Efficient Distributed Stochastic Dual Coordinate Ascent

Mingrui Liu, Jeff Hajewski
Department of Computer Science
University of Iowa
Iowa City, IA 52242
mingrui-liu@uiowa.edu, jeffery-hajewski@uiowa.edu

Abstract

We propose the design and analysis of an efficient, distributed, SDCA algorithm that uses GPUs to improve computational efficiency for parameter updates. This work builds off the previous work of T. Yang [**?**], creating a more scalable and efficient SDCA algorithm. Specifically, we propose to use CUDA to improve the runtime efficiency of the gradient and parameter update calculations along with the use of MPI for network communication. The implementation is written in pure C++.

1 Introduction

In recent years, the amount and size of available data has grown at an incredible rate. As the size of the data grows, the challenge of applying standard machine learning algorithms to the data has become increasingly complex. Two common countermeasures used to deal with this are employing stochastic optimization algorithms, and utilizing computational resources in a parallel or distributed manner [**?**].

In this paper, we consider a class of convex optimization problems with special structure, whose objective can be expressed as the sum of a finite sum of loss functions and a regularization function:

$$\min_{w\in\mathbb{R}^d} P(w), \text{where } P(w) = \frac{1}{n}\sum_{i=1}^{n}\phi(w^\top x_i, y_i) + \lambda g(w), \tag{1}$$

where $w \in \mathbb{R}^d$ denotes the weight vector, $(x_i, y_i), x_i \in \mathbb{R}^d, y_i \in \mathbb{R}$, $i = 1, \ldots, n$ are training data, $\lambda > 0$ is a regularization parameter, $\phi(z, y)$ is a convex function of z, and $g(w)$ is a convex function of w. We refer to the problem in (1) as Regularized Finite Sum Minimization (RFSM) problem. When $g(w) = 0$, the problem reduces to the Finite Sum Minimization (FSM) problem.

Both RFSM and FSM problems have been extensively studied in machine learning and optimization literature. When n is large, numerous sequential stochastic optimization algorithms have been proposed [**? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?**], and there also exist several parallel or distributed stochastic algorithms [**? ? ? ? ? ? ? ?**]. Specifically, S. Shalv-Shwartz and T. Zhang [**?**] proposed the Stochastic Dual Coordinate Ascent (SDCA) which provided new analysis with strong theoretical guarantees regarding the duality gap. T. Yang [**? ?**] developed two Distributed Stochastic Dual Coordinate Ascent (DisDCA) algorithms and analyzed the tradeoff between network communication between nodes and the difficulty of the performed computation (task). However, the problem of developing a more efficient distributed SDCA algorithm is still open. In this paper, we first provide a GPU implementation of the vanilla distributed SDCA [**?**], and then give an asynchronous distributed approach to SDCA to make full use of computational resources that scale well.

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

2 Related Work

First we review the related work of sequential stochastic convex optimization for solving FSM and RFSM problems. The first numerical scheme of stochastic optimization stems from stochastic gradient descent (SGD) [**?** **?**], which was designed to avoid the calculation of full gradient and gets faster convergence than full gradient descent (FGD). To improve the converge rate of SGD, many new algorithms were proposed by exploiting the finite sum structure, including the Stochastic average gradient (SAG) [**?**], stochastic dual coordinate ascent (SDCA) [**?**], stochastic variance reduced gradient (SVRG) [**?**], accelerated proximal coordinate gradient method (APCG) [**?**], SAGA [**?**], Prox-SDCA [**?**], Prox-SVRG [**?**], and stochastic primal-dual coordinate method (SPDC) [**?**]. Recently, the optimal first-order stochastic optimization method were developed [**?** **?**]. Although there exist rich literature studying sequential stochastic optimization with strong theoretical guarantee, less efforts have been devoted to considering them in a parallel or distributed manner. It constitutes a huge gap between theory and practice, since nowadays the size of data increases at a rapid speed, which makes one-core processor or one computer very difficult to handle it properly.

