FPGA Implementation of an Improved OMP for Compressive Sensing Reconstruction

Jun Li, Paul Chow, Yuanxi Peng, and Tian Jiang

***Abstract*—This article proposes an improved orthogonal matching pursuit (OMP) algorithm and its implementation with Xilinx Vivado high-level synthesis (HLS). We use the Gram–Schmidt orthogonalization to improve the update process of signal residuals so that the signal recovery only needs to perform the least-squares solution once, which greatly reduces the number of matrix operations in a hardware implementation. Simulation results show that our OMP algorithm has the same signal reconstruction accuracy as the original OMP algorithm. Our approach provides a fast and reconfigurable implementation for different signal sizes, different measurement matrix sizes, and different sparsity levels. The proposed design can recover a 128-length signal with measurement number $M = 32$ and sparsity $K = 5$ and $K = 8$ in 13.2 and 21 μs, which is at least a 21.9% and 22.2% improvement compared with the existing HLS-based works; a 256-length signal with $M = 64$ and $K = 8$ in 20.6 μs, which is a 24% improvement compared with the existing work; and a 1024-length signal with measurement number $M = 256$ and sparsity $K = 12$ and $K = 36$ in 150.3 and 423 μs, respectively, which are close to the results of traditional hardware description language (HDL) implementations. Our results show that our improved OMP algorithm not only offers a superior reconstruction time compared with other recent HLS-based works but also can compete with existing works that are implemented using the traditional field-programmable gate array (FPGA) design route.**

***Index Terms*—Compressed sensing (CS), field-programmable gate array (FPGA) implementation, orthogonal matching pursuit (OMP) algorithm, sparse signal recovery, Vivado high-level synthesis (HLS).**

I. Introduction

TRADITIONAL signal processing is based on the Nyquist–Shannon sampling theorem [1]. The digital sampling frequency must be at least twice the highest frequency in the original analog signal to fully retain all the information in the analog signal. Tao *et al.* demonstrated that if the signal is sparse, then it can be reconstructed with far fewer samples than required by the Nyquist sampling theorem. The concept of compressed sensing [2] was formally proposed in 2006. It is a technique with the potential to benefit multiple fields such as image and video processing [3], [4], where compressed sensing is used to reconstruct images from a small number of features. It can also be applied in remote sensing, hyperspectral imaging [5], and synthetic aperture radar imaging [6].

Within all the practical applications, one critical issue that the compressive sensing needs to solve is how to reliably recover the original signals from the measured signal in an efficient way. Various algorithms have been proposed to reconstruct signals from the compressively sensed samples. There are several approaches, such as matching pursuit (MP) [7], orthogonal matching pursuit (OMP) [8], and simultaneous OMP (SOMP) [9]. The MP algorithm iteratively looks for the measurement matrix column that is most relevant to the current signal estimation and then performs a simple calculation to update the estimated signal iteratively. The OMP algorithm incorporates a least-squares step to perform signal estimation, which is more accurate than MP. The least-squares step in OMP reduces the number of iterations but increases the complexity at each iteration. The SOMP applies multiple measurement vectors based on OMP, but it significantly increases the complexity and implementation cost. Because of its tradeoff between complexity and accuracy, the OMP algorithm is a good target for hardware implementation to obtain real-time compressive sensing signal reconstruction. The complexity of each iteration of the OMP algorithm is mainly due to a large number of inner product operations and matrix inversion operations. The overall complexity of the OMP is also mainly determined by the size of the measurement matrix and the amount of sparsity. The bottleneck of this algorithm is the solution to the least-squares problem [8].

The traditional normalized matrix solution method needs to invert the matrix directly, which is extremely complicated and not favorable to hardware implementation. The mainstream solutions for least squares are based on matrix decomposition, which includes singular value decomposition (SVD), QR decomposition, and Cholesky factorization. The SVD is the most accurate decomposition method, but it is more complex and not easy to implement in hardware. The most popular method for least squares is based on QR decomposition and Cholesky factorization, which improves the computation efficiency of the matrix inversion.

However, there are still significant challenges in how the hardware is implemented. This article presents a novel approach to computing the OMP, implemented with high-level synthesis (HLS), which requires fewer computations and

Manuscript received April 30, 2020; revised July 11, 2020 and August 24, 2020; accepted October 11, 2020. Date of publication October 23, 2020; date of current version January 28, 2021. This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1301104 and Grant 2017YFB1001900 and in part by the National Natural Science Foundation of China under Grant 91648204 and Grant 61803375. *(Corresponding author: Yuanxi Peng.)*

Jun Li and Yuanxi Peng are with the College of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: lijun14@nudt.edu.cn; pyx@nudt.edu.cn).

Paul Chow is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S, Canada (e-mail: pc@eecg.toronto.edu).

Tian Jiang is with the College of Advanced Interdisciplinary Studies, National University of Defense Technology, Changsha 410073, China (e-mail: tjiang@nudt.edu.cn).

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2020.3030906

results in shorter reconstruction times than previous HLS implementations, and has results comparable to traditional field-programmable gate array (FPGA) implementations using hardware description languages (HDLs).

The rest of this article is organized as follows. Section II presents the related works and briefly introduces our work. Section III gives a review of OMP-based compressive sensing reconstruction. In Section IV, we present our improved OMP algorithm and its verification. The FPGA implementation of the improved OMP is given in Section V. Implementation results and analysis are presented in Section VI and the conclusion is given in Section VII.

II. Related Work and Our Approach

Recently, many hardware architectures for OMP implementations have been proposed in the literature. There are primarily two categories: one is traditional FPGA design using HDLs such as Verilog and VHDL, and the other is HLS design. HLS takes a complete behavioral C/C++ description of a system instead of using HDL to implement a design architecture and automatically generates an HDL design description. HLS is a new design trend. HLS has the advantage of working at a higher level of abstraction making it easier to do design exploration and parameterization to achieve different architectures.

We first provide implementations using HDL with a focus on FPGAs. Septimus *et al.* [10] presented one of the first hardware implementations of the OMP algorithm. Similar to our work, they also use Gram–Schmidt orthogonalization to achieve a more efficient hardware implementation. They demonstrated that this approach can achieve better performance than CPUs and GPUs for one specific configuration, but little is given about the design, such as the number representation and its impact on accuracy. We have done a more complete exploration of a larger configuration and design space enabled by using an HLS approach. Bai *et al.* [11] developed a fixed-point implementation of OMP based on QR decomposition on an FPGA with a certain range of supported signal sparsity. Blache *et al.* [12] presented a design where the matrix inversion is computed based on a coordinate rotation digital computation (CORDIC) algorithm. Several operators, such as matrix–vector multiplication, are reused to optimize both area and execution time. Ren *et al.* [13] provided a single-precision FPGA implementation that shares the computing resources among different tasks of OMP by using configurable processing elements. Rabah *et al.* [14] reduced the execution time of OMP by improving the matrix inversion based on the Newton–Raphson iteration and reusing the hardware for matrix–vector multiplications.

The study in [15] is an FPGA implementation using the technique referred to as matrix inversion bypass (MIB) that decouples the computations of intermediate signal estimates and matrix inversions to reduce the computational complexity.

In recent years, there have been a number of HLS-based implementations of compressed sensing. A Xilinx Vivado HLS implementation for OMP using QR decomposition is presented in [16] that solves the least-squares problem and avoids the square root operations. Kerdjidj *et al.* [17] proposed a hardware architecture consisting of compressing and recovering for electrocardiogram (ECG) signal analysis. Korrai *et al.* [18] developed the least-squares process using the QR decomposition for mmWave indoor sparse channel estimation. Knoop *et al.* [19] provided a rapid digital design flow based on HLS for OMP. Kim *et al.* [20] reported an HLS-based FPGA design of the OMP algorithm that is improved by a novel partitioned inversion technique.

The related works mentioned earlier all focus on simplifying the challenging least-squares problem in OMP. However, because the least squares must be solved in each iteration, the computational complexity remains high. In this article, we propose an efficient OMP implementation on FPGA for compressive sensing reconstruction. The most distinctive contribution is that, different from other related works, we introduce Gram–Schmidt orthogonality [21] to the signal residual update, which enables the omission of the least-squares solution in each iteration. The least-squares solver only needs to be executed once in the final iteration. At the final iteration, we develop the least-squares process by using the Cholesky factorization.

III. OMP-Based Signal Reconstruction

In this section, we present a brief overview of the theory behind OMP-based signal reconstruction. Compressed sensing is a breakthrough theory that acquires and reconstructs a signal from fewer samples. It takes advantage of the fact that most natural signals are sparse in a specific transform domain. For example, consider image reconstruction, a common application of compressed sensing. If an image is transformed into the wavelet domain, where a few coefficients can represent most of the information, the remaining small coefficients can be approximated by zero.

Consider signal $x \in R^N$, which is generally not sparse in nature, but sparse in an $N \times N$ transform domain $\Psi = [\Psi_1\ \Psi_2, \ldots, \Psi_N]$

$$x = \Psi\theta \tag{1}$$

where $\theta \in R^N$ are the coefficients of the expansions, which is an equivalent representation of the signal in the Ψ domain. Here, θ is K-sparse, $K \ll N$, where only a small number of elements are nonzero, whereas the other elements are either zero or nearly zero.

Compressed sensing employs an observation matrix $\Phi \in R^{M\times N}$, which is fixed and independent of the signal x. The observation vector $y \in R^M$ is written as

$$y = \Phi x = \Phi\Psi\theta = A\theta \tag{2}$$

where $A = \Phi\Psi \in R^{M\times N}$, $M < N$, is the measurement matrix. Therefore, an original signal $x \in R^N$ is encoded into a measurement $y \in R^M$ through a linear transformation $y = A\theta$ by a measurement matrix $A \in R^{M\times N}$.

To reconstruct the original signal, many efforts have been made to find an approximate solution. The l_0-minimization can be described as

$$\hat{x} = \arg\min_{x} \|x\|_0 \quad \text{s.t. } Ax = y \tag{3}$$

where $\hat{x}$ is the estimated sparse representation of x. $\|*\|_0$ is the pseudonorm of l_0, which represents the number of nonzero elements.

