Real-time and Low Power Embedded ℓ_1-Optimization Solver Design

Zhi Ping Ang
Electronic Systems Division
DSO National Laboratories
20 Science Park Drive, Singapore 118230
Email: azhiping@dso.org.sg

Akash Kumar
Department of Electrical & Computer Engineering
National University of Singapore
4 Engineering Drive 3, Singapore 117583
Email: akash@nus.edu.sg

***Abstract*—Basis pursuit denoising (BPDN) is an optimization method used in cutting edge computer vision and compressive sensing research. Although hosting a BPDN solver on an embedded platform is desirable because analysis can be performed in real-time, existing solvers are generally unsuitable for embedded implementation due to either poor run-time performance or high memory usage. To address the aforementioned issues, this paper proposes an embedded-friendly solver which demonstrates superior run-time performance, high recovery accuracy and competitive memory usage compared to existing solvers. For a problem with 5000 variables and 500 constraints, the solver occupies a small memory footprint of 29 kB and takes 0.14 seconds to complete on the Xilinx Zynq Z-7020 system-on-chip. The same problem takes 0.19 seconds on the Intel Core i7-2620M, which runs at 4 times the clock frequency and 114 times the power budget of the Z-7020. Without sacrificing run-time performance, the solver has been highly optimized for power constrained embedded applications. By far this is the first embedded solver capable of handling large scale problems with several thousand variables.**

***Index Terms*—ℓ_1-optimization; Basis pursuit denoising; LASSO optimization; Xilinx Zynq Z-7020; Embedded implementation**

I. Introduction

Sparse recovery has made inroads within several fields of research such as computer vision [1], compressive sensing [2] and medical imaging [3]. By relaxing NP-hard sparse recovery problems into convex ℓ_1-optimization programs, problems that are previously thought to be intractable are now solvable within reasonable time complexity. Hosting an embedded real-time solver is desirable because optimization-based decisions can be made in-situ instead of deferring to off-line processing. For example, a facial recognition module that performs ℓ_1-optimization can be hosted alongside an image sensor on an embedded system-on-chip without the assistance of a workstation, thus allowing such a device to be deployed for space or power constrained use cases. A subcategory of sparse recovery, collectively known as basis pursuit denoising (BPDN) [4], recovers the solution from measurements corrupted by Gaussian noise. Therefore, this method is highly relevant for dealing with real world data and shall be the focus of this paper.

Solving BPDN problems generally require more time as compared to problems with closed-form linear algebraic solutions (i.e. least squares regression). This is because optimized software libraries for matrix computations, most notably the Basic Linear Algebra Sub-programs (BLAS), can be used to accelerate problems formulated using linear algebra, something that cannot be easily applied to BPDN problems. The main difficulty with ℓ_1-optimization is the minimization of non-smooth functions. Most solvers minimize a smooth approximation of the objective function, progressively refining the solution by solving a sequence of sub-problems approaching that of the original objective. In this aspect, ℓ_1-solvers are often control-flow rather than data-flow intensive, making acceleration on parallel hardware a nontrivial task. With this combination of factors, it is rare to find ℓ_1-solvers that are deployed to solve real-time large scale problems in embedded devices.

This research shall advance a BPDN solver where computationally intensive bottlenecks are formulated as matrix operations that can be accelerated either by using BLAS libraries, or by hardware acceleration on programmable logic. Compared to existing solvers, this algorithm exhibits superior run-time performance, recovery fidelity, and competitive memory footprint, making it an attractive candidate for embedded implementation. The solver is implemented and benchmarked on the Xilinx Zynq® Z-7020 system-on-chip. For a problem instance of 5000 variables and 500 constraints, the solver takes 0.14 seconds to complete using a single ARM Cortex-A9 CPU and 7% of slice logic on-board the Z-7020. The estimated upper bound for power usage is 305 mW, where further power savings can be achieved by clock gating FPGA logic during inactivity.

Implementation codes are accessible at http://x.co/fastBPDN

II. Basis Pursuit Denoising

Basis pursuit denoising recovers a sparse $\mathbf{x} \in \mathbb{R}^n$ with s underlying non-zero entries, given a wide matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the observation $\mathbf{y} \in \mathbb{R}^m$ that has been corrupted by Gaussian noise (Equation 1). The parameter λ trades signal fidelity ($\lambda \to 0$) for solution sparsity ($\lambda \to \infty$).

$$x^* = \underset{x}{\operatorname{argmin}} \frac{1}{2}||y - Ax||_2^2 + \lambda||x||_1 \tag{1}$$

A. Unsuitability of Targeting Existing Solvers to Embedded Platforms

Existing solvers are either time-consuming, memory intensive or unsuitable for parallelization because of inappropriate

embedded design methodology.

1) Floating Point Division: Solver designers commonly assume that all mathematical operations run in constant time complexity on the CPU, which is inaccurate because different operations have critical paths of varying lengths. Between two n-bit operands, addition takes 1 clock cycle because the critical path has $\log_2 n$ levels of logic. Multiplication takes slightly longer as the critical path is $\log_{1.5} n$-deep based on the Wallace tree multiplier, whereas division is the hardest because there is no trivial circuitry which computes the quotient given the dividend and divisor. Instead, iterative algorithms such as Goldschmidt division are used to successively approximate the quotient, requiring several clock cycles to complete. For example, the Cortex-A9 has an initiation interval of 10 cycles for floating point division. The prudent designer would be wise to avoid frivolous use of division that heavily impacts on run-time performance.

2) Expensive Transcendental Functions: A handful of solvers liberally use transcendental functions. For example, ℓ_1-MAGIC [5] uses the log-barrier method to solve BPDN, but it is rare to find hardware support for computing logarithms. These functions are often implemented as polynomial approximations in software that typically take hundreds of clock cycles per evaluation. Hardware support for transcendental functions is sporadic, and if available, expensive in run-time. The ARM Cortex-A9 on-board the Z-7020 has support for floating point square root that takes 13 clock cycles per computation.

3) Large Software Libraries: Several solvers rely on software libraries that provide advanced matrix operations (i.e. LAPACK). An example would be SparseLab [6] which uses Cholesky and LU factorization. Hosting such a library on an embedded platform is memory intensive: a pre-compiled LAPACK library[1] for x86 targets consumes 7.4 MB of program space, possibly greater for a RISC architecture like ARM.

4) Inefficient Memory Usage: A number of solvers achieve speed up by pre-computing $\mathbf{A}^\mathsf{T}\mathbf{A}$. These solvers are likely to run out of memory when handling problems with thousands of variables because memory usage scales quadratically with n. CGIST [7] and L1-LS [8] caches $\mathbf{A}^\mathsf{T}$ along with $\mathbf{A}$, and ADMM LASSO [9] saves the LU factorization of $\mathbf{A}$, making them consume an extra $O(mn)$ memory.

5) Lack of Parallelism: Some solvers are control-flow intensive, complicating any attempts of pipelined processing or data-flow acceleration. Examples would be TFOCS [10] and SpaRSA2 [11].

B. Poor Recovery Performance of Orthogonal Matching Pursuit

Because hosting an embedded ℓ_1-solver is challenging, the majority of FPGA systems that require ℓ_1-optimization [12]–[19] use orthogonal matching pursuit (OMP) [20] to approximately solve the BPDN problem. Since OMP is basically a greedy heuristic, implementation is straightforward and the run-time is short provided $\mathbf{x}$ is very sparse. Although OMP has a recovery accuracy that rivals standard ℓ_1-solvers for Gaussian $\mathbf{A}$, performance is poor for correlated matrices. A comparison of the recovery accuracy between BPDN and OMP for problems with non-random $\mathbf{A}$ can be found in [21].

