A Scalable FPGA Architecture for Nonnegative Least Squares Problems

Alric Althoff and Ryan Kastner
Computer Science and Engineering
University of California, San Diego
Email: {aalthoff, kastner}@eng.ucsd.edu

***Abstract*—Nonnegative least squares (NNLS) optimization is an important algorithmic component of many problems in science and engineering, including image segmentation, spectral deconvolution, and reconstruction of compressively sensed data. Each of these areas can benefit from high performance implementations suitable for embedded applications. Unfortunately, the NNLS problem has no solution expressible in closed-form, and many popular algorithms are not amenable to compact and scalable hardware implementation. Classical iterative algorithms generally have a per-iteration computational cost with cubic growth, and interdependencies which limit parallel approaches.**

In this paper we develop two efficient hardware architectures. One is based on a novel algorithm we develop in this paper specifically to reduce FPGA area consumption while preserving performance. We implement our architectures on a very small FPGA and apply them to reconstruction of a compressively sensed signal showing residual error results competitive with traditional algorithms.

***Keywords*-FPGA, nonnegative least squares, nonnegative quadratic programming, high-level synthesis, compressive sensing**

I. Introduction

Optimization problems in which the solution must be nonnegative are pervasive in many areas of science, engineering, and economics. This stems in part from the desire to end up with meaningful quantities relevant to the problem domain, such as counts, distances, or otherwise. A large subset of such optimization problems are least squares problems, concerned with finding the solution, or a close approximation when A has no inverse, to $x \geq 0$ in the equation $y = Ax$.

One example application is in computer vision. An image is a nonnegative matrix, and decomposing this matrix into nonnegative components can allow identification of simpler geometric components of objects in the image [1], [2]. Another application is in the identification of chemical components of a mixture [3]—in this case the matrix A is a "dictionary" of known spectrographic signatures, while y is the captured transmission spectrum of the mixture. Mixture components in x must be nonnegative to be sensible. An important application in machine learning is the training of support vector machines (SVMs), a staple classification algorithm. In the application we use as an example in section IV, a signal x containing many zero entries (a sparse signal) is compressed via a random linear transform into a smaller vector y. We then recover a close approximation to x using our hardware architecture.

In this paper we discuss an efficient hardware implementation of a recently developed algorithm for such *nonnegative least squares* (NNLS) problems. All implementation sources were generated by the Vivado HLS tool by Xilinx, and tests were run on a ZedBoard evaluation board. To motivate our development of such a compact architecture we consider that the relevant image processing and machine learning applications are usually limited to systems which are not portable. Our work makes these tasks possible on extremely compact and efficient devices.

The key contributions of this paper are:

- An algorithm and architecture specifically designed to solve large NNLS problems on very small FPGAs.
- A high performance and compact NNLS core for compressed sensing recovery, SVM training, etc.
- The first FPGA implementation of an algorithm for NNLS.
- A complete and functional system using a ZedBoard.

A. Mathematical Background

Consider the ordinary linear least squares problem of minimizing the sum of squared differences:

$$x^* = \operatorname*{argmin}_{x} \left\{ \frac{1}{2} \|y - Ax\|_2^2 \right\} \tag{1}$$

for $A \in \mathbb{R}^{m \times n}$. This is the standard way to solve for x in problems of the form $y = Ax$. This form can be used when A is not invertible, and has the closed form solution

$$x^* = A^{\dagger} y = \left(A^T A\right)^{-1} A^T y,$$

where A^T and A^{-1} are, respectively, the transpose and inverse of A, and $A^{\dagger}$ is called the *pseudoinverse* of A. Problems of this type are ubiquitous in all areas science and engineering. Due to their importance, a great deal of work has been done to accelerate least squares solvers, for some FPGA examples, see [4]–[6].

A modification of (1), identical except that x is required to be nonnegative, is the following:

$$x^* = \operatorname*{argmin}_{x \geq 0} \left\{ \frac{1}{2} \|y - Ax\|_2^2 \right\} \tag{2}$$

This can be rewritten as

$$
\begin{aligned}
x^* &= \underset{x\geq 0}{\operatorname{argmin}} \left\{ \frac{1}{2}\left(x^T A^T A x - y^T A x\right) \right\} \\
&= \underset{x\geq 0}{\operatorname{argmin}} \left\{ x^T H x + c^T x \right\}
\end{aligned} \tag{3}
$$

where the symmetry and positive semi-definiteness of H makes this a convex problem. In the case where H is rank deficient (or the problem is poorly conditioned), a small factor δI, where I is the identity and δ is a constant, can be added to H as a preconditioner.

Unfortunately, no closed-form solution exists for such problems. Numerous iterative optimization-based algorithms have been developed [7]–[9]. Despite this effort, fast algorithms suitable for real-time applications and/or economical hardware implementation are uncommon. This is due to their use of intermediate algorithms requiring relatively complex hardware, such as the square root in the otherwise simple technique of [7].

B. Related Work

While a great deal of research has gone into the development of efficient algorithms to solve NNLS problems [8], [10], [11], a search for a hardware implementation of any NNLS algorithm (including those used by nonnegative matrix and tensor factorizations) presently yields no results. Only recently have algorithms been developed that have a structure simple enough for very compact hardware implementation [7], [12], [13]. In [12] Brand et al. develop the algorithm upon which part of this work is based, targeting model predictive control applications.

