FLUD: A Scalable and Configurable Systolic Array Design for LU Decomposition on FPGAs

Xingyu Tian
Simon Fraser University
xingyut@sfu.ca

Geng Yang
Xidian University
gengyang@stu.xidian.edu.cn

Zhenman Fang
Simon Fraser University
zhenman@sfu.ca

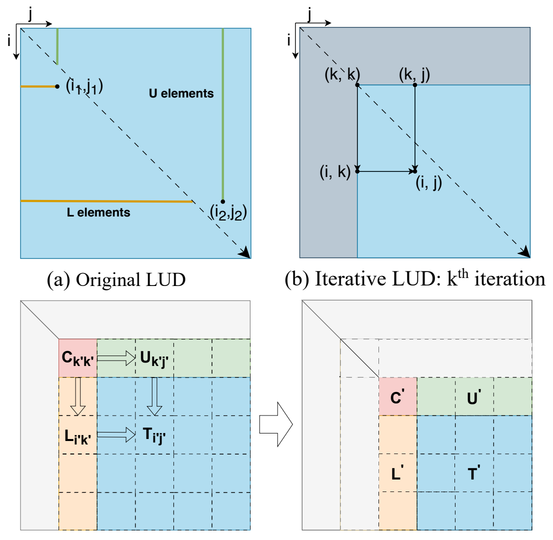
***Abstract*—Lower-upper decomposition (LUD) is one of the most popular matrix factorization techniques in linear algebra and has been widely used in many scientific and engineering applications. While prior studies have investigated various strategies to accelerate block LUD on FPGAs for arbitrary input sizes, they often suffer from one or more of the following limitations: 1) excessive resource utilization due to separate PE (processing element) designs for different matrix blocks with diverse computation patterns; 2) excessive on-chip memory usage due to buffer-based designs; and 3) insufficient parallelism as only one-level parallelism (either row-level or iteration-level parallelism) was exploited due to complex dependencies.**

To address those limitations, we propose FLUD, a streaming-based systolic array design on the FPGA to accelerate block LUD, which shares the systolic array to accelerate different matrix blocks and exploits both column-level parallelism and iteration-level parallelism. First, FLUD implements a configurable systolic array that is shared by different LUD blocks and scalable to arbitrary input sizes. To further optimize its hardware resource efficiency, FLUD groups a column of PEs together to replace their FIFO connections with lightweight registers and reduce multiple copies of local control logic inside each PE (for the purpose of resource sharing among different LUD blocks) into a global one. Moreover, FLUD devises a computation schedule to effectively share the highly-optimized systolic array design among the execution of different LUD blocks. Lastly, to enable fast design space exploration on a given FPGA platform, we develop an automation tool to automatically generate the optimized FLUD design in Vitis high-level synthesis (HLS), where users can configure the design size and data precision based on their needs. Experimental results demonstrate that FLUD achieves a peak throughput of 427.95 GFLOPS for single-precision floating-point LUD, which is about 3x faster than state-of-the-art FPGA design. Compared to the LAPACK library running on a 12-core Xeon Silver 4214 CPU, FLUD achieves 4.71x higher throughput and 10.25x better throughput/watt.

I. Introduction

LUD algorithm is one of the fundamental matrix factorization techniques in linear algebra, which is widely used in many scientific and engineering computations [1], [2], [3], [4], [5], [6] . It factors a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U ($A = LU$): elements on the diagonal line are ones in L and zeros in U. Basically, LUD is a matrix form of Gaussian elimination to solve various dense linear systems of equations, which has complex data dependencies [7]. As shown in Figure 1a, the calculation of an element $L(i_1, j_1)$ is dependent on a row of elements to its left and a column of elements to its top (above the diagonal line in U). Similarly, the calculation of an element $U(i_2, j_2)$ is dependent on a row of elements to its left (left of the diagonal line in L) and a column of elements to its top.

In the context of large matrices, block LUD [8] was proposed to enhance the data locality: It iteratively performs



(a) Original LUD (b) Iterative LUD: kth iteration

(c) Block LUD: kth to (k+1)th iteration

Fig. 1: Data dependency illustration of LUD

panel factorization and trailing submatrix updates, where panel factorization results are used to update the trailing submatrix. As shown in Figure 1c and 2, block LUD introduces complex data dependencies both inter-block and intra-block. A detailed analysis of these dependencies is provided in Section II-B.