[1] http://icl.cs.utk.edu/lapack-for-windows/lapack/

C. Questionable Scalability of OMP

OMP has good run-time performance under the assumption $\mathbf{x}$ is highly sparse, which is unnecessarily pessimistic because an ℓ_1 problem is recoverable as long as $\mathbf{x}$ is $O(\sqrt{m})$-sparse. Since the run-time of OMP scales quadratically with sparsity, these solvers are only capable of handling problems with low sparsity. The timings reported by studies that implement OMP on FPGA are therefore highly optimistic because the sparsity (defined to be the fraction $\frac{s}{n}$) used are close to 0. The following are some sparsity parameters: 0.03 ($n = 256$) [12], 0.04 ($n = 128$) [13], 0.04 ($n = 128$) [15], 0.008 ($n = 255$) [16] and 0.04 ($n = 128$) [18]. Given that most of the publications reported timings for $n \leq 256$, it is doubtful whether these solvers will gracefully scale with problem size, not to mention when problem sparsity increases.

III. Proposed BPDN Solver

To address the constraints of implementing a BPDN solver for real-time embedded applications, the proposed solver is formulated such that computationally intensive bottlenecks are amenable towards efficient pipelined processing or data-flow parallelization. The use of transcendental functions is eschewed and floating point divisions are kept to a minimum. Economical memory usage is ensured by in-place manipulation of $\mathbf{A}$ and judicious pre-computation of intermediate results.

A. Overview

At all times the solver maintains a prediction of the set of non-zero entries, denoted Ω, in the sparse $\mathbf{x}$. At the beginning, $\mathbf{x}$ is initialized to a rough estimate and Ω is updated from this estimate. After which, the algorithm iterates between two phases: during the first phase, Equation 1 is solved with the additional constraint that entries indexed by Ω^c are 0. In the second phase, Ω is intelligently updated based on the current estimate of $\mathbf{x}$, gradually introducing more correct non-zeros after each iteration. The algorithm iterates until $\mathbf{x}$ converges (Algorithm 1).

Algorithm 1: Proposed Solver Overview

Input : $\mathbf{y} \in \mathbb{R}^m, \mathbf{A} \in \mathbb{R}^{m \times n}, \lambda \in \mathbb{R}^+$ from Equation 1, number of sparse entries num_nonzeros, convergence parameter η
Output: Optimal $\mathbf{x}^*$ satisfying Equation 1

begin
  $\mathbf{x} \leftarrow \left(\mathbf{A}^\mathsf{T}\mathbf{y}\right) ./ \texttt{diag}\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)$;
  [∼,sorted_index] ← sort(|x|,1,'descend');
  Ω ← sorted_index(1:num_nonzeros);
  idx ← sorted_index(1);
  while $\mathbf{x}$ *has not converged* **do**
    $\mathbf{x} \leftarrow \texttt{EstimateFromNonzeros}\left(\mathbf{y}, \mathbf{A}, \mathbf{x}, \lambda, \Omega, \eta\right)$;
    $\Omega \leftarrow \texttt{GuessNonzeros}\left(\mathbf{y}, \mathbf{A}, \mathbf{x}, \lambda, \Omega, \text{idx}, \text{num\_nonzeros}\right)$;
  return $\mathbf{x}$;

B. Initializing $\mathbf{x}$ and Ω

An approximate solution to Equation 1 is $\mathbf{x} = \left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)^\dagger \mathbf{A}^\mathsf{T}\mathbf{y}$. If the problem is designed such that $\mathbf{A}$ has mutually incoherent columns (as that would have been the case for compressive sensing applications), $\mathbf{A}^\mathsf{T}\mathbf{A}$ would be strongly diagonal and the pseudoinverse can be approximated as $\texttt{diag}\left(1 ./ \texttt{diag}\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)\right)$. The i-th entry of $\mathbf{x}$ can be efficiently computed as $\frac{\mathbf{a}_i^\mathsf{T}\mathbf{y}}{\mathbf{a}_i^\mathsf{T}\mathbf{a}_i}$ by using vectorization. Ω is populated by picking the largest entries of $\mathbf{x}$ sorted by magnitude.

C. Phase I: Solving for $\mathbf{x}$ Given Ω

Given that only entries indexed by Ω are non-zero, $\mathbf{x}$ can be incrementally solved by minimizing Equation 1 one variable at a time. Equation 1 is differentiated with respect to the perturbation Δx_i (where $x_i = x_{i0} + \Delta x_i, i \in \Omega$ and $\mathbf{y}_0 = \mathbf{y} - \mathbf{A}_{(:,\Omega)}\mathbf{x}_{\Omega 0}$) to give:

$$\frac{df}{d\Delta x_i} = -\mathbf{y}_0^\mathsf{T}\mathbf{a}_i + \mathbf{a}_i^\mathsf{T}\mathbf{a}_i\Delta x_i + \lambda\,\texttt{sign}(x_{i0} + \Delta x_i) \quad (2)$$

The optimal perturbation Δx_i^* is obtained when Equation 2 vanishes, or due to the gradient discontinuity introduced by the modulus term, when the derivative changes sign at $\Delta x_i = -x_{i0}$. Determining x_i^* can be reasoned as follows: Setting $\lambda = 0$ would mean only the quadratic term is minimized, and $\Delta x_{i,\lambda=0}^* = \frac{\mathbf{y}_0^\mathsf{T}\mathbf{a}_i}{\mathbf{a}_i^\mathsf{T}\mathbf{a}_i}$. Setting $\lambda = \infty$ would mean only the modulus term is minimized, and $\Delta x_{i,\lambda=\infty}^* = -x_{i0}$. Thus, Δx_i^* lies between $\Delta x_{i,\lambda=0}^*$ and $\Delta x_{i,\lambda=\infty}^*$, and can be computed from Equation 3.

$$\Delta x_i^* = \begin{cases} \min\left(\frac{\lambda+\mathbf{y}_0^\mathsf{T}\mathbf{a}_i}{\mathbf{a}_i^\mathsf{T}\mathbf{a}_i}, \Delta x_{i,\lambda=\infty}^*\right), \Delta x_{i,\lambda=0}^* \le \Delta x_{i,\lambda=\infty}^* \\ \max\left(\frac{-\lambda+\mathbf{y}_0^\mathsf{T}\mathbf{a}_i}{\mathbf{a}_i^\mathsf{T}\mathbf{a}_i}, \Delta x_{i,\lambda=\infty}^*\right), \text{otherwise} \end{cases} \quad (3)$$

$$\Rightarrow \Delta\mathbf{x}_\Omega^* = \Delta\mathbf{x}_{\Omega,\lambda=0}^* + \texttt{sign}(\Delta\mathbf{x}_{\Omega,\lambda=\infty}^* - \Delta\mathbf{x}_{\Omega,\lambda=0}^*) \,.^*\, \min\left(\lambda ./ \texttt{diag}\left(\mathbf{A}_{(:,\Omega)}^\mathsf{T}\mathbf{A}_{(:,\Omega)}\right), \left|\Delta\mathbf{x}_{\Omega,\lambda=\infty}^* - \Delta\mathbf{x}_{\Omega,\lambda=0}^*\right|\right) \quad (4)$$

By applying the identities $\max(a,b) \equiv -\min(-a,-b)$ and $\min(a,b) \equiv \min(a-x, b-x)+x$, Equation 3 can be vectorized to give Equation 4. If the matrix $\mathbf{A}$ remains constant from problem to problem (commonly the case for compressive sensing applications), the reciprocal $\frac{1}{\mathbf{a}_i^\mathsf{T}\mathbf{a}_i}$ can be precomputed to avoid performing expensive divisions. The optimal perturbations of every variable, $\Delta\mathbf{x}_\Omega^*$, are aggregated to update $\mathbf{x}$ and repeated until convergence. Due to the fact that one variable is perturbed at a time, applying all changes at once does not yield the optimal solution, therefore the change is weighted to ensure convergence, i.e. $\mathbf{x}_\Omega \leftarrow \mathbf{x}_\Omega + \eta\,\Delta\mathbf{x}_\Omega^*, 0 < \eta < 1$. Algorithm 2 summarizes what has been mentioned so far. Note that $\mathbf{b}$ is loop invariant and can be pre-computed in the program preamble.