The reconstruction of the original signal x is an NP-hard problem. The greedy algorithms are directed to finding alternative solutions. The MP [7] is one of the most reliable greedy algorithms. The idea of the MP algorithm is simply to choose an atom (each column vector of the measurement matrix is an atom) that best matches the signal y from the measurement matrix A, construct a sparse approximation, and find the signal residual. Then, continue to select the atom that best matches the signal residual and repeat the iteration. The signal y can be represented by the linear sum of these selected atoms and the final residual. If the residual value is in a negligible range, the signal y is a linear combination of these selected atoms. Projecting the signal y to the column vector can be divided into two parts, the vertical projection component and the residual. If the vertical projection of the signal on the selected column vector is nonorthogonal, this will make the result of each iteration nonoptimal, resulting in many iterations for convergence.

The improvement of the OMP algorithm is that all selected atoms are orthogonalized in each iteration, which makes the OMP algorithm obtain the optimal solution in each iteration. The residual error in each iteration of the algorithm is orthogonal to the selected column vector, which makes the currently selected column vector linearly independent of the previously selected column vector, that is, the column vector will not be selected repeatedly in each iteration. This is why OMP is faster to converge than MP. The basic OMP [8] is now described.

Generally, the basic idea of OMP is to find the indexes of all the nonzero components in the sparse coefficients through an iterative search and recovering one nonzero component by solving the least-squares problem at each iteration. In the nth iteration, the best atom in measurement matrix A is selected to match the observation signal residual r (initialized as the observation vector y). The best matching atom is selected and added to a subset of the observation matrix, named $\hat{A}$. The selected column does not need to be cleared because it is orthogonal to the residual r and will not be selected again in the next iteration. Based on the latest selected subset $\hat{A}$, a new estimated sparse coefficients $\hat{\theta}$ and a new observation vector residual r are calculated through the least squares. In the next iteration, update $\hat{A}$ by adding the atom that best matches the observation vector residual. Then, calculate $\hat{\theta}$ and r until the end of the iteration. Finally, the reconstructed signal $\hat{x}$ can be obtained by (1). Algorithm 1 shows the pseudocode of the OMP-based signal reconstruction.

Lines 3 and 5 of Algorithm 1 are the key parts of the algorithm. Line 3 is the step for finding one atom in matrix A that is most correlated with the current residual r, which is the atom selection step. In the atom selection, a larger inner product value indicates greater correlation. Line 5 is the step to solving the least-squares problem, named the least-squares step, which is a key part of the OMP algorithm.

For the least squares, the purpose is to find the distance of l_2 norms in the Euclidean space and to minimize the distance between the vectors $A\theta$ and y, that is,

$$\min_{\theta} \|y - A\theta\|_2. \tag{4}$$

The vector that minimizes the distance is the same as the vector that minimizes the squared distance, so (4) is equivalent

Algorithm 1 OMP Reconstruction Algorithm

```
Input:   The measurement matrix $A \in R^{M\times N}$, the observation signal $y \in R^M$, the target sparsity $K$;
Output:  The reconstructed signal $\hat{x}$;
    %main program
1:  Initialization: $r_0 = y$, $\Lambda = \emptyset$, $\hat{A} = \emptyset$;
2:  for $t < K$ do
3:      Atom Selection: $\lambda_t = arg \max_{j=1,2,\cdots,n} |< r_{t-1}, a_j >|$
4:      $\hat{A}$ Update: $\Lambda_t = \Lambda_{t-1} \cup \lambda_t$, $\hat{A} = \Lambda_{t-1} \cup a_{\lambda_t}$;
5:      Least Squares: $\hat{\theta}_t = arg \min_{\theta_t} \left\| y - \hat{A}\theta_t \right\|$
                                   $= (\hat{A}^T\hat{A})^{-1}\hat{A}^T y$;
6:      Residual Update: $r_t = r_{t-1} - \hat{A}\hat{\theta}_t$;
7:      t=t+1;
8:  end for
9:  $\hat{x} = \Psi\hat{\theta}$
    Where:
    $r_i$: observation signal residual;
    $t$: $t$th iteration;
    $\emptyset$: null set;
    $\Lambda$: set of indexes;
    $\hat{A}$: set of selected atoms;
    $\lambda$: indexes of atoms;
    $a_j$: $j$th column of matrix $A$;
    $\cup$: set union operation;
    $< a, b >$: inner product of a and b;
    $\hat{\theta}_t$: sparse coefficients;
```

to

$$\min_{\theta} \|y - A\theta\|_2{}^2 \tag{5}$$

$$\|y - A\theta\|_2{}^2 = \theta^T A^T A\theta - y^T A\theta - \theta^T A^T y + y^T y. \tag{6}$$

Manipulating the above formula, then, we can get the matrix θ from of the least-squares solution

$$\frac{\partial \|y - A\theta\|_2{}^2}{\partial\theta} = 2A^T A\theta - 2A^T y = 0 \tag{7}$$

$$\theta = \left(\hat{A}^T\hat{A}\right)^{-1}\hat{A}^T y \tag{8}$$

where (8) is a solution in the form of a projection matrix.

Let $C = \hat{A}^T\hat{A}$, (8) is transformed into

$$\theta = (C)^{-1}\hat{A}^T y. \tag{9}$$

Equation (9) involves a matrix inversion multiplication. Matrix C is usually decomposed by the QR decomposition (QRD) or Cholesky decomposition. In this article, we use an alternative Cholesky decomposition-based matrix inversion to solve the least-squares problem. The alternative Cholesky decomposition (ACD) [22] decomposes a symmetric and positive definite matrix into a lower triangular matrix L and a diagonal matrix D

$$C = \mathrm{LDL}^T. \tag{10}$$

The ACD takes $(k^3 + 3k^2 - 4k)/6$ multiplications and $(k^3 - k)/6$ subtractions. The total cost of operations is approximately equal to $k^3/3$. Polat [23] compared the complexity of

QRD and ACD. The most efficient QR (Householder) method takes $4k^3/3$ operations, more than the ACD. Most importantly, the ACD has no square root, which makes it more appropriate for hardware implementation. The computation of C^{-1} is given in the following equations:

$$C^{-1} = \left(L^{-1}\right)^T D^{-1} L^{-1} \tag{11}$$

$$L_{ij} = \frac{1}{D_{ij}}\left(C_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} D_{kk}\right), \quad i > j \tag{12}$$

$$D_{ii} = C_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 D_{kk} \tag{13}$$

$$L_{ij}^{-1} = -\sum_{k=1}^{i-1} L_{ik} L_{kj}^{-1}, \quad i \neq j. \tag{14}$$

Algorithm 2 Gram–Schmidt Orthogonalization Algorithm

```
Input:   Input matrix $A \in R^{M \times N}$;
Output:  Output matrix $Q \in R^{M \times N}$;
     %main program
 1: $Q_0 = \frac{A_0}{\|A_0\|}$;
 2: for $0 < j < N$ do
 3:     for $0 <= i < j - 1$ do
 4:         $h = \sum_i^{j-1} (A_j{}^T Q_i) Q_i$
 5:     end for
 6:     $\hat{Q}_j = A_j - h$
 7:     $Q_j = \frac{\hat{Q}_j}{\|\hat{Q}_j\|}$
 8: end for
     Where: A subscript represents the column of the matrix.
     e.g. $Q_j$ is the $j$th column of $Q$.
```

IV. Our OMP Algorithm and Verification

In the OMP algorithm, the total iteration number is equal to the sparsity of signal K. It is easy to identify that the computation complexity increases as the sparsity of the original signal increases. As the survey of related work showed in Section II, previous research has focused on improving the solution of the least-squares problem, and many efforts have been focused on matrix inversion. In this article, we propose a new method to update the signal residuals, which results in reducing the number of least-squares operations and improving the efficiency of the atom selection.

A. Our OMP Algorithm

The least-squares step in the original OMP algorithm can be solved by matrix inversion. The computational complexity of direct matrix inversion is $O(N^3)$. The computational complexity of inversion through the matrix factorization algorithm is $O(N^3/3)$. If the dimension of the matrix A is large, the operation amount of matrix inversion is still significant. Therefore, if the least-squares solution is performed in each iteration of the OMP algorithm, significant computation will be required. To solve the abovementioned problem, we consider finding the atom set first, which is the relevant columns of A. Then, the reconstructed values of the signal are obtained by solving an overdetermined least-squares equation. Theoretically, an orthonormal set of atoms can be created by using Gram–Schmidt [21]. For a set of orthogonal atoms and d-dimensional vectors, at most d orthogonal directions can be found. Our procedure sequentially subjects the selected atoms to Gram–Schmidt orthogonalization, and then, the residual is obtained by subtracting the orthogonalized components of atoms from the signal to be decomposed. Hence, our strategy can achieve the same atom set as the original OMP algorithm. The details of the improvement are as follows.

In Line 5 of Algorithm 1, we introduce the Gram–Schmidt orthogonalization to obtain the orthogonal matrix Q_t of atom set Λ. Then, the observation vector residual is updated. Line 6 can be expanded as follows:

$$r_t = r_{t-1} - \hat{A}\left(\hat{A}^T \hat{A}\right)^{-1} \hat{A}^T r_{t-1} = r_{t-1} - P r_{t-1} \tag{15}$$

where $P = \hat{A}(\hat{A}^T \hat{A})^{-1} \hat{A}^T$ is a projection matrix. Equation (16) and Line 6 of Algorithm 1 lead to (18). The projection matrix can be simplified as

$$Q_t \left(Q_t{}^T Q_t\right)^{-1} Q_t{}^T = Q_t Q_t{}^T \tag{16}$$

where Q_t is a standard orthogonal matrix, which can be obtained by the Gram–Schmidt orthogonalization algorithm. For our improved OMP algorithm, Lines 5 and 6 in Algorithm 1 are replaced by (17) and (18), respectively

$$Q_t = \mathrm{GS}\left(\hat{A}\right) \tag{17}$$

$$r_t = r_{t-1} - Q_t Q_t{}^T r_{t-1} \tag{18}$$

where function GS represents the Gram–Schmidt orthogonalization. The Gram–Schmidt orthogonalization is a method to find the orthogonal basis of a Euclidean space. Starting from any linearly independent vector group $\alpha_1, \alpha_2, \ldots, \alpha_m$ in Euclidean space, the orthogonal vector group $\beta_1, \beta_2, \ldots, \beta_m$ is obtained. The two vector groups are equivalent. Then, each vector in the orthogonal vector group is normalized to get a standard orthogonal vector group. Algorithm 2 shows the pseudocode for Gram–Schmidt orthogonalization.

Algorithm 3 shows the flow of our OMP algorithm. It can be observed that the optimized OMP algorithm does not need to perform the least-squares operation in each iteration, but it only performs the least squares once after all iterations are completed, which greatly reduces the computational complexity.