II. Description of the Algorithms

A. Parallel Quadratic Programming

Following [13] we may derive algorithm 1 directly from the Karush-Kuhn-Tucker (KKT) first order optimality conditions. These are conditions for optimality commonly used in mathematical optimization. They are very closely related to the method of Lagrange multipliers from early calculus—in that we are turning a constrained problem to an unconstrained one by adding the constraints to the original problem, then minimizing this sum.

Consider the Lagrangian of (3):

$$
\Lambda(x;\lambda) = x^T H x - c^T x - \lambda^T x,\ \lambda \geq 0 \in \mathbb{R} \tag{4}
$$

Minimizing Λ with respect to x is the same as solving the NNLS problem. We have included the constraint $x \geq 0$ within a single equation instead of writing it externally. This gives the KKT conditions (where $\cdot\circ\cdot$ is element-wise multiplication):

$$
x \circ \nabla_x \Lambda = x \circ (Hx - c - \lambda) = 0 \tag{5a}
$$

$$
\lambda \circ \nabla_\lambda \Lambda = \lambda \circ x = 0 \tag{5b}
$$

That is to say, either an entry of x is zero, or the partial derivative of Λ with respect to x is zero, and the same entry in x and λ cannot both be nonzero. We can see that for x to be optimal, it must satisfy these conditions, in that setting a derivative to zero locates possible optimal solutions.

Let $v^+ = \max\{0, v\}$, $v^- = \max\{0, -v\}$, and $\div$ be elementwise division. Then algebraic manipulation of (5a) gives:

$$
\begin{aligned}
x \circ (Hx - c - \lambda) &= x \circ \left(\left[H^+ - H^-\right]x - \left[c^+ - c^-\right] - \lambda\right) \\
&= x \circ \left(-H^- x - c^+ - \lambda\right) + x \circ \left(H^+ x + c^-\right) \\
&= 0
\end{aligned}
$$

By (5b), $x \circ \lambda \to 0$ as $x \to x^*$, so we obtain

$$
x \circ \left(H^+ x + c^-\right) = x \circ \left(H^- x + c^+\right) \implies x = x \circ \frac{H^- x + c^+}{H^+ x + c^-}
$$

This is only true when x is optimal, but given certain properties of this equation for x [13] we can infer that the sequence defined by the recursion

$$
x^{(k+1)} = x^{(k)} \circ \frac{H^- x + c^+}{H^+ x + c^-} \tag{6}
$$

eventually converges to the correct value. This convergence is linear, and in some cases superlinear, similar to gradient descent. For proof of convergence we refer the reader to [13].

Note that maintenance of nonnegativity is implicit to the algorithm, given that no element is ever negative given a feasible initial guess for $x^{(0)}$.

1) Numerical Considerations: Numerical error buildup is not a major concern by virtue of the necessary convergence of the ratio $\left(H^- x + c^+\right) / \left(H^+ x + c^-\right)$ to the vector of all ones. This implies that the algorithm will tend to iteratively correct numerical errors that may occur early on, assuming initialization constraints have been met. If H is very poorly conditioned, a small constant may be added to the diagonal as mentioned in section I-A.

As is the case with any algorithm involving division, we must take care to ensure that the denominator is nonzero, specifically $[H^+ x + c^-]_i \neq 0$, at any iteration. Due to the numerically corrective property discussed above, indices having a very small denominator can simply be skipped for that iteration.

Incorporating these corrective measures leads to the final parallel quadratic programming algorithm (PQP) as shown in Algorithm 1.

B. Algorithm 2

When we consider a hardware implementation of PQP, several disadvantages are immediately apparent—for each iteration there are two symmetric matrix-vector multiply (SYMV) operations and an elementwise vector division. Additionally, while each element of $x^{(k)}$ may be computed in parallel, all such computations must finish prior to beginning computation of any element of $x^{(k+1)}$. In a software implementation this would allow for good parallelization under the fork-join model, but in an FPGA design, this means we must provide a way to concurrently access elements of the $H^{+/-}$ matrices. While necessary for PQP, this can lead to storage inefficiencies.

In this section, we develop a novel algorithm to address these deficiencies inspired by the notion of *coordinate descent*.

Algorithm 1 PQP

```
Require: $x^0 > 0,\ c,\ H \succeq 0 \in \mathbb{R}^{n\times n},\ \delta > 0,\ \varepsilon > 0,\ maxit > 0$
  $H^+ \leftarrow \delta I + \max\{0, H\}$
  $H^- \leftarrow \delta I + \max\{0, -H\}$
  $c^+ \leftarrow \max\{0, c\}$
  $c^- \leftarrow \max\{0, -c\}$
  for $k \in [maxit]$ do
    $a \leftarrow H^- x + c^+$
    $b \leftarrow H^+ x + c^-$
    for $i \in [n]$ do
      if $b_i > \varepsilon$ then
        $x_i \leftarrow \frac{a_i}{b_i} \cdot x_i$
      end if
    end for
  end for
```

1) Coordinate Descent: In the familiar gradient descent approach to optimization a function $f(x)$ is (perhaps only locally) optimized by repeated iterations of the update equation

$$x^{(k+1)} = x^{(k)} - \delta^{(k)} \nabla f\left(x^{(k)}\right)$$

where $\delta^{(k)} > 0$ is a step size parameter. While gradient descent updates the entire vector at each step coordinate descent seeks to find one entry (a coordinate) of the optimal x completely—usually by a line search—before moving on to the next coordinate. For separable f, repeated sweeps over all entries of x will converge to an optimum. The intuitive advantage of this approach for our purposes is that the amount of data used during an iteration may be considerably reduced, allowing more efficient use of storage.