Algorithm 2: `EstimateFromNonzeros(`$\mathbf{y}$,$\mathbf{A}$,$\mathbf{x}$,λ,Ω,η`)`

Input : $\mathbf{y} \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{m\times n}$, current estimate of $\mathbf{x} \in \mathbb{R}^n$, $\lambda \in \mathbb{R}^+$ from Equation 1, indices of sparse entries Ω, convergence parameter η

Output: Optimal solution to Equation 1 among the family of $\mathbf{x}$ with non-zero entries in Ω

$\mathbf{b} \leftarrow \texttt{diag}\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)$;
begin
  $\mathbf{c} \leftarrow \mathbf{A}_{(:,\Omega)}^\mathsf{T}\mathbf{y}$;
  while $\mathbf{x}_\Omega$ *has not converged* **do**
    $\Delta\mathbf{x}_{\Omega,\lambda=0}^* \leftarrow \mathbf{c} - \left(\mathbf{A}_{(:,\Omega)}^\mathsf{T}\mathbf{A}_{(:,\Omega)}\right)\mathbf{x}_\Omega$;
    $\Delta\mathbf{x}_{\Omega,\lambda=\infty}^* \leftarrow -\mathbf{x}_\Omega \,.^*\, \mathbf{b}_\Omega$;
    $\Delta\mathbf{x}_\Omega^* \leftarrow \left(\Delta\mathbf{x}_{\Omega,\lambda=0}^* + \texttt{sign}\left(\Delta\mathbf{x}_{\Omega,\lambda=\infty}^* - \Delta\mathbf{x}_{\Omega,\lambda=0}^*\right) \,.^*\, \min\left(\lambda ./ \mathbf{b}_\Omega, \left|\Delta\mathbf{x}_{\Omega,\lambda=\infty}^* - \Delta\mathbf{x}_{\Omega,\lambda=0}^*\right|\right)\right)$;
    $\mathbf{x}_\Omega \leftarrow \mathbf{x}_\Omega + \eta\,\Delta\mathbf{x}_\Omega^*$;
  $\mathbf{x}_{\Omega^c} \leftarrow \mathbf{0}$;
  return $\mathbf{x}$;

D. Phase II: Updating Ω Given $\mathbf{x}$

The current Ω may not contain all the non-zero entries of the underlying true solution, therefore phase two refines Ω based on the current $\mathbf{x}$. The gist is to optimize Equation 1 with respect to (x_i,x_j) with the rest held constant. A large magnitude for x_i/x_j indicates a higher probability that the respective variable should be included in Ω. To solve (x_i,x_j), consider $\lambda = 0$ where Equation 1 simplifies to the ℓ_2-penalty $f(x_i,x_j) = \frac{1}{2}\|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2$. Letting $\mathbf{b} = -\mathbf{A}^\mathsf{T}\mathbf{y}$ and $\mathbf{a}_0 = \mathbf{A}\mathbf{x} - \mathbf{a}_i x_i - \mathbf{a}_j x_j$, the global minimum occurs at $(x_i^{\ell_2}, x_j^{\ell_2})$ satisfying Equation 5, a linear system that can be easily solved.

$$\begin{aligned} \frac{\partial f}{\partial x_i} &= \|\mathbf{a}_i\|^2 x_i^{\ell_2} + (\mathbf{a}_i \cdot \mathbf{a}_j)\, x_j^{\ell_2} + \mathbf{a}_0 \cdot \mathbf{a}_i + \mathbf{b}_i &= 0 \\ \frac{\partial f}{\partial x_j} &= (\mathbf{a}_i \cdot \mathbf{a}_j)\, x_i^{\ell_2} + \|\mathbf{a}_j\|^2 x_j^{\ell_2} + \mathbf{a}_0 \cdot \mathbf{a}_j + \mathbf{b}_j &= 0 \end{aligned} \quad (5)$$

When the ℓ_1 penalty term is included, the minimum point shifts from $(x_i^{\ell_2}, x_j^{\ell_2})$ to $(x_i^{\ell_2\ell_1}, x_j^{\ell_2\ell_1})$ according to Equation 6.

$$\begin{aligned} \|\mathbf{a}_i\|^2 x_i^{\ell_2\ell_1} + (\mathbf{a}_i \cdot \mathbf{a}_j)\, x_j^{\ell_2\ell_1} + \mathbf{a}_0 \cdot \mathbf{a}_i \\ + \mathbf{b}_i + \lambda\,\texttt{sign}\left(x_i^{\ell_1}\right) &= 0 \\ (\mathbf{a}_i \cdot \mathbf{a}_j)\, x_i^{\ell_2\ell_1} + \|\mathbf{a}_j\|^2 x_j^{\ell_2\ell_1} + \mathbf{a}_0 \cdot \mathbf{a}_j \\ + \mathbf{b}_j + \lambda\,\texttt{sign}\left(x_j^{\ell_1}\right) &= 0 \end{aligned} \quad (6)$$

If $x_j^{\ell_1}$ and $x_j^{\ell_1\ell_2}$ are opposite in signs, x_j is not a candidate for Ω. x_i is fixed to be the entry with the largest magnitude from the initialization in Section III-B, and x_j is independently solved for all variables. Ω is subsequently updated by selecting the largest magnitudes among $\mathbf{x}^{\ell_1\ell_2}$. Algorithm 3 outlines what has been mentioned. Due to loop invariance, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ can be pre-computed in the program preamble.