Recent studies [9], [10], [11], [12], [13], [14] have made great progress in accelerating block LUD on FPGAs. However, they still have one or more of the following limitations:

1. **Excessive Computing Resource Utilization:** In block LUD, as detailed in Section II-B, different types of blocks (i.e., corner, lower perimeter, upper perimeter, and trailing blocks) have diverse computation patterns, which require specific optimizations. A straightforward solution is to utilize different sets of processing elements (PEs) to process various types of blocks, as demonstrated in [10], [14]. However, this approach leads to high resources consumption and thus limits the scalability of the design.
2. **Excessive On-chip Memory Consumption:** Many of the existing LUD accelerator designs, such as [10], [14], [11], manage the inter-block dependency by buffering prior dependent blocks on-chip. Additionally, to mitigate memory access latency, ping-pong buffering is frequently used. Such excessive usage of on-chip memory often constrains the scalability of these designs.
3. **Insufficient Parallelism:** Most previous studies only explore one-level parallelism due to the complex intra-block dependencies. Buffer-based designs [10], [14], [11] often explore row-level or column-level parallelism within the

corner, lower perimeter, and upper perimeter blocks. On the other hand, the streaming-based design in [9] explores iteration-level parallelism, where multiple PEs process different iterations of a single block concurrently in a dataflow fashion. The scalability of these designs is limited, as the block size must exceed the number of PEs, and consequently, more on-chip memory is required.

To tackle the above challenges, we propose FLUD, a streaming-based systolic array accelerator for block LUD on FPGAs, which shares the same set of PEs among different LUD blocks and exploits both column-level and iteration-level parallelism. FLUD presents the following novel features:

1. **Scalable Systolic Array Design with Resource Sharing and Two-level Parallelism:** FLUD designs a highly-optimized streaming-based systolic array architecture that can scale to arbitrary input matrix sizes. It exploits parallelism across both the column level and iteration level, and enables resource sharing across various computation patterns of LUD blocks. This approach significantly reduces the resource consumption and improves its scalability.
2. **PE Grouping Optimization for Systolic Array:** FLUD groups PEs within the same column together to further improve resource efficiency. With such PE grouping, FIFO connections between one column of PEs are replaced with lightweight registers, and local control logic in each PE (for resource sharing) is replaced with a global one.
3. **Optimized Computation Schedule:** FLUD devises an optimized computation schedule to manage the complex data dependencies to realize two-level parallelism for various LUD blocks and their resource sharing. All blocks are transferred in a unique streaming manner column-by-column. PEs within the same column process column-level parallelism, while various columns of PEs explore iteration-level parallelism. This approach guarantees memory access efficiency and provides notable throughput improvement.
4. **Design Automation Support:** Given any FPGA platform, FLUD can automatically explore the design space and generate the optimized accelerator in Vitis HLS, which is guided by our throughput modeling in Section III-D. It also allows users to configure the number of PEs and data precision (e.g., half, single, and double-precision).

We evaluated the on-board performance of FLUD on the AMD-Xilinx Alveo U280 FPGA, running at 255 MHz for single-precision floating-point design. FLUD can reach a throughput of 427.95 GFLOPS and an energy efficiency of 10.97 GFLOPS/W, which is about 3x faster over state-of-the-art FPGA design [14]. Compared to LAPACK library [15] running on a 12-core Xeon Silver 4214 CPU, FLUD can reach 4.71x speedup and 10.25x improvement in energy efficiency.

II. Background and Related Work

A. LU Decomposition (LUD) Algorithm

The computation of LUD can be described as $A = LU$. As shown in Figure 1a, in the original LUD algorithm, the calculation of an element $L(i_1, j_1)$ is dependent on a row of elements to its left and a column of elements to its top (above the diagonal line in U). Similarly, the calculation of an element $U(i_2, j_2)$ is dependent on a row of elements to its left (left of the diagonal line in L) and a column of elements to its top. The computation needs to start from the top-left to the bottom-right, and the complex data dependencies hinder the effective parallelization of LUD.