Following this general idea we iterate for a fixed number of iterations on

$$x_i^{(k+1)} = \frac{\langle H^-_{i,:}, x^{(k)}\rangle + c_i^+}{\langle H^+_{i,:}, x^{(k)}\rangle + c_i^-}$$

where x_i is the i^{th} entry of x, and $H_{i,:}$ is the i^{th} row of H. Since at each iteration, only one value of $x^{(k+1)}$ is altered, we may precompute the partial inner product $c_i^{-/+} + \langle H^{+/-}_{i,:}, x^{(k)}\rangle - H_{ii}x_i^{(k)}$, greatly simplifying updates.

Despite our update rule not being coordinate descent in a strict sense, we will use this term to indicate iteration on a single entry of $x^{(k)}$ for the remainder of this paper.

2) Division: To address the division operation, we start by formalizing our earlier observation with respect to the PQP update equation:

$$\lim_{k\to\infty} \frac{H^- x^{(k)} + c^+}{H^+ x^{(k)} + c^-} = 1$$

This being so, we can think of this quotient as a step size much like δ in gradient descent, albeit one that adjusts implicitly. We then recognize that if an entry of $x^{(k)}$ is too large, then $H^-_{i,:}x^{(k)} + c_i^+ < H^+_{i,:}x^{(k)} + c_i^-$, and conversely if an entry of $x^{(k)}$ is too small. This "seesaw-like" effect allows us to save

Algorithm 2

```
Require: $x^0 > 0,\ c,\ H \succeq 0 \in \mathbb{R}^{n\times n},\ \delta > 0,\ maxit > 0,\ passes > 0$
  $H^+ \leftarrow \delta I + \max\{0, H\}$
  $H^- \leftarrow \delta I + \max\{0, -H\}$
  $c^+ \leftarrow \max\{0, c\}$
  $c^- \leftarrow \max\{0, -c\}$
  for $j \in [passes]$ do
    $\mu \in (0, 0.5)$
    for $i \in [n]$ do
      $p \leftarrow c_i^+ + \langle H^-_{i,:},\ x\rangle - H^-_{i,i}x_i$
      $q \leftarrow c_i^- + \langle H^+_{i,:},\ x\rangle - H^+_{i,i}x_i$
      for $k \in [maxit]$ do
        $a \leftarrow H^-_{i,i}x_i + p$
        $b \leftarrow H^+_{i,i}x_i + q$
        if $a < b$ then
          $x_i \leftarrow (1-\mu)x_i$
        else
          $x_i \leftarrow (1+\mu)x_i$
        end if
      end for
    end for
  end for
```

ourselves from a series of expensive division computations. Instead of computing $\left(H^- x^{(k)} + c^+\right) / \left(H^+ x^{(k)} + c^-\right)$, we first determine the numerator and denominator, then compare them. If $\left(H^-_{i,:}x^{(k)} + c_i^+\right) > \left(H^+_{i,:}x^{(k)} + c_i^-\right)$, we heuristically select a step size $\mu \in (0, 0.5)$ and update $x_i^{(k)}$ to be $(1+\mu)x_i^{(k)}$, and if the denominator is greater than the numerator we do update as $(1-\mu)x_i^{(k)}$ instead. This also gives us the flexibility to take larger steps if we are more uncertain about the initial guess $x^{(0)}$, and decrease the step size during each coordinate descent pass.

Given these optimizations, we greatly alter the original algorithm while still preserving much of the spirit, leading to Algorithm 2.

3) Convergence :

Theorem 1. *If $x^{(k)}$ is the estimate of x^* made by algorithm 2 at the k^{th} pass, then $\lim_{k\to\infty} x^{(k)} = x^* \pm \mu x^*$, where $x^*$is the solution of equation 2.*

Proof: We rely on the proof of convergence for algorithm 1 [13], and call $x^{(k)}_{PQP}$ the estimate of x^* at the k^{th} iteration of that algorithm. We recognize that for some heuristic multiplicative step μ we have selected μ such that $x_i^{(k+1)} = (1\pm\mu)x_i^{(k)} = x_i^{(k)} \pm \mu x_i^{(k)}$ will be "tracking" $x^{(k)}_{PQP}$. After several coordinate steps, $x^{(k)}$ will either be closer to the respective coordinate of $x^{(k)}_{PQP}$, or no more than $\mu x_i^{(k)}$ away from it (i.e. it could oscillate about $x^{(k)}_{PQP}$). Since $\lim_{k\to\infty} x^{(k)}_{PQP} = x^*$, $\lim_{k\to\infty} x^{(k)} = x^* \pm \mu x^*$. Note also that since with each coordinate descent pass we may decrease μ, $x^{(k)}$ may become arbitrarily close to x^*. ■

$$\begin{pmatrix} a_{00} & a_{01} & a_{01} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \Longrightarrow \left(a_{00}, a_{10}, a_{20} \middle| a_{11}, a_{21} \middle| a_{22}\right)$$

Figure 1. An example of packed lower triangular symmetric matrix storage for a 3×3 matrix. The packed matrix is indexed at a_{ij} via the formula $i + j \cdot \frac{2n-(j+1)}{2}$.

It is important to realize that the above proof does not imply that descent will be monotonic, in fact, we should expect monotonicity only if the step size is small enough to never oscillate about the best estimate. In practice this oscillation does little damage so long as μ decreases sufficiently with each pass.

