Acceleration of Sparse Vector Autoregressive Modeling using GPUs

Shreenivas Bharadwaj Venkataramanan
Computer Science and Engineering
University of California San Diego
La Jolla, California, US
vshreenivasbharadwaj@gmail.com

Rahul Garg
Indian Institute of Technology
Hauz Khas, New Delhi, INDIA
rahulgarg@cse.iitd.ac.in

Yogish Sabharwal
IBM Research - India
New Delhi, INDIA
ysabharwal@in.ibm.com

***Abstract*—Autoregressive modeling is a standard approach to mathematically describe the behavior of a time series. The vector autoregressive model (VAR) describes the behavior of multiple time series. The VAR modeling is a fundamental approach which has applications in multiple domains such as time series forecasting, Granger causality, system identification and stochastic control. Solving high dimensional VAR model requires the use of sparse regression techniques from machine learning. Efficient algorithms to solve the sparse regression problems are too slow to be useful in solving large high dimensional sparse VAR modeling problems. Earlier application of sparse VAR modeling in the neuroimaging domain required the use of the IBMs Blue Gene supercomputers. In this paper we describe an approach to accelerate large scale sparse VAR problems when solved using the lasso regression algorithm on state-of-the-art GPUs. Our accelerated implementation on NVIDIA GTX 1080 GPU takes a few seconds to solve the problem, reaching up to 4 TFLOPs of single-precision performance which is close to 55% of its peak matrix-multiply (GEMM) performance.**

I. Introduction

Autoregressive modeling is a standard mathematical approach to represent the behaviour of time series data [1], [2]. The autoregressive modeling approach has applications in multiple domains such as time series forecasting [3], prediction of economic data [4], modeling of gene expression dynamics [5] among many others. In many practical situations, we have the data from multiple time series [6]. In such cases, the time series may be represented as a vector autoregressive (VAR) model or a multi-variate autoregressive model [7]. The VAR modeling is a fundamental approach to represent the behaviour of multiple time series and has many applications [8], [9], [10], [11], [12], [13], [14], [15], [16], [17].

A fundamental problem in autoregressive modeling is that of model identification. The standard approach to estimate the model parameters in the case of autoregressive as well as VAR modeling uses the least square solution [1], [2], [6], [7]. The least square solution has several desirable properties. It has been shown to be the best linear unbiased estimator of the model parameters. It is very simple to compute, and in most of the cases, it gives good results.

With the popularity of big data analytics [18] and Internet of things [19], the amount of data has grown significantly. The dimensionality of observations as well as the number of observations have grown substantially. In order to solve the model identification problem for high dimensional data using the conventional approaches, the number of observations also need to grow linearly with the dimensionality of the data. This has restricted the applicability of VAR modeling in several newer domains involving high dimensional data.

Sparse VAR modeling [20], [21], [22], [23], [24] provides a new and promising approach to solve the model identification problem for high dimensional VAR models. The key assumption in a sparse VAR model is that the number of non-zero parameters in the model is small. This enables several machine learning approaches based on sparse regression to be effectively applied to solve the model identification problem.

The lasso regression [25], [26], which is a standard approach to enforce sparsity [26] has also been found effective in solving the sparse VAR model problems [21], [24]. In order to solve the sparse VAR model identification problem, one needs to solve N independent lasso regression problems (where N is the dimensionality of the data). The lasso regression problem is a quadratic programming problem in general. Efficient algorithms based on coordinate descent [27] and lasso modification of the least angle regression LARS [28] are available. The lasso modification to LARS [28], in addition to giving much better solutions also gives the complete path to the solution. Each point in the path is a solution for a different value of the parameter λ. This gives an additional advantage that the cross validation may be done using the path of the solutions produced without having to solve the original problem multiple times for different values of the parameter λ. The lasso modification to LARS [21] has been shown to give good results in the context of sparse VAR problems in neuroimaging [24], [29].

The time taken to solve one sparse VAR model identification problem in the domain of functional neuroimaging [24], using Matlab on a conventional server, ranges from 1 hour to 50 hours depending on the data size and parameter values. A conventional experiment in this domain may comprise 30-50 brain scans (requiring 30-50 sparse VAR model identification problems to be solved) while a bigger study may have 1000 to 1500 brain scans [30], [31]. The earlier work in this domain was only possible using supercomputers such as Blue Gene [21], [32]. Now with the advent of very fast GPUs, it is possible to significantly accelerate the computation time of the sparse VAR problem to acceptable levels.

In this paper we describe an approach to quickly solve the sparse VAR model identification problem using graphical processor units (GPUs). This approach achieves more than 650 times speedup on NVIDIA GTX 1080 GPU as compared to a Matlab implementation using MKL on a 4-core 3.5GHz Intel-Xeon based server. For the neuroimaging application, the model identification time now ranges from 3.5 seconds to 250 seconds, making it possible to solve large high-dimensional sparse VAR problems in reasonable amounts of time. The peak single precision performance obtained with this approach is 4.06 TFLOPS which is close to 50% of its theoretical peak performance. On other GPUs, the approach yields similar performance results relative to the GPUs peak performance.

Interestingly, although this problem is *embarassingly parallel* (EP) in nature, solving it as an EP problem on the GPU does not give much performance improvements. The performance is improved by synchronizing the loop iterations of multiple parallel problems and converting the matrix-vector multiplication operations (level 2 BLAS) into matrix-matrix multiplication operations (level 3 BLAS). Since each EP problem works with different active set of different sizes, doing this in synchrony is non-obvious and requires several other techniques to make it work efficiently.

One such technique is designed to efficiently handle different sizes of active sets in each iteration. Our optimization comes by grouping the models for the processing of certain kernels (such as Gram computations) having comparable active set sizes. This reduces the load imbalance amongst the models for these kernels.

Another key improvement comes from batching of kernels for different models to solve larger problems collectively. This reduces the overheads of invoking many kernels for small problem sizes and also overcomes the limitations in parallelism that arise due to the limited support for only a few streams on the GPUs.

To the best of our knowledge, this is the first work that accelerates the sparse VAR modeling problem using GPUs. There have been prior works that accelerate the lasso regression algorithm using parallel and distributed processing [33], [34], [35], [36]. Most of these works focus on single lasso regression problem, which is generally not as compute intensive as the sparse VAR modeling problem. The lasso algorithm is likely to be bound by the matrix-vector product operations which are level 2 BLAS operations limited by the performance of memory bandwidth. There have been other works that speed up generic sparse regression algorithms [37], [38], [39], [40] and are not applicable to the lasso regression problem.