Alg. 1 Iterative non-pivoting LU decomposition

```
1: for ($k = 0$; $k < N$; $k++$) do ▷ Iterative computation on $N \times N$ matrix
2:     for ($i = k + 1$; $i < N$; $i++$) do            ▷ Dependent on $A[k][k]$
3:         $A[i][k] \leftarrow A[i][k] \ / \ A[k][k]$
4:     for ($i = k + 1$; $i < N$; $i++$) do            ▷ Dependent on $A[i][k]$
5:         for ($j = k + 1$; $j < N$; $j++$) do                ▷ and $A[k][j]$
6:             $A[i][j] \leftarrow A[i][j] - A[i][k] * A[k][j]$
```

To expose more parallelism, Dongarra et al. [7] first introduced an iterative non-pivoting LUD algorithm as shown in Figure 1b and Algorithm 1. In the k-th iteration, it first calculates the elements on column k, which only depends on $A[k][k]$ (lines 2-3 in Algorithm 1). Then, it computes the trailing matrix, which only depends on row k (updated in $(k-1)$-th iteration) and column k (lines 4-6); that is, the computation of $A[i][j]$ is only dependent on $A[i][k]$ and $A[k][j]$. This simplifies the data dependency in each iteration: the computation for all elements in the same column (and the same row in the trailing matrix) can be processed in parallel.

Both non-iterative and iterative LUD algorithms encounter data locality challenges when processing large matrix sizes.

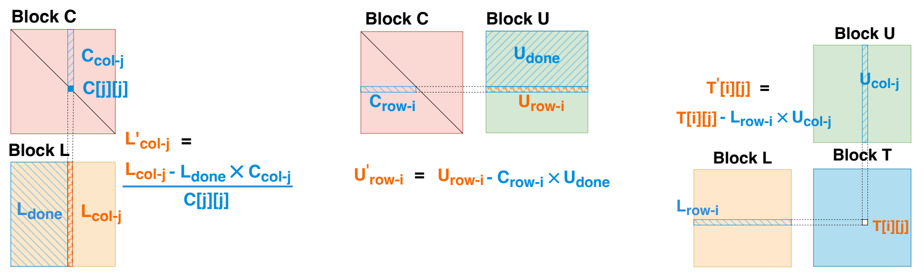
B. Block LUD: Diverse Computations and Dependencies

For large matrices, block LUD was proposed to improve the data locality [8], [17], [18]. As shown in Figure 1c, at each iteration of block LUD, the current (trailing) matrix is first split into four types of blocks, which are the corner block (C), upper perimeter blocks (U), lower perimeter blocks (L), and trailing blocks (T). In each iteration, C block is first processed, and then its results are used to compute U and L blocks. After that, it utilizes blocks U and L to update T blocks. Block LUD algorithm iteratively repeats those operations on the trailing-matrix until it reaches the bottom-right corner.

1. The computation of C block is the same as the iterative LUD shown in Figure 1b and Algorithm 1.
2. In L and U blocks, the computation is executed in a column-by-column and row-by-row manner, respectively.
3. T blocks are updated via matrix multiplication operations.

While block LUD improves the data locality, it introduces additional data dependencies. Figure 1c demonstrates the inter-block dependency, both blocks $U_{k'j'}$ and $L_{i'k'}$ are dependent on block $C_{k'k'}$. Block $T_{i'j'}$ is dependent on $L_{i'k'}$ and $U_{k'j'}$. Figure 2 illustrates more details of inter-block and intra-block dependencies for L, U, and T type blocks; the intra-block dependencies for C type block are shown in Figure 1b.

