Flexible High-Level Synthesis Library for Linear Transformations

Journal:	*IEEE Transactions on Circuits and Systems II: Express Briefs*
Manuscript ID	TCAS-II-17456-2023
Manuscript Type:	Regular Paper - Letters
Date Submitted by the Author:	14-May-2023
Complete List of Authors:	Zhao, Wuqiong; Southeast University Li, Changhan; Southeast University Ji, Zhenhao; Southeast University Guo, Zhichen; Southeast University Chen, Xuanbo; Southeast University You, You; Purple Mountain Laboratories Huang, Yongming; Southeast University You, Xiaohu; Southeast University Zhang, Chuan; Southeast University,
EDICS:	COMM - Circuits and Systems for Communications, CAD100A5 - Computer aided high level synthesis < CAD100 - Computer aided design and synthesis < CAD - Computer Aided Design and Electronic Design Automation
TCAS-II Subject Category Please select the subject category that most closely fits with the scope of your manuscript:	Circuits and Systems for Communications

Flexible High-Level Synthesis Library for Linear Transformations

Wuqiong Zhao, Changhan Li, Zhenhao Ji, Zhichen Guo, Xuanbo Chen, You You, Yongming Huang, Xiaohu You, *Fellow, IEEE*, and Chuan Zhang, *Senior Member, IEEE*

***Abstract*—Efficient hardware design for signal processing based on linear algebra has been pursued for decades. However, the traditional design workflow involving hardware description languages (HDLs) can be challenging and time-consuming. High-level synthesis (HLS) provides an easier approach, but still requires thorough designs of basic modules concerning linear transformations to achieve acceptable hardware efficiency. To simplify the HLS workflow, we propose the FLAMES library, which provides efficient and ready-to-use linear transformation modules. Users can easily implement algorithms via the FLAMES library In this paper, we demonstrate the effectiveness of the library by implementing the orthogonal matching pursuit list (OMPL) algorithm for compressed sensing in FPGA, achieving 1.56× and 1.12× throughput/slice compared with traditional HLS for the sequential and parallel architecture, respectively.**

***Index Terms*—High-level synthesis (HLS), linear transformations, compressed sensing, hardware implementation, field programmable gate array (FPGA).**

I. Introduction

HIGH-LEVEL synthesis (HLS) for hardware implementation can transcompile high-level programming languages like C/C++ into register-transfer level (RTL) designs [1]–[3]. HLS simplifies the hardware design workflow and lowers the barrier to obtaining efficient architectures [4]. Moreover, it can explore design space to achieve better performance with Pareto-optimal ones [5]. Despite the promising future of HLS, some limitations are inevitable. To optimize hardware design, users have to follow a strict HLS writing guideline [6]. Therefore, hardware design with HLS remains a formidable challenge, violating the HLS concept for a simpler hardware design paradigm.

To alleviate these limitations of HLS, higher-level libraries are constructed for complexity reduction and performance enhancement. The proposed template-based method in [7] shows its superiority in several cases. FBLAS [8] ports the basic linear algebra subprograms (BLAS) library to HLS, promoting HLS design productivity. However, the complicated interfaces of [7], [8] impose inflexibility for users in practical scenarios. We take advantage of `class` and `template` of C++ with Vitis HLS [9] to further reduce the gap between HLS design and software algorithms by providing a user-friendly library with concise and customizable interfaces like Armadillo C++ library [10]. Algorithms can be efficiently implemented with the matrix-based library since an algorithm can be viewed as matrices and their transformations. The optimized hardware can be obtained by simply mapping the matrix transformations into HLS code without the need for extensive design. Our contributions are as follows:

1) We present the novel matrix-based HLS library concept FLAMES (**F**lexible **L**inear **A**lgebra with **M**atrix-**E**mpowered **S**ynthesis) for fast algorithm implementation, which is designed as a header-only C++ library for Vitis HLS, simplifying the design workflow of signal processing modules based on linear transformations;
2) With our designed HLS library, efficient architectures for orthogonal matching pursuit list (OMPL) compressed sensing (CS) are implemented as a verification. The sequential and parallel designs achieve throughput 94.81 Mb/s and 184.5 Mb/s (1.56× and 1.12× throughput/slice compared with traditional HLS), respectively, with significantly reduced design complexity.

Notations: In this paper, lower-case and upper-case boldface letters $\mathbf{a}$ and $\mathbf{A}$ stand for vector and matrix, respectively. $\mathbf{A}^\mathsf{T}$ and $\mathbf{A}^{-1}$ represent the transpose and inverse of matrix $\mathbf{A}$, respectively. $\|\mathbf{a}\|_1$ and $\|\mathbf{a}\|_2$ take the ℓ_1- and ℓ_2-norm of vector $\mathbf{a}$, respectively. $\langle \mathbf{a}, \mathbf{b} \rangle$ denotes the inner product of vector $\mathbf{a}$ and $\mathbf{b}$. $\lfloor x \rfloor$ is the floor function returning the largest integer smaller or equal to x. Finally, $\mathcal{N}(\mu, \sigma^2)$ denotes the Gaussian distribution with mean μ and variance σ^2.