B. Complexity Analysis

Our OMP algorithm can be divided into four main parts: atom selection, $\hat{A}$ update, Gram–Schmidt, residual update, and least squares. The atom selection and $\hat{A}$ update steps are the same as the original OMP in Algorithm 1. The atom selection step is the computation of the most correlated vector, which mainly involves the inner product of a matrix and a vector. Then, the set of selected atoms, $\hat{A}$, is updated by augmenting the column from A with the largest inner product value. The Gram–Schmidt step computes the standard orthogonal matrix of $\hat{A}$. Note that when the Gram–Schmidt is integrated into our OMP algorithm, the outer for-loop of Gram–Schmidt is

TABLE I

COMPUTATION COMPLEXITY OF OMP ALGORITHM AND THE IMPROVED OMP ALGORITHM (M IS THE SIZE OF OBSERVATION VECTOR, N IS THE SIZE OF ORIGIN SIGNAL, AND K IS THE DEGREE OF SPARSITY)

Algorithm	Steps		Multiplication	Addition/Subtraction
OMP	Atom Selection		MNK	$(M-1)NK$
	Least Squares	C	$M\sum_{k=1}^{K} k$	$(M-1)\sum_{k=1}^{K} k$
		C^{-1}	$(4K^3+3K^2-7K)/6$	$(K^3-K^2)/2$
		$\hat{\theta}$	$(2K^3+3K^2-5K)/6$	$(K^3-K)/6$
	Residual Update		$M\sum_{k=1}^{K} k$	$M\sum_{k=1}^{K} k$
	Total		$K^3+(M+1)K^2+(MN+M-2)K$	$(4K^3+(6M-9)K^2+(6MN+6M-6N-4)K)/6$
Our OMP	Atom Selection		MNK	$(M-1)NK$
	Gram-Schmidt		$4MK+2M\sum_{k=1}^{K-1} k$	$2(M-1)K+(2M-1)\sum_{k=1}^{K-1} k$
	Residual Update		$M^2K+M\sum_{k=1}^{K} k$	$M^2K+M\sum_{k=1}^{K}(k-1)$
	Total		$(3MK^2+(2M^2+2MN+7M)K)/2$	$((3M-1)K^2+(2M^2+2MN+M-2N-3)K)/2$

Algorithm 3 Our OMP Reconstruction Algorithm

```
Input: The measurement matrix $A \in R^{M\times N}$, The observation signal $y \in R^M$, The target sparsity $K$;
Output: The reconstructed signal $\hat{x}$;
    %main program
1:  Initialization: $r_0 = y$, $\Lambda = \emptyset$, $\hat{A} = \emptyset$;
2:  for $t < K$ do
3:      Atom Selection: $\lambda_t = arg \max_{j=1,2,\cdots,n} |< r_{t-1}, a_j >|$
4:      $\hat{A}$ Update: $\Lambda_t = \Lambda_{t-1} \cup \lambda_t$, $\hat{A} = \Lambda_{t-1} \cup a_{\lambda_t}$;
        Gram-Schmidt (Lines 5 to 11):
5:      $Q_0 = \frac{\hat{A}_0}{\|\hat{A}_0\|}$;
6:      j = t;
7:      for $0 <= i < j-1$ do
8:          $h = \sum_i^{j-1}({A_{tj}}^T Q_i) Q_i$
9:      end for
10:     $\hat{Q}_j = A_{tj} - h$
11:     $Q_j = \frac{\hat{Q}_j}{\|\hat{Q}_j\|}$
12:     Residual Update: $r_t = r_{t-1} - Q_t Q_t^T r_{t-1}$;
13:     t=t+1;
14: end for
15: Least Squares: $\hat{\theta} = arg \min_{\theta} \left\| y - \hat{A}\theta_t \right\|$
                              $= (\hat{A}^T\hat{A})^{-1}\hat{A}^T y$;
16: $\hat{x} = \Psi\hat{\theta}$
```

merged with the main outer loop OMP. Otherwise, it will cause many duplicate calculations for Q_j. For the residual update step, the new residual is computed by (18). At the final iteration, the sparse coefficients are obtained by solving the least-squares problem with the full updated set of selected atoms.

Table I compares the number of arithmetic operations (multiplications and additions/subtractions) in the basic OMP shown in Algorithm 1 and our OMP shown in Algorithm 3. Each algorithm is broken into the various parts of the computation, which are then summed and shown as the totals. Since the Λ update step does not involve complex matrix operations, it is not included here. The number of arithmetic operations in the basic OMP is $O(K^3)$, while it is $O(K^2)$ for our OMP. However, for a more concrete example consider $M = 32$, $N = 128$. Fig. 1 shows how the numbers of multiplications and additions grow as a function of K. For these parameters, it can be seen that the number of multiplications in the two algorithms is very similar but for additions and subtractions, OMP is significantly worse as K increases.

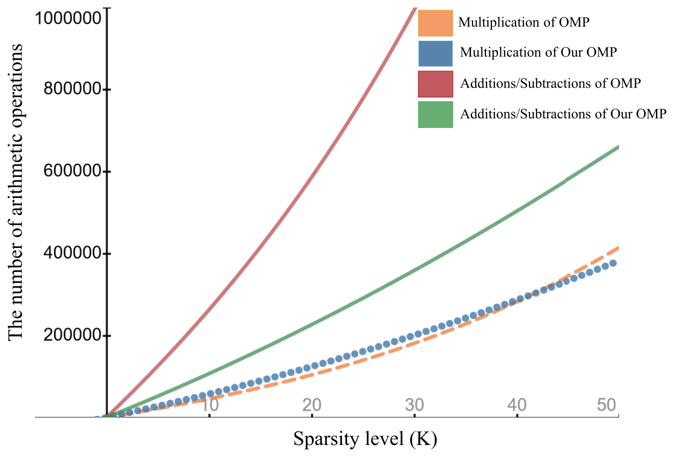


Fig. 1. Growth trend of the number of arithmetic operations.

Our OMP algorithm reduces the total number of operations, especially with larger K values. However, our algorithm still requires a large number of computations. The steps in the main loop have to be performed sequentially and repeat for each iteration because of its iterative nature. The functions within the main loop are also data dependent. As a result, there is no opportunity for parallelism within iterations and between functions. The computational complexity is mainly due to the matrix operations and is related to the size and sparsity of the signal.

Based on the above-mentioned analysis, the only possible way to speedup the algorithm is to introduce parallelism within the functions. By analyzing each function in the algorithm, it is easy to find that there are a large number of matrix and vector multiplications, vector and vector multiplications, and matrix–matrix multiplications. There exist several computing components that can be reused. Details are discussed in Section V.

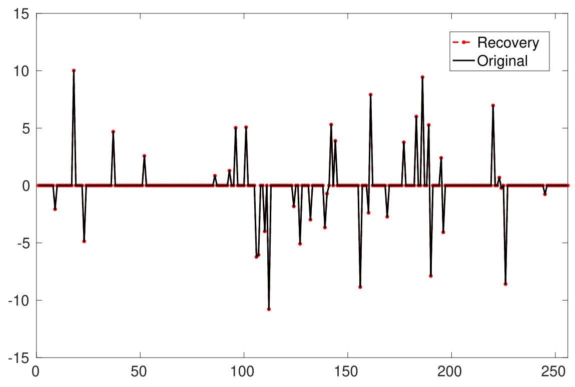


Fig. 2. Reconstruction result of our OMP algorithm, $M = 200$, $N = 256$, and $K = 36$.

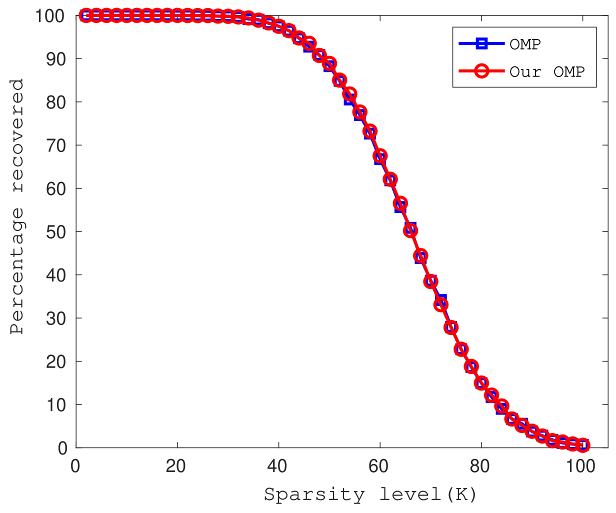


Fig. 3. Success rate comparison, $M = 200$ and $N = 256$, using 1000 trials.

C. Simulations

This section shows that our OMP is an efficient algorithm for signal reconstruction. While it reduces the computational complexity, it does not compromise on the accuracy of the algorithm. The simulation experiments are performed using MATLAB.

To ensure the credibility of the results, 1000 trials are performed in each simulation experiment. The success rate is defined as the proportion of successfully reconstructed trials. For each trial, the original signal is a random sparse signal, where K components out of N are chosen randomly and set equal to a random value, whereas other components are set to zero. The measurement matrix $\Phi \in R^{M \times N}$ is generated with a Gaussian random distribution. The observation vector y is obtained by (2). If the reconstructed signal $\hat{x}$ is identical to the original signal x, which means $\|x - \hat{x}\|_2 < 10^{-6}$, then we claim that the reconstruction is successful.

Fig. 2 shows the reconstruction result of one of our trial signals to visually show how well our reconstruction is working. The signal is with a length of $N = 256$ and $K = 36$, which represents 36 nonzero elements in a total of 256 elements. The reconstructed signal matches well with the original signal visually, and the Euclidean distance is less than 10^{-6}. Fig. 3 shows the success rate of OMP and our OMP as a function of sparsity K. K ranges from 5 to 125 with a step of 5. The two curves are basically coincident. The slight deviation is because the success rate is a statistical result based on random inputs. It can be seen that as the sparsity increases, the success rate decreases. Our OMP shows no sacrifice for reconstruction accuracy.

