Efficient Distributed Stochastic Dual Coordinate Ascent

Jeff Hajewski
Mingrui Liu

May 3, 2017

University of Iowa

Outline

Problem Overview

Implementation

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

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

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

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

$$\phi(z, y) \text{ is convex in } z$$

$$g(w) \text{ is convex in } w$$

Problem Overview - The Dual Problem

We consider the case when $g(w) = \frac{1}{2}\|w\|_2^2$, then the dual problem is given by

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

where

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

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

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

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

Related Work

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)

SDCA Algorithm

SDCA procedure:

- Let $w^{(0)} = w(\alpha^{(0)})$
- **Iterate**: for $t = 1, 2, \dots, T$
    - Randomly pick i
    - Find $\triangle\alpha_i$ to maximize $-\phi_i^*(-(\alpha^{(t-1)} + \triangle\alpha_i)) - \frac{\lambda n}{2}\|w^{(t-1)} + (\lambda n)^{-1}\triangle\alpha_i x_i\|^2$
    - $\alpha^{(t)} \leftarrow \alpha^{(t-1)} + \triangle\alpha_i e_i$
    - $w^{(t)} \leftarrow w^{(t-1)} + (\lambda n)^{-1}\triangle\alpha_i x_i$
- Output (Random option):

  Let $\bar{\alpha} = \alpha^{(t)}$ and $\bar{w} = w^{(t)}$ for some random $t \in T_0 + 1, \dots, T$

  Return $\bar{w}$

Remark: The red steps spend the largest proportion of computing resources.

What if we ran SDCA on the GPU?

Parallelizing SDCA

Two approaches:

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

Parallelizing SDCA

Two approaches:

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

Implementation

Main Points

The key area of concern:

- Memory
    - Allocation

Main Points

The key area of concern:

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

Main Points

The key area of concern:

- Memory
    - Allocation
    - Communication (via PCIE bus rather than network)
    - How can we reduce 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?

Communication Costs

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)

Communication Costs

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

Communication Costs

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
}
```

Wrappers

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?

Distributed SDCA

- We have a closed-form solution for $\Delta\alpha$

Distributed SDCA

- We have a closed-form solution for $\Delta\alpha$
- The computationally intensive task is $\omega^\top x_i$

Distributed SDCA

- We have a closed-form solution for $\Delta\alpha$
- The computationally intensive task is $\omega^\top x_i$
- This means for k coordinate axes, we want to compute $X\omega$, where $X = (x_{i_1}^\top x_{i_2}^\top \cdots x_{i_k}^\top)^\top$.

Distributed SDCA

- We have a closed-form solution for $\Delta\alpha$
- The computationally intensive task is $\omega^\top x_i$
- This means for k coordinate axes, we want to compute $X\omega$, where $X = (x_{i_1}^\top x_{i_2}^\top \cdots x_{i_k}^\top)^\top$.

We can reformulate the distributed version as matrix-vector multiplication. This is just an extension of the naive approach!

Distributed SDCA - The Challenge

So what is the challenge?

Distributed SDCA - The Challenge

So what is the challenge?

Our memory management class is poorly written

- X is a matrix now, not a vector

Distributed SDCA - The Challenge

So what is the challenge?
Our memory management class is poorly written

- X is a matrix now, not a vector
- However, we have only allocated memory for vectors

Results - Challenges

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

- Loading data

Results - Challenges

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

- Loading data
- CUDA can be challenging - communication is slowing us down

Results - Challenges

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

- Loading data
- CUDA can be challenging - communication is slowing us down
- Algorithm

Results - Challenges

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

- Loading data
- CUDA can be challenging - communication is slowing us down
- Algorithm
- Algorithm performance tracking

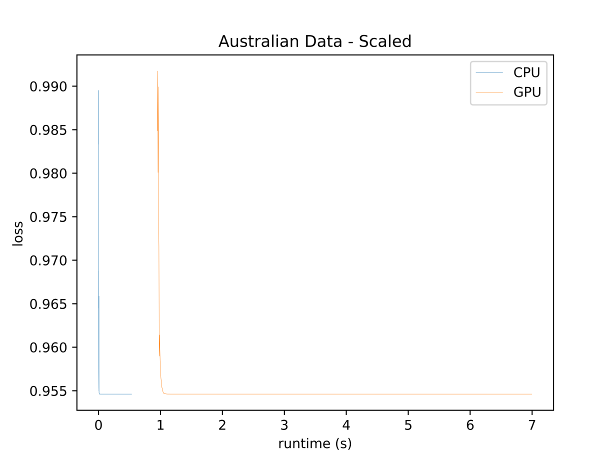
Results - Challenges

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

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

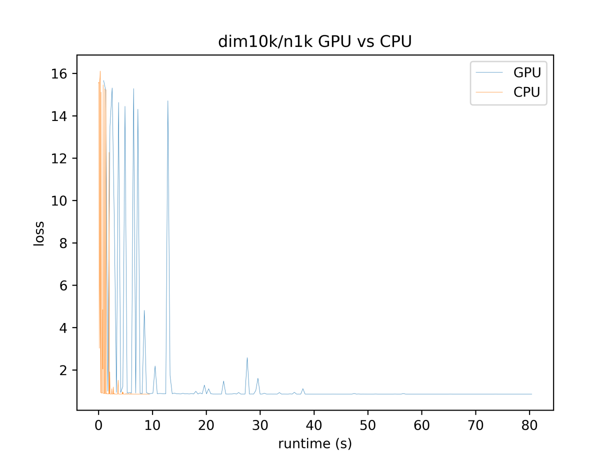
Results - Australian Dataset ($\mathbb{R}^{14}$, 690 points)

Naive approach



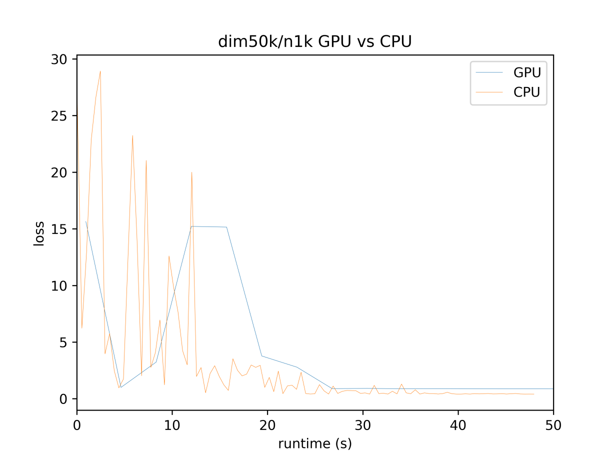
Results - Generated Dataset ($\mathbb{R}^{10k}$, 1000 points)

Naive approach



Results - Generated Dataset ($\mathbb{R}^{50k}$, 1000 points)

Naive approach



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.

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.