II. HLS Library Design

A. Framework and Syntax

Despite the power of HLS tools, designing hardware-efficient solutions still require substantial knowledge and experience in hardware design. Existing libraries like [7], [8] are not as user-friendly as popular software-based linear algebra libraries like Armadillo [10]. To address the issue, we propose a matrix-based library design where matrix operations and functions are at the center of the design, which enables signal processing algorithms to be expressed in their original form with minimal adjustments without the need for data decomposition or pipeline designs. Table I provides several examples of library syntax. One notable feature of the FLAMES library is its unified user interface, which does not rely on low-level variable types such as C arrays, C++ standard containers, or streams. Furthermore, the library is designed to be header-only, making it easy to install and fully portable. Users only need to include the required header file to access the classes and utilities provided by the FLAMES library.

W. Zhao and C. Li contributed equally to the paper.

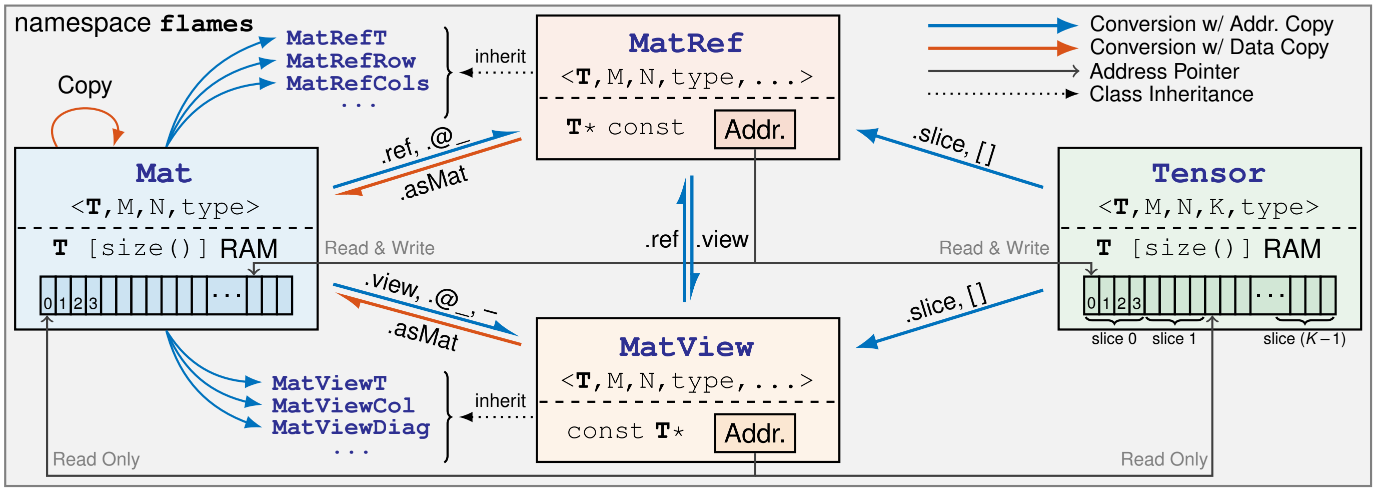


Fig. 1. Classes and their relationships in the FLAMES library, where '@' is a placeholder for function names, such as 't' and 'col'. T is the data type.

TABLE I
LIBRARY SYNTAX EXAMPLES.

HLS Code	Description
`A + B` and `A - B`	addition/subtraction
`A * B` and `A % B`	multiplication/element-wise multiplication
`A += B` or `A.add(B)`	self adding another matrix
`A.t()` and `-A`	transpose/negation
`A(2,3)` or `A.at(2,3)`	accessing element of row 2 column 3
`A.row(1)`	accessing row 1
`A.cols({2,3,4})`	accessing columns 2, 3 and 4
`A.diagMat()`	diagonal as matrix

A and B are both matrices.

B. Basic Structure

The FLAMES library provides several primary classes for matrix and tensor operations, depicted in Fig. 1. These classes include Mat for matrices, MatRef for matrix references, MatView for read-only matrix views, and Tensor for tensors. MatRef and MatView can be used for simple matrix transformations without data copying. To be specific, MatRef has a constant pointer to the original data, while MatView has a pointer to constant values with read-only access. In addition, MatView and MatRef can be converted back to Mat using the .asMat method to allocate the required RAM. The Tensor class allocates RAM slice by slice, and slices can be accessed using a MatRef or MatView with the .slice method or the [] operator by specifying the slice index. Matrices and tensors have different types that determine how the data is arranged. Since FLAMES is written with modern C++, the auto keyword can be used to simplify the code and leave MatView and MatRef to be auto inferred [11].

C. Hardware-friendly Design

In addition to the simple user interface, the FLAMES library is carefully designed for efficient hardware implementation.

1) Optimized RAM Usage: Traditionally, algorithm implementations separate matrices for performance enhancement, adding difficulties to algorithm implementation [12]–[14]. In contrast, the FLAMES library offers a neat matrix-based interface while optimizing RAM usage with different MatTypes. For instance, diagonal matrices (MatType::DIAGONAL) and upper triangular matrices (MatType::UPPER) are efficiently stored exploiting their structural properties, while providing a unified user interface. Furthermore, MatView avoids unwanted intermediate RAMs due to the lack of return value optimization (RVO) in Vitis HLS. For example, the permutation of the transposed matrix (Fig. 2) is achieved using the MatViewT class inherited from MatView, which connects wires to the original data instead of copying and allocating RAM for the transposed matrix.