C. *Convergence Detection*

While several possible methods for detecting convergence are possible, we choose to simply limit the iteration count using a function parameter. This seems most practical for a hardware implementation and allows the user to determine exactly how long the system should run, regardless of the accuracy of the results. If for some reason a solution x is unsatisfactory after a fixed amount of time, the algorithm can simply continue where it left off, using the resulting x as the new $x^{(0)}$. It is more efficient to execute convergence checks in an external control module than spend time tracking convergence on every iteration.

One such check would be to compute the error $||c - Hx||^2$ and stop when it becomes sufficiently small. While simple to understand, this is $O(n^2)$ due to the SYMV, vector subtraction, and magnitude computations. Fortunately there are other possibilities. One attractive option is to check the sum of the entries of x. Convergence has been reached if the difference between sums in two consecutive iterations is below a user specified threshold.

D. *Symmetry*

Significant memory savings can be gained by using packed symmetric array storage such as that used in many BLAS implementations (e.g. [14]), see Figure 1 for details. In Algorithm 2 we access H a single row at a time. Since we perform multiple iterations using a row or block of rows, it is feasible to store the matrix off the FPGA fabric in this format, and unpack/rearrange the target row(s) in software just prior to transfer from DRAM.

III. FPGA Architecture

We implement both algorithms on a ZedBoard platform with a Zynq-7000 SOC (xc7z020clg484-1 type FPGA—the programmable logic portion is similar to a medium-large Artix 7, i.e. quite small). For evaluation we generate problem instances—H, c, and $x^{(0)}$—at random on the ARM CPU running Xillinux 1.3 [15] and communicate with the FPGA fabric over 32-bit wide generic Xillybus FIFOs. We have chosen Xillinux on account of it being a readily available core-to-fabric solution for Zynq SOCs, though our experiments show that a custom solution may be necessary for optimal throughput.

Table I
Post-synthesis performance for differing C++ implementations

Code	Latency	Period	II
3a	49,152	13.83 ns	3
3b	16,385	8.71 ns	1

A. *Algorithm 1*

To efficiently decompose algorithm 1 into hardware components, we recognize three distinct steps, splitting, SYMV, and division-multiplication. Figure 2 is a high level block diagram showing the flow of data between these components.

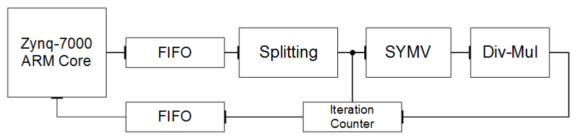

Figure 2. The complete system highlighting discrete algorithmic steps.

1) ARM: In this architecture, H, c, and $x^{(0)}$ are initially stored in RAM. The ARM core manages memory access and feeding data to the input FIFO, and reading the final result from the output FIFO. No preprocessing is done prior to splitting. Data transmission overhead is minimal.

2) Splitting: Splitting is represented by the max operations shown in algorithm 1. δ is added to the diagonal of H^+ and H^- simultaneous to splitting. While this approach increases area utilization by a small amount, it reduces latency by allowing a branch-free inner loop during SYMV.

While this may seem a trivial to implement in C++, functionally identical code will be translated by Vivado HLS into very different architectures. Figure 3a shows how a naïve split might be accomplished, while 3b is less obvious and produces a much better architecture.

Table I shows that Vivado HLS is unable to pipeline code 3a as efficiently as 3b, and the clock period change shows that implementation 3a is on the critical path. In this table II is the initiation interval, and both implementations are pipelined. Latency and II values isolate the splitting component, while period considers the entire system.

The lesson here is that results from Vivado HLS (and likely any HLS tool) are effective in direct proportion to the user's experience and knowledge of how the tool interprets code. Efficient C++ does not imply efficient hardware.

3) SYMV: We can see that each of $H^-x + c^+$ and $H^+x + c^-$ represent the same function of a matrix and two vectors (this common sub-problem will be called $H^{+/-}x + c^{+/-}$ in the following). The SYMV operation is the performance bottleneck, with a naive complexity of $O(n^2)$. There are several possible approaches, we adopt the column-major approach discussed in [16].

```
[...]
bool Hij_geq0 = val >= 0;
if (Hij_geq0) {
    Hp[i*SIZE+j] = val;
    Hn[i*SIZE+j] = 0;
} else {
    Hp[i*SIZE+j] = 0;
    Hn[i*SIZE+j] = -val;
}
if (i == j) {
    Hp[i*SIZE+j] += delta;
    Hn[i*SIZE+j] += delta;
}
[...]
```

(a) Within the innermost loop of a naïve split of H into H^+ and H^-.

```
[...]
bool Hij_geq0 = val >= 0;
bool diag = i == j;
Hp[i*SIZE+j] = (Hij_geq0 ? val : 0) + (diag ? delta : 0);
Hn[i*SIZE+j] = (Hij_geq0 ? 0 : -val) + (diag ? delta : 0);
[...]
```

(b) A more HLS amenable implementation.

Figure 3. Two implementations of splitting, (a) shows an obvious implementation, while (b) is functionally identical, but much more Vivado-HLS-friendly. Both (a) and (b) are inside double loops indexed by i and j.

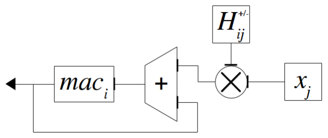

Figure 4. An example column-wise signal flow diagram of $H_{ij}^{+/-} \cdot x_i$ which will be duplicated to match the unroll factor of the innermost loop. In this case j is the outer loop counter. mac_i resets to $c_i^{+/-}$. Multiplexed mac_i resets and outputs to the divider, comparator, and multiplier are not shown.