IV. BPDN Solver Benchmark

The proposed solver is benchmarked against a range of state-of-the-art solvers for run-time performance, recovery accuracy

Algorithm 3: GuessNonzeros($\mathbf{y}$,$\mathbf{A}$,$\mathbf{x}$,λ,Ω,idx,num_nonzeros)

Input : $\mathbf{y} \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, current estimate of $\mathbf{x} \in \mathbb{R}^n$, $\lambda \in \mathbb{R}^+$ from Equation 1, indices of sparse entries Ω, index of the entry with the largest initial magnitude idx, number of sparse entries num_nonzeros

Output : Optimal solution to Equation 1 among the family of $\mathbf{x}$ with non-zero entries in Ω

$\mathbf{b} \leftarrow \mathbf{A}^\mathsf{T}\mathbf{A}_{(:,\mathsf{idx})}$;
$\mathbf{c} \leftarrow \mathbf{A}^\mathsf{T}\mathbf{y}$;
$\mathbf{d} \leftarrow \mathtt{diag}\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)$;
begin
  $\mathbf{a} \leftarrow \mathbf{A}^\mathsf{T}\left(\mathbf{A}_{(:,\Omega)}\mathbf{x}_\Omega\right)$;
  $\mathbf{x}^{\ell_2} \leftarrow \mathbf{b} .* \left(\mathbf{a}_{\mathsf{idx}} - (\mathbf{b} .* \mathbf{x}) - \mathbf{c}_{\mathsf{idx}}\right) - \mathbf{d}_{\mathsf{idx}}\left(\mathbf{a} - (\mathbf{d} .* \mathbf{x}) - \mathbf{c}\right)$;
  $\mathbf{x}^{\ell_2\ell_1} \leftarrow \mathbf{x}^{\ell_2} + \lambda\ \mathtt{sign}\left(\mathbf{x}_{\mathsf{idx}}\right)\mathbf{b} - \lambda\ \mathbf{d}_{\mathsf{idx}}\ \mathtt{sign}\left(\mathbf{x}^{\ell_2}\right)$;
  $\mathbf{x}^{\mathsf{nonsparse}} \leftarrow \left(\mathbf{x}^{\ell_2} .* \mathbf{x}^{\ell_2\ell_1}\right) > 0$;
  $[\sim, \mathsf{sorted\_index}] \leftarrow \mathtt{sort}\left(\left|\mathbf{x}^{\mathsf{nonsparse}} .* \mathbf{x}^{\ell_2\ell_1}\right|, 1, \mathtt{'descend'}\right)$;
  $\Omega \leftarrow \mathsf{sorted\_index}(1{:}\mathsf{num\_nonzeros})$;
  return Ω;

and peak memory usage. Default settings recommended by respective authors are used. MEX-files are used if provided or instructed for compilation. The benchmark runs on an Intel® Core™ i7-2620M (2.7 GHz) machine with 8 GB memory. MATLAB R2011b (7.13.0.564 64-bit) is used as the benchmarking environment.

A. List of Solvers

The following solvers are included in the benchmark:

1) ADMM LASSO *[9]:* Alternating Direction Method of Multipliers efficiently solves large scale problems by processing sub-problems across distributed computing resources.

2) CGIST *[7]:* Conjugate Gradient Iterative Shrinkage/Thresholding solves by using a forward-backward splitting method with an acceleration step.

3) FPC-BB *[22]:* Fixed-Point Continuation is advertised for large scale image and data processing. The solver uses Barzilar-Borwein steps to accelerate convergence.

4) GLMNET *[23]:* Generalized Linear Model for elastic-net regularization is the the reference algorithm used to implement the MATLAB `lasso` function.

5) GPSR-BB6 *[24]:* Gradient Projection for Sparse Reconstruction uses special line search and termination techniques to yield faster solutions as compared to SparseLab [6], ℓ_1-MAGIC [5], bound-optimization method [25] and interior-point methods. Similar to FPC-BB, Barzilar-Borwein steps are used to accelerate convergence.

6) HOMOTOPY *[26]:* Homotopy refers to a class of methods that solves BPDN by solving a sequence of intermediate problems with varying λ.

7) L1-LS *[8]:* L1-LS is an interior point method for large-scale sparse problems, or dense problems where $\mathbf{A}$ contains structure that admits fast transform computations.

8) OMP *[20]:* Although Orthogonal Matching Pursuit approximately solves BPDN, the recovered accuracy for Gaussian $\mathbf{A}$ is comparable to ℓ_1-solvers and has excellent run-time performance. It is therefore an attractive candidate for FPGA implementation.

9) SESOP_PACK *[27]:* Sequential Subspace Optimization solves large-scale smooth unconstrained optimization.

10) SPAMS *[28]:* Sparse Modeling Software is a MATLAB toolbox for sparse recovery problems. Its C++ library makes use of the Intel Math Kernel Library for floating point computations.

11) SPARSA2 *[11]:* Sparse Reconstruction by Separable Approximation is an iterative method where each step is an optimization sub-problem involving a separable quadratic term plus the sparsity-inducing term.

12) TFOCS *[10]:* Templates for First-Order Conic Solvers provides a set of modules that can be mixed-and-matched to create customized solvers.

13) TWIST2 *[29]:* Two-Step Iterative Shrinkage / Thresholding implements a nonlinear two-step iterative version over the original iterative shrinkage/thresholding procedure to provide faster convergence for ill-conditioned problems.

14) YALL1 *[30]:* Your Algorithms for L1 is a suite of solvers that uses alternating direction algorithms, with the option of enforcing joint sparsity among related variables.

B. Test Input

For various m, n and s, the entries of $\mathbf{A}$ are drawn from the standard normal distribution. Positions of the non-zero entries in $\mathbf{x}$ are randomly picked, and the values follow a uniform distribution in the interval $(-1, 1)$. The ideal measurement $\mathbf{y} = \mathbf{A}\mathbf{x}$ is corrupted with scaled Gaussian noise of zero mean and 0.1 variance.

C. Run-Time Performance

The overall run-time complexity of the proposed solver is $O(k_1(mn + k_2 s))$, where k_1 is the number of iterations in Algorithm 1, and k_2 is the number of iterations in Algorithm 2. From Figure 1, it is evident that the proposed solver has superior run-time performance over state-of-the-art BPDN solvers due to extensive use of matrix multiplication and vectorized operations. For large problems, the run-time performance of the proposed solver is at least ten times that of the next fastest solver.

D. Accuracy of Recovered Results

For varying problem sparsity, $\mathbf{x}$ is recovered and debiased. Recovery accuracy is expected to decrease as problem sparsity increases because the problem progressively enters the ill-conditioned region of the Donoho-Tanner phase transition diagram [31]. The error measure between the debiased solution $\mathbf{x}_{\mathsf{recovered}}$ and the underlying true solution $\mathbf{x}_{\mathsf{actual}}$ is given by $\frac{\|\mathbf{x}_{\mathsf{actual}} - \mathbf{x}_{\mathsf{recovered}}\|_2}{n}$. From Figure 2, the proposed solver has superior recovery accuracy compared to all the solvers.

E. Memory Usage

The proposed solver requires $O(n + sm + s^2)$ memory in addition to the inputs $\mathbf{A}$ and $\mathbf{y}$, making the implementation memory efficient if the underlying $\mathbf{x}$ is sparse. The solver does not require advanced linear algebraic decomposition, (i.e. LU, Cholesky or singular value decomposition), hence there is no hidden memory requirement whatsoever. Because the