The rest of the paper is organized as follows. In Section 2 more details about sparse VAR models and algorithms to solve the model identification problem. In Section 3, we present our approach approach to parallelize the high dimensional sparse VAR problem on GPUs. In Section 4, the performance results of implementing the proposed approach on two different GPUs are presented. Section 5 concludes the paper.

II. Sparse Vector Auroregressive Modeling

A autoregressive model of order k, represents a time series of observations $x(t)$ as:

$$x(t) = \sum_{\tau=1}^{k} a(\tau)x(t-\tau) + \eta(t) \text{ for all } t \in [1 \dots T]$$

where $x(t)$ represents the t^{th} observation of the time series, $a(\tau), \tau \in [1 \dots k]$ represent the coefficients of the order-k autoregressive model and $\eta(t)$ represents the uncorrelated noise at time t in the model. Given the time series data $x(t)$, the model identification problem is to reliably estimate the model parameters $a(\tau), \tau \in [1 \dots k]$.

A. Vector Autoregressive (VAR) Models

A time series with multiple observations is modeled as a vector autoregressive model. Mathematically,

$$X(t) = \sum_{\tau=1}^{k} A(\tau)X(t-\tau) + \eta(t) \tag{1}$$

where $X(t) \in \mathcal{R}^N$ represents the vector of N observations at time t, $A(\tau) \in \mathcal{R}^{N \times N}$ represents the k matrices of model coefficients for $\tau \in [1 \dots k]$ and $\eta(\tau)$ represents the N-dimensional vector of temporally uncorrelated stationary noise at time t.

The model identification problem in the context of VAR models is to estimate the coefficients of the $N \times N$ model matrices $A(\tau), \tau \in [1 \dots k]$, using the observations $X(t), t \in [1 \dots T]$.

Given the model parameters $A(\tau)$, the values of next observations of $X(t)$ can be predicted using its past k observations as:

$$\hat{X}(t) = \sum_{\tau=1}^{k} A(\tau)X(t-\tau)$$

The least square solution chooses the model parameters A that minimize the sum of square of the forecast error. The forecast error is mathematically given as:

$$e(t) = X(t) - \hat{X}(t) = X(t) - \sum_{\tau=1}^{k} A(\tau)X(t-\tau)$$

Formally, the least square solution is given by:

$$\min_{A} \sum_{t=k+1}^{T} \left\| X(t) - \sum_{\tau=1}^{k} A(\tau)X(t-\tau) \right\|_2^2 \tag{2}$$

where $\|v\|_2^2$ denotes the square of the $\mathcal{L}_2$ norm of the vector v, which is the same as the sum of square of all its components.

B. High Dimensional VAR Models and Sparsity

The least squares solution of Eq. (2) has kN^2 unknown model parameters and $N(T-k)$ prediction errors which need to be minimized. When $kN^2 \geq N(T-k)$ then the system becomes underdetermined and the prediction error becomes zero. This is the case of overfitting to the data which has very little or no predictive value. The model can be solved reliably only when $kN^2 \ll N(T-k)$, i.e., the number of time points for which the system is observed is significantly larger than the dimensionality of the data.

A newer approach to solve high-dimensional VAR modeling problem has been proposed in the context of neuroimaging data analysis [21], [24], [29]. The key assumption in the approach is that of sparsity in the VAR models. The model coefficients $A_{ij}(\tau)$ represent the influence of time series X_j in predicting the future value of time series X_i.

A non-zero value of A_{ij} represents an interaction between two time series. The sparse VAR model assumes that such interactions are very few as compared to the total number of possible kN^2 interactions. The VAR model identification problem problem using the least squares is decomposed into N independent regression problems as follows:

$$\text{For all } i, \min_{A_{i,\cdot}} \sum_{t=k+1}^{T} \left\| X_i(t) - \sum_{\tau=1}^{k} A_{i,\cdot}(\tau) X(t-\tau) \right\|_2^2$$

where $A_i, .(\tau)$ represents the i^{th} row vector of the model matrix $A(\tau)$. In case of the sparse VAR problem, instead of solving the above N linear regression problems using the ordinary least squares technique (OLS) [7], the lasso regression [25] is used to solve each of the problems independently. The lasso regression uses $\mathcal{L}_1$ penalty to enforce sparsity. So the equivalent lasso regression problems become: For all i,

$$\min_{A_{i,\cdot}} \sum_{t=k+1}^{T} \left\| X_i(t) - \sum_{tau=1}^{k} A_{i,\cdot}(\tau) X(t-\tau) \right\|_2^2 + \lambda \sum_{j=1}^{N} \sum_{\tau=1}^{k} |A_{ij}(\tau)| \quad (3)$$

For the neuroimaging application[1] [21], [24], it is sufficient to solve for model of order 1[2] (i.e., $k=1$). Thus the equivalent problem (3) may be rewritten as

$$\min_{\beta^{(i)} \in \mathcal{R}^N} \left\| y^{(i)} - W\beta^{(i)} \right\|_2^2 + \lambda \left\| \beta^{(i)} \right\|_1 \quad (4)$$

where $y^{(i)}$ is a $T-1$ dimensional vector given as $y(i) = X_i(2...T)$, W is a $T-1 \times N$ dimensional matrix given as $W = X(2 \ldots T)$, and $\beta^{(i)}$ is an N dimensional vector given by $\beta^{(i)} = A_i, .(1)$. This amounts to solving N lasso problems for different values of the vector $y^{(i)}$, but the same matrix W and the same value of the parameter λ.

In the context of fMRI imaging, typically N ranges from 20,000 to 100,000 and T ranges from 200 to 1000. For these sizes, solving the N lasso problems on a state-of-the art server may require 1 to 50 hours per brain, depending on the value of the parameter λ. This necessitates the need to solve this problem more efficiently.

[1]For the neuroimaging application, we have an additional constraint $A_{ii}(1) = 0$ for all i, which can be incorporated easily in the algorithm, by setting the corresponding correlations to zero in each iteration.

[2]The proposed approach is not restricted to model order 1, but works for general model order. For model order k, the size of W matrix becomes $(T-k) \times kN$ and the size of β becomes kN.