Then we review several related work of distributed optimization algorithms. In the existing literature, many distributed algorithms have been developed on top of stochastic gradient descent (SGD), alternating direction method of multipliers (ADMM), and stochastic dual coordinate ascent (SDCA). The two main approaches used in developing parallel algorithsm for SGD are based on shared memory and distributed memory architectures. Some work [**?**] looks at both settings, in addition to removing the syncrhonization requirement in the distributed memory setting. A number of approaches [**?** **?** **?**] consider the asynchronous or lock-free setting, making use of parameter servers [**?**], sparsity [**?**], as well as unique data-flow architectures for parameter updates [**?**]. ADMM stems from [**?**], which was developed to solve the equality constrained optimization problem. Recently, two independent works of stochastic ADMM were proposed [**?** **?**]. A standard reference for distributed ADMM is [**?**]. The advances of SDCA algorithms [**?**] and its variant [**?** **?** **?**] enjoy faster convergence than SGD and ADMM, and the distributed SDCA (DisDCA) [**?** **?**] was developed along with novel analysis of tradeoff between computation and communication. [**?**] serves as the starting point for our work.

In [**?**], SDCA is implemented in a distributed, synchronized fashion. While the achieved results were quite promising, they did not take advantage of hardware acceleration or asynchronous communication. We will build off of work from T. Yang's work on distributed SDCA [**?**], and add GPU capabilities.

3 GPU Acceleration for Sequential SDCA

Algorithm 1 Sequential SDCA

Require: $\alpha^{(0)}$
Ensure: $\bar{w}$
1: Let $w^{(0)} = w(\alpha^{(0)})$, where $w(\alpha) = \frac{1}{n}\sum_{i=1}^{n} \alpha_i x_i$
2: **for** $t = 1, 2, \ldots, T$ **do**
3: Randomly pick i
4: Find $\Delta\alpha_i$ to maximize $-\phi_i^*(-(\alpha^{(t-1)} + \Delta\alpha_i)) - \frac{\lambda n}{2}\|w^{(t-1)} + (\lambda n)^{-1}\Delta\alpha_i x_i\|^2$
5: $\alpha^{(t)} \leftarrow \alpha^{(t-1)} + \Delta\alpha_i e_i$
6: $w^{(t)} \leftarrow w^{(t-1)} + (\lambda n)^{-1}\Delta\alpha_i x_i$
7: **end for**
8: Output (Random option):
Let $\bar{\alpha} = \alpha^{(t)}$ and $\bar{w} = w^{(t)}$ for some random $t \in T_0 + 1, \ldots, T$
9: **return** $\bar{w}$

The traditional SDCA algorithm [**?**] is described in Algorithm 1, where $g(w) = \frac{1}{2}\|w\|_2^2$. In that procedure, the most expensive work is in line 4–6. We employ the GPU acceleration technique to make those lines run in a faster manner.

3.1 Naive GPU Implementation

The naive implementation (hereafter referred to as the sequential GPU algorithm) is simply the CPU-based algorithm but we replace all linear algebra/vector operations with CUDA kernels and carrying them out on the GPU. In other words, there is no change to the SDCA algorithm, we simply perform some of our calculations on the GPU, rather than performing everything on the CPU.

3.2 Refined GPU Implementation

The refined GPU implementation improves upon the naive implementation by optimizing memory allocations. This is done by using static pointers to hold the address of the allocated memory on the GPU. Once this memmory is allocated, there is no need to re-allocate, since we can access it via the pointers. The memory is re-allocated only in the event that the amount of memory needed changes.

4 GPU Acceleration for Distributed SDCA

The distributed SDCA [**?**] is described in Algorithm 2. We restrict our implementation over the case when $g(w) = \frac{1}{2}\|w\|_2^2$. The procedure **SDCA-mR** is the procedure on k-th machine (process).

Algorithm 2 Distributed SDCA

1: Start K processes by calling the following procedure **SDCA-mR** with input m and T.

Procedure **SDCA-mR**

Require: Number of Iterations T, number of samples m at each iteration
Ensure: w^T
2: Let $\alpha_k^{(0)} = 0$, $v^{(0)} = 0$, $w^0 = 0$
3: **Read Data:** $(x_{k,i}, y_{k,i})$, $i = 1, \ldots, n_k$
4: **for** $t = 1, \ldots, T$ **do**
5: **for** $j = 1, \ldots, m$ **do**
6: Randomly pick $i \in \{1, \ldots, n_k\}$ and let $i_j = i$
7: Find $\Delta\alpha_{k,i}$ by

$$\Delta\alpha_{k,i} = \max_{\Delta\alpha} -\phi_{k,i}^*(-(\alpha_{k,i}^{t-1} + \Delta\alpha)) - \Delta\alpha x_{k,i}^\top w^{t-1} - \frac{mK}{2\lambda n}(\Delta\alpha)^2\|x_{k,i}\|_2^2$$