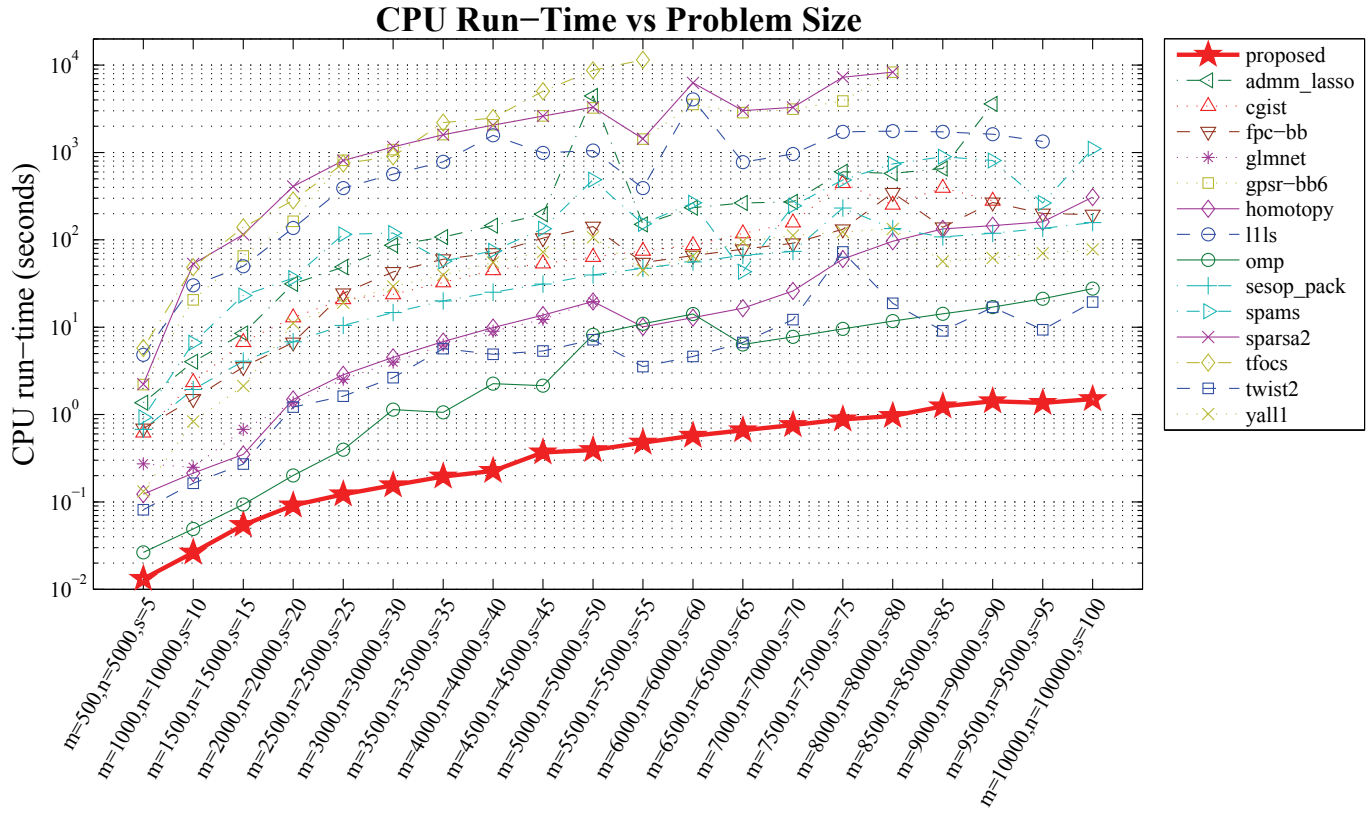

Figure 1. Run-time for various problem sizes, where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and there are s non-zero entries in $\mathbf{x}$

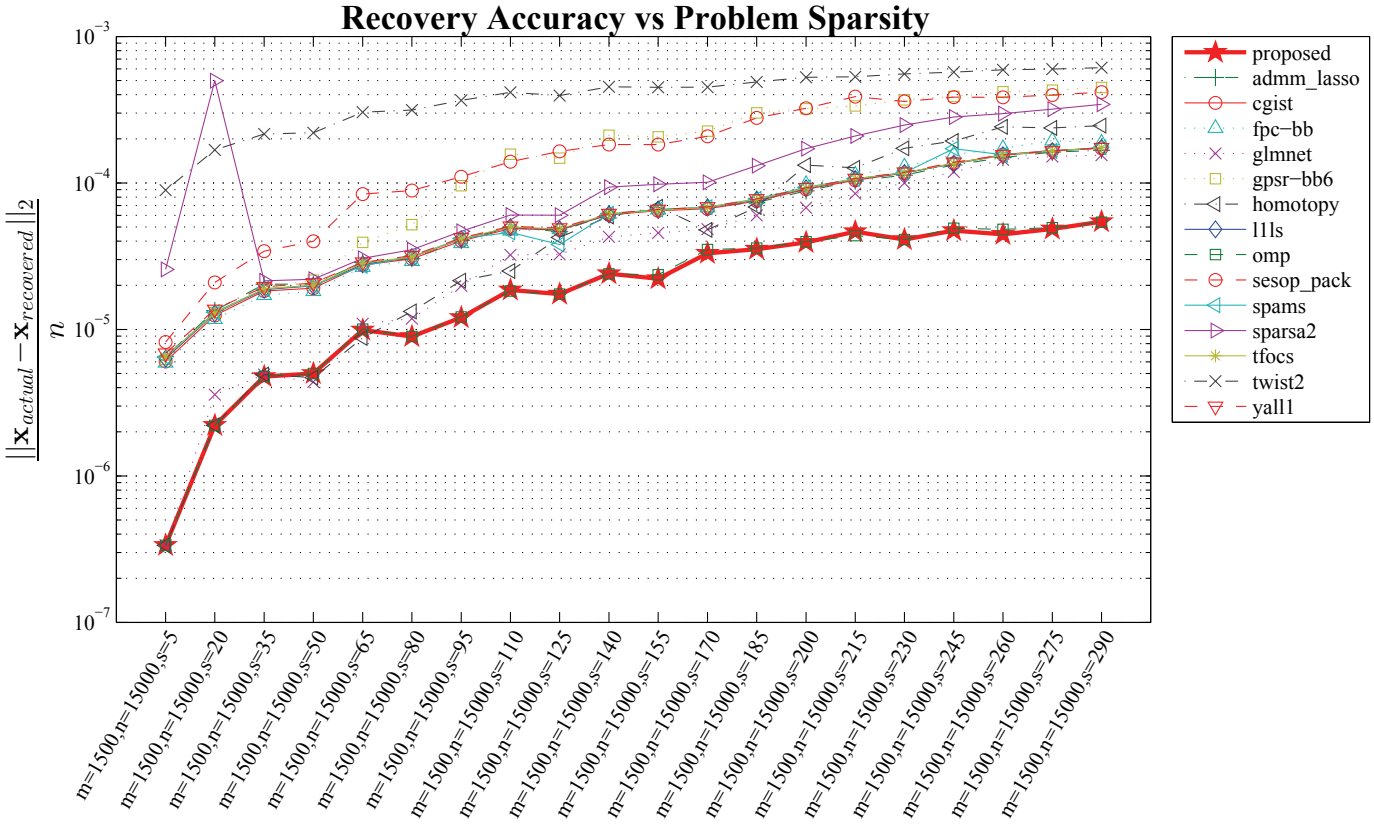

Figure 2. Recovery accuracy for various problem sparsity, where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and there are s non-zero entries in $\mathbf{x}$

solver caches pre-computed values, its memory footprint is not the lowest but nevertheless remains as one of the highly competitive among the benchmarked (Figure 3).

V. Implementation on the Xilinx Zynq Z-7020 SoC

The solver is implemented on the ZedBoard development board (Rev. C), comprising of a Zynq™-XC7Z020-CLG484-1 All Programmable System-on-Chip with 512 MB DDR3 memory. The Z-7020 chip features a dual ARM® Cortex™-A9 MPCore™ that is tightly coupled with the Artix™-7 FPGA fabric. Each core has separate 32 kB L1 instruction and data caches, and both share a unified 512 kB L2 cache. Matrix and vector operations are efficiently handled by BLAS libraries that use the NEON™ SIMD engine on-board each CPU. Custom hardware data-paths can be instantiated within the FPGA fabric if hardware acceleration is necessary. The CPU is clocked at 667 MHz and the FPGA fabric at 125 MHz. All implementations are benchmarked with respect to the problem size of $m = 500$, $n = 5000, s = 75$. The reference MATLAB solver takes 0.19 seconds to complete on the i7-2620M.