Algorithm 1 `Lasso-LARS`: high level overview

```
Input: $W \in \mathcal{R}^{T\times N}, y \in \mathcal{R}^T, \lambda \in "\mathcal{R}^+$
Output: $\beta \in \mathcal{R}^N$ that minimizes $||y - W\beta||_2^2 + \lambda||\beta||_1$
Initialization: $\mathcal{A} = \phi, \mathcal{I} = \{1 \ldots N\}, \beta = 0, \mu = 0$
Normalize $y$ and columns of $W$

/* S1: Process while not solved */
while (not done) do

    /* S2: Update residue and compute inner product
    of residue with columns of matrix $W$ */
    $r = y - \mu$                                              (S2.1)
    $c = W^T r$                                                (S2.2)

    /* S3: Update active set using max inner product */
    $c_{max} = \max_{j\in\mathcal{I}}|c_j|$                    (S3.1)
    $c_{index} = \arg\max_{j\in\mathcal{I}}|c_j|$              (S3.2)
    if(variable not dropped) then
        $\mathcal{A} = \mathcal{A} \cup \{c_{index}\}$         (S3.3)
    end if

    /* S4: Find projection on active set */
    $R = W_{.,\mathcal{A}}$                                    (S4.1)
    $\beta^p = (R^T R)^{-1} R^T y$                             (S4.2)
    $d = R * \beta^p - \mu$                                    (S4.3)

    /* S5: Find step size */
    /* First compute LARS step size $\gamma_1$ */
    $c^d = W^T d$                                              (S5.1)
    $\gamma_1 = \min^+_{j\in\mathcal{I}} \left\{ \frac{c_j - c_{max}}{c_j^d - c_{max}}, \frac{c_j + c_{max}}{c_j^d + c_{max}} \right\}$   (S5.2)
    /* Compute Lasso step size $\gamma_2$ */
    $\gamma_2 = \min^+_{j\in\mathcal{A}-\{c_{index}\}} \left\{ \frac{\beta_j}{\beta_j - \beta_j^p} \right\}$   (S5.3)
    $v = \arg\min^+_{j\in\mathcal{A}-\{c_{index}\}} \left\{ \frac{\beta_j}{\beta_j - \beta_j^p} \right\}$   (S5.4)
    $\gamma = \min(\gamma_1, \gamma_2)$                        (S5.5)

    /* S6: Update $\beta$, $\mu$ and inactive set */
    $\beta = \beta + \gamma * (\beta^p - \beta)$               (S6.1)
    $\mu = \mu + r.d$                                          (S6.2)
    if $\gamma_2 < \gamma_1$ then
        $\mathcal{I} = \mathcal{I} \cup \{v\}$                 (S6.3)
    end if

    /* S7: Check for terminating condition */
    continue while $\Delta||\beta||_1\lambda + \Delta||y - W\beta||_2^2 \leq 0$   (S7.1)

end while
```

C. Overview of the lasso modification to LARS

Algorithm 1 outlines the main steps in the algorithm. It takes $y \in \mathcal{R}^T$, $W \in \mathcal{R}^{T\times N}$ and $\lambda \in \mathcal{R}^+$ as its input and produces $\beta \in \mathcal{R}^N$ that minimizes $\|y - W\beta\|_2^2 + \lambda\|\beta\|_1$.

The algorithm initially normalizes y and columns of the matrix W such that they have zero mean and unit variance. It maintains an *active set* of columns of W for which the corresponding values of β are non-zero. At any stage in the algorithm, the columns of W in the active set have the smallest angle with the residue vector r (i.e., the absolute values of the inner product of r with the active set of columns of W are identical).

The algorithm begins by adding the column of W subtending the smallest angle with the vector y (given by the largest absolute inner product in steps S2 and S3). At any general step in the algorithm, the residual vector $r = y - W\beta$, is projected on to the active set of columns of W (step S4). However, the solution β is not updated all the way to the projection, but is updated incrementally till one of the two conditions become satisfied. The LARS condition (steps S5.1 and S5.2), checks if any other column of W becomes *equiangular* to the residue vector obtained after updating β. If such a column is found, then it is added to the active set (steps S2, S3). The lasso condition (steps S5.3, S5.4) checks if any of the βs corresponding to the active set may become zero after the update. The corresponding column is then removed from the active set (step S6.3) and the projections are recomputed (steps S2, S3, S4). The current solution β is updated in step S6 and the terminating condition is checked in step S7.

III. Parallelization of Sparse VAR Modeling

We start by describing an embarrassingly parallel scheme for the GPU; we call this the `GPU-EP` implementation. We analyze the performance of this scheme and identify the major bottlenecks. We then present optimized algorithms designed to tackle these bottlenecks.

A. An embarrassingly parallel GPU scheme

The main idea behind this GPU parallelization scheme is that the processing of all the models is independent. Therefore, we can design an embarrassingly parallel algorithm that solves the models in parallel, subject to the number of models that can be loaded into the GPU memory.

Algorithm 2, named `GPU-EP`, provides a high level pseudocode for this algorithm. We maintain a queue, called the model queue, of all the unsolved models. We determine the maximum number of models that can fit in the GPU memory and launch as many threads. Each thread works on one model at a time. Note that the threads take different amounts of time to determine the solutions; this is because the final active set sizes may be different for each model and hence a different number of iterations of Algorithm 1 are executed by the threads. When a thread finishes the processing of its model, the model is replaced with a new model from the model queue. The queue handling is accomplished via atomics. The new model is then copied from the host memory to the GPU memory.

As expected, the model processing is dominated by matrix and vector operations. In order to effectively use threads on the GPU, we use cuBLAS kernels for performing these operations within each thread. These kernels are launched as child kernels thereby exploiting dynamic parallelism capabilities on the GPU with the launch of additional threads and also exploiting the GPU optimized code of cuBLAS.

Algorithm 2 `GPU-EP`: high level pseudocode

```
On every GPU:
    while $\exists$ an unsolved model in the model queue do
        Copy next model from queue to GPU memory
        /* Process the model */
        Execute Steps S1-S7 of Algorithm 1
    end while
```

Profile Analysis: In order to identify the bottlenecks in the `GPU-EP` implementation, we profiled the code in order to determine the most time consuming kernels. Our analysis shows that the matrix-vector products involved in steps S2.2, S5.1, S4.2 and S4.3 account for more than 90% of the execution time.