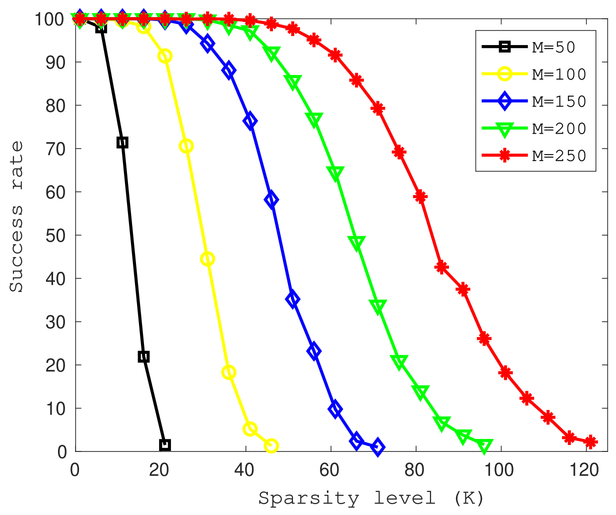


Fig. 4. Success rate as a function of sparsity, for $N = 256$ and varying M, the number of measurements.

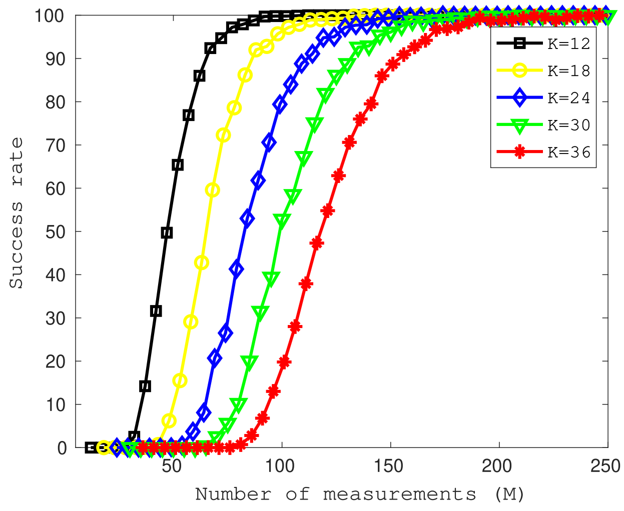


Fig. 5. Success rate as a function of measurement number, for $N = 256$ and varying K, the sparsity level.

Parameters K and M are the key factors for our OMP algorithm. In Fig. 4, we show the success rate of the improved OMP as a function of sparsity, where N is set to 256 and M varies. It shows that for a fixed measurement number M, the success rate decreases as the sparsity increases. For a fixed sparsity, taking more measurements leads to a better result. Fig. 5 shows the percentage of signals reconstructed correctly as a function of the measurement number with various sparsities. It shows that when the sparsity increases, more measurements are needed to maintain signal reconstruction effectiveness.

For applications more complex than signal reconstruction, OMP helps to improve real-world signal processing, such as image, audio, and ECG. Fig. 6 shows the reconstruction results of a 256 × 256 image "Lena" and the peak signal-to-noise ratio (PSNR) performance at different sample rates (M/N). The sample rate is defined as the ratio of the number

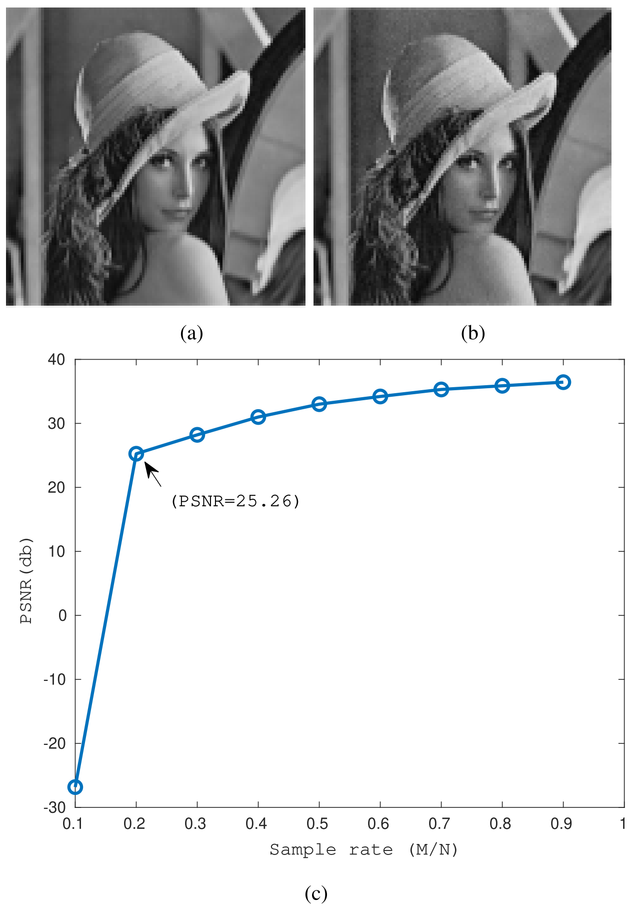


Fig. 6. Image reconstruction results and PSNR using our OMP algorithm. (a) Original image. (b) Reconstructed image. (c) PSNR at different sample rates (M/N).

of measurements M to the signal length N. The higher the sample rate, the better performance for image reconstruction. It can be seen that the image reconstruction fails when M is too small. An inflection point appears when the sample rate is 0.2, where the number of measurements satisfies the restricted isometry property (RIP) [24]. When the number of measurements $M > K * \log(N/K)/\delta^2$, OMP can provide the exact recovery of arbitrary signals, where δ is an isometry constant.

V. Implementation

This section first introduces the Vivado HLS tool [25] and then provides details of our Vivado HLS implementation. Fig. 7(a) shows the architecture and Fig. 7(b) shows the data-flow graph for the implementation of our OMP algorithm. The architecture can be divided into four units: atom selection, Gram–Schmidt, residual update, and least squares, and we will explain the implementation according to these units. In the data-flow graph, we use different colored lines to represent different steps, where Steps 1–3 correspond to the atom selection unit, Steps 4–6 correspond to the Gram–Schmidt unit, and Steps 7 and 8, respectively, correspond to the residual update and least-squares steps. The symbol $*$ indicates the beginning and end of each cycle.

In our OMP algorithm, there are many matrix–vector and matrix–matrix multiplications. Their basic operations consist of inner products between two vectors. A parallel matrix multiplier is designed to process vectors, and the same design is used in each stage of the algorithm: atom selection, Gram–Schmidt, residual update, and least squares. Since this multiplier is used in all stages, we will describe it before describing each stage.

The OMP algorithm is widely used in many complex applications that require higher data accuracy. The variant nature of different applications and the unpredictability of data range make it infeasible to mathematically ensure the effect of fixed-point computation. The floating-point operation ensures a much larger dynamic range, which is particularly important in providing a general solution for processing unpredictable dynamic data. Floating point enables our OMP algorithm to be easily used in the realization of different tasks without requiring the user to adjust the implementation corresponding to the target application. Rabah *et al.* [14] and Liu *et al.* [26] showed the influence of data precision on the quality of signal reconstruction. It has been shown that 32-bit data precision with 22 bits for the fractional part, which is close to floating point, allows reconstruction with better performance. Hence, all computations in our work are performed using a single-precision floating-point format. An advantage of our use of HLS is that it is possible to change to fixed point without too much difficulty if the dynamic range of the target application space is understood.

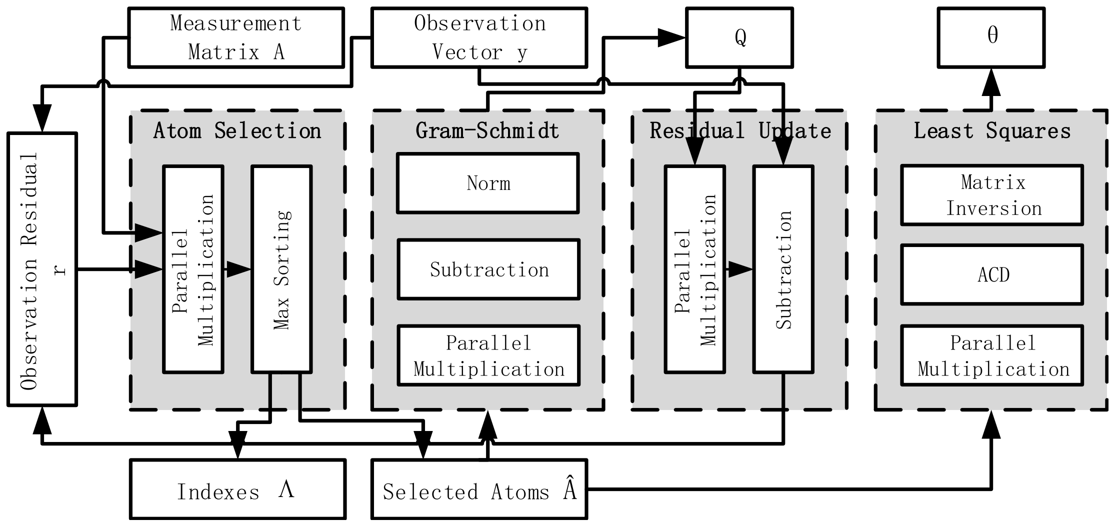
A. Vivado HLS

The complexity of hardware designs is growing quickly, but the time-to-market needs to be reduced to improve design productivity. The main goal of HLS tools is to reduce the time-to-market of the design [27].