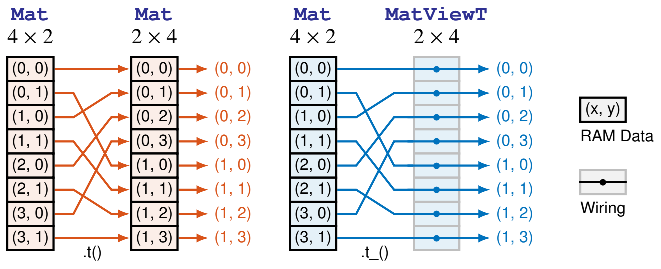


Fig. 2. Permutation network for 4 × 2 transposed matrix.

2) Configurable Parallelism: The library employs the HLS pragma internally provided by Vitis HLS, controlling the pipelining, unrolling, loop flattening, etc. The parallelism for matrix multiplication, matrix copying, etc. can be configured separately, by defining the corresponding macro such as FLAMES_MAT_TIMES_UNROLL_FACTOR. Similarly, the data array partitioning can also be configured.

3) Optimized Matrix Operations: The FLAMES library optimizes matrix operations by utilizing the unique properties of different matrix types. Additionally, FLAMES offers commonly used functions in linear transformations and signal processing. For example, for a diagonally dominant matrix **A**, its inverse can be computed efficiently with Neumann series approximation (NSA) [15] as

$$\mathbf{A}^{-1} = \lim_{n\to\infty} \sum_{i=0}^{n} (-\mathbf{D}^{-1}\mathbf{E})^{i}\mathbf{D}^{-1}, \tag{1}$$

where $\mathbf{A} = \mathbf{D} + \mathbf{E}$, $\mathbf{D}$ is the diagonal part while $\mathbf{E}$ is the off-diagonal part. In FLAMES, $\mathbf{D}$ and $\mathbf{E}$ can be of types MatViewDiag and MatViewOffDiag, obtained by methods .diagMat_ and .offDiag_, respectively.

III. Case Study and Verification

Compressed sensing [16] is a powerful technique with numerous applications, including millimeter wave (mmWave) channel estimation in the angular domain for multiple-input multiple-output (MIMO) systems [17], [18]. It is modeled as

$$\mathbf{y} = \mathbf{\Phi}\mathbf{x} + \mathbf{n}, \tag{2}$$

where $\mathbf{y} \in \mathbb{R}^{M\times 1}$ is the compressed signal, $\mathbf{\Phi} \in \mathbb{R}^{M\times N}$ ($M < N$) is the sensing matrix, $\mathbf{x} \in \mathbb{R}^{N\times 1}$ is the original L-sparse signal, $\mathbf{n} \sim \mathcal{N}(0, \sigma_n^2)$ is the additive white Gaussian noise (AWGN), and σ_n^2 is the noise variance. The OMP algorithm in [19] can be employed to achieve high efficiency sparse signal recovery. However, the performance of OMP is not satisfactory enough, and an efficient OMPL algorithm in [20] can be applied to enhance the recovery performance. The OMPL algorithm extends the OMP iteration to n lists, which are branched and merged in each iteration. Every set of support indices is a candidate in OMPL.

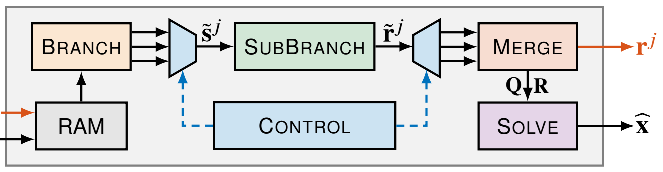
A. Algorithm and Implementation

With the square-root-free QR decomposition [21] method for OMP proposed in [22], the OMPL algorithm can be represented as Algorithm 1.