B. Collating Matrix-Vector products

The profile analysis of the `GPU-EP` code suggests that the matrix-vector product kernels that in turn invoke the cuBLAS GEMV calls are the most time consuming kernels. We note that, of these, the size of the matrix-vector products in Steps 2.2 and 5.1 are substantially larger than the other matrix-vector products. These two GEMV kernels together account for over 90% of the total time. The first matrix vector product computes the inner-product of the residue with the columns of matrix W in step 2.2. The second matrix-vector product computes the inner-product of the projection with the columns of matrix W in step 5.1. Note that even though each model maintains its own active set and the computation of each model is independent, the matrix, W, used in all the GEMV calls of step S2.2 is the same. Similarly, the matrix W used in all the GEMV calls of step S5.1 is also the same. This is not true of the other matrix vector products involved in steps S4.2 and S4.3.

It is well known that GEMV being a BLAS 2 routine, is memory bandwidth bound and does not exploit the full computational capacity of the GPU. Therefore we can synchronize all the models to perform steps S2.2 and S5.1 in tandem allowing us to replace the GEMV calls by GEMM calls. However, since each model has a different active set (having different size), the time taken in performing other operations in every iteration varies for every model. Therefore, there is a trade-off in synchronizing the models to perform matrix matrix multiply in steps S2.2 and S5.1 as different amount of work done by different models for the other steps leads to load imbalance before the synchronization.

To analyze the projected benefits of using GEMM instead of GEMV, we measure the performance of these two routines for the problem sizes arising in our application. We vary the number of models m from 128 to 2048 - this corresponds to the number of models being processed in parallel at any point of time. For both GEMV and GEMM we take the size of the first matrix to be $40,000 \times 300$. This is approximately the size of our W matrix. For GEMV, we take the size of the vector to be 300×1 and we process all the m GEMVs in parallel. For GEMM, we take the size of the second matrix to be $300 \times m$ (m being the model size) and we perform only a single matrix multiply. The results of our analysis are shown in Table I.

Number of models (m)	Time taken by GEMV (s)	Time Taken by GEMM (s)
128	29.54	0.51
256	59.00	0.99
512	118.03	1.90
1024	236.04	3.42
2048	471.94	6.74

TABLE I
GEMM AND GEMV PERFORMANCE COMPARISON

These results indicate that the GEMM performance is significantly better (factor 50 or more) compared to GEMV performance for the problem sizes of interest. Moreover, we know that more than 90% of the time is spent in the GEMV calls. Therefore the gains due to collating the matrix-vector products are expected to outweigh the performance loss due to load imbalance.

In order to synchronize the threads to execute GEMM collectively, we redesign our algorithm to orchestrate the execution of the iterations of all the models on the GPU. Algorithm 3 provides a high level pseudocode for this algorithm. It works as follows. A queue of unsolved models is maintained on the host as before. Let m be the number of models currently being processed by the GPU. The data for these models is loaded on to the GPU memory. The host executes the algorithm iteratively. In each iteration, it launches GPU kernels to perform one iteration of algorithm 1 (Steps S2-S7) for each of the m models residing on the GPU. For all sub-steps other than S2.2 and S5.1, this is done by invoking one GPU kernel for each of the m models. These GPU kernels are equally distributed amongst the streams available on the GPU. For sub-steps S2.2 and S5.1, the cuBLAS GEMM kernel is directly invoked to perform the collated matrix-multiply for all the models. At the end of each iteration, the host checks if any of the models has completed, i.e., a solution has been found for the model. All such models are replaced with unsolved models from the model queue. In case the model queue is empty, the model is replaced by a dummy model. We call this algorithm, that incorporates matrix-vector products into the `GPU-EP` code, as `GPU-MVColl`.

Algorithm 3 `GPU-MVColl`: high level pseudocode

```
On Host:
    Copy $m$ models to GPU memory
    while all the models are not completed do
        /* Process the m models */
        for each sub-step Sx.y of S2-S7 in Algorithm 1 do
            if sub-step is S2.2 then
                Invoke GPU cuBLAS GEMM kernel to
                compute $c = W^T r$
            else
                if sub-step is S5.1 then
                    Invoke GPU cuBLAS GEMM kernel to
                    compute $c^d = W^T d$
                else
                    for each model $i$ in $[1, m]$ do
                        Invoke GPU-Kernel for sub-step
                        Sx.y and assign to stream
                        $i$ % max-streams
                    end for
                end if
            end if
        end for

        for all completed models do
            if all the models are not completed then
                Copy next model from model queue
                to GPU memory.
            end if
        end for
    end while
```

C. Batched Kernels

We profile the code `GPU-MVColl` again after incorporating the optimizations for Matrix-Vector products as explained in the previous section. The profile, presented in Table 2, shows the statistics for the 7 most time consuming kernels in the `GPU-MVColl` implementation. The profile indicates that after optimizing the matrix-vector products, more than 60% of the total time is spent on kernels that are memory bound (based on vector and matrix-vector operations). We further analyze the performance of these kernels. We calculate the aggregate performance (in GFLOPs) obtained for these kernels (in `GPU-MVColl`). This aggregate performance is reported in Table 2 when run with 512 threads. We also analyzed the performance for these kernels for large problem sizes; this is reported in Table 2 as peak performance. We note that the performance we obtain is much lower in comparison to the peak performance in some cases. There are several reasons for this. Firstly, as the active set lies in the range of 1-31 with an average of 5, some of the problem sizes are quite skewed. Secondly, each model launches a separate kernel. Therefore there are a lot of kernel invocations to solve small problems. Each kernel call has its overheads and these become prominent for small problem sizes. Thirdly, even though as many kernels are launched as the number of models, only a limited number of kernels may end up executing in parallel depending on the number of parallel streams supported by the GPU (for instance the GTX 1080 only supports 16 threads). Therefore the limited number of kernels executing in parallel do not efficiently utilize the GPU resources.

We therefore design batched versions of these kernels. Instead of invoking separate kernels to solve for each model, we directly perform the operations (listed in Table 2) for all the m models collectively using a large number of threads propor-

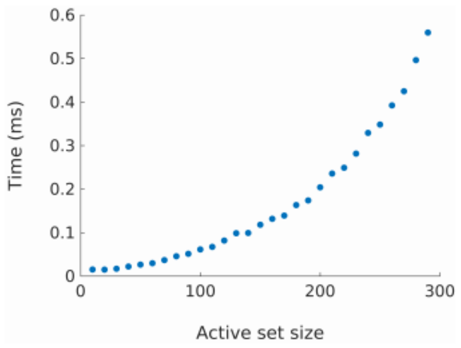

Fig. 1. Analysis of time per iteration vs active set size

tional to the size of the vectors. We call this implementation, that incorporates batched kernels in to the `GPU-MVColl` code, as `GPU-Batched`.