The column-wise architecture focuses on the idea of caching the intermediate results of the inner product operation (Figure 4). This forms the entire result of the SYMV prior to the compare-divide-and-multiply step required to complete an iteration.

A benefit of this approach is the ability to increase the unroll-factor without a corresponding increase in the depth of the adder tree. This yields throughput directly proportional to the unroll-factor of the matrix multiply loop, plus overhead. In practice partial, rather than complete, unrolling is required to maintain feasible area consumption. The process of the SYMV can then be thought of as a thin block-wise operation, tiling the matrix columns (Figure 5).

As shown in the figure, this architecture iterates first over the rows, then over the columns. This allows each x_j, where j is the current column, to be fetched from memory only once per SYMV, (noting that both H^+ and H^-matrix multiplies occur simultaneously in the hardware).

4) Division-Multiplication: Outputs from the multiply-accumulate are compared with ε, then fed to pipelined dividers and multipliers. We conserve DSP resources by applying the same unroll-factor as used in the SYMV step.

5) Memory Model: Each of H^+, H^-, c^+, and c^- are copied from the input FIFO through a MUX which assigns $H^{+/-}$ to dual-port block RAMs and $c^{+/-}$ to distributed RAM during

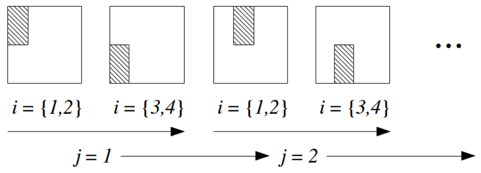

Figure 5. A partial run of the column-wise architecture showing the matrix memory access pattern when n is four with an unroll factor of two.

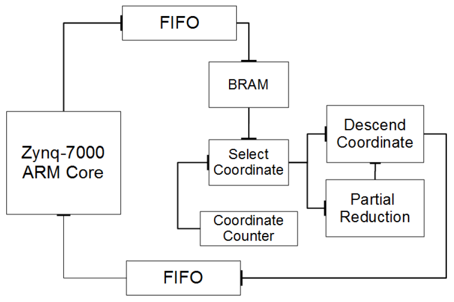

Figure 6. The complete system highlighting discrete algorithmic steps.

the splitting step prior to the first iteration of the optimization procedure.

The design is parametrized to allow user control over the partitioning and unroll-factors of SYMV and the division loop. The user can alter this parameter to generate compact designs at the expense of throughput. As a result of the memory model, the unroll-factors of all loops can be chosen to maximize throughput without the risk of memory port contention or severe under-utilization of available BRAM memory due to partitioning schemes.

Table II shows 32-bit floating point utilization and performance at several unroll-factors. This table demonstrates the inverse linear relationship between performance and area. Latency (in cycles) and throughput is for a single iteration of SYMV and division-multiplication. Note that area is proportional, and latency is inversely proportional, to unroll-factor.

B. Algorithm 2

As with the previous architecture, we break the algorithmic flow into discrete stages, shown in 6. Unlike the first architecture which is more performance oriented, algorithm 2 and the resulting architecture is designed for very small resource utilization while maintaining good performance.

1) ARM: The algorithm 2 architecture requires that the ARM core manage DRAM access, unpacking symmetric matrix storage, counting coordinate descent passes, and breaking the matrix into discrete blocks which are to decrease data transmission overhead. In this context, a “block” is a set of rows of H with a fixed size.

2) Select Coordinate: After a block of rows has been sent to the programmable logic and locally stored in block RAM, we select a row to descend, and keep track of the current set of rows-of-interest. In our experiments, we have chosen to sweep through all the rows of a block for a fixed number of iterations each, but in certain cases it may be useful to have additional

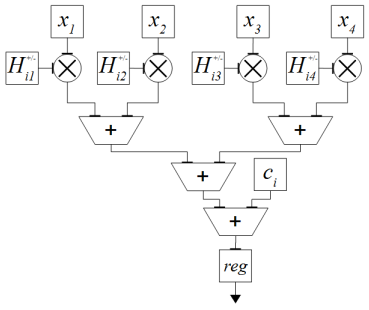

Figure 7. A diagram of the top-level adder-tree used for pipelined partial reduction in algorithm 2. *reg* is the pipeline register, and outputs to more tree stages. Note that the current value of $H_{ii}x_i^{(k)}$, where we a descending the i^{th} coordinate, will not be an input to the tree.

logic here to detect whether a given entry of $x^{(k)}$ is still of interest. For example, we can assume an application where we are only concerned with values in x^* that are within a range, in which case we can choose not to descend on coordinates converging to values outside of that target, hence accelerating the solution process.

3) Partial Reduction: Partial reduction is performed once prior to iteration on a particular coordinate. This is equivalent to $c_i^{-/+} + \langle H_{i,:}^{+/-}, x^{(k)}\rangle - H_{ii}x_i^{(k)}$, which we compute via a pipelined tree while omitting $H_{ii}x_i^{(k)}$. We compute partial reductions in parallel for a subset of the rows minus the results they depend on. We can complete the dependency immediately before descending that coordinate. In mathematical notation we compute

$$\begin{aligned} c_i^{-/+} + \langle H_{i,:}^{+/-}, x^{(k)}\rangle - H_{i,:}x_i^{(k)} \\ c_{i+1}^{-/+} + \langle H_{i,:}^{+/-}, x^{(k)}\rangle - H_{i,:}x_i^{(k)} - H_{i+1,:}x_{i+1}^{(k)} \\ \vdots \\ c_{i+d}^{-/+} + \langle H_{i,:}^{+/-}, x^{(k)}\rangle - \sum_{j=1}^{d} H_{i+j,:}x_{i+j}^{(k)} \end{aligned}$$

concurrently using pipelined arithmetic cores to reduce DSP use. It is easy to see that there will be a point of diminishing returns for both speed and area. We choose $d = 4$ in our experiments.