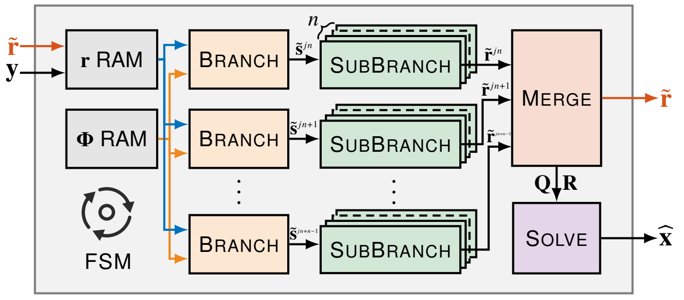
Algorithm 1 OMPL with square-root-free QR decomposition.

```
Input: $\mathbf{\Phi} \in \mathbb{R}^{M\times N}$, $\mathbf{y} \in \mathbb{R}^{M\times 1}$.
Initialization: $\widehat{\mathbf{x}} = \mathbf{O}_{N\times 1}$.
 1: for $i = 0, 1, \cdots, L-1$ do
 2:     for $j = 0, 1, \cdots, n-1$ do                                  ▷ BRANCH.
 3:         $\tilde{\mathbf{s}}_i^{jn}, \tilde{\mathbf{s}}_i^{jn+1}, \cdots, \tilde{\mathbf{s}}_i^{(j+1)n-1} = \arg\max_k^{(n)} |\langle \mathbf{r}^j, \mathbf{\Phi}_{:,k} \rangle|$;
 4:         for $k = 0, 1, \cdots, n-1$ do                              ▷ SUBBRANCH.
 5:             $\mathbf{w} = \mathbf{\Phi}_{:,\tilde{\mathbf{s}}_i^{jn+k}}$;
 6:             for $q = 0, 1, \cdots, i-1$ do
 7:                 $\widetilde{\mathbf{R}}_{q,i}^{jn+k} = (\widetilde{\mathbf{Q}}_{:,q}^{jn+k})^\mathsf{T}\mathbf{w}$;
 8:                 $\mathbf{w} = \mathbf{w} - (\widetilde{\mathbf{R}}_{q,i}^{jn+k}/\widetilde{\mathbf{R}}_{q,q}^{jn+k})\widetilde{\mathbf{Q}}_{:,q}^{jn+k}$;
 9:             $\widetilde{\mathbf{R}}_{i,i}^{jn+k} = \|\mathbf{w}\|_2^2$, $\widetilde{\mathbf{Q}}_{:,i}^{jn+k} = \mathbf{w}$;
10:             $\tilde{\mathbf{r}}^{jn+k} = \tilde{\mathbf{r}}^{jn+k} - (\langle \widetilde{\mathbf{Q}}_{:,i}^{jn+k}, \tilde{\mathbf{r}}^{jn+k} \rangle / \widetilde{\mathbf{R}}_{i,i}^{jn+k})\widetilde{\mathbf{Q}}_{:,i}^{jn+k}$;
11:     $\mathbf{a} = \mathrm{u}\arg\min_k^{(n)} \|\tilde{\mathbf{r}}^k\|_1$;                    ▷ MERGE.
12:     for $j = 0, 1, \cdots, n-1$ do
13:         $\mathbf{r}^j = \tilde{\mathbf{r}}^{\mathbf{a}_j}$, $\mathbf{Q}^j = \widetilde{\mathbf{Q}}^{\mathbf{a}_j}$, $\mathbf{R}^j = \widetilde{\mathbf{R}}^{\mathbf{a}_j}$, $\mathbf{s}^j = \tilde{\mathbf{s}}^{\mathbf{a}_j}$;
14:     for $j = 0, 1, \cdots, n^2-1$ do
15:         $\tilde{\mathbf{r}}^j = \mathbf{r}^{\lfloor j/n \rfloor}$, $\widetilde{\mathbf{Q}}^j = \mathbf{Q}^{\lfloor j/n \rfloor}$, $\widetilde{\mathbf{R}}^j = \mathbf{R}^{\lfloor j/n \rfloor}$, $\tilde{\mathbf{s}}^j = \mathbf{s}^{\lfloor j/n \rfloor}$;
16: $m = \arg\min_k \|\mathbf{r}^k\|_1$; $\mathbf{Q} = \mathbf{Q}^m$, $\mathbf{R} = \mathbf{R}^m$, $\mathbf{s} = \mathbf{s}^m$;
17: for $l = 0, 1, \cdots, L-1$ do                                        ▷ SOLVE.
18:     $\widehat{\mathbf{x}}_{\mathbf{s}_{L-l-1}} = (\langle \mathbf{Q}_{:,L-l+1}, \mathbf{y} \rangle - \mathbf{R}_{L-l-1,:}\widehat{\mathbf{x}}_{\mathbf{s}})/\mathbf{R}_{L-l-1,L-l-1}$;
Output: Estimated $L$-sparse signal $\widehat{\mathbf{x}}$.
```

The hardware architecture of OMPL consists of 4 modules. For a given candidate, the Branch module computes n sub-candidates. The SubBranch module updates the QR decomposition result and residual of a sub-candidate. The Merge module selects the best n candidates among the n^2 sub-candidates according to their residual. The Solve module solves the least square (LS) problem. The hardware architecture can be divided into sequential (Fig. 3(a)) or parallel (Fig. 3(b)) designs. The sequential architecture reuses one Branch module and one SubBranch module, whereas the parallel architecture contains n Branch modules and n^2 SubBranch modules. In each iteration, the parallel architecture process all the sub-candidates in parallel. The timeline traces of these two architectures are shown in Fig. 4.

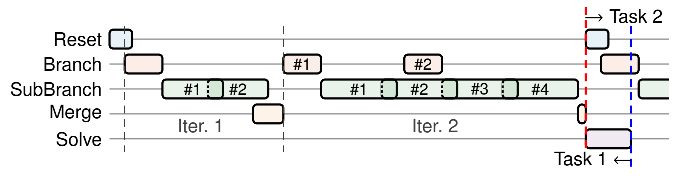


(a) Sequential architecture.