D. *Grouping by active set size*

Our profile analysis of `GPU-Batched` shows that about 24% of the time is spent in the Gram computations of step S4.2; the amount of computation in this step is dependent on the size of the active set of the model. Though the load imbalance amongst threads working on models with different active set sizes was not prominent in the `GPU-EP` implementation, this imbalance starts to show with all the optimizations performed in `GPU-Batched`. To see this, we analyze the time taken per iteration for models with different active set sizes for `GPU-Batched`. The scatter plot in Figure 1 plots the active set size vs. time taken for an iteration.

The plot shows that there is significant variation in the iteration execution time depending on the active set size. Therefore, in an effort to resolve this load imbalance for the Gram computations in step S4.2, we partition the models in each iteration into groups based on their active set sizes. We perform the Gram computations of step S4.2 in phases, once for each group of models with the same active set size by invoking the corresponding cuBLAS kernels. We call this implementation as `GPU-Opt`; this version incorporates the active set grouping for computing Gram optimization on the `GPU-Batched` version.

IV. Experimental Results

In this Section, we provide results from a comprehensive set of experiments performed with our optimized code for solving high-dimensional sparse VAR model identification problem using Lasso-LARS. We start with a description of our experimental setup.

A. *Experimental Setup*

In this section, we describe hardware platforms and the application data from the fMRI imaging domain that we used in conducting our experiments.

Application Data: The optimized GPU implementation for sparse VAR model identification problem was motivated by application in the fMRI imaging domain [20], [21], [24]. One functional brain scan from the 1000 functional connectomes project [30] was taken. The fMRI data was preprocessed and flattenned to a matrix of size 295x39658 (representing 295 time points and 39658 brain voxels). Most of the results were reported on this data.

Hardware Platforms: We used three hardware platforms for performance comparisons. In order to obtain a baseline assessment, we used a server grade system based on 4-core 3.5 GHz Intel Xeon E5-1620 processor with 32 GB RAM. We used a low-end GPU configuration NVIDIA 940mx with 4GB RAM that is commonly available on desktop or laptop environments. Finally we used a high end GPU configuration of NVIDIA GTX 1080 with 8GB RAM attached to the Intel Xeon system.

B. *Performance with different optimizations*

We first study how our different optimizations affect the performance of the Lasso-LARS algorithm and how well our optimized code performs on different systems.

Profile of Key Kernels: In order to analyze the performance with different versions of our code optimizations, we divide the kernels into three groups. The first group, G1, is the matrix-vector products of steps S2.2 and S5.1 - these are the kernels that were converted to GEMM calls in `GPU-MVColl`. The second group, G2, is the kernels corresponding to vector and matrix-vector operations of steps S3.1, S5.2, S5.3, S7.1, S4.2 and S4.3 as listed in Table 2 - these are the kernels that we batched in `GPU-Batched`. All the remaining kernels are clubbed into group G3.

The time taken by these group of kernels is shown in Figure 2 for different code optimizations, namely, `GPU-EP`, `GPU-MVColl`, `GPU-Batched` and `GPU-Opt`. In Figure 2 (a), we can see that the group of kernels G1 take substantial percentage of the total time in `GPU-EP` whereas they only take a very small percentage of the total time in `GPU-MVColl`. The time of the remaining kernels remains the same. This shows that collating matrix-vector products into matrix-matrix products significantly improves the performance of the algorithm. In Figure 2 (b), we compare the timing of these groups of kernels for `GPU-MVColl`, `GPU-Batched` and `GPU-Opt` (these are plotted in a separate chart for the sake of visual clarity as the time taken by `GPU-EP` is quite large). We note that the performance of the G2 kernels improves substantially in the `GPU-Batched` version over `GPU-MVColl` whereas the performance of the other group of kernels remains largely the same. This is primarily due to the batching of G2 kernels in the `GPU-Batched` version. Finally, we note that the performance of the G3 kernels improves in the `GPU-Opt` version over `GPU-Batched` version while the performance of the other group of kernels remains largely the same. This is due to the grouping of models by active set size in `GPU-Opt` that improves the performance of the Gram computations.

Kernel	Time (%)	Measured GFLOPs	Peak GFLOPs
S3.1: $c_{max} = \max_{j\in\mathcal{I}} \lvert c_j \rvert$	9.5%	14	21
S4.2: Matrix-Vector products	4.5%	1.14	86.1
S4.3: Matrix-Vector product	5.4%	0.95	86.1
S5.2: $\gamma_1 = \min^{+}_{j\in\mathcal{I}} \left\{ \frac{c_j - c_{max}}{c_j^d - c_{max}}, \frac{c_j + c_{max}}{c_j^d + c_{max}} \right\}$	9.7%	48	65
S5.3: $\gamma_2 = \min^{+}_{j\in\mathcal{A}-\{c_{index}\}} \left\{ \frac{\beta_j}{\beta_j - \beta_j^p} \right\}$	11.5%	0.001	6
S7.1: $\Delta \lvert\lvert y - W\beta \rvert\rvert_2^2$	9.5%	0.15	30
S7.1: $\Delta \lvert\lvert \beta \rvert\rvert_1 \lambda$	9.5%	14.21	20