4) Descend Coordinate: After the first partial reduction, we iterate a fixed number of times on a single coordinate using the architecture shown in figure 8. Each iteration updates $x_i^{(k)}$, while holding $c_i^{-/+} + \langle H_{i,:}^{+/-}, x^{(k)}\rangle - H_{ii}x_i^{(k)}$ (the partial result) constant. After the counter has reach a predefined number of iterations, we choose eight in our experiments, x_i is updated, and x_{i+1} is pulled from storage.

After this stage results are sent to the ARM core via FIFO, and a new block is loaded. This is repeated for a fixed number of passes, or until convergence.

Tables II and III show performance and area results for the two architectures. We observe that single-iteration latencies are similar for both architectures.

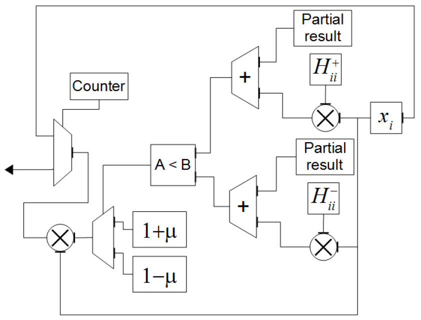

Figure 8. A block diagram showing the descent stage of algorithm 2. Partial results are read from the the *Partial Reduction* stage. While the external connections are not shown here, $1 \pm \mu$ are reloaded at the beginning of the each pass, usually with diminished values of μ.

Table II
UTILIZATION AND PERFORMANCE OF OUR ALGORITHM 1 ARCHITECTURE WITH VARYING UNROLL-FACTORS (UF) NOTE: LATENCIES ARE FOR ONE ITERATION, CLOCK PERIOD IS 8.63 NS FOR ALL.

Matrix size $n = 64$

UF	BR	DSP	FF	LUT	Lat.	Thr. (KHz)
2	20	24	5,044	8,607	2,183	53.08
4	16	44	9,626	16,140	1,111	104.30
8	16	84	18,326	29,834	576	201.17

Matrix size $n = 128$

UF	BR	DSP	FF	LUT	Lat.	Thr. (KHz)
2	64	24	7,244	9,922	8,359	14.14
4	64	44	13,471	18,420	4,199	28.15
8	64	84	27,298	36,989	2,120	55.76

IV. EXAMPLE APPLICATION

Though many applications exist, we choose an example of decompressing a sparse signal compressed using the method of compressed sensing.

A. Compressed Sensing (CS)

CS is a mathematical framework based on pioneering work by Candés and Tao [17], and Donoho [18]. The importance of their result lies in showing that the Nyquist rate is not a limiting factor for exact signal recovery for certain classes of signals under a linear sampling criterion. They show that a signal vector $x \in \mathbb{R}^n$ possessing at most k nonzero coefficients—said to be *k-sparse*—can be reconstructed from a set of linear measurements of the form $y = \Phi x$ where $\Phi \in \mathbb{R}^{m\times n}$, $m < n$. While not a strict requirement, Φ should, for robustness, also obey the *Restricted Isometry Property* (RIP): $(1-\delta)\|x\|_2^2 \leq \|\Phi x\|_2^2 \leq (1+\delta)\|x\|_2^2$, which is to say that Φ should not alter the magnitude of x too greatly. Φ matrices having the RIP can be constructed by drawing i.i.d. (independent and identically distributed) random elements from a sub-Gaussian distribution and ensuring that $m = \Omega(k \log n/k)$.

The reconstruction of x from y is obtained as the solution of the (NP-hard) optimization problem

$$\min_x \|x\|_0 \text{ s.t. } y = \Phi x \tag{7}$$

Table III
UTILIZATION AND PERFORMANCE OF OUR ALGORITHM 2 ARCHITECTURE WITH VARYING BLOCK SIZES. CLOCK PERIOD IS 8.63 NS FOR ALL.

Block size 32

n	BR	DSP	FF	LUT	Lat.	Thr. (KHz)
128	12	10	3,240	4,908	8,086	14.33
512	36	10	3,258	4,945	94,552	1.23
4096	267	10	3,285	4,998	5,401,280	0.025

Block size 128

n	BR	DSP	FF	LUT	Lat.	Thr. (KHz)
128	36	10	3,262	4,949	8,033	14.42
512	132	10	3,280	4,986	93,766	1.24
4096	1035	10	3,307	5,038	5,351,984	0.022

where $\|x\|_0$ represents the number of nonzero entries, or Hamming weight, of x. This objective function may be relaxed to

$$\min_x \|x\|_1 \text{ s.t. } y = \Phi x \tag{8}$$

where $\|x\|_1 = \sum_{i=1}^n |x_i|$, while still ensuring recovery with overwhelming probability. Problems of this form are solvable with a variety of convex optimization techniques, such as linear programming [19].

Reconstruction has been well studied, and numerous successful algorithms have been proposed. Despite this, it is still often true that reconstruction is much more computationally intensive than compression, motivating the use of our accelerated algorithm.