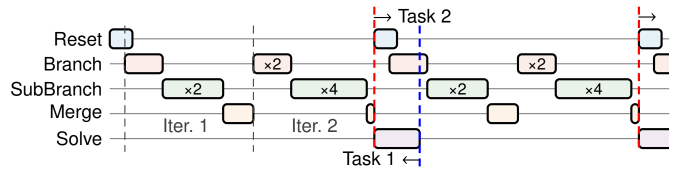


(b) Parallel architecture.

Fig. 3. Hardware architecture of OMPL.



(a) Sequential.



(b) Parallel.

Fig. 4. Timeline of sequential and parallel designs with $L = 2$, $n = 2$.

The main iterations of OMPL compressed sensing are between step 1 and 15, where Branch and Merge operations are repeated.

Branch. For each candidate, Branch selects n new support indices as sub-candidates through step 3 according to the correlation of $\mathbf{\Phi}$ and $\mathbf{r}^j$ as illustrated in Fig. 5. A binary sensing matrix with 1-bit quantization is used [23]–[25] to simplify the correlation calculation, resulting in $\mathbf{\Phi}$ being of type `Mat<bool,M,N>`. The multiplication of a binary matrix and a vector is specially optimized in FLAMES with configurable parallelism p. The optimized process employs a multiplexer array followed by an adder and accumulator array to sum up the results. The accumulator array outputs the $\mathbf{\Phi}^\mathsf{T}\mathbf{r}^j$ vector every $\frac{M}{p}$ clock cycles, which is then processed with an absolute value calculating array before the sorting network locates the n largest elements of the vector. Each of the sub-candidates leads to one SubBranch. Notably, only one Branch operation is required in the first iteration.

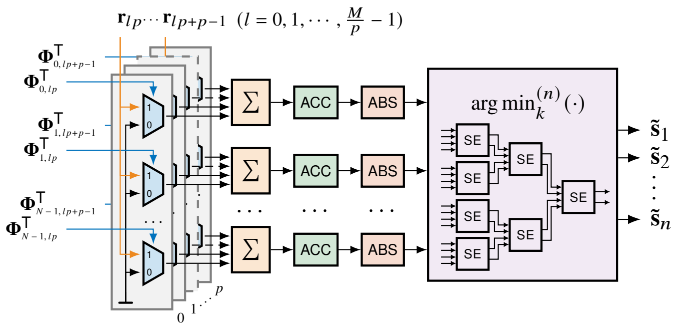
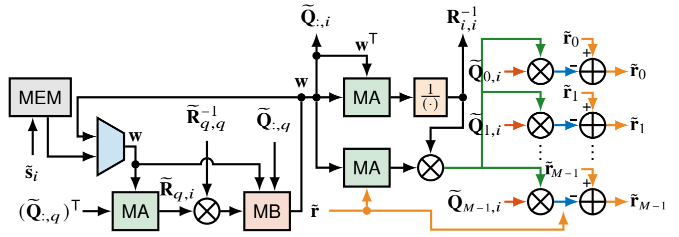
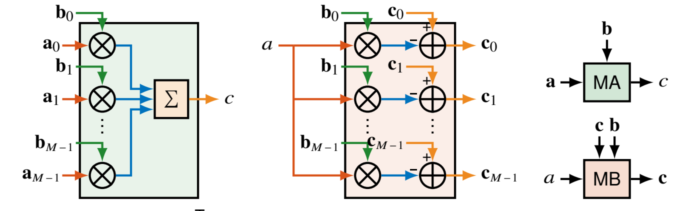

$\mathbf{r}_{lp} \cdots \mathbf{r}_{lp+p-1}$ $(l = 0, 1, \cdots, \frac{M}{p} - 1)$
$\mathbf{\Phi}^{\mathsf{T}}_{0,lp+p-1}$
$\mathbf{\Phi}^{\mathsf{T}}_{0,lp}$
$\mathbf{\Phi}^{\mathsf{T}}_{1,lp+p-1}$
$\mathbf{\Phi}^{\mathsf{T}}_{1,lp}$
$\mathbf{\Phi}^{\mathsf{T}}_{N-1,lp+p-1}$
$\mathbf{\Phi}^{\mathsf{T}}_{N-1,lp}$
ACC
ABS
$\arg\min^{(n)}_{k}(\cdot)$
SE
$\tilde{\mathbf{s}}_1$
$\tilde{\mathbf{s}}_2$
$\tilde{\mathbf{s}}_n$

Fig. 5. Hardware architecture of the BRANCH module.

SUBBRANCH. This module follows the square-root-free QR decomposition in [22], which is shown in step 4 to 10 of Algorithm 1. In SUBBRANCH, $\mathbf{w}$ is calculated iteratively by reusing linear transformation units including vector-vector multiplication and vector-scalar multiplication. The calculation of $\widetilde{\mathbf{R}}_{:,i}$ and $\widetilde{\mathbf{Q}}_{:,i}$ are implemented similarly. To simplify the architecture, $\widetilde{\mathbf{R}}^{-1}_{i,i}$ is stored instead of $\widetilde{\mathbf{R}}_{i,i}$ and the reciprocal operation is implemented with a look-up table. In Fig. 6(e), `dtype` is the data type of $\mathbf{w}$, and the `innerProd` function calculates the inner product of two vectors.