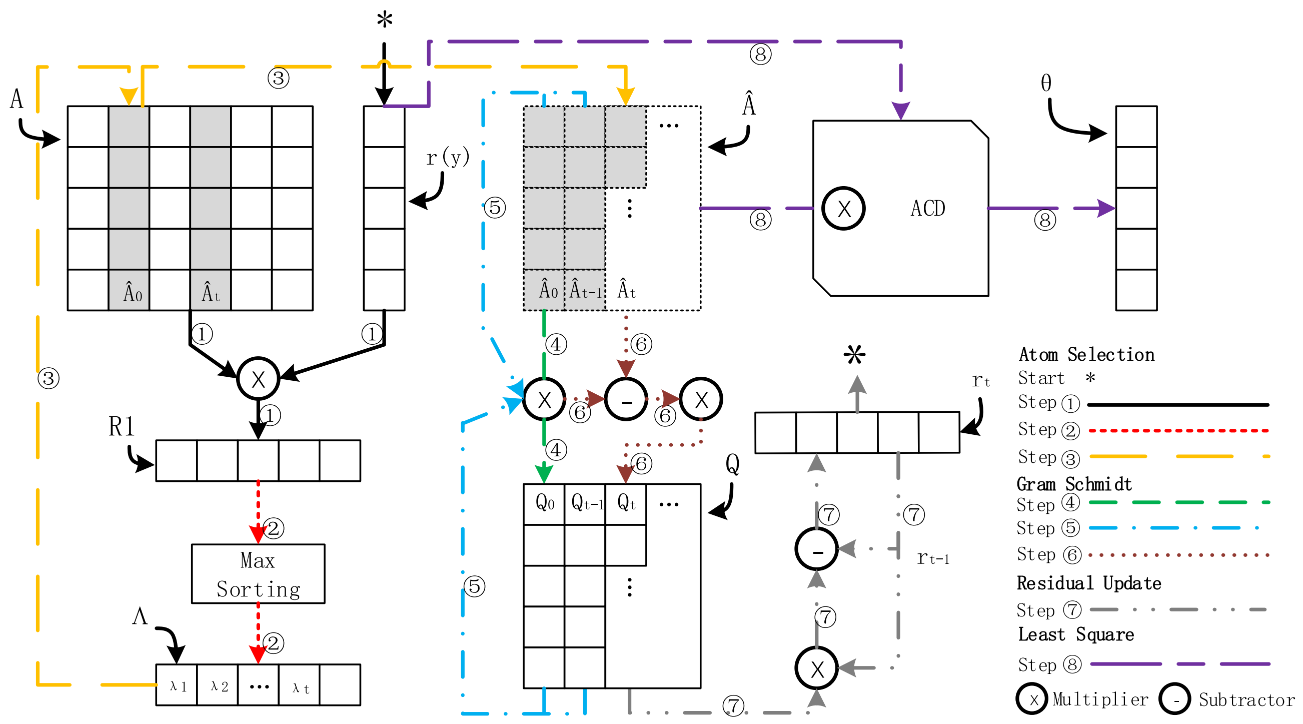
Xilinx Vivado HLS [25] uses C, C++, or SystemC as the description language and the tool will generate VHDL, Verilog, and SystemC register-transfer-level (RTL) descriptions from the HLS model after considering the input constraints. Vivado HLS has different directives and pragmas to provide constraints that are used to optimize the hardware of the generated design according to specifications and enable customization of features, such as interfaces, parallelization levels, and data types. A C/C++ testbench is applied in C/C++ simulation to ensure that the model works with different test vectors. Once the design passes C/C++ simulation and it is functionally correct, the HLS synthesis generates the corresponding RTL description. This RTL code can be verified with C or RTL cosimulation by using the previously defined C/C++ testbench and test vectors. Finally, designers can know whether the RTL design complies with the timing and area constraints by observing the HLS synthesis results [25].

B. Parallel Matrix Multiplier

The inner product operations formed as $a = a + b[i] \times c[i]$ occur frequently in loops. The simplest approach is to execute the computations using the loop as described. To achieve more parallelism, the multiplication can be decoupled from the accumulation so that loop unrolling and pipelining can be used. All of the multiplications can then be done in parallel and the results stored in an intermediate vector register. When



(a)



(b)

Fig. 7. (a) Architecture and (b) data-flow graph for implementation of our OMP algorithm.

the multiplications are finished, an adder tree can sum the values in the intermediate vector register to obtain the inner product. Fig. 8(a) shows the structure of the simple loop-based multiplier and Fig. 8(b) shows the parallel design that we used.

Every for-loop of the adder tree is unrolled and each multiplier is pipelined to handle more operations simultaneously. By doing so, the implementation requires multiple concurrent loads and stores from memory. To achieve the concurrence requirement, the input data and the intermediate variables in our OMP kernel are cyclicly interleaved in multiple small-sized block RAMs instead of stored in a large block RAM. This enables multiple load/store accesses to the memory block in each clock cycle. An appropriate partition and unroll factor gives a reasonable tradeoff between the throughput and resource utilization. We choose to expand the matrices and for-loops to the greatest extent to obtain the best computing efficiency, but the cost is more resource consumption.

Our OMP algorithm has a large number of multiply and add/subtract operations that can be performed in parallel. Architecture synthesis by Vivado HLS heavily depends on compiler directives (pragmas). The pragma HLS PIPELINE is used to automatically pipeline stages to reduce loop latencies. The code in Fig. 9 gives an example of matrix and vector multiplication demonstrating how pragmas are used.

Cooperating with the pragma HLS UNROLL, all sublevel loops are unrolled and thus highly parallelized. As such,

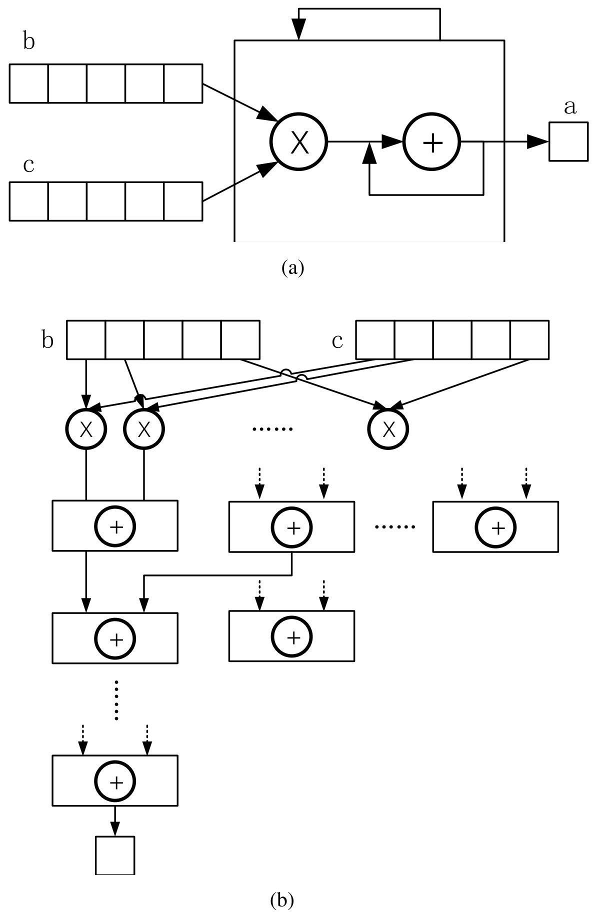
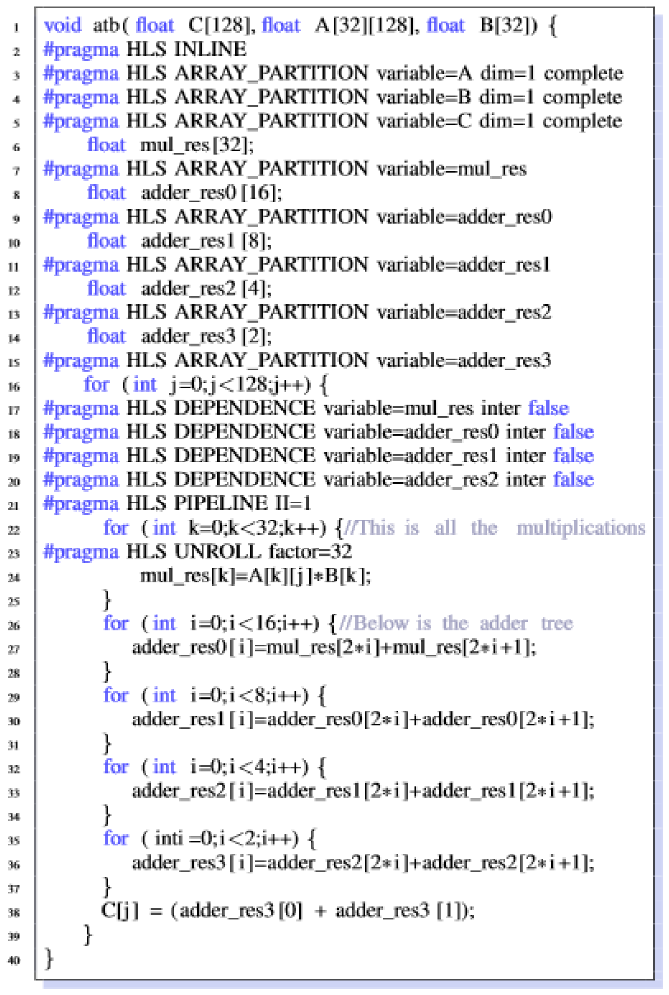


Fig. 8. Schematics of vector–vector multiplication designs. (a) Simple loop-based multiplier. (b) Parallel multiplier that we used.

our implementation requires multiple concurrent loads and stores from a particular RAM to access multiple elements of the matrices and removing the dependence in the for-loops. To ensure that the PIPELINE and UNROLL pragmas work effectively, the HLS ARRAY_PARTITION is used for cyclic array partitioning. Then, the matrixes are filled in smaller sized block RAMs by cyclic interleaving. As the adder-tree-based vector–vector multiplication is widely used in our implementation, the HLS INLINE pragma ensures that the adder_tree functions and their submodules are dissolved and raised to the top level. Several HLS DEPENDENCE pragmas are used in the multiplier to avoid unnecessary dependence in the temp arrays, so the sublevel loops can operate in parallel. Through reasonable parallel design and necessary loop unrolling along with pipeline optimization, we are finally able to reduce the iteration interval and handle as many operations simultaneously as possible.

Table II shows the tradeoff between resource usage and latency with various pragma settings for the matrix–vector multiplier in Fig. 9. Solution 1 has no pipelining or loop unrolling. Solutions 2 and 3 have no pipelining in the outer loop, but the inner loop is unrolled 16 and 32 times, respectively. Solutions 4 and 5 show the impact of outer loop pipelining with the Initiation Intervals (II) set to 2 and the default of 1, respectively. When the outer loop is pipelined, the inner loop will automatically be fully unrolled. UNROLL



```
void atb( float C[128], float A[32][128], float B[32]) {
#pragma HLS INLINE
#pragma HLS ARRAY_PARTITION variable=A dim=1 complete
#pragma HLS ARRAY_PARTITION variable=B dim=1 complete
#pragma HLS ARRAY_PARTITION variable=C dim=1 complete
     float mul_res[32];
#pragma HLS ARRAY_PARTITION variable=mul_res
     float adder_res0 [16];
#pragma HLS ARRAY_PARTITION variable=adder_res0
     float adder_res1 [8];
#pragma HLS ARRAY_PARTITION variable=adder_res1
     float adder_res2 [4];
#pragma HLS ARRAY_PARTITION variable=adder_res2
     float adder_res3 [2];
#pragma HLS ARRAY_PARTITION variable=adder_res3
     for ( int j=0;j<128;j++) {
#pragma HLS DEPENDENCE variable=mul_res inter false
#pragma HLS DEPENDENCE variable=adder_res0 inter false
#pragma HLS DEPENDENCE variable=adder_res1 inter false
#pragma HLS DEPENDENCE variable=adder_res2 inter false
#pragma HLS PIPELINE II=1
        for ( int k=0;k<32;k++) {//This is all the multiplications
#pragma HLS UNROLL factor=32
            mul_res[k]=A[k][j]*B[k];
        }
        for ( int i=0;i<16;i++) {//Below is the adder tree
            adder_res0[i]=mul_res[2*i]+mul_res[2*i+1];
        }
        for ( int i=0;i<8;i++) {
            adder_res1[i]=adder_res0[2*i]+adder_res0[2*i+1];
        }
        for ( int i=0;i<4;i++) {
            adder_res2[i]=adder_res1[2*i]+adder_res1[2*i+1];
        }
        for ( inti =0;i<2;i++) {
            adder_res3[i]=adder_res2[2*i]+adder_res2[2*i+1];
        }
        C[j] = (adder_res3[0] + adder_res3 [1]);
     }
}
```

Fig. 9. Code of a multiplier.