This compression method is more general than it might appear at first. Any signal x for which there exists an invertible $\Psi \in \mathbb{C}^{n\times n}$—the cosine or Fourier transform, for example—such that Ψx is k-sparse may be compressed so long as $\Phi = \Theta\Psi$, where $\Theta \in \mathbb{R}^{m\times n}$. Recovery then becomes $x^* = \Psi^{-1}z$ for z obtained via a CS recovery algorithm.

While this and the following information are sufficient for the purposes of this paper, we direct the reader interested in a more detailed treatment of CS to [18].

B. Decompression via NNLS

The method that we utilize for decompression is derived from that introduced in [20].

Let $x \in \mathbb{R}^n$ be a signal vector, $\Phi \in \mathbb{R}^{m\times n}$ be a compression matrix, and $y = \Phi x$ be a compressed vector. First consider a smoother version of equation (8):

$$\min_x \left\{ \frac{1}{2}\|y - \Phi x\|_2^2 + \lambda \|x\|_1 \right\} \tag{9}$$

In (9) we have relaxed the equality constraint on Φx and y, instead we will accept an approximation. We can also see that augmenting the parameter λ increases the weight of the $\|x\|_1$ term, which reduces the smoothness of the problem. This minimization problem is a general regression technique known as the *least absolute shrinkage and selection operator* (LASSO) introduced in [21].

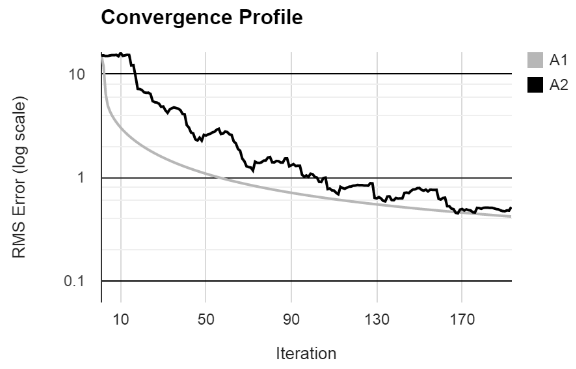

Figure 9. Convergence results for the compressed sensing application. A1 and A2 refer to the first and second algorithm respectively. Final errors are 0.41850 and 0.51593. Nonmonotonicity is clearly evident in A2.

We can see that any $x \in \mathbb{R}^n$ can be written $x = u - v$ for some nonnegative u and v, and then (9) can be written

$$\min_{u,v} \left\{ \frac{1}{2}\|y - \Phi(u-v)\|_2^2 + \lambda e_n^T(u-v) \right\} \text{ s.t. } u, v \geq 0 \tag{10}$$

where e_n is a vector of length n were each entry equals one. Letting $C = \Phi^T\Phi$, and recognizing that $\|y - \Phi x\|_2^2 = (u-v)^T C(u-v) + (u-v)^T\Phi^T y$ we can distribute to get

$$\min_{z\geq 0} \left\{ z^T H z + c^T z \right\} \tag{11}$$

$$\text{where } z = \begin{bmatrix} u \\ v \end{bmatrix},\ c = \lambda e_{2n} + \begin{bmatrix} -\Phi^T y \\ \Phi^T y \end{bmatrix},$$

$$\text{and } H = \begin{bmatrix} C & -C \\ -C & C \end{bmatrix}.$$

We can see that (11) is exactly the NNLS problem. Even though forming H looks computationally daunting, due to our foreknowledge of Φ in the CS framework we can precompute and cache this result. Also note that in practice general sparse recovery only requires two independent runs of the NNLS algorithm to determine u and v, and hence x. For simplicity, we assume the sparse signal is nonnegative when presenting results.

We've chosen a problem size of $n = 128$ for this experiment. As mentioned in section III-A5, problems with larger vectors scale only at the cost of memory required to contain $H^{+/-}$, $c^{+/-}$, and x. Additionally, though more advanced and complex algorithms exist for this particular application, this example demonstrates the numerical stability of the algorithm in the face of a rank-deficient (and therefore not strictly convex) quadratic program.

Figure 9 shows for a problem size of 128 the RMS error over 192 iterations for algorithm 1, and 24 passes of 8 iterations per coordinate for algorithm 2. We see that while algorithm 2 is nonmonotonic, it has periods where it bounces down sharply, catching up with algorithm 1. Final errors are 0.41850 and 0.51593 for algorithm 1 and algorithm 2 respectively.

Table IV
RUN TIME (SECONDS) FOR THE COMPRESSED SENSING EXAMPLE GIVEN AN ITERATION COUNT OF 192. NOTE: RESULTS FOR A1 ARE TRUNCATED DUE TO INSUFFICIENT FPGA RESOURCES.