MEM
$\tilde{\mathbf{s}}_i$
$(\widetilde{\mathbf{Q}}_{:,q})^{\mathsf{T}}$
MA
MB
$\widetilde{\mathbf{R}}_{q,i}$
$\widetilde{\mathbf{R}}^{-1}_{q,q}$
$\widetilde{\mathbf{Q}}_{:,q}$
$\widetilde{\mathbf{Q}}_{:,i}$
$\mathbf{w}^{\mathsf{T}}$
$\mathbf{w}$
$\tilde{\mathbf{r}}$
$\mathbf{R}^{-1}_{i,i}$
$\frac{1}{(\cdot)}$
$\widetilde{\mathbf{Q}}_{0,i}$
$\widetilde{\mathbf{Q}}_{1,i}$
$\widetilde{\mathbf{Q}}_{M-1,i}$
$\tilde{\mathbf{r}}_0$
$\tilde{\mathbf{r}}_1$
$\tilde{\mathbf{r}}_{M-1}$

(a) Overall architecture. The superscript $jn+k$ is omitted for simplicity.



(b) MA ($c = \mathbf{ab} = \langle \mathbf{a}^{\mathsf{T}}, \mathbf{b} \rangle$). (c) MB ($\mathbf{c} = \mathbf{c} - a \cdot \mathbf{b}$). (d) Legend.

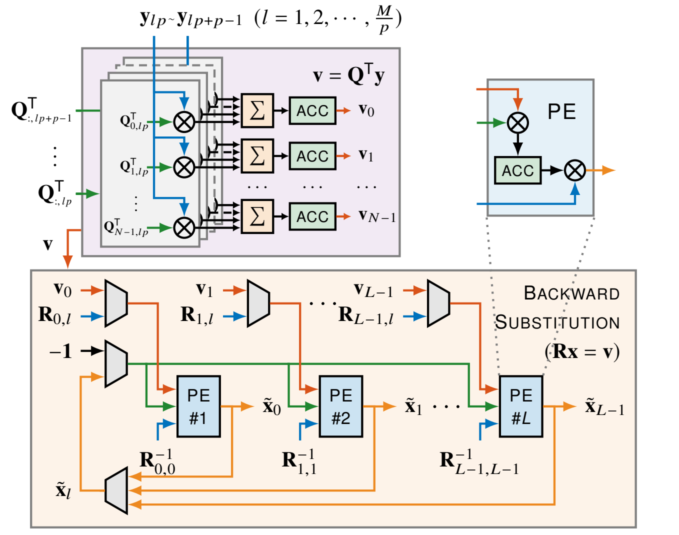
```
auto w = Phi.col<dtype>(cand_s[j*n+k]);
SB_LOOP: for (int p = 0; p + 1 < i; ++p)
  w -= (Ri(p, i) = innerProd(Qi.col_(p), w))
       / Ri(p, p) * Qi.col_(p);
auto wp = Ri(i, i) = w.power(); /* l2-norm square */
Qi.col(i, w);                    /* Or: Qi.col_(i) = w; */
ri -= innerProd(w, ri) / wp * w;
```

(e) Corresponding C++ code.

Fig. 6. Hardware architecture and C++ code of the SUBBRANCH module.

MERGE. This module seeks the n least residual support sets in n^2 candidates, using ℓ_1-norm instead of ℓ_2-norm for hardware simplicity. To avoid repetitions, the n selected indices forming $\mathbf{a}$ in step 11 are unique. The operation is denoted as $\mathrm{u}\arg\min^{(n)}_{k}$. The corresponding matrices are updated until step 15. MERGE is implemented with a sorting tree with multiple layers. Each layer contains sorting nodes that select the n smallest values of its input. The output of the final layer is the desired output. Finally, the selected set is obtained by finding the set with minimum residual of the n lists.

SOLVE. This module implements step 17 to 18 of Algorithm 1. It performs matrix-vector multiplication, backward substitution, and location mapping to solve the LS problem. The matrix-vector multiplication is implemented with an architecture similar to that in the BRANCH module, but with a multiplier array replacing the multiplexer array. Backward substitution is then performed with a PE array, where PE is a 3-input, 1-output processing unit shown in Fig. 7. Finally, $\tilde{\mathbf{x}}_l$ is mapped to its location $\mathbf{s}_l$ via the mapping unit.


$\mathbf{y}_{lp} \cdots \mathbf{y}_{lp+p-1}$ $(l = 1, 2, \cdots, \frac{M}{p})$
$\mathbf{v} = \mathbf{Q}^{\mathsf{T}}\mathbf{y}$
$\mathbf{Q}^{\mathsf{T}}_{:,lp+p-1}$
$\mathbf{Q}^{\mathsf{T}}_{:,lp}$
$\mathbf{Q}^{\mathsf{T}}_{0,lp}$
$\mathbf{Q}^{\mathsf{T}}_{1,lp}$
$\mathbf{Q}^{\mathsf{T}}_{N-1,lp}$
ACC
$\mathbf{v}_0$
$\mathbf{v}_1$
$\mathbf{v}_{N-1}$
$\mathbf{v}$
PE
BACKWARD SUBSTITUTION
$(\mathbf{Rx} = \mathbf{v})$
$\mathbf{v}_0$
$\mathbf{R}_{0,l}$
$\mathbf{v}_1$
$\mathbf{R}_{1,l}$
$\mathbf{v}_{L-1}$
$\mathbf{R}_{L-1,l}$
-1
PE #1
PE #2
PE #L
$\tilde{\mathbf{x}}_0$
$\tilde{\mathbf{x}}_1$
$\tilde{\mathbf{x}}_{L-1}$
$\mathbf{R}^{-1}_{0,0}$
$\mathbf{R}^{-1}_{1,1}$
$\mathbf{R}^{-1}_{L-1,L-1}$
$\tilde{\mathbf{x}}_l$