TABLE II
PROFILE OF `GPU-MVColl` IMPLEMENTATION: MOST TIME CONSUMING KERNELS

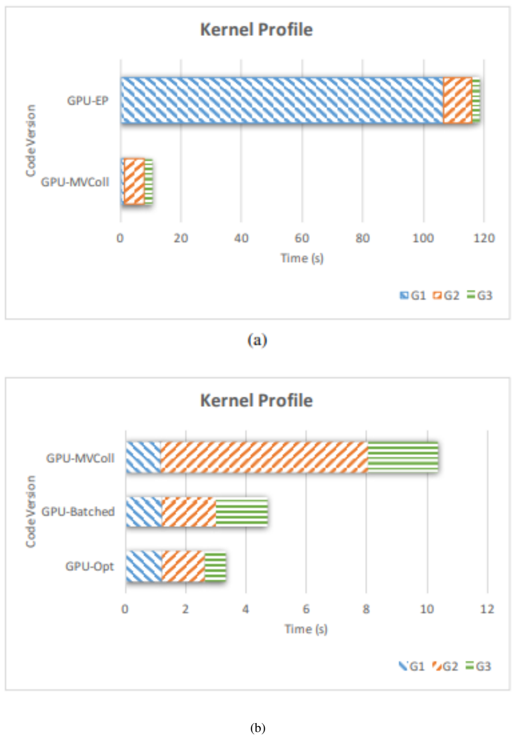

Fig. 2. Performance of key kernels with different optimizations

Performance on different systems: We next analyze the performance of our optimizations on different systems. We ran a MATLAB version of the code on the server grade 4-core Intel Xeon E5-1620 system using the Intel high performance computing MKL library. For double precision, this code took over 1 hour to solve all the models performing at a rate of 2.2 GFLOPs. For single precision, it took over 45 minutes to solve the models performing at a rate of 3.8 GFLOPs. We also computed the peak matrix multiply performance on the server (by multiplying large matrices of size 10000×10000). The peak GEMM performance obtained was 125 GFLOPs for double precision and 225 GFLOPs for single precision.

We next measured the performance on the 1080 and 940mx GPUs. The single precision as well as double precision performance of different versions of the code, namely, `GPU-EP`, `GPU-MVColl`, `GPU-Batched` and `GPU-Opt` on both these GPUs is presented in Table 3. To put our performance in context, we also measured the peak GEMM (matrix-multiply) performance on these GPUs. These are also presented in the table.

Code Version	Single Precision		Double Precision	
	1080	940mx	1080	940mx
`GPU-EP`	70	5	9	0.48
`GPU-MVColl`	810	165	212	23
`GPU-Batched`	1779	192	242	23.5
`GPU-Opt`	2534	205	233	22
Peak GEMM	7350	400	280	27

TABLE III
PERFORMANCE ON DIFFERENT SYSTEMS (GFLOPS)

The `GPU-Opt` version of the single precision code gave a performance of 2.5 Teraflops and finished processing all the models in mere 3.5 seconds. This performance is about 35% of the peak GEMM performance on the GPU. This code is 650x faster than the Matlab implementation and 35x faster than the embarrassingly parallel `GPU-EP` implementation. Even on the lower end 940mx, the `GPU-Opt` code achieved 205 GFLOPs, finishing the models in 41 seconds; this performance is within 50% of the peak GEMM performance on this GPU.

We also modified our implementation to support double precision and measured the performance on these GPUs. As can be seen the `GPU-Opt` code achieved 233 GFLOPs on the 1080 and 22 GFLOPs on the 940mx. In both cases, it performed at more than 80% of the peak GEMM performance. We also observe that `GPU-Batched` actually performs better for the case of double precision than `GPU-Opt`. The double precision performance of the GPUs is considerably lower than the single precision performance and hence the load imbalance due to different sized active sets does not play a significant role in case of double precision.

Varying the number of models: We measure the performance of our `GPU-Opt` algorithm by varying the number of models, m, that are processed in any iteration on the GPU. The results are shown in Figure 3. We see that in general the performance improves with increasing number of models. The performance peaks when $m = 512$ models are processed every iteration and then drops slightly when the number of models is increased further.

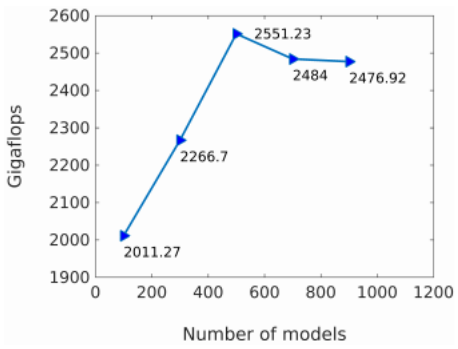

Fig. 3. Performance of `GPU-Opt` with varying number of models

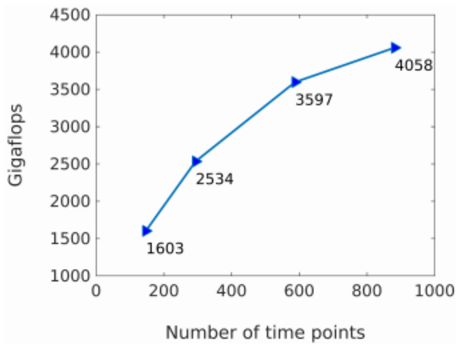

Fig. 4. Performance of `GPU-Opt` with varying number of time points

Stability runs: We next check how stable the performance results are. For this we ran the `GPU-Opt` code 100 times. We measured the time taken by the top 5 time-consuming kernels as well as the time taken by the complete code. We report the mean and standard deviations for these across the runs in Table 4. We observe that the standard deviation of these kernels is no more than 5%. The standard deviation of the complete `GPU-Opt` opt code is just within 2% of the mean. We therefore conclude that the `GPU-Opt` code is very stable - there is little variation in performance across runs.

Kernel	Mean (s)	Std dev
S2.2: $c = W^T r$	0.60	0.002
S5.1: $c^d = W^T d$	0.60	0.002
S3.1: $c_{max} = \max_{j \in \mathcal{I}} \lvert c_j \rvert$	0.35	0.0016
S5.2: $\gamma_1 = \min^+_{j \in \mathcal{I}} \left\{ \frac{c_j - c_{max}}{c^d_j - c_{max}}, \frac{c_j + c_{max}}{c^d_j + c_{max}} \right\}$	0.47	0.002
S7.1: $\Delta \lvert\lvert \beta \rvert\rvert_1 \lambda$	0.33	0.001
`GPU-Opt`	3.39	0.064

TABLE IV
STABILITY OF `GPU-Opt`

C. *Effect of varying VAR parameters*

As problems from different domains will have different VAR parameters, we evaluate the performance of our algorithm by varying these parameters in this section.

Varying the number of time points: Next, we measure the performance of our `GPU-Opt` algorithm by varying the number of time points. Note that this affects the size of the problems being solved through the value of T. The results are shown in Figure 4. We see that as the number of time points is increased, the problem size increases and the algorithm gives better performance. The time spent by the algorithm in the G1-GEMM kernel increases from 26% for T = 147 to 52% for T = 882. The GEMM kernel gives performance in the range of 6-7.6 TFLOPS leading to an overall improvement in the performance.