The lower perimeter block L is processed in a column-by-column manner, which exposes column-level parallelism. For j^{th} column $L[:][j]$, it depends on the results of previous columns $L[:][0 : j-1]$ and the same column of the corner


Block C
$C_{col\text{-}j}$
$C[j][j]$
Block L
L_{done}
$L_{col\text{-}j}$
$L'_{col\text{-}j} = \frac{L_{col\text{-}j} - L_{done} \times C_{col\text{-}j}}{C[j][j]}$
Block C
Block U
U_{done}
$C_{row\text{-}i}$
$U_{row\text{-}i}$
$U'_{row\text{-}i} = U_{row\text{-}i} - C_{row\text{-}i} \times U_{done}$
Block U
$U_{col\text{-}j}$
$T'[i][j] = T[i][j] - L_{row\text{-}i} \times U_{col\text{-}j}$
Block L
Block T
$L_{row\text{-}i}$
$T[i][j]$

(a) Lower perimeter block L: column-level parallelism

(b) Upper perimeter block U: row-level parallelism

(c) Trailing block T: column- and row-level parallelism

Fig. 2: Data dependency in L (lower perimeter), U (upper perimeter), and T (trailing) types of LUD blocks

TABLE I: Comparison of FLUD with recent LUD accelerators on FPGA, assuming a block size of $B \times B$.

Design	Year	Arbitrary Input Size	PE Sharing	On-chip Memory System			Parallelism Level	
				Buffer-based	Streaming-based	Buffer Size	Row/column level	Iteration level
[10]	2012	✓		✓		$>16B^2$	✓	
[14]	2023	✓		✓		N/A	✓	
[11]	2023	✓	✓	✓		$5B^2$	✓	
[9]	2012	✓	✓		✓	2#PE$\times B$		✓
[16]	2021				✓	N/A	✓	✓
FLUD	2024	✓	✓		✓	#PEG$\times B$	✓	✓

block $C[0:j][j]$ above the diagonal line. The computation of each column can be demonstrated as:

$$L[:][j] = \frac{L[:][j] - L[:][0:j-1] \times C[0:j-1][j]}{C[j][j]} \quad (1)$$

The upper perimeter block U is processed row by row, which exposes row-level parallelism. Each row $U[i][:]$ depends on all previous rows $U[0:i-1][:]$ and one row in the corner block $C[i][0:i-1]$ to the left of the diagonal line:

$$U[i][:] = U[i][:] - C[i][0:i-1] \times U[0:i-1][:] \quad (2)$$

The trailing block T updates its data using corresponding lower perimeter block L and upper perimeter block U, which is basically matrix multiplication and exposes both column-level and row-level parallelism:

$$T[:][:] = T[:][:] - L[:][:] \times U[:][:] \quad (3)$$

As analyzed above, these blocks in block LUD have diverse computation patterns and complex data dependencies, which creates challenges to design efficient hardware accelerators.

C. Prior LUD Accelerators on FPGA

Multiple prior studies have explored different acceleration strategies of block LUD on FPGAs, yet each approach comes with its own limitations as summarized in Table I. Generally, the following considerations influence the performance of an LUD accelerator design on the FPGA.

Separate PE v.s. Shared PE Design: To handle the diverse computation patterns, designs like [10], [14] implement separate PEs for various types of blocks. This solution is straightforward but consumes duplicate resources. [14] utilizes specialized PEs for different types of blocks. The corner block is first processed by LUD PEs and forwarded to upper PEs and lower PEs to compute both perimeter blocks. Then the results will be used by trailing PEs to update the trailing blocks. PEs of the same type simultaneously process one row or column of the block in each iteration; for the trailing blocks, both row-level and column-level parallelism is exploited. This design also supports scaling across multiple FPGAs through external communication channels. Another design [10] includes a matrix inverse unit to compute the inverse of the corner block. By utilizing both the inverse and original corner blocks, the computation of perimeter blocks and trailing blocks can be simplified to a uniform operation, matrix multiplication.

The other approach is implementing shared PEs to process different blocks. For example, in [11], [9], each PE equips a multiply-accumulate (MAC) unit to compute for all blocks, while a single divider is shared across all PEs. The shared PE design is more resource efficient and can accommodate more PEs on the same FPGA to achieve higher performance.