A. Eigen BLAS Library

Eigen[2] is an open-source library providing optimized assembly routines for matrix operations. The library supports hardware vectorization on ARM targets. Run-time benchmarks indicate that Eigen outperforms the Intel Kernel Math Library for operations such as $\mathbf{y} \leftarrow \alpha\mathbf{x}+\beta\mathbf{y}$, $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ and $\mathbf{Y} \leftarrow \mathbf{A}\mathbf{A}^\mathsf{T}$, therefore this library has been chosen to replace MATLAB's BLAS library. The Eigen-compiled executable occupies 29 kB of .text memory, 40 kB of .bss memory and takes 0.30 seconds to complete. The compact .text program size allows the solver to be fully loaded within the L1 instruction cache, ensuring fast program execution without expensive memory fetches. Pre-computed .bss data structures used by the solver also fits economically within the L2 cache.

B. Accelerating $\mathbf{A}^\mathsf{T}_{(:,\Omega)}\mathbf{A}_{(:,\Omega)}$ Using Programmable Logic

Table I shows the run-time summary of the solver running on a single Cortex-A9 CPU without FPGA acceleration. 89% of the overall run-time is spent on executing Eigen library code. Further profiling using the SCU[3] timer reveals that the matrix operations $\mathbf{A}^\mathsf{T}\left(\mathbf{A}_{(:,\Omega)}\mathbf{x}_\Omega\right)$ and $\mathbf{A}^\mathsf{T}_{(:,\Omega)}\mathbf{A}_{(:,\Omega)}$ occupy 34% and 55% of the run-time respectively. $\mathbf{A}^\mathsf{T}\left(\mathbf{A}_{(:,\Omega)}\mathbf{x}_\Omega\right)$ cannot be further accelerated because for every entry of $\mathbf{A}$ that is read from memory, one multiply-and-accumulate (MAC) is performed, making the operation susceptible towards I/O-boundness. A ballpark estimate illustrates the problem: The maximum read bandwidth from DDR memory to the programmable logic over an AXI_HP interface is 1.2 GB/s [32, §22.3]. This means it takes 8.3 ms to deliver $\mathbf{A}$ to the programmable logic. The average number of fetches for $\mathbf{A}$ per run is around 10, giving a paltry speed-up of 1.23×. It is possible but infeasible to utilize all the four AXI_HP buses on the Z-7020 to achieve a 4.9× speed-up as this would deprive other hardware modules of any routing resources to interface with the CPU. On the contrary, $\mathbf{A}^\mathsf{T}_{(:,\Omega)}\mathbf{A}_{(:,\Omega)}$ is compute-bound because $\frac{s(s+1)}{2}$ MACs have to be performed for every s entries read, making it an excellent candidate for FPGA acceleration.

Table I
Run-time profiling results on the Z-7020 using Gprof

% Total Time	Function
45.8	`Eigen::general_matrix_vector_product`
40.9	`Eigen::gebp_kernel`
10.4	Preamble of `fastBPDN`
1.47	`Eigen::gemm_pack_lhs`
0.95	`Eigen::gemm_pack_rhs`
0.48	Rest of the code

Table II
MAC engine hardware specifications

Operating Frequency		125 MHz
Power		55 mW
Resource Usage	Slice Logic	13735 (7%)
	Block RAM (RAMB36E1)	48 (34%)
	DSP Slices (DSP48E1)	46 (21%)
Timing	Speed-Up	7.00×
	Initiation Interval	224680 cycles
	Latency	224679 cycles
	Worst Negative Slack	0.280 ns
	Worst Hold Slack	0.052 ns
	Worst Pulse Width Slack	2.750 ns

Figure 4 shows a hardware engine capable of parallelizing $\mathbf{A}^\mathsf{T}_{(:,\Omega)}\mathbf{A}_{(:,\Omega)}$ by pipelining 9 MACs per clock cycle. The matrix $\mathbf{A}^\mathsf{T}_{(:,\Omega)}\mathbf{A}_{(:,\Omega)}$ is partitioned into 3×3 sub-matrices, and the engine computes all elements of a sub-matrix in parallel. Since $\mathbf{A}^\mathsf{T}_{(:,\Omega)}\mathbf{A}_{(:,\Omega)}$ is symmetric, sub-matrices lying on and above the main diagonal are computed on the FPGA, and entries beneath the diagonal are populated by replicating above-diagonal entries on the CPU. Inputs from $\mathbf{A}_{(:,\Omega)}$ are served over the AXI_ACP bus and results are written over the same interface. Transactions over the AXI_ACP are also cache coherent because the bus has access to the SCU that governs both caches. Prior to operation, the engine is configured by the CPU over the AXI_GP interface.

Figure 5 shows the post-routed layout consisting of both the Cortex-A9 CPU and MAC engine, and Table II details the engine specifications. The MAC engine provides a seven-fold speed-up over the software implementation of $\mathbf{A}^\mathsf{T}_{(:,\Omega)}\mathbf{A}_{(:,\Omega)}$, while utilizing a modest 7% of slice logic available on the Z-7020. Although it is possible to increase the hardware speed-up by increasing the sub-matrix size, a larger multiplier mesh has to be synthesized. By running the accelerator in parallel with CPU code, the solver takes 0.14 seconds to complete the test instance, a speed-up of 26% over running on the i7-2620M. Although the speed-up may seem modest, keep in mind that the clock frequency of the Cortex-A9 on-board the Z-7020 (667 MHz) is a quarter of i7-2620M (2.7 GHz), and the power consumption of the i7-2620M (35 W[4]) is 114 times that of the Z-7020 (305 mW[5]). Hence, the solver implementation on the Z-7020 has been optimized for low power applications without