8: Set $\alpha_{k,i} = \alpha_{k,i}^{t-1} + \Delta\alpha_{k,i}$
9: **end for**
10: **Reduce:** $v^t : \frac{1}{\lambda n}\sum_{j=1}^m \Delta\alpha_{k,i_j} x_{k,i_j} \to v^{t-1}$
(Remark: the reduce step is implemented in two steps
- The master node adds the increment from each machine (process) together
- Each machine (process) receives v^t from the broadcasting process from master node

)
11: **Update:** $w^t = v^t$
12: **end for**
13: **return** w^T

We took the idea of parallelizing SDCA over a number of machines, and replaced the machines with threads on a GPU. Instead of network communication we have communication via the PCIE bus, which, while slower than RAM access, is considerably faster than ethernet. Additionally, rather than send a batch of data to each thread (to mimic sending data to a worker node), we send a single data point to k threads, rather than m data points to k threads. This simplifies communication and computation, and results ing a matrix-vector calculation that is performed on the GPU. While we could further optimze this algorithm by performing additional calculations on the GPU, the low-level implementations and handling of shared and non-shared GPU memory as well as thread synchronization proved to be too much to tackle for this project given our time constraint. However, moving forward these are great areas to explore.

5 Implementation

In our experiments, we use $\phi(z, y) = \max(0, 1 - yz)$ and $\lambda = 10$. The CPU implementation outperformed the GPU implementations across the board. While this was not expected, the results are very interesting from an implementation point of view. There are a few key areas that hurt the GPU implementation – unfortunately we did not have enough time to fix/implement these properly.

- Memory allocation
- Device-host communication
- Optimized kernel implementations

5.1 Memory Allocation

This was the first issue we ran into during our work on the sequential GPU algorithm (which we implmented prior to the distributed version). Our initial implementation made no optimizations in terms of memory allocation. Each time we performed any work on the GPU, we would allocate memory, transfer the data, perform the operation, transfer the data bac, and the release the allocated device memory. This resulted in over a million allocations on a small dataset and made up about 13s of compute time for the same run. After profiling our code using `nvprof` we realized the excessive allocations were really degrading performance and largely unnecessary.

To fix this issue, we created static pointers to allocated memory on the device. These pointers were allocated once, reducing our device memory allocations from the millions down to just three allocations (two allocations for vectors, and a third allocation for the result). This was an immense savings, and reduce the compute time for the dot product kernel from 13s down to less than a second for the test data set (roughly 100 points in $\mathbb{R}^3$).

For the distributed version, we added two additional static pointers for a matrix (which held the entire mini-batch) and a second result vector for the result of a matrix-vector multiplication.

5.2 Device-Host Communication

This is the second issue we ran into, but did not have enough time to correct. Throughout our algorithm, we transfer data to and from the GPU each time we encounter an operation that can be efficiently computed on the GPU. This means there are situations where we transfer data to or from the GPU when we don't necessarily need to – either the data already resides on the GPU or we don't actually need the result vector. Using `nvprof` we saw that communication was now dominating our GPU compute time, consisting of nearly 50% of total GPU compute time.

However, fixing this issue is quite a bit more complicated than fixing the allocation issue mentioned in the previous section. There are two improvements that can improve the communication between host and device: memory pinning and warpping our data in a class that handles memory synchronization in the background.

Memory Pinning Memory pinning is declaring that the memory allocations on the host (that are the source for the transfer to device) will not be paged. This means the data stays in pinned memory so no transfer from paged out memory to pinned memory is necessary (or even checking if it's necessary). However, since we are using fairly modern hardware, the impact from using pinned memory is not as significant as being more efficient with when we transfer between host and device.

Data Wrapper Creating a data wrapper around the data types we used from Eigen would allow us to more intelligently handle transfers between host and device. Specifically, we would be able to determine where the data is currently located (device or host) and decide whether a transfer is needed or not. For example, if the weight vector is currently sitting on the GPU, no updates are made to the weights on the host, more data is transferred to the GPU, we could avoid both a transfer of the weight from device to host and then again from host to device by simply leaving the weight vector on the GPU.

