. While each thread/core has to compute their own prediction and gradients, the weights are shared among all the cores. This means that any thread is free to update the weights even at the risk of race conditions. Each thread processes the same batch over 100 epochs. We did so to avoid massive communication between the host and the device GPU for shuffling

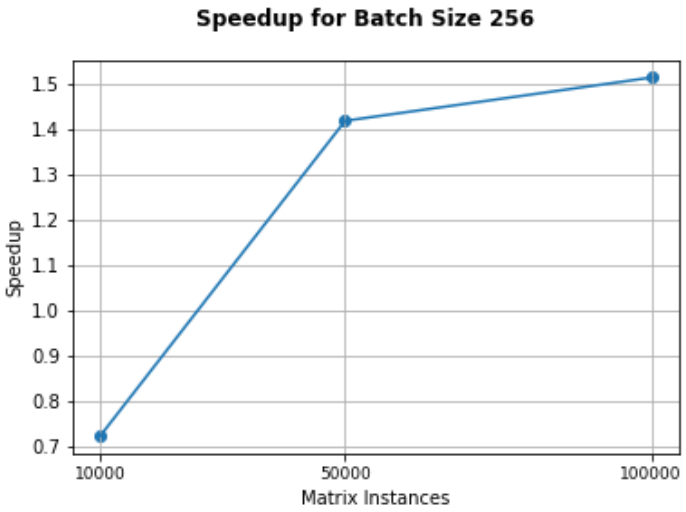
the batches between epochs. In our implementation, we shuffled the whole dataset once in the beginning to achieve randomness when choosing batches. The number of threads is equal to the number of mini-batches. The same loss as above is computed per configuration and the results are plotted.

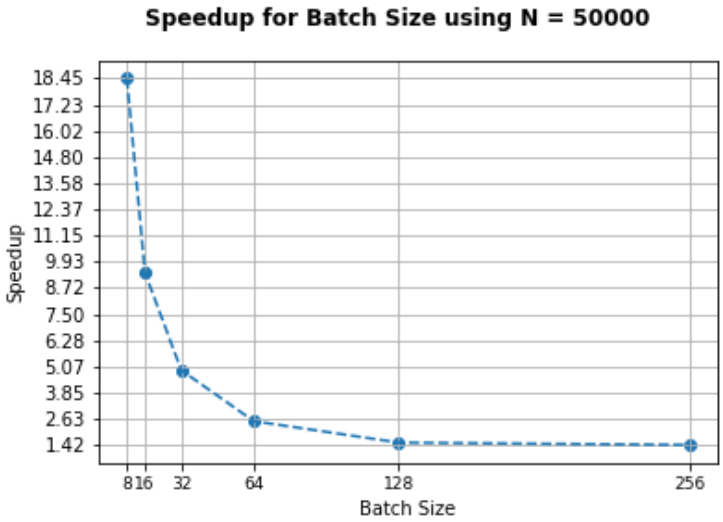
3.3 Difficulties Encountered

Serialized Batch SGD implemented using C++ using native libraries and the utilities (for timing) provided by the instructor. We identified several logic errors in regards to matrix indexing after using valgrind on the script and observed both leakages and memory overwrite warnings; additionally, we forgot to free allocated memory on the first few runs. These errors were quickly fixed.

Since CUDA doesn't support dynamic indexing for arrays in local memory, another difficulty we met is to find a way to save thread-owned intermediate results like prediction and gradients. Saving these intermediate results in the host memory is not a valid option since the overhead to transfer data between the host and the device is very expensive. We first tried to solve the problem by saving these intermediate results inside GPU's shared memory for each block since shared memory access speed is faster than that of global memory. However, RTX 2080 Ti only has 64 KB shared memory per SM, eventually it will not be large enough for all the threads to store their intermediate results as the size of input grows. In the end, we choose to allocate space for each thread inside GPU's global memory using careful indexing to avoid threads overwriting each other. We did the same for storing the loss value for the evualation by storing the loss of each epoch inside the global memory and transferring them back at once after the kernel finishes its job to avoid transferring data between epochs.

4. Results and Discussion

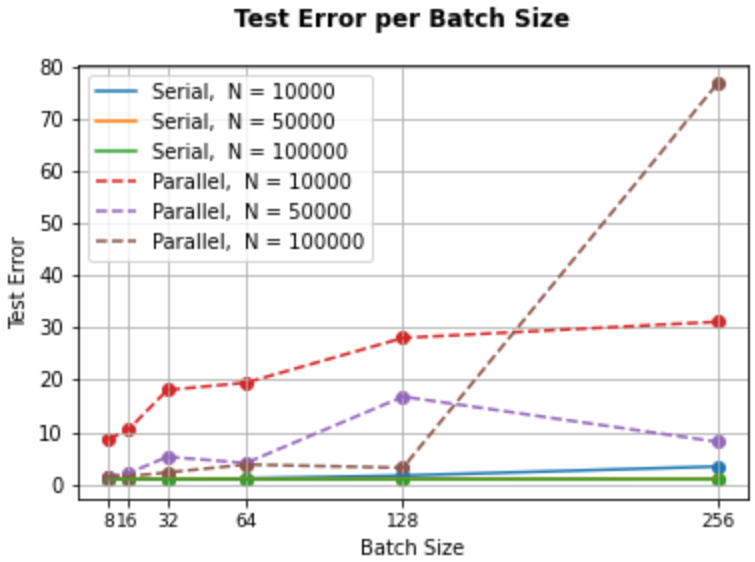




Based on the first graph, the speedup grows as the number of matrix instrances increases, but it doesn't scale well. The reason for that is the large batch size causes the number of threads to be small, therefore, all the threads can fit in one block to be run in one SM, making other SMs idle.

For a fixed matrix size, speedup decreases exponentially with increasing batch size. This happens for the similar reason as that of the first graph. The larger batch size causes the number of threads generated to be decreased. Since we fix the number

of threads per block to be 500, fewer threads results in fewer blocks. CUDA allows a whole block to be runned on a single SM, more blocks means more SMs and more parallel computing power can participate in the computation. As a result, when the number of blocks is small, only one block is generated for the case of batch size 256, the speedup goes down dramatically.



Test error for the parallel implementations is generally greater than that of their serial counterparts. The test error was the greatest for the largest matrix in the parallel regime, probably due to the lock-free scheme where the different threads are free to access and overwrite the same weights all at once. This leads to a lot of race conditions. Therefore, the weights are not the most optimal as compared to the serial version. On the other hand, since we are averaging losses over an entire batch, smaller batch size leads to less error.

5. Conclusion

After running our test setups we find that, for fixed matrix size, smaller batch sizes led to greater speedups than those for larger batch sizes. Contrarily, the test loss for

the parallel version is greater than that of the serial version. The parallel versions of batch SGD produced greater test errors on the whole due to the lock-free scheme adversely affecting the final values of the weights. However, with a careful choice of batch size (usually a small batch size), the test error can be reduced to the same level as the serial version while still having about 20 times speedup for a very large problem size.

6. References

1. Niu, Feng et al. "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent." (2011).

2. “Programming Guide :: CUDA Toolkit Documentation.” *(C) Copyright 2005*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 8 May 2022.