

Efficient Distributed Stochastic Dual Coordinate Ascent

Jeff Hajewski

Mingrui Liu

May 3, 2017

University of Iowa

Problem Overview

Problem Overview - The Primal Problem

Many machine learning problems can be formulated as the Regularized Finite Sum Minimization (RFSM) problem.

$$\min_{w \in \mathbb{R}^d} P(w) \tag{1}$$

where

$$P(w) = \frac{1}{n} \sum_{i=1}^n \phi(w^\top x_i, y_i) + \lambda g(w)$$

$$w, x_i \in \mathbb{R}^d, \text{ for } i = 1, \dots, n$$

$$y_i \in \mathbb{R}, \text{ for } i = 1, \dots, n$$

$\phi(z, y)$ is convex in z

$g(w)$ is convex in w

Problem Overview - The Dual Problem

The dual problem is given by

$$\max_{\alpha \in \mathbb{R}^n} D(\alpha) \quad (2)$$

where

$$D(\alpha) = \frac{1}{n} \sum_{i=1}^n -\phi_i^*(-\alpha_i) - \frac{\lambda}{2} \left\| \frac{1}{\lambda n} \sum_{i=1}^n \alpha_i x_i \right\|^2$$

$$x_i \in \mathbb{R}^d, \text{ for } i = 1, \dots, n$$

$$\alpha \in \mathbb{R}^n$$

$$\phi_i^*(u) = \max_z (zu - \phi_i(z))$$

The two key papers influencing our work are:

- Stochastic Dual Coordinate Ascent (SDCA)
[Shalev-Shwartz and Zhang, 2013]
- Distributed SDCA [Yang, 2013, Yang et al., 2013]

What is SDCA?

- **SDCA** - randomly pick a coordinate axis of $\alpha \in \mathbb{R}^n$, find update that best improves the objective
- **Distributed SDCA** - randomly pick k coordinate axes of $\alpha \in \mathbb{R}^n$, simultaneously find updates that best improve the objective (independently)

What if we ran SDCA on the GPU?

Two approaches:

- Naive approach: simply parallelize all tensor operations (e.g., dot product, matrix-vector multiplication, etc.)

Two approaches:

- Naive approach: simply parallelize all tensor operations (e.g., dot product, matrix-vector multiplication, etc.)
- Better: mimic the distributed approach by [Yang, 2013] on a GPU

Implementation

The key area of concern:

- Memory
 - Allocation

The key area of concern:

- Memory
 - Allocation
 - Communication (via PCIE bus rather than network)

The key area of concern:

- Memory
 - Allocation
 - Communication (via PCIE bus rather than network)
 - How can we cognitive load of writing this code?

Dealing with Memory Allocation

Naive approach:

```
void MemSync::PushToGpu(const vector &x) {  
    double *dx;  
    int n_bytes = sizeof(double) * x.size();  
  
    // Allocate GPU memory  
    cudaMalloc((double**) &dx, n_bytes);  
  
    // Copy data to GPU  
    cudaMemcpy(dx, &x[0], n_bytes,  
               cudaMemcpyHostToDevice);  
}
```

On a small dataset(200 points in \mathbb{R}^3) we hit over 200k allocations, which comprised nearly **95%** of the GPU compute time (≈ 13 seconds).

Can we do better?

Can we do better?

Yes!

Dealing with Memory Allocation

Better approach:

```
class MemSync {  
    // class code  
    static double *dx_;  
};  
  
void MemSync::PushToGpu(const vector &x) {  
    int n_bytes = sizeof(double) * x.size();  
  
    // Copy data to GPU  
    cudaMemcpy(MemSync::dx_, &x[0],  
               n_bytes, cudaMemcpyHostToDevice);  
}
```

The use of static class pointers reduced the 200k allocations down to only **3 memory allocations**, which comprised only **0.03%** compute time ($\approx 180\mu s$).

What about the cost of communication?

Copying data to and from the GPU is the next most expensive operation.

- About 50% of the compute time, or 768 ms (using the same toy dataset, after we have fixed the memory allocation issue)

Copying data to and from the GPU is the next most expensive operation.

- About 50% of the compute time, or 768 ms (using the same toy dataset, after we have fixed the memory allocation issue)
- Mostly unnecessary!