Table 1: Results of sequential CPU, sequential GPU, and distributed GPU for a sample data set of 10,000 points each having 100 features. Times are given in MM:SS.

Algorithm	**mini-batch = 1**	**mini-batch = 32**	**mini-batch = 64**	**mini-batch = 128**
Sequential CPU	7:36	0:23	0:16	0:13
Sequential GPU	7:50	0:25	0:19	0:15
Distributed GPU	33:33	1:10	0:40	NA

5.3 Optimized CUDA kernel implementations

The last issue is the implementation of our CUDA kernels. The dot-product kernel was fairly straightforward, but the matrix-vector kernel leaves room for improvement. Overall, there are a large number of kernel optimizations that can be made to improve overall kernel throughput, but these optimizations can be very low-level. Using a library such as cuBLAS would have been a better approach in hind-sight. However, at the onset of the project we determined writing a wrapper class for our data that would allow us to interface with the cuBLAS kernels more easily would require more time than we had available for the project.

Aside from taking advantage of cuBLAS, we can also perform more operations in the update phase of the distributed GPU algorithm on the GPU. Currently we simply perform a matrix-vector multiplication and then handle everything else on the host. By performing more of the update work on the GPU, we could squeeze even more performance out of this approach.

6 Experimental Results

Our results were not as promising as we anticipated. Table 1 shows the results using a created data set in $\mathbb{R}^{100}$ that contained 10,000 data points. We ran three algorithms: sequential CPU, sequential GPU, and distributed GPU. Sequential CPU is just the standard implementation of SDCA. Sequential GPU extends this implmentation by performing dot product operations on the GPU. The distributed GPU version is the implementation of distributed SDCA on the GPU (rather than a cluster). All algorithms were run for 5 epochs without any type of early termination. We are measuring, roughly speaking, how these algorithms compare in terms of the time it takes to complete a predefined workload. As we will see in a later discussion, our algorithms also converge to similar training losses.

Sequential CPU outperformed both GPU implementations due to implementation issues. However, as expected, the distributed GPU version improved in terms of compute time as the size of the mini-batch increased. As the mini-batch size increases, the number of coordinate directions optimized in the distributed SDCA algorithm increases, and these optimizations are done in parallel on the GPU. Performing more of these updates simultaneously on the GPU reduces the number of communications with the GPU and thus improving the overall runtime. Conversely, as the mini-batch size decreases we would expect GPU compute times to degrade, since a small mini-batch requires more data transfers back and forth between host RAM (on the computer) and device RAM (VRAM on the GPU). Lastly, the NA result for the distributed GPU implementation on a batch size of 128 is the result of a memory corruption error. At the time of writing this report we were still chasing down the cause of the memory corruption. We believe this is a bad memory access while indexing through the matrix of data on the GPU, but are still working on this issue. As we will discuss later, this is another reason we would have been better off using a third-party library for these kernels (such as cuBLAS, implemented by NVIDIA).

Figure 1 shows a graph comparing sequential CPU versus sequential GPU for the Australian dataset (from the LibSVM data archives). This data set only contains 54 features and roughly 690 data points, making it a good data set for quick tests and analyses.

To get a better sense of the CPU versus GPU comparison, we generated data that would allow us to control the number of features so we could better understand how more features impacted performace on the GPU and CPU implementations. Figure 2 shows a comparison of sequential CPU versus sequential GPU for data with 10,000 features while Figure 6 shows the same comparison but for data with 50,000 features.

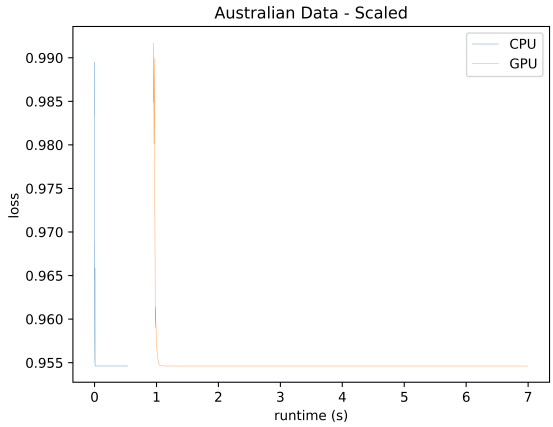

Figure 1: Comparison of training loss for sequential CPU and sequential GPU for the scaled Australian data set.