[2] http://eigen.tuxfamily.org/

[3] Snoop control unit

[4] http://ark.intel.com/products/52231

[5] http://www.arm.com/products/processors/cortex-a/cortex-a9.php

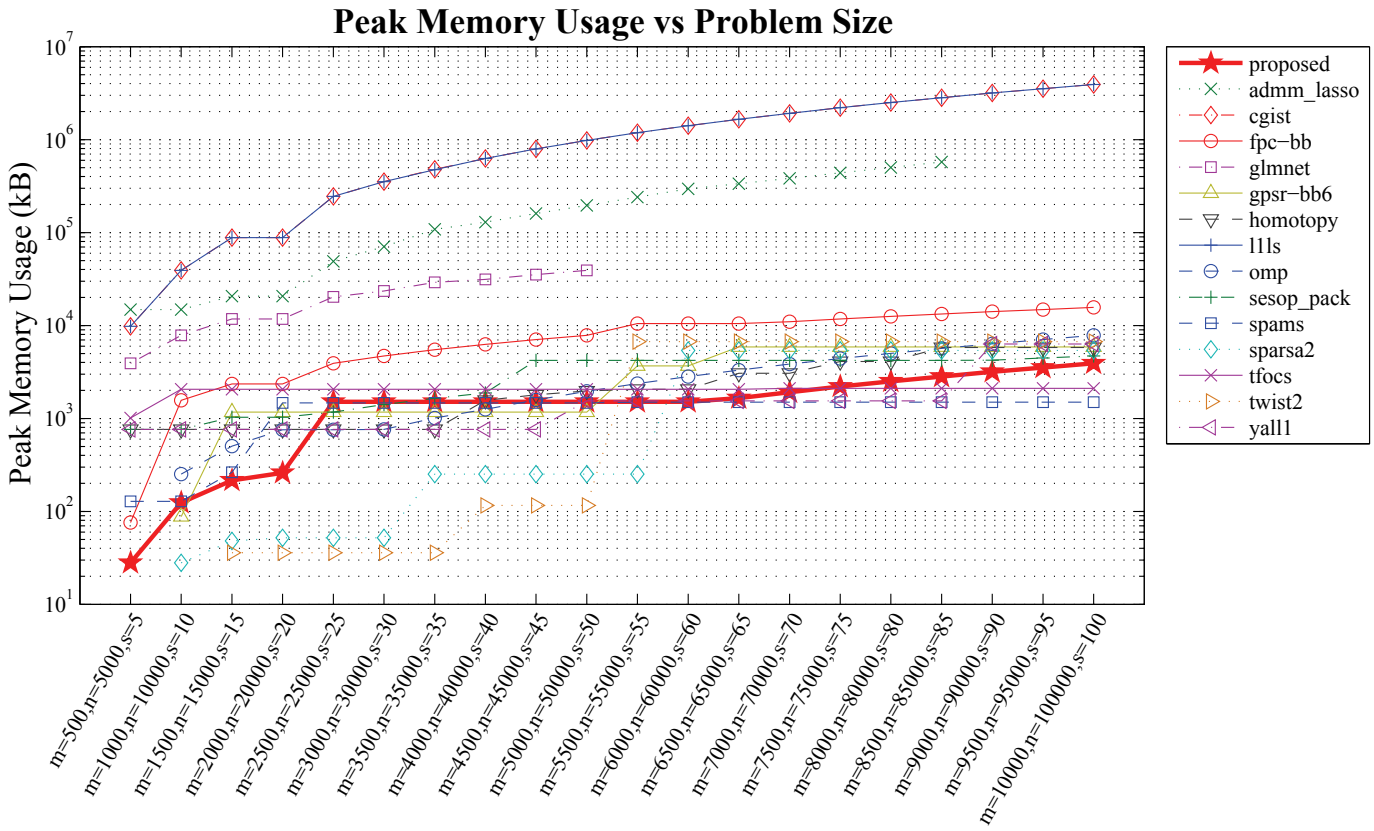

Figure 3. Peak memory usage for various problem sizes, where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and there are s non-zero entries in $\mathbf{x}$

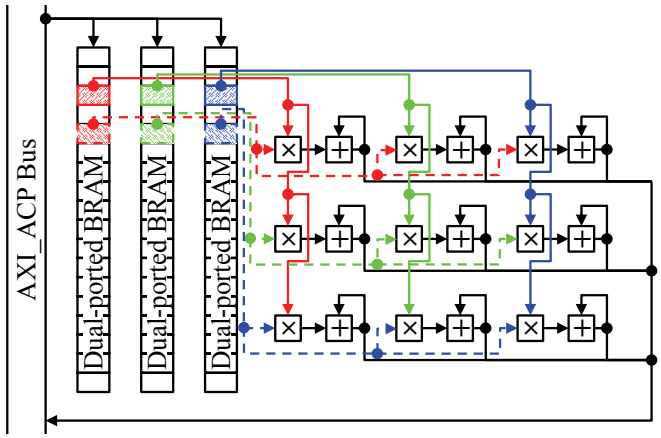

Figure 4. Hardware engine comprising of 9 floating point MAC units

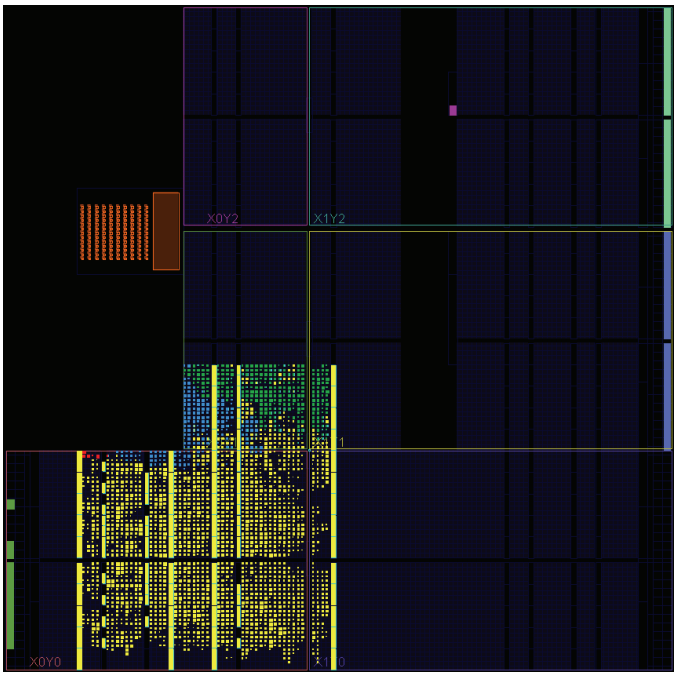

Figure 5. Post-routed layout on the Z-7020 using Vivado 2013.2 (Legend: Yellow – MAC engine, brown – ARM Cortex-A9 and DDR3 memory bus, green – AXI_ACP interconnect logic, blue – AXI_GP interconnect logic, red – reset logic)

sacrificing run-time performance. Note that the Z-7020's power figure is pessimistic as the hardware logic is only active for 17% of the final run-time; clock gating can be used to power down FPGA logic during inactivity.

VI. Conclusion

With advances in chip technology, parallel computing structures are increasingly becoming ubiquitous in modern embedded processors. Examples of embedded architectures that already feature SIMD instructions include the ARM

(NEON), the Power Architecture (AltiVec) and the Intel Atom (SSE). Hybrid CPU-FPGA system-on-chips, like the Xilinx Zynq-7000 Extensible Processing Platform and Altera's Hard Processor System, are also becoming the norm. Therefore, algorithms intended to be executed on an embedded target should be designed to have as much data-flow parallelism as possible, so as to exploit these parallel hardware.

In this paper an embedded-friendly BPDN solver is proposed. Compared to state-of-the-art solvers, the proposed solver exhibits superior run-time performance by formulating compute-intensive routines as matrix-matrix and matrix-vector multiplications, both of which are efficiently handled by BLAS libraries. Since these libraries are tuned to use architectural-specific SIMD instructions, computations are guaranteed to execute as efficiently as possible. Program memory running on the Cortex-A9 CPU is economical enough to fit within the L1 cache, and data structures within the L2 cache. The bottleneck of the solver is implemented in programmable logic which achieves a seven-fold speed-up over software code running on the embedded processor. Without sacrificing run-time performance, the embedded implementation on the Z-7020 is at least 114 times as power efficient as the MATLAB prototype on the i7-2620M.

APPENDIX

A. Mathematical Notation

1) Operators: $.^*$ and $./$ denote element-wise multiplication and division respectively. If a scalar operates with a vector, the scalar value acts on all the vector entries. $|\mathbf{x}|$ denotes element-wise magnitude, and $(\mathbf{x} > 0)$ denotes element-wise comparison that returns a vector where 1 if true, otherwise 0 entry-wise. $\min(\mathbf{x}, \mathbf{y})$ picks the minimum of the two in an entry-wise manner, and correspondingly for $\max$.