Buffer-based v.s. Streaming-based Design: To handle the complex data dependencies between LUD blocks, designs like [10], [14], [11] incorporate a buffer-based on-chip memory system to buffer all dependent blocks with high on-chip memory consumption. For example, [10] buffers multiple upper perimeters and lower perimeter blocks for the computation of trailing matrix for better data reuse. Moreover, it applies ping-pong buffering to hide the off-chip memory latency of each single block. [11] buffers block C to compute perimeter blocks, and buffers block U and block L to compute block T in a ping-pong buffering scheme. Due to the limitation of one-level parallelism, block size must exceed the number of PEs processing one row or column in parallel, which consumes a substantial amount of on-chip memory and impacts scalability.

In streaming-based designs, each PE operates in a data-driven pattern. In [9], the output of each PE becomes the input for the next PE. To facilitate iteration-level parallelism, described below, each PE buffers at most one row and one column of dependent block. [16] proposes a streaming-based systolic array that supports only impractically small matrices and does not adequately handle dependencies between PEs.

Parallelism Dimension: Row-level, Column-level, and Iteration-level: Shared PE designs only explore one-level

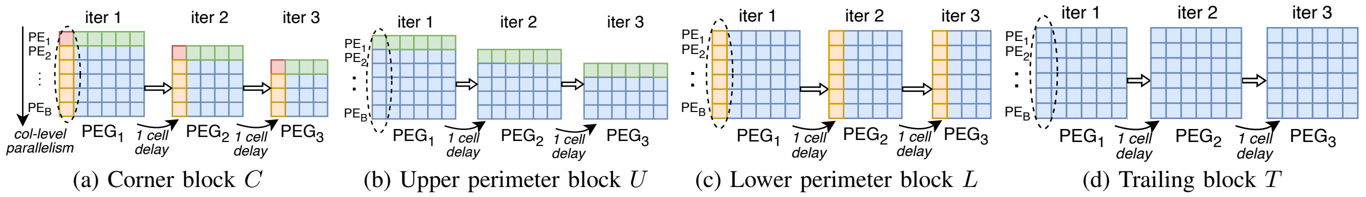


Fig. 3: Combination of column-level and iteration-level parallelism for different LUD blocks

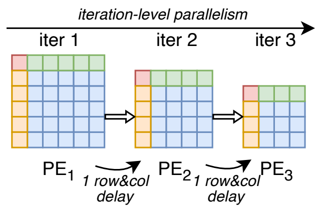


Fig. 4: Iteration-level parallelism for block C. Delay between PEs is the time to process one row and one column.

parallelism, either row/column-level parallelism or iteration-level parallelism. For example, in [11], all PEs process either one row of block U or one column of block L in parallel. Note [11] only provided HLS synthesis results and did not run on the actual FPGA board. [9] organizes each PE to process one iteration of a block. For example in block C shown in Figure 4, all PEs compute different iterations in a dataflow manner and the trailing matrix for each PE keeps shrinking. Note the current PE can only start when its prior PE finishes computing one row and one column of data that the current PE depends on, which creates a long inter-PE delay.

In existing separate PE designs like [10], [11], they only explore row-level or column-level parallelism for the corner, lower perimeter, and upper perimeter blocks. While for the trailing block, which is basically matrix multiplication, both row-level and column-level parallelism are explored. However, none of them explored iteration-level parallelism.

Summary and Our Novelty: As summarized in Table I, all prior FPGA studies for LUD acceleration suffer from one or more of the limitations discussed above. To address these limitations, we propose FLUD, a streaming-based systolic array design that shares PEs to accelerate different LUD blocks and exploits both column-level and iteration-level parallelism.

III. FLUD Design and Implementation

First, we analyze how to combine column-level and iteration-level parallelism for different LUD blocks. Second, we present our streaming-based systolic array design that exploits both levels of parallelism and is shared by different LUD blocks. In addition, we further optimize our systolic array design with column-level PE grouping to improve resource efficiency. After that, we devise the computation schedule for resource sharing among different blocks. Finally, we develop an automation tool, together with an analytical model, to automatically generate the optimized design for a given FPGA platform and user configurations.

A. Combination of Column- and Iteration-level Parallelism

Figure 3 presents how FLUD combines column-level and iteration-level parallelism. Each cell in the figure presents one element of a block in the current iteration. Each block is transferred column by column. Different iterations of the same block are distributed among various PE groups (PEGs), thus implementing iteration-level parallelism. Inside each PEG, various PEs exploit the column-level parallelism by processing elements within the same column in parallel.