TABLE II
TRADEOFF BETWEEN RESOURCE USAGE AND LATENCY WITH VARIOUS PRAGMA SETTINGS

	PIPELINE/ UNROLL	Block RAMs	DSP	FF	LUT	Latency
Solution 1	No/No	0	5	2806	3285	43137
Solution 2	No/factor=16	0	5	4606	4544	31361
Solution 3	No/factor=32	0	98	7491	7121	22913
Solution 4	II=2/No	0	80	9817	6630	280
Solution 5	II=1/No	0	158	17319	12090	152

"factor $= 16$" indicates the code in the for-loop is expanded into 16 copies, which is equivalent to using 16 times the resources to implement the structure. The maximum number of unrolled copies is the number of iterations of the for-loop. We can see that Solution 3 obtains a lower latency with more resource usage compared with Solutions 1 and 2. Solution 4 shows that pipelining can significantly reduce latency and even uses fewer resources than Solution 3, except for an increase in flip flops, which is usually not an issue with FPGAs since the flip flops are available whether or not they are used. By roughly doubling the resource usage of Solution 4, Solution 5 further reduces the latency by almost half. These

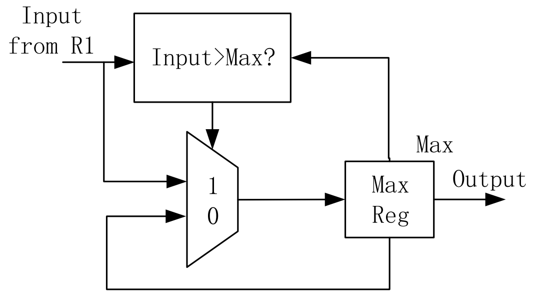


Fig. 10. Structure of the max sorting unit.

results show that by using HLS, it is possible to rapidly explore different architectures and compare their performance and resource use. For this example, it is clear that pipelining is the most effective way to get performance in our multiplier.

C. Atom Selection

The atom selection is used to compute the inner product between the columns of the measurement matrix A and the residual vector r. In Step 1 of Fig. 7(b), N columns of A are read from memory, one column at a time, and multiplied with the current residual in the r vector. The absolute value of each dot product of a column of A with r is stored in register R1. In Step 2, the output of R1 is compared with the current maximum dot product stored in the Max Reg of the Max Sorting unit shown in Fig. 10. If the new dot product for the current column is larger than Max Reg, the Max Reg is updated with the new dot product. These operations are repeated until all columns are completed. After N iterations, the index of the column corresponding to the largest atom is held in Register Λ. Due to the delay in the adder tree, the comparator unit needs to wait for the inner product value to arrive. This results in the atomic selection being completed in $N+m$ clock cycles, where m is the delay of the adder tree.

It should be noted that the number of block RAMs in FPGAs is limited. It is necessary to optimize the use of block RAMs. As mentioned in Section III, the selected columns of matrix A would not be selected twice during one call of the algorithm. This means that A is also unchanged throughout the computation; therefore, instead of separately storing the selected columns of A in another memory to build the $\hat{A}$ matrix, we are able to just build an index table pointing to the relevant columns in A that comprise $\hat{A}$. We do this as part of the Λ register. This is Step 3 and occurs in Line 4 of Algorithm 3. In Fig. 7(b), we show a virtual $\hat{A}$ in dotted lines, with the columns of $\hat{A}$ corresponding to the columns in A that are colored in gray.

D. Gram–Schmidt

The main purpose of the Gram–Schmidt unit is to compute the standard orthogonal matrix Q of the selected atoms matrix $\hat{A}$. Matrix Q is then used in the residual update to ensure that the estimated signal is orthogonal to the residual. Because of the orthogonality, a selected atom will not be selected twice in the following iterations.

In Fig. 7(b), Step 4 shows that in the first iteration, Q_0 is calculated from $\hat{A}_0$. Steps 5 and 6 show the subsequent iterations. In Step 5, the intermediate variable h, shown in Line 8 of Algorithm 3, is computed from Q and $\hat{A}$ in the previous iterations. In Step 6, h is subtracted from the atom of the current loop and divided by its norm to get Q_t. In each iteration, the number of columns in $\hat{A}$ increases, so does Q.

The Gram–Schmidt unit mainly consists of multiple vector–vector multiplications and norm operations, which are also the inner product of two vectors. In the inner loop of this unit, a vector–vector multiplication result is multiplied by another vector and then add to a vector. It is $2(j-1)M$ multiplication and $(j-1)(2M-1)$ addition operations, where j is the current iteration number. For each norm operation, $2M$ multiplications and $M-1$ additions are performed. The total number of operations is shown in Table I.

E. Residual Update

In our OMP, the residual update is defined by (18), where the dimension of the matrix Q_t is $M \times K$. In Fig. 7(b) Step 7, corresponding to Line 12 in Algorithm 3, the updated residual vector of the current iteration r_t is computed form Q_t and the residual vector of the last iteration r_{t-1}. This step mainly involves matrix multiplication and subtraction operations. After this execution is completed, the algorithm returns to the beginning of the next iteration. Computation in the residual update unit consists of an $M \times K$ matrix product with its transposition, multiplied by the residual vector from the last iteration and then subtracted from the last residual vector to obtain the new residual vector for the next iteration. The number of multiplications in the residual update unit in an iteration is $Mk+M^2$, and the number of addition/subtraction is $M(k-1)+M(M-1)+M$. The total numbers are accumulated by iterations, as shown in Table I.

F. Least Squares

When the main loop finishes iterating, $\hat{A}$ with the selected atoms and the observation vector y is input to the ACD-based least-squares unit, as shown in Fig. 7(b) (Step 8).

In the original OMP algorithm, selected atoms from the measurement matrix are used to compute an entry of the signal that is nonzero. The size of the selected atom matrix is gradually increased iteration by iteration. The least-squares estimation must be performed at every iteration to update the residual. Our improved OMP algorithm only needs the least squares to be carried out once after all atoms are selected by the proposed Gram–Schmidt-based approach.

Direct implementation of least squares requires high hardware complexity because of complicated matrix operations and data flows. An approximate solution using alternative Cholesky decomposition can reduce the hardware implementation complexity and can be more suitable for hardware accelerators.

As described in Section III, the matrix inversion for the least-squares estimation is affected by the diagonal matrix D and the lower triangular matrix L. Therefore, it is important to develop parallelism in the decomposition of D and L. The inversion of matrix L is computed in parallel with the inversion of D.

Equations (12) and (13) show that L and D are interdependent with each other. Expanding the calculation of each iteration indicates the relationship between individual components. As an example, the following equations show the first three iterations.

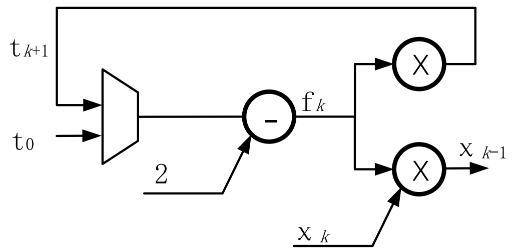


Fig. 11. Architecture of the Goldschmidt algorithm.

Iteration 1:

$$D_{11}^{-1} = C_{11}^{-1}. \tag{19}$$

Iteration 2:

$$\begin{aligned} D_{21} &= C_{21} \\ L_{21} &= D_{11}^{-1} D_{21} \\ D_{22} &= C_{22} - L_{21} D_{21}. \end{aligned} \tag{20}$$

Iteration 3:

$$\begin{aligned} D_{31} &= C_{31} \\ L_{31} &= D_{11}^{-1} D_{31} \\ D_{32} &= C_{32} - L_{21} D_{31} \\ L_{32} &= D_{22}^{-1} D_{32} \\ D_{33} &= (C_{33} - L_{31} D_{31}) - L_{32} D_{32} \end{aligned} \tag{21}$$

$$\begin{aligned} L_{21}^{-1} &= -L_{21} \\ L_{31}^{-1} &= -\left(L_{31} + L_{32} L_{21}^{-1}\right) \\ L_{32}^{-1} &= -L_{32}. \end{aligned} \tag{22}$$

It can be seen that the LDL decomposition only uses multiplication and subtraction operations. The inversion of matrix L is computed iteratively and incrementally by using the results of the previous iterations. The operations involved are multiplication, addition, and arithmetic negation. A Goldschmidt-based reciprocal operator is developed to compute the inversion of matrix D [28]. Rabah [14] has used the Newton–Raphson division [28], which requires finding a function $f(x)$ that has a zero at $X = 1/D$. The key iteration is $x_{k+1} = x_i - (f(x_i)/f'(x_i)) = x_i(2 - Dx_i)$, consisting of multiplication and subtraction operations. However, there is a data dependence between the two multiplication operations in one iteration. While for Goldschmidt division, as shown in Fig. 11, the iterative formula is shown in the following:

$$\begin{aligned} f_k &= 2 - t_k \\ x_{k+1} &= x_k \times f_k \\ t_{k+1} &= t_k \times f_k. \end{aligned} \tag{23}$$

The multiplication operations in the Goldschmidt algorithm can be computed in parallel within an iteration.

As described in (11), the inverse matrix $C^{-1} = (L^{-1} D^{-1} L^{-1})$. Due to the nature of the matrices L and D, the multiplication of L^{-1} and D^{-1} is a symmetric matrix, so C^{-1} is also a symmetric matrix. Therefore, we only need to compute a half triangular of C^{-1}.