Size	A1	A2 32	A2 128	ARM	i7
64	0.00231	0.13919	0.02883	0.08073	0.00101
128	0.00612	0.95766	0.23707	0.32037	0.00901
256	~	1.91780	0.47837	1.26987	0.02302
512	~	3.83701	0.95702	6.48921	0.09407

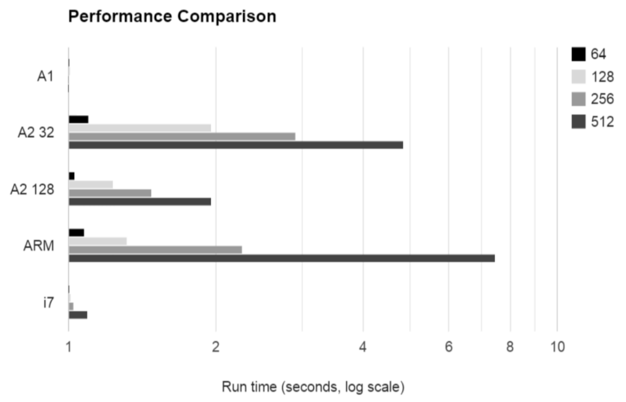

Figure 10. Performance results from table IV in graphical form. Time is in seconds for 192 iterations on differing hardware. A2 32 and A2 128 are for the second algorithm with block sizes of 32 and 128. We can see that data transfer takes a majority of the time for A2—increased block size allows fewer data transfers.

In table IV and figure 10 we compare performance for this test across multiple architectures. In this table A1, A2 32, and A2 128 refer to architecture 1 and architecture 2 with block sizes of 32 and 128. A1 would not fit on the programmable logic portion of the Zynq for matrices larger than 128. These measurements were taken both on the ARM core of a Zynq-7000 (Cortex-A9 dual-core APU at 866 MHz) and an Intel i7-4850HQ CPU at 3.5 GHz (i.e. in "Turbo Boost 2.0" mode). Software versions were compiled with g++ at the O2 optimization level. During testing both software versions were pinned to a single core. We believe the reason for the dramatic speed increase for matrix size 64 on the i7 is due to the entirety of H, c, and $x^{(k)}$ being contained in the 256 Kb L1 data cache. This is not possible for other sizes.

What is most apparent here is the role of transfer overhead in overall performance—greater block size decreases the total number of transfers. While at matrix sizes of 128 and 64, A1 and A2 transfer the same amount of data, A2 makes $(MatrixSize/BlockSize) \times Passes$ transfers, A1 only requires two data transfers total.

V. CONCLUSIONS AND FUTURE WORK

In this work we have discussed two FPGA-based architectures for the general NNLS problem, the second of which based on a novel algorithm. Results show fast convergence and high throughput for one architecture, and superb scalability for the second. Our future efforts will focus on optimizing data transfer using a custom implementation and direct access to off-chip memory rather than a generic FIFO with access arbitrated by a CPU. We will also focus on integration into systems for specific application domains.

REFERENCES

[1] P. O. Hoyer, "Non-negative matrix factorization with sparseness constraints," *The Journal of Machine Learning Research*, vol. 5, pp. 1457–1469, 2004.

[2] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788–791, 1999.

[3] A. Matteson and M. M. Herron, "Quantitative mineral analysis by fourier transform infrared spectroscopy," in *SCA Conference*, no. 9308, 1993.

[4] J. Gonzalez and R. C. Núñez, "Lapackrc: Fast linear algebra kernels/-solvers for fpga accelerators," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012042.

[5] D. Boppana, K. Dhanoa, and J. Kempa, "Fpga based embedded processing architecture for the qrd-rls algorithm," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*. IEEE, 2004, pp. 330–331.

[6] D. Yang, G. D. Peterson, H. Li, and J. Sun, "An fpga implementation for solving least square problem," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, 2009, pp. 303–306.

[7] F. Sha, Y. Lin, L. K. Saul, and D. D. Lee, "Multiplicative updates for nonnegative quadratic programming," *Neural Computation*, vol. 19, no. 8, pp. 2004–2031, 2007.

[8] R. Bro and S. De Jong, "A fast non-negativity-constrained least squares algorithm," *Journal of chemometrics*, vol. 11, no. 5, pp. 393–401, 1997.

[9] C. L. Lawson and R. J. Hanson, *Solving least squares problems*. SIAM, 1974, vol. 161.

[10] M. H. Van Benthem and M. R. Keenan, "Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems," *Journal of chemometrics*, vol. 18, no. 10, pp. 441–450, 2004.

[11] J. Kim and H. Park, "Toward faster nonnegative matrix factorization: A new algorithm and comparisons," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. IEEE, 2008, pp. 353–362.

[12] M. Brand, V. Shilpiekandula, C. Yao, S. A. Bortoff, T. Nishiyama, S. Yoshikawa, and T. Iwasaki, "A parallel quadratic programming algorithm for model predictive control," in *Proc. 18th World Congress of the International Federation of Automatic Control (IFAC)*, vol. 18, no. 1, 2011, pp. 1031–1039.

[13] S. D. Cairano, M. Brand, and S. A. Bortoff, "Projection-free parallel quadratic programming for linear model predictive control," *International Journal of Control*, vol. 86, no. 8, pp. 1367–1385, 2013.

[14] (2015) Packed storage. [Online]. Available: http://www.netlib.org/lapack/lug/node123.html

[15] (2015) Xillinux: A linux distribution for zedboard, zybo, microzed and sockit. [Online]. Available: http://xillybus.com/xillinux

[16] L. Zhuo and V. K. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *IEEE Trans. Computers*, vol. 57, no. 8, pp. 1057–1071, 2008.

[17] E. J. Candes and T. Tao, "Near-optimal signal recovery from random projections: Universal encoding strategies?" *Information Theory, IEEE Transactions on*, vol. 52, no. 12, pp. 5406–5425, 2006.

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

[19] E. J. Candes and T. Tao, "Decoding by linear programming," *Information Theory, IEEE Transactions on*, vol. 51, no. 12, pp. 4203–4215, 2005.

[20] M. A. Figueiredo, R. D. Nowak, and S. J. 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.

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