Communication Costs

Consider the following algorithm using the GPU:

$$1: \Delta\omega_i \leftarrow f(\mathbf{x}, \omega)$$

$$2: \omega_i \leftarrow \omega_i + \Delta\omega_i$$

To handle this efficiently we should:

Communication Costs

Consider the following algorithm using the GPU:

$$1: \Delta\omega_i \leftarrow f(\mathbf{x}, \omega)$$

$$2: \omega_i \leftarrow \omega_i + \Delta\omega_i$$

To handle this efficiently we should:

- Reuse ω in step 2 since we already moved it to the GPU for step 1

Communication Costs

Consider the following algorithm using the GPU:

$$1: \Delta\omega_i \leftarrow f(\mathbf{x}, \omega)$$

$$2: \omega_i \leftarrow \omega_i + \Delta\omega_i$$

To handle this efficiently we should:

- Reuse ω in step 2 since we already moved it to the GPU for step 1
- Perform step 2 on the GPU since the data is already there. No need to pull it off and then move it back to the GPU

The sad reality is this is quite complicated.

- Lots of book-keeping
- Are there edge cases?
- Need to watch out for memory leaks. Remember, **no** GC!

How can we handle this complexity?

Wrappers

We use wrappers (also known as decorators) to add additional functionality to our code. For example,

```
double VectorDotProduct(const vector &x,
                        const vector &y) {
#ifdef GPU
    return VectorDotProduct_gpu(x, y);
#else
    return VectorDotProduct_cpu(x, y);
#endif
}
```

To handle the flow of data from GPU to CPU, as well as book-keeping, we can use something like this:

```
class Data {  
    enum DataLocation { Gpu, Cpu };  
  
    // Pointer to GPU memory  
    std::unique_ptr<double> dx_  
    // Local reference (RAM)  
    Eigen::VectorXd x_  
    // DataLocation::Gpu or DataLocation::Cpu  
    DataLocation location_  
};
```

What about results?

We are still finalizing results. There is a lot of non-trivial structural code behind this.

- Loading data

We are still finalizing results. There is a lot of non-trivial structural code behind this.

- Loading data
- CUDA can be challenging

We are still finalizing results. There is a lot of non-trivial structural code behind this.

- Loading data
- CUDA can be challenging
- Algorithm

We are still finalizing results. There is a lot of non-trivial structural code behind this.

- Loading data
- CUDA can be challenging
- Algorithm
- Algorithm performance tracking

We are still finalizing results. There is a lot of non-trivial structural code behind this.

- Loading data
- CUDA can be challenging
- Algorithm
- Algorithm performance tracking
- Expect distributed SDCA to be the fastest, followed by CUDA accelerated SDCA, followed by SDCA



Bottou, L. (2010).

Large-scale machine learning with stochastic gradient descent.

In *Proceedings of COMPSTAT'2010*, pages 177–186.
Springer.



Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. (2011).

Distributed optimization and statistical learning via the alternating direction method of multipliers.

Foundations and Trends® in Machine Learning,
3(1):1–122.



Johnson, R. and Zhang, T. (2013).

Accelerating stochastic gradient descent using predictive variance reduction.

In *Advances in Neural Information Processing Systems*, pages 315–323.



Lian, X., Huang, Y., Li, Y., and Liu, J. (2015).

Asynchronous parallel stochastic gradient for nonconvex optimization.

In *Advances in Neural Information Processing Systems*, pages 2737–2745.



Nemirovski, A., Juditsky, A., Lan, G., and Shapiro, A. (2009).

Robust stochastic approximation approach to stochastic programming.

SIAM Journal on optimization, 19(4):1574–1609.



Shalev-Shwartz, S. and Zhang, T. (2013).

Stochastic dual coordinate ascent methods for regularized loss minimization.

Journal of Machine Learning Research, 14(Feb):567–599.



Shalev-Shwartz, S. and Zhang, T. (2014).

Accelerated proximal stochastic dual coordinate ascent for regularized loss minimization.

In *ICML*, pages 64–72.



Xiao, L. and Zhang, T. (2014).

A proximal stochastic gradient method with progressive variance reduction.

SIAM Journal on Optimization, 24(4):2057–2075.



Yang, T. (2013).

Trading computation for communication: Distributed stochastic dual coordinate ascent.

In *Advances in Neural Information Processing Systems*, pages 629–637.



Yang, T., Zhu, S., Jin, R., and Lin, Y. (2013).

Analysis of distributed stochastic dual coordinate ascent.

arXiv preprint arXiv:1312.1031.