Varying the number of voxels: Next, we measure the performance of our `GPU-Opt` algorithm by varying the number of voxels. Note that this affects the size of the problems being solved through the value of N. The results are shown in Figure 5. We again see that performance of the algorithm improves with increase in the problem size. Again, increasing the problem size results in more time being spent in the GEMM kernel which operates at much higher floating point performance leading to an improvement in the overall floating point performance.

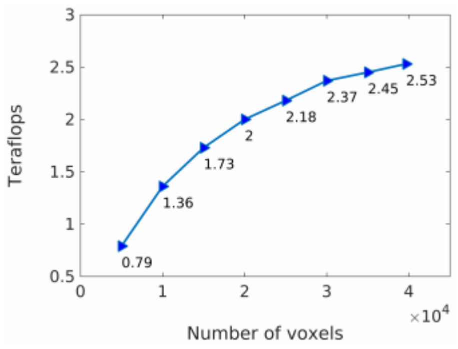

Fig. 5. Performance of `GPU-Opt` with varying number of voxels

Impact of Sparsity: The sparsity of the model can be adjusted by changing the regularization parameter λ. As the value of λ is decreased there is less penalty on selecting more columns of W in the active set. Hence the active set size increases and the number of iteration to obtain the final solution also increases. Table 5 shows the impact of the parameter λ on the size of active set, overall performance, total number of floating operations performed and the time spend in various kernel groups.

As λ is decreased from 0.4 to 0.15, the average active set size increases from 5.02 to 106.58. Total number of floating point operations also go through a nearly 30x increase. The overall floating point performance decreases from 2.54 to 1.09 TFLOPS. Notice that with increase in the active set sizes, the fraction of time spent on G3 kernels increases from 21% to 65%. This increase is primarily due to increase in time taken to find projection onto the active set (steps S4.2 and S4.3). Since the size of the matrix R is generally small, this step operates between 15-20 GFLOPS thereby leading to a reduction in the overall performance.

The average active set size for $\lambda = 0.15$ is 106.58. Smaller values of λ, are not likely to give meaningful results since $T = 295$ for this experiment. When the active set size becomes equal to 295, problem will become underdetermined leading to overfitting. The performance for $\lambda = 0.15$ is reasonable and building the model for the full brain takes 275 seconds.

λ value	0.40	0.35	0.25	0.20	0.15
Average active set size	5.02	7.65	21.71	42.12	106.58
Performance (GFLOPS)	2540	2480	2347	2039	1087
FLOP (x1012)	10.47	16.08	48.85	102.08	299.51
Time (G1)	1.48	2.29	6.90	14.35	39.86
Time (G2)	1.75	2.72	8.32	17.74	57.19
Time (G3)	0.88	1.45	5.52	17.81	177.57

TABLE V
IMPACT OF SPARSITY ON PERFORMANCE

V. CONCLUSION

Solving large-scale high-dimensional vector autoregressive (VAR) modeling problems has applications in neuroimaging [20], [24] and other domains [22]. This is a computationally challenging problem that requires the use of supercomputers. Modern Graphical Processing Units (GPUs) offer significant computational performance at a fraction of the cost of supercomputers. In this paper we presented an approach to speed up the solution to large scale high dimensional sparse VAR modeling problems on GPUs.

Although the problem appears to be embarrassingly parallel (EP), solving iterations of multiple sub-problems in lock-step synchronization enables one to replace matrix vector product operations (level2 BLAS) with matrix-matrix product operations (level3 BLAS), thereby giving very good speedups. Batching of kernels of different sub-problems improves the performance further. Finally, grouping certain kernels according to the *active set* sizes leads to the final gains in the performance.

On the NVIDIA GTX 1080 GPU, our implementation gives up to 4.05 TFLOPS single precision performance which is close to 50% of the theoretical peak GPU performance. On a laptop GPU such as NVIDIA 940mx, the implementation achieves 205 GFLOPS of single precision performance, which is again close to 50% of its theoretical peak performance.

The use of GPUs and efficient parallelization techniques have made it possible to solve large scale sparse VAR problems in neuroimaging and other domains, on an inexpensive desktop. This is likely to spur the development of more applications making use of sparse high-dimensional VAR models.

ACKNOWLEDGMENTS

The authors would like to thank Dushyant Sahoo for sharing his Matlab implementation of the sparse VAR modeling code.