TABLE III
PARAMETERS SETTINGS

Sets	$M \times N$	K	Application example
Small	32×128	5/8	e-Health monitoring [29]
Medium	256×256	100	ECG Signal Analysis [17]
Largre	256×1024	12/36	AIC [30]

VI. Implementation Results and Analysis

The Xilinx Vivado HLS tool version 2018.3 is used in our experiments. All synthesis results given in this article target a Xilinx Zynq UltraScale+ MPSoC (XCZU19EGFFVC1760-2-I) at 113 MHz. All computations are performed using single-precision floating-point operations to ensure accuracy.

Several different sets of parameters are used to characterize our OMP algorithm, which we call: 1) small; 2) medium; and (3) large, as shown in Table III. The settings correspond to small-to-large-scale signal reconstruction representing many applications, such as medical signal analysis, mmWave channel estimation, and radar signal processing. Therefore, this design has a universal value for applications based on compressed sensing reconstruction.

The performance of our design is compared with the existing implementations for the OMP algorithm in Table IV. All the small implementations in Table IV use Vivado HLS except for Septimus [10] in Line 1 and Polat [23] in Line 4. Rabah [14] used the MATLAB–Simulink tool along with Xilinx system generator (XSG) and Xilinx LogiCore for the system-level modeling for FPGA implementation. Other than Our OMP in the medium and large categories, all the others are implemented using HDL.

Row 1 shows the result of a VHDL implementation. It obtained a usage below 40% of all resources in a Xilinx Virtex-5 FPGA, and a specific device is not given, with a frequency of 39 MHz. This work has the same approach for OMP reconstruction as our algorithm. Comparing our design to Septimus is difficult. If we just scale the clock speeds, then Our OMP would have a reconstruction time of $113/39 \times 13.2 = 38$ μs, which is slower, but we use floating point and Septimus just reports that they use fixed point, without any indication of the precision. This can account for a significant fraction of the performance difference. In Row 4, we compare with Polat [23] and see that the reconstruction time is less than our time most likely because Polat is using the fixed point compared with our use of floating point, which has much higher latency operations.

The results also show that our OMP has the lowest reconstruction time for all of the other small designs. This is even for the case where some of the other designs use fixed-point numbers and our implementation is using floating-point numbers as well as after scaling for clock frequency differences. However, it can be seen from comparing Rows 2 and 6 that there is a significant resource cost when comparing our OMP with Knoop [19]. Some of that can be attributed to our use of floating point and the rest would be due to the algorithm where we require significantly more block RAMs. Block RAMs are mainly used to store the vectors and matrices involved in the

TABLE IV
COMPARISON OF OUR OMP RESULT WITH EXISTING WORKS

Implementation	$M \times N$	K	FPGA Type	Frequency	Computation Precision	Block RAMs	DSP	FF	LUT	Reconstruction Time
1. Septimus [10]	32 × 128 (Small)	5	Virtex-5	39 MHz	fixed point	-	-	-	-	24 µs
2. Knoop [19]			Virtex-7	100 MHz	fixed point: Q4.14	4	130	13.7K	22.3K	16.9 µs
3. Yu [16]			Kintex-7	100 MHz	-	-	-	-	-	205.985 µs
4. Polat [23]			Virtex-5	115 MHz	fixed point: 18-bit	42	43	8.2K	14.9K	6.26 µs
5. Kim [20]		8	Kintex UltraScale	250 MHz	fixed point: 16- and 32-bit	>307[1]	>2K	>210K	>153K	27 µs
6. Our OMP		5	Zynq UltraScale	113 MHz	floating point: 32-bit	68	254	55.7K	61.1K	13.2 µs
7. Our OMP		8	Zynq UltraScale	113 MHz	floating point: 32-bit	70	254	55.1K	61.3K	21 µs
8. Stanislaus [31]	64 × 256 (Medium)	8	Virtex-5	85 MHz	-	-	-	-	-	27.1 µs
9. Polat [23]			Virtex-5	110 MHz	fixed point: 18-bit	80	81	13.5K	22.6K	23.2 µs
10. Polat [23]		12	Virtex-5	101 MHz	fixed point: 18-bit	88	89	24.7K	40.1K	39.56 µs
11. Our OMP		8	Zynq UltraScale	113 MHz	floating point: 32-bit	132	446	150K	114K	20.6 µs
12. Our OMP		12	Zynq UltraScale	113 MHz	floating point: 32-bit	134	446	151K	116K	33.54 µs
13. Bai [11]	256 × 1024 (Large)	12	Virtex-6	100 MHz	fixed point: 18- and 42-bit	-	-	-	-	158.7 µs
14. Bai [11]		36	Virtex-6	100 MHz	fixed point: 18- and 42-bit	258	261	-	96K	630 µs
15. Rabah [14]			Virtex-6	120 MHz	fixed point: 18-bit	576	589	-	18K	340 µs
16. Polat [23]			Virtex-6/Virtex-7	94/135 MHz	fixed point: 18-bit	-	-	-	-	450/314 µs
17. Our OMP		12	Zynq UltraScale	113 MHz	floating point: 32-bit	518	1.7K	635K	453K	150.3 µs
18. Our OMP		36	Zynq UltraScale	113 MHz	floating point: 32-bit	521	1.7K	596K	440K	423 µs

[1] The number of resources used in [20] is only for the inverting of the matrix, not the entire design.

algorithm, such as y, r, and A. To enable the concurrence, the vectors and matrices are filled in block RAMs by cyclic array partitioning. The block RAMs are also configured for parallel access, so a faster computation can be performed by accessing multiple successive elements of the vectors and matrices in parallel.

For the medium comparisons, we have Stanislaus [31] and Polat [23] in Rows 8–10 to compare with our OMP in Rows 11 and 12. Our reconstruction time is 24% lower than Stanislaus, but the times are similar if the clock frequencies are normalized ($113/85 \times 20.6 = 27.4$ for Our OMP compared with 27.1 for Stanislaus). Stanislaus does not provide any information about resource usage. For Polat, we can observe that our OMP is about 11.2% and 15.2% better than Polat with $K = 8$ and $K = 12$ but requires significantly more hardware. We discuss the hardware size when comparing large designs in the following.

When comparing the large designs, first, note that we have used HLS and the other designs use HDL. We can see that in Rows 13 and 17, the reconstruction times are comparable for $K = 12$ given that we have a slightly faster clock, but we are also using the floating point compared with the use of fixed point in Row 13. Bai did not report resources in Row 13, so we cannot compare that.

For $K = 36$, we compare Rows 15 and 18 and see that the reconstruction time for Rabah in Row 15 is about 20% lower than our time. If we make an adjustment for the difference in clock frequencies, then Rabah is about $(113/120 \times 423 - 340)/423 \times 100 = 14\%$ lower. Rabah uses the 18-bit fixed point compared with our use of 32-bit floating point, which can account for some of the remaining difference as reducing the precision should allow us to increase our clock frequency.

Bai and Rabah used "occupied slices" as the metric of resource usage, but counting slices is not a direct measure of actual resources used. What is actually consumed in terms of logic is lookup tables (LUTs). The number of slices used is a function of LUT packing, which is affected by how hard the tools decide to pack LUTs into fewer slices. If the chip is large, the tools will not work so hard to pack because there are lots of resources available. Also, Bai and Rabah used a different generation of FPGA and slice architecture will also have some impact on LUT packing. A Virtex-6 has four six-input LUTs per slice and the UltraScale has eight six-input LUTs per slice. To make a comparison more consistent, we have made a crude approximation to scale the number of Virtex-6 slices used to a number of LUTs.

We estimate LUT packing to be 75% and multiply the slice counts for Bai and Rabah by $0.75\times4=3$ LUTs/slice. Bai used 32k slices in [11], which we estimate as 96k LUTs. Rabah used 6k slices in [14], which we estimate as 18k LUTs and we enter those scaled values in Table IV. Even with the scaling of LUT counts, it is clear that our OMP architecture uses significantly more LUTs than Bai and Rabah. To understand this difference, we focus on Rabah, who used less resources than Bai. Rabah reused a 64-point inner product unit across the data-dependent sections of the algorithm, which shared the hardware for matrix–vector multiplication, thus leading to significant resource optimization. For our HLS-based design, our first focus was to parallelize all operations as much as possible to try to achieve maximum performance. However, the HLS-synthesized design does not reuse hardware resources even when it might seem possible, but it allocates separate resources for each of the matrix–vector multiplications. This leads to multiple times the number of resources used compared with the other implementations. Our understanding is that this is a current limitation of the HLS tool we are using.

Our OMP unrolls all expandable for-loops and parallelizes a large number of multiplication operations. The amount of parallel computation depends on the size of the measurement matrix ($M \times N$). The length of the signal N only appears at the top level of the for-loop in the atom selection step, while M is the range of many for-loops. Therefore, as shown in Table IV, the number of resources used by our OMP increases significantly as M increases from 32 to 256. Also, it is important to consider the sparsity (K), which is equal to the number of iterations of the main loop. Table I shows that the number of multiplications and additions grows as $O(K^2)$, so we can also expect that as K increases, more hardware will be required based on our aggressive strategy of unrolling loops.

VII. Conclusion

This article has presented an improved OMP algorithm for compressive sensing reconstruction. In our OMP algorithm, Gram–Schmidt orthogonality is introduced to update the residual instead of performing the least-squares solver at every iteration. Due to a very large number of inner products, the original OMP algorithm results in a high computational complexity for the least-squares problem. Therefore, the improvement described in this article can effectively reduce the computational complexity of the matrix operations. We have verified our OMP algorithm using MATLAB. The simulation results show that our OMP algorithm not only reduces the computational complexity but also does not compromise on performance.

To meet the requirement for real-time applications, hardware architecture for our OMP is implemented on an FPGA using the Xilinx Vivado HLS design suite. Due to the advantages of HLS and careful parallel design, the proposed architecture supports different sets of dimensions of the measurement matrix ($M \times N$) and sparsity (K). The implementation results show that our OMP algorithm offers a superior reconstruction time compared with other recent HLS-based works and has a competitive real-time performance with existing HDL-based works for large input signal sizes but with a severe cost in the amount of hardware required because of the way our HLS tool currently works. Differing from most other existing works, our architecture is effective for signals of any size with any sparsity instead of fixed parameters. Unfortunately, our approach of using HLS results in significant hardware growth as M and K increase so that, from a resource perspective, our HLS approach works best for the small designs. We believe that the hardware usage can be significantly reduced in the medium and large designs with more sharing of the hardware, but unfortunately, the current implementation of HLS is unable to do the required sharing to reduce total resource usage.