(a) Hardware architecture (the mapping unit is omitted).

```
auto v = Qf.t_() * y;
Vec<x::value_type, L> xs(init::zeros);
SOLVE_BACK_SUB: for (int i = 0; i < L; ++i)
  x[sf[j = L-i-1]] = xs[i] = (v[j] -
    innerProd(Rf.row_(j), xs) / Rf(j, j));
```

(b) Corresponding C++ code.

Fig. 7. Hardware architecture and C++ code of the SOLVE module.

IV. EXPERIMENTAL RESULTS

A. Performance Verification

Fig. 8 compares the normalized mean square error (NMSE) performance defined as $\mathbb{E}[\|\widehat{\mathbf{x}} - \mathbf{x}\|_2 / \|\mathbf{x}\|_2]$ versus the signal-to-noise ratio (SNR), where the OMPL list size is $n = 2$. The LS

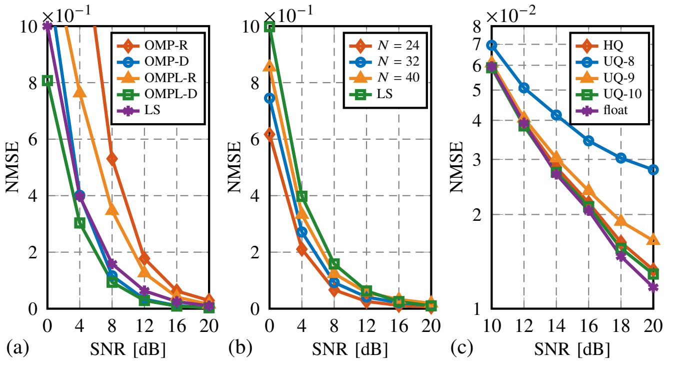


Fig. 8. NMSE performance of: (a) OMP and OMPL, (b) OMPL under different measurement size, (c) OMPL under different quantization schemes.

method is compared as a benchmark with no dimension compression. OMPL outperforms OMP [19] using both random binary ('-R' suffix) and designed binary sensing matrices ('-D' suffix) in Fig. III-A(a), where $M = 64$, $N = 128$, $L = 8$. In Fig. 8(b), N varies among 24, 32 and 40, and $M = 16$, $L = 2$. In Fig. 8(c), we use hybrid quantization (HQ) with an average of around 7 fixed-point bits ($M = 16$, $N = 32$, $L = 2$), which is close to the performance of 10 bits unified quantization (UQ) and float-point precision. In our HQ scheme, $\mathbf{r}$, $\widehat{\mathbf{x}}$, $\mathbf{w}$, $\mathbf{Q}$, $\mathbf{R}$ have 8, 10, 3, 7 and 7 bits, respectively.

B. Hardware Implementation

The OMPL algorithm for compressed sensing is synthesized and implemented for Xilinx Zynq-7000 ZC-706 evaluation board, with Vitis HLS 2022.2. The FPGA resource consumption and performance comparisons are listed in Table II, which shows the effectiveness of hardware design for matrix-based linear transformations compared with the implementation with traditional HLS (T-HLS) [9]. Designs assisted by the FLAMES library achieve 3.96× and 1.67× latency reduction and 1.56× and 1.12× throughput/slice for the sequential and parallel designs, respectively. The parallel design achieves higher throughput and lower latency than the sequential design at the expense of more resource consumption.

TABLE II
FPGA RESOURCE AND PERFORMANCE COMPARISONS FOR 16×32 COMPRESSED SENSING WITH OMPL.

Metric	**Sequential**		**Parallel**	
	T-HLS [9]	FLAMES	T-HLS [9]	FLAMES
Slice	2,566	6,912	6,022	8,181
LUT	4,360	15,704	17,129	9,669
FF	5,958	22,114	10,899	39,170
DSP	81	36	75	5
BRAM	14	1	5	1
Latency [μs]	13.91	3.510	3.113	1.864
Frequency [MHz]	134	104	128	107
Throughput [Mb/s]	23.63	94.81	120.9	184.5
Throughput/slices [Mbps/K slices]	8.819	13.72	20.08	22.55

Remarks: The sequential and parallel designs only differ in the HLS pragma/directive, so the OMPL algorithm can be readily implemented as different architectures. The FLAMES library significantly simplifies the hardware design with the matrix-based syntax shown in Table I. Complicated operations like branching and merging in OMPL are implemented easily with matrix-based coding, significantly reducing the complexity and difficulty of hardware design. Notably, the FLAMES library excels at achieving high throughput in contrast to HLS's conservative optimization strategy.