2) Matrix/vector indexing: Given a matrix $\mathbf{A}$ and a set of indices Ω, $\mathbf{A}_{(:,\Omega)}$ is analogous to the MATLAB expression `A(:,Ω)`, which returns a matrix comprising of columns indexed by Ω. For a vector $\mathbf{x}$, $\mathbf{x}_\Omega$ is another vector which is a subset of $\mathbf{x}$ indexed by Ω. For example, given a vector $\mathbf{v} = [3, 1, 4, 1, 5, 9]$ and $\Omega = [2, 3, 5]$, $\mathbf{v}_\Omega = [1, 4, 5], \mathbf{v}_{\Omega^c} = [3, 1, 9]$. If $|\Omega| = 1$, $\mathbf{x}_\Omega$ is interpreted as a scalar.

REFERENCES

[1] J. Wright, Y. Ma, J. Mairal, G. Sapiro, T. Huang, and S. Yan, "Sparse representation for computer vision and pattern recognition," *Proceedings of the IEEE*, vol. 98, no. 6, pp. 1031–1044, 2010.

[2] D. Donoho, "Compressed sensing," *Information Theory, IEEE Transactions on*, vol. 52, no. 4, pp. 1289–1306, 2006.

[3] M. Lustig, D. Donoho, and J. M. Pauly, "Sparse mri: The application of compressed sensing for rapid mr imaging," *Magnetic resonance in medicine*, vol. 58, no. 6, pp. 1182–1195, 2007.

[4] S. Chen and D. Donoho, "Basis pursuit," in *Signals, Systems and Computers. Conference Record of the Twenty-Eighth Asilomar Conference on*, vol. 1, 1994, pp. 41–44 vol.1.

[5] E. Candes and J. Romberg, "l1-magic: Recovery of sparse signals via convex programming," *URL: www. acm. caltech. edu/l1magic/downloads/l1magic. pdf*, vol. 4, 2005.

[6] A. Maleki and D. Donoho, "Optimally tuned iterative reconstruction algorithms for compressed sensing," *Selected Topics in Signal Processing, IEEE Journal of*, vol. 4, no. 2, pp. 330–341, 2010.

[7] T. Goldstein and S. Setzer, "High-order methods for basis pursuit," *Methods*, pp. 1–17, 2011.

[8] K. Koh, S. Kim, and S. Boyd, "l1 ls: A matlab solver for large-scale l1-regularized least squares problems," *Stanford University*, 2007.

[9] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[10] S. R. Becker, "Practical compressed sensing: modern data acquisition and signal processing," Ph.D. dissertation, California Institute of Technology, 2011.

[11] S. Wright, R. Nowak, and M. A. T. Figueiredo, "Sparse reconstruction by separable approximation," *Signal Processing, IEEE Transactions on*, vol. 57, no. 7, pp. 2479–2493, 2009.

[12] J. L. Stanislaus and T. Mohsenin, "Low-complexity fpga implementation of compressive sensing reconstruction," in *International Conference on Computing, Networking and Communications*, 2013.

[13] K. Karakus and H. Ilgin, "Implementation of image reconstruction algorithm using compressive sensing in fpga," in *Signal Processing and Communications Applications Conference (SIU), 20th*, 2012, pp. 1–4.

[14] H. Rabah, A. Amira, and A. Ahmad, "Design and implementaiton of a fall detection system using compressive sensing and shimmer technology," in *Microelectronics (ICM), 24th International Conference on*, 2012, pp. 1–4.

[15] P. Blache, H. Rabah, and A. Amira, "High level prototyping and fpga implementation of the orthogonal matching pursuit algorithm," in *Information Science, Signal Processing and their Applications (ISSPA), 11th International Conference on*, 2012, pp. 1336–1340.

[16] J. Lu, H. Zhang, and H. Meng, "Novel hardware architecture of sparse recovery based on fpgas," in *Signal Processing Systems (ICSPS), 2nd International Conference on*, vol. 1, 2010, pp. V1–302–V1–306.

[17] L. Bai, P. Maechler, M. Muehlberghuber, and H. Kaeslin, "High-speed compressed sensing reconstruction on fpga using omp and amp," in *Electronics, Circuits and Systems (ICECS), 19th IEEE International Conference on*, 2012, pp. 53–56.

[18] A. Septimus and R. Steinberg, "Compressive sampling hardware reconstruction," in *Circuits and Systems (ISCAS), Proceedings of IEEE International Symposium on*, 2010, pp. 3316–3319.

[19] Y. Zakharov and V. Nascimento, "Orthogonal matching pursuit with dcd iterations," *Electronics Letters*, vol. 49, no. 4, pp. 295–297, 2013.

[20] Y. Pati, R. Rezaiifar, and P. S. Krishnaprasad, "Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition," in *Signals, Systems and Computers, Proceedings of 27th Asilomar Conference on*, 1993, pp. 40–44 vol.1.

[21] P. Gill, A. Wang, and A. Molnar, "The in-crowd algorithm for fast basis pursuit denoising," *Signal Processing, IEEE Transactions on*, vol. 59, no. 10, pp. 4595–4605, 2011.

[22] E. T. Hale, W. Yin, and Y. Zhang, "Fixed-point continuation for ℓ_1-minimization: Methodology and convergence," *SIAM Journal on Optimization*, vol. 19, no. 3, pp. 1107–1130, 2008.

[23] J. Friedman, T. Hastie, and R. Tibshirani, "glmnet: Lasso and elastic-net regularized generalized linear models," *R package version*, vol. 1, 2009.

[24] M. A. T. Figueiredo, R. Nowak, and S. Wright, "Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems," *Selected Topics in Signal Processing, IEEE Journal of*, vol. 1, no. 4, pp. 586–597, 2007.

[25] M. A. T. Figueiredo and R. Nowak, "A bound optimization approach to wavelet-based image deconvolution," in *Image Processing. ICIP. IEEE International Conference on*, vol. 2, 2005, pp. II–782–5.

[26] M. S. Asif and J. Romberg, "Fast and accurate algorithms for re-weighted l1-norm minimization," *arXiv preprint arXiv:1208.0651*, 2012.

[27] G. Narkiss and M. Zibulevsky, *Sequential subspace optimization method for large-scale unconstrained problems*. Technion-IIT, Department of Electrical Engineering, 2005.

[28] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski, "Optimization with sparsity-inducing penalties," *arXiv preprint arXiv:1108.0775*, 2011.

[29] J. Bioucas-Dias and M. A. T. Figueiredo, "A new twist: Two-step iterative shrinkage/thresholding algorithms for image restoration," *Image Processing, IEEE Transactions on*, vol. 16, no. 12, pp. 2992–3004, 2007.

[30] J. Yang and Y. Zhang, "Alternating direction algorithms for \ell_1-problems in compressive sensing," *SIAM journal on scientific computing*, vol. 33, no. 1, pp. 250–278, 2011.

[31] D. Donoho and J. Tanner, "Observed universality of phase transitions in high-dimensional geometry, with implications for modern data analysis and signal processing," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 367, no. 1906, pp. 4273–4293, 2009.

[32] *Zynq-7000 AP SoC Technical Reference Manual*, v1.6.1 ed., Xilinx Inc., September 2013.