Future work can be carried out in two directions. One is to address the large resource usage caused by the inability of HLS to do proper resource sharing. While it is possible to simply do an HDL implementation that would have to be for a particular configuration and precision and the flexibility of HLS is lost by reverting to HDL. A more constructive approach would be to address the HLS limitations we have found by exploring different HLS coding techniques or even working with the HLS developers to improve the tool. A second direction is to explore the application of our OMP to more complex real-world applications.

Acknowledgment

The authors would like to thank the State Key Laboratory of High Performance Computing, National University of Defense Technology, China. They would also like to thank the China Scholarship Council (CSC) and Dusan and Anne Miklas Chair for Engineering Design of the University of Toronto for the support.

References

[1] S. H. W. Oppenheim, A. S. Alan, and A. V. Nawab, *Signals and Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, 1996.

[2] D. L. Donoho, "Compressed sensing," *IEEE Trans. Inf. Theory*, vol. 52, no. 4, pp. 1289–1306, Apr. 2006.

[3] R. G. Baraniuk, T. Goldstein, A. C. Sankaranarayanan, C. Studer, A. Veeraraghavan, and M. B. Wakin, "Compressive video sensing: Algorithms, architectures, and applications," *IEEE Signal Process. Mag.*, vol. 34, no. 1, pp. 52–66, Jan. 2017.

[4] J. Li, M. Song, and Y. Peng, "Infrared and visible image fusion based on robust principal component analysis and compressed sensing," *Infr. Phys. Technol.*, vol. 89, pp. 129–139, Mar. 2018.

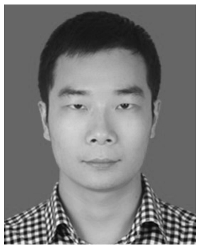
[5] K. Gunasheela and H. Prasantha, "Compressive sensing approach to satellite hyperspectral image compression," in *Information and Communication Technology for Intelligent Systems*. New York, NY, USA: Springer, 2019, pp. 495–503.

[6] J. Yang, T. Jin, C. Xiao, and X. Huang, "Compressed sensing radar imaging: Fundamentals, challenges, and advances," *Sensors*, vol. 19, no. 14, p. 3100, Jul. 2019.

[7] S. G. Mallat and Z. Zhang, "Matching pursuits with time-frequency dictionaries," *IEEE Trans. Signal Process.*, vol. 41, no. 12, pp. 3397–3415, Dec. 1993.

[8] J. A. Tropp and A. C. Gilbert, "Signal recovery from random measurements via orthogonal matching pursuit," *IEEE Trans. Inf. Theory*, vol. 53, no. 12, pp. 4655–4666, Dec. 2007.

[9] J. A. Tropp, A. C. Gilbert, and M. J. Strauss, "Algorithms for simultaneous sparse approximation. Part I: Greedy pursuit," *Signal Process.*, vol. 86, no. 3, pp. 572–588, Mar. 2006.
[10] A. Septimus and R. Steinberg, "Compressive sampling hardware reconstruction," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2010, pp. 3316–3319.
[11] L. Bai, P. Maechler, M. Muehlberghuber, and H. Kaeslin, "High-speed compressed sensing reconstruction on FPGA using OMP and AMP," in *Proc. 19th IEEE Int. Conf. Electron., Circuits, Syst. (ICECS)*, Dec. 2012, pp. 53–56.
[12] P. Blache, H. Rabah, and A. Amira, "High level prototyping and FPGA implementation of the orthogonal matching pursuit algorithm," in *Proc. 11th Int. Conf. Inf. Sci., Signal Process. Their Appl. (ISSPA)*, Jul. 2012, pp. 1336–1340.
[13] F. Ren, R. Dorrace, W. Xu, and D. Markovic, "A single-precision compressive sensing signal reconstruction engine on FPGAs," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–4.
[14] H. Rabah, A. Amira, B. K. Mohanty, S. Almaadeed, and P. K. Meher, "FPGA implementation of orthogonal matching pursuit for compressive sensing reconstruction," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 10, pp. 2209–2220, Oct. 2015.
[15] G. Huang and L. Wang, "An FPGA-based architecture for high-speed compressed signal reconstruction," *ACM Trans. Embed Comput. Syst.*, vol. 16, no. 3, p. 89, May 2017.
[16] Z. Yu *et al.*, "Fast compressive sensing reconstruction algorithm on FPGA using orthogonal matching pursuit," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 249–252.
[17] O. Kerdjidj, N. Ramzan, A. Amira, K. Ghanem, and F. Chouireb, "Design and evaluation of vivado HLS-based compressive sensing for ECG signal analysis," in *Proc. IEEE 16th Int. Conf. Dependable, Autonomic Secure Comput., 16th Int. Conf. Pervas. Intell. Comput., 4th Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol. Congr. (DASC/PiCom/DataCom/CyberSciTech)*, Aug. 2018, pp. 457–461.
[18] P. K. Korrai, K. Deergha Rao, and C. Gangadhar, "FPGA implementation of OFDM-based mmWave indoor sparse channel estimation using OMP," *Circuits, Syst., Signal Process.*, vol. 37, no. 5, pp. 2194–2205, May 2018.
[19] B. Knoop, J. Rust, S. Schmale, D. Peters-Drolshagen, and S. Paul, "Rapid digital architecture design of orthogonal matching pursuit," in *Proc. 24th Eur. Signal Process. Conf. (EUSIPCO)*, Aug. 2016, pp. 1857–1861.
[20] S. Kim *et al.*, "Reduced computational complexity orthogonal matching pursuit using a novel partitioned inversion technique for compressive sensing," *Electronics*, vol. 7, no. 9, p. 206, Sep. 2018.
[21] D. Clayton, "Algorithm AS 46: Gram-schmidt orthogonalization," *J. Roy. Stat. Soc. C (Appl. Statist.)*, vol. 20, no. 3, pp. 335–338, 1971.
[22] S. J. Bellis, W. P. Marnane, and P. J. Fish, "Alternative systolic array for non-square-root Cholesky decomposition," *IEE Proc.-Comput. Digit. Techn.*, vol. 144, no. 2, pp. 57–64, Mar. 1997.
[23] Ö. Polat and S. K. Kayhan, "High-speed FPGA implementation of orthogonal matching pursuit for compressive sensing signal reconstruction," *Comput. Electr. Eng.*, vol. 71, pp. 173–190, Oct. 2018.
[24] E. Livshitz, "On efficiency of orthogonal matching pursuit," 2010, *arXiv:1004.3946*. [Online]. Available: http://arxiv.org/abs/1004.3946
[25] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx vivado high level synthesis: Case studies," in *Proc. 25th IET Irish Signals Syst. Conf. China-Ireland Int. Conf. Inf. Commun. Technol. (ISSC/CIICT)*, Limerick, Republic of Ireland, 2014, pp. 352–356, doi: 10.1049/cp.2014.0713.
[26] S. Liu, N. Lyu, and H. Wang, "The implementation of the improved OMP for AIC reconstruction based on parallel index selection," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 2, pp. 319–328, Feb. 2018.
[27] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.
[28] *Division Algorithm*. Accessed: Mar. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Division_algorithm
[29] O. Kerdjidj, A. Amira, K. Ghanem, N. Ramzan, S. Katsigiannis, and F. Chouireb, "An FPGA implementation of the matching pursuit algorithm for a compressed sensing enabled e-Health monitoring platform," *Microprocessors Microsyst.*, vol. 67, pp. 131–139, Jun. 2019.
[30] S. Liu, N. Lyu, and Z. Wang, "High-speed FPGA implementation of orthogonal matching pursuit for analog to information converter," in *Proc. IEEE 12th Int. Conf. (ASIC ASICON)*, Oct. 2017, pp. 152–155.
[31] J. L. V. M. Stanislaus and T. Mohsenin, "High performance compressive sensing reconstruction hardware with QRD process," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2012, pp. 29–32.



Jun Li received the B.S. degree in microelectronics from Sichuan University (SCU), Chengdu, China, in 2014, and the M.S. degree in microelectronics from the National University of Defense Technology (NUDT), Changsha, China, in 2017, where he is currently pursuing the Ph.D. degree in microelectronics and solid-state electronics.

He was a Visiting Student with the Department of Electronics and Computer Engineering, University of Toronto, Toronto, ON, Canada, from 2018 to 2020. His research interests include hyperspectral image processing, multisource data fusion, and embedded systems and applications.

Paul Chow received the B.A.Sc. degree (Hons.) in engineering science and the M.A.Sc. and Ph.D. degrees in electrical engineering from the University of Toronto, Toronto, ON, Canada, in 1977, 1979, and 1984, respectively.

In 1984, he joined the Computer Systems Laboratory, Stanford University, Stanford, CA, USA, as a Research Associate, where he was a major contributor to an early RISC microprocessor design called MIPS-X, one of the first microprocessors with an ON-chip instruction cache. He joined the Department of Electrical and Computer Engineering with the University of Toronto, in January 1988, where he is currently a Professor and holds the Dusan and Anne Miklas Chair in engineering design. His research interests include high-performance computer architectures, reconfigurable computing, heterogeneous cloud computing, embedded and application-specific processors, and field-programmable gate array architectures and applications.

Yuanxi Peng was born in 1966. He received the B.S. degree in computer science from Sichuan University, Chengdu, China, in 1988, and the M.S. and Ph.D. degrees in computer science from the National University of Defense and Technology (NUDT), Changsha, China, in 1998 and 2001, respectively.

He was a Visiting Professor with the Department of Electronic and Computer Engineering, University of Toronto, Toronto, ON, Canada, from 2010 to 2011. He has been a Professor with the Computer School, NUDT, since 2011. His research interests are in the areas of high-performance computing, multi and many-core architectures, on-chip networks, cache coherence protocols, and architectural support for parallel programming.

Tian Jiang received the B.S., M.S., and Ph.D. degrees in optical engineering from the National University of Defense Technology (NUDT), Changsha, China, in 2007, 2010, and 2012, respectively.

He was a Post-Doctoral Researcher with Tsinghua University (THU), Beijing, China, from 2014 to 2017. He is currently a Faculty with NUDT.