V. Conclusion

In this paper, we propose FLAMES for matrix-based linear transformations, a high-level synthesis library implemented on Vitis HLS. FLAMES achieves hardware and coding efficiency for fast algorithm implementations. Moreover, we demonstrate the effectiveness of FLAMES through the OMPL algorithm, which verifies the library's capability to synthesize efficient hardware designs.

References

[1] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y.-L. Lin, *High—Level Synthesis: Introduction to Chip and System Design*, D. D. Gajski, Ed. New York, NY: Springer, 1992.
[2] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 8–17, 2009.
[3] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.
[4] Z. Zhong, W. J. Gross, Z. Zhang, X. You, and C. Zhang, "Polar compiler: Auto-generator of hardware architectures for polar encoders," *IEEE Trans. Circuits Syst. I*, vol. 67, no. 6, pp. 2091–2102, Jun. 2020.
[5] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, Oct. 2020.
[6] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for FPGAs," 2018, *arXiv:1805.03648*. [Online]. Available: https://arxiv.org/abs/1805.03648
[7] J. Matai, D. Lee, A. Althoff, and R. Kastner, "Composable, parameterizable templates for high-level synthesis," in *Proc. IEEE Design Autom. Test Europe Conf. Exhibit. (DATE)*, Mar. 2016, pp. 744–749.
[8] T. De Matteis, J. de Fine Licht, and T. Hoefler, "FBLAS: Streaming linear algebra on FPGA," in *Proc. IEEE Int. Conf. HPC Netw. Stor. Anal.*, Nov. 2020, pp. 1–13.
[9] Xilinx, "Vitis high-level synthesis user guide (UG1399)," 2023, accessed: Feb. 27, 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls
[10] C. Sanderson and R. Curtin, "Armadillo: a template-based C++ library for linear algebra," *J. Open Source Softw.*, vol. 1, no. 2, p. 26, 2016.
[11] *ISO/IEC 14882:2014(E) – Programming Languages – C++*, ISO Std., 2014, sec. 7.1.6.4: auto specifier.
[12] M. Ayinala, M. Brown, and K. K. Parhi, "Pipelined parallel FFT architectures via folding transformation," *IEEE Trans. VLSI Syst.*, vol. 20, no. 6, pp. 1068–1081, Jun. 2012.
[13] B. Yuan and K. K. Parhi, "Architecture optimizations for BP polar decoders," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, May 2013, pp. 2654–2658.
[14] H. Wang, Y. Ji, Y. Shen, W. Song, M. Li, X. You *et al.*, "An efficient detector for massive MIMO based on improved matrix partition," *IEEE Trans. Signal Process.*, vol. 69, pp. 2971–2986, 2021.
[15] X. Tan, W. Xu, Y. Zhang, Z. Zhang, X. You, and C. Zhang, "Efficient expectation propagation massive MIMO detector with Neumann-series approximation," *IEEE Trans. Circuits Syst. II*, vol. 67, no. 10, pp. 1924–1928, Oct. 2020.
[16] D. L. Donoho, "Compressed sensing," *IEEE Trans. Inf. Theory*, vol. 52, no. 4, pp. 1289–1306, Apr. 2006.
[17] J. Lee, G.-T. Gil, and Y. H. Lee, "Channel estimation via orthogonal matching pursuit for hybrid MIMO systems in millimeter wave communications," *IEEE Trans. Commun.*, vol. 64, no. 6, pp. 2370–2386, Jun. 2016.
[18] Y. You and L. Zhang, "Bayesian matching pursuit-based channel estimation for millimeter wave communication," *IEEE Commun. Lett.*, vol. 24, no. 2, pp. 344–348, Feb. 2020.
[19] 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.
[20] W. Zhao, Y. You, L. Zhang, X. You, and C. Zhang, "OMPL-SBL algorithm for intelligent reflecting surface-aided mmWave channel estimation," *IEEE Trans. Veh. Technol.*, 2023, to be published.
[21] S.-F. Hsieh, K. R. Liu, and K. Yao, "A unified square-root-free approach for QRD-based recursive-least-squares estimation," *IEEE Trans. Signal Process.*, vol. 41, no. 3, pp. 1405–1409, Mar. 1993.
[22] X. Ge, F. Yang, H. Zhu, X. Zeng, and D. Zhou, "An efficient FPGA implementation of orthogonal matching pursuit with square-root-free QR decomposition," *IEEE Trans. VLSI Syst.*, vol. 27, no. 3, pp. 611–623, Mar. 2019.
[23] S.-T. Xia, X.-J. Liu, Y. Jiang, and H.-T. Zheng, "Deterministic constructions of binary measurement matrices from finite geometry," *IEEE Trans. Signal Process.*, vol. 63, no. 4, pp. 1017–1029, Feb. 2015.
[24] W. Lu, T. Dai, and S.-T. Xia, "Binary matrices for compressed sensing," *IEEE Trans. Signal Process.*, vol. 66, no. 1, pp. 77–85, Jan. 2018.
[25] M. Fardad, S. M. Sayedi, and E. Yazdian, "A low-complexity hardware for deterministic compressive sensing reconstruction," *IEEE Trans. Circuits Syst. I*, vol. 65, no. 10, pp. 3349–3361, Oct. 2018.