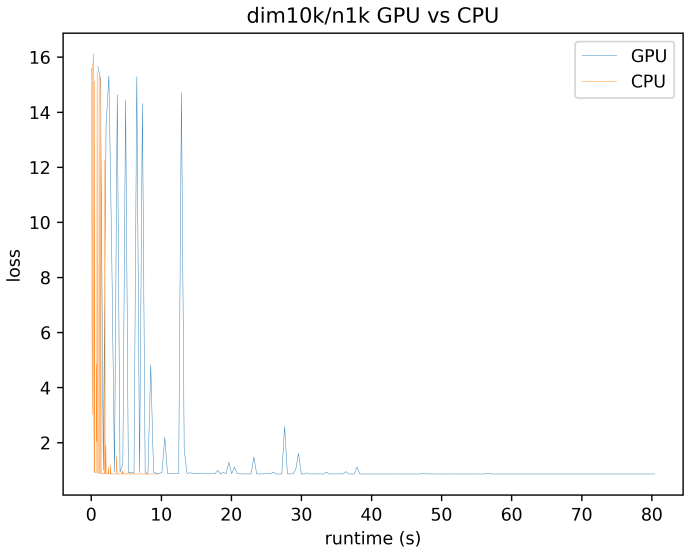

Figure 2: Comparison of training loss for generated data with 10,000 features. The algorithms used are sequential CPU and sequential GPU.

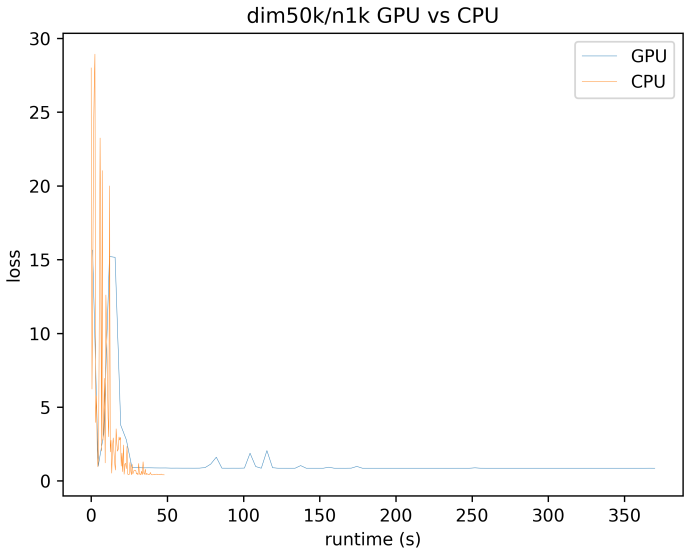

Figure 3: Comparison of training loss for generated data with 50,000 features. the algorithms used are sequential CPU and sequential GPU.

As expected, as the dimension of the feature space increases (Figure 6), we see an improvement in the sequential GPU version relative to the sequential CPU version. This is because the GPU implementation can more efficiently handle vector dot product as the dimension of the vector increases. are other limiting factors preventing us from achieving ideal performance.

7 Lessons Learned

We learned a lot about what can go wrong and what considerations need to be made when carrying out low-level implementations of machine learning algorithms. It was surprising how big of a difference some of these optimizations made (for example, memory allocations). If we were to start over (which would probably be a good idea if this work were to continue), we would make two major changes. First, we would create a data wrapper class to handle all the synchronization work in a more intelligent fashion. Second, we would take advantage of cuBLAS, a library of highly optimized CUDA kernels for various linear algebra operations. Not only are the cuBLAS kernels better optimized in terms of implementation, they also can better handle making thread and block allocations on the GPU. It was a mistake not using this from the beginning, but it was a lesson well-learned.

8 Conclusion

While our results were not what we were hoping for, we firmly believe there is promise in this approach. The failure of our implementation is more due to lack of low-level optimizations and smart memory usage than it is a failure of the approach in general. This is seen by looking at the improvements in the GPU algorithm runtimes for both sequential and distributed implementations as we increased the size of the mini-batch. We believe further work on this topic is merrited, both from theoretical and implementation point-of-views.

9 Future Work

We plan on continuing to complete the work to get more efficient implementation of GPU programming. In this report, we didn't successfully derive the theory of an asynchronous version of distributed SDCA, and we plan on doing it in the future work.