REFERENCES

[1] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, Time series analysis: Forecasting and control. John Wiley & Sons, 2015.
[2] P. J. Brockwell and R. A. Davis, Introduction to time series and forecasting. springer, 2016.
[3] H. Akaike, Fitting autoregressive models for prediction, Annals of the institute of Statistical Mathematics, vol. 21, no. 1, pp. 243–247, 1969.
[4] R. S. Pindyck and D. L. Rubinfeld, Econometric models and economic forecasts. Irwin/McGraw-Hill Boston, 1998, vol. 4.
[5] M. F. Ramoni, P. Sebastiani, and I. S. Kohane, Cluster analysis of gene expression dynamics, Proceedings of the National Academy of Sciences, vol. 99, no. 14, pp. 9121–9126, 2002.
[6] G. C. Tiao and G. E. Box, Modeling multiple time series with applications, journal of the American Statistical Association, vol. 76, no. 376, pp. 802–816, 1981.
[7] W. Penny and L. Harrison, Multivariate autoregressive models, Statistical Parametric Mapping: The analysis of functional brain images, pp. 534–540, 2007.
[8] G. Reinsel, Some results on multivariate autoregressive index models, Biometrika, vol. 70, no. 1, pp. 145–156, 1983.
[9] L. Harrison, W. D. Penny, and K. Friston, Multivariate autoregressive modeling of fmri time series, Neuroimage, vol. 19, no. 4, pp. 1477–1491, 2003.
[10] X. He and G. De Roeck, System identification of mechanical structures by a high-order multivariate autoregressive model, Computers & structures, vol. 64, no. 1-4, pp. 341-351, 1997.
[11] C. W. Anderson, E. A. Stolz, and S. Shamsunder, Multivariate autoregressive models for classification of spontaneous electroencephalographic signals during mental tasks, IEEE Transactions on Biomedical Engineering, vol. 45, no. 3, pp. 277–286, 1998.
[12] M. Ding, S. L. Bressler, W. Yang, and H. Liang, Short-window spectral analysis of cortical event-related potentials by adaptive multivariate autoregressive modeling: data preprocessing, model validation, and variability assessment, Biological cybernetics, vol. 83, no. 1, pp. 35–45, 2000.
[13] B. Ghosh, B. Basu, and M. OMahony, Multivariate short-term traffic flow forecasting using time-series analysis, IEEE transactions on intelligent transportation systems, vol. 10, no. 2, pp. 246–254, 2009.
[14] E. Ekheden and O. Hossjer, Multivariate time series modeling, estimation and prediction of mortalities, Insurance: Mathematics and Economics, vol. 65, pp. 156–171, 2015.
[15] R. Lewis and G. C. Reinsel, Prediction of multivariate time series by autoregressive model fitting, Journal of multivariate analysis, vol. 16, no. 3, pp. 393–411, 1985.
[16] W. Wang and A. K. Wong, Autoregressive model-based gear fault diagnosis, Journal of Vibration and Acoustics, vol. 124, no. 2, pp. 172–179, 2002.
[17] C. W. Granger, Investigating causal relations by econometric models and cross-spectral methods, Econometrica: Journal of the Econometric Society, pp. 424–438, 1969.
[18] P. Zikopoulos, C. Eaton et al., Understanding big data: Analytics for enterprise class hadoop and streaming data. McGraw-Hill Osborne Media, 2011.
[19] L. Atzori, A. Iera, and G. Morabito, The internet of things: A survey, Computer networks, vol. 54, no. 15, pp. 2787–2805, 2010.
[20] P. A. Valdes-Sosa, J. M. S anchez-Bornot, A. Lage-Castellanos, M. Vega-Hernandez, J. Bosch-Bayard, L. Melie-Garc a, and E. Canales-Rodrguez, Estimating brain functional connectivity with sparse multivariate autoregression, Philosophical Transactions of the Royal Society of London B: Biological Sciences, vol. 360, no. 1457, pp. 969–981, 2005.
[21] G. A. Cecchi, R. Garg, and A. R. Rao, Inferring brain dynamics using granger causality on fmri data, in Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on. IEEE, 2008, pp. 604–607.
[22] J. Chiang, Z. J. Wang, and M. J. McKeown, Sparse multivariate autoregressive (mar)-based partial directed coherence (pdc) for electroencephalogram (eeg) analysis, in Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on. IEEE, 2009, pp. 457–460.
[23] A. J. Rothman, E. Levina, and J. Zhu, Sparse multivariate regression with covariance estimation, Journal of Computational and Graphical Statistics, vol. 19, no. 4, pp. 947–962, 2010.

[24] R. Garg, G. A. Cecchi, and A. R. Rao, Full-brain auto-regressive modeling (farm) using fmri, Neuroimage, vol. 58, no. 2, pp. 416–441, 2011.

[25] R. Tibshirani, Regression shrinkage and selection via the lasso, Journal of the Royal Statistical Society. Series B (Methodological), pp. 267–288, 1996.

[26] E. J. Candes and T. Tao, Decoding by linear programming, IEEE transactions on information theory, vol. 51, no. 12, pp. 4203–4215, 2005.

[27] J. Friedman, T. Hastie, and R. Tibshirani, Regularization paths for generalized linear models via coordinate descent, Journal of statistical software, vol. 33, no. 1, p. 1, 2010.

[28] B. Efron, T. Hastie, I. Johnstone, R. Tibshirani et al., Least angle regression, The Annals of statistics, vol. 32, no. 2, pp. 407–499, 2004.

[29] R. Garg, G. A. Cecchi, and A. R. Rao, Characteristics of voxel prediction power in full-brain granger causality analysis of fmri data, in SPIE Medical Imaging. International Society for Optics and Photonics, 2011, pp. 796 502–796 502.

[30] B. B. Biswal, M. Mennes, X.-N. Zuo, S. Gohel, C. Kelly, S. M. Smith, C. F. Beckmann, J. S. Adelstein, R. L. Buckner, S. Colcombe et al., Toward discovery science of human brain function, Proceedings of the National Academy of Sciences, vol. 107, no. 10, pp. 4734–4739, 2010.

[31] D. C. Van Essen, S. M. Smith, D. M. Barch, T. E. Behrens, E. Yacoub, K. Ugurbil, W.-M. H. Consortium et al., The wu-minn human connectome project: an overview, Neuroimage, vol. 80, pp. 62–79, 2013.

[32] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay et al., Overview of the Blue Gene/L system architecture, IBM Journal of Research and Development, vol. 49, no. 2.3, pp. 195–212, 2005.

[33] B. Geng, Y. Li, D. Tao, M. Wang, Z.-J. Zha, and C. Xu, Parallel lasso for large-scale video concept detection, IEEE Transactions on Multimedia, vol. 14, no. 1, pp. 55–65, 2012.

[34] J. A. Bazerque, G. Mateos, and G. B. Giannakis, Distributed lasso for in-network linear regression, in Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on. IEEE, 2010, pp. 2978–2981.

[35] D. Yu, J.-H. Won, T. Lee, J. Lim, and S. Yoon, High-dimensional fused lasso regression using majorizationminimization and parallel processing, Journal of Computational and Graphical Statistics, vol. 24, no. 1, pp. 121–153, 2015.

[36] G. K. Chen, A scalable and portable framework for massively parallel variable selection in genetic association studies, Bioinformatics, vol. 28, no. 5, pp. 719–720, 2012.

[37] D. Luo, C. Ding, and H. Huang, Parallelization with multiplicative algorithms for big data mining, in Data Mining (ICDM), 2012 IEEE 12th International Conference on. IEEE, 2012, pp. 489–498.

[38] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin, Parallel coordinate descent for l 1-regularized loss minimization, in Proceedings of the 28th International Conference on International Conference on Machine Learning. Omnipress, 2011, pp. 321–328.

[39] G. Mateos, J. A. Bazerque, and G. B. Giannakis, Distributed sparse linear regression, IEEE Transactions on Signal Processing, vol. 58, no. 10, pp. 5262–5276, 2010.

[40] Z. Peng, M. Yan, and W. Yin, Parallel and distributed sparse optimization, in Signals, Systems and Computers, 2013 Asilomar Conference on. IEEE, 2013, pp. 659–646.