1. For the C and L blocks, each PEG buffers the results of the first column and updates the remaining columns, then forwards them to the next PEG to process the next iteration.
2. For the U blocks, data dependency exists across rows and each iteration performs computation of the trailing matrix dependent on one row. To realize column-level parallelism, each PEG uses the first element of a column to update the remaining elements in the same column.
3. For the T blocks, each PEG receives one T block along with one row of dependent U blocks. Each iteration it computes the intermediate results of matrix multiplication with the buffered column of the L blocks.

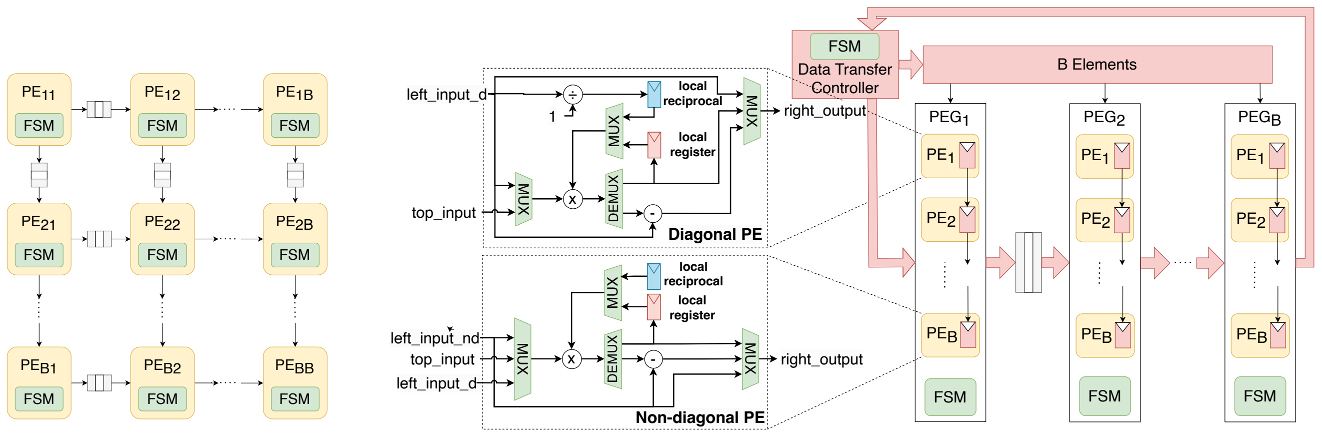
In summary, all blocks can be accelerated using a combination of column-level and iteration-level parallelism. Note since each column is now processed in parallel, the inter-PEG delay is only the time to process one cell. The underlying architecture can be implemented by a streaming-based 2D systolic array, which includes a grid of PEs.

B. FLUD Shared Systolic Array Design

1) Baseline 2D Systolic Array (Non-Grouped): Based on the above analysis, we adopt a 2D systolic array architecture to support the block LUD algorithm: where a column of PEs support column-level parallelism and a row of PEs support iteration-level parallelism. As shown in Figure 5a, PEs are organized as a grid-like structure, and adjacent PEs are connected through FIFOs. Each PE contains one MAC unit (multiply-accumulate, adder can be used to implement subtraction) and PEs on the diagonal line have one additional divider; please refer the basic operations to Algorithm 1. All PEs are data-driven and capable of operating independently, each scheduled by their own finite state machine (FSM) to work for different LUD blocks.

While this design can handle diverse computation patterns with shared computing resources, the FIFO channels between PEs and local FSMs per PE consumes substantial amount of LUTs, which limits the total number of PEs that can be placed on the FPGA and thus constraints the parallelism.

2) Grouped Systolic Array: To tackle the limitation of the 2D systolic array, FLUD groups PEs on the same column into a PEG. As shown in Figure 5b, communication between PEs inside a PEG is implemented by economical registers



(a) Baseline 2D systolic array (b) Grouped systolic array

Fig. 5: Overview of FLUD architecture: comparison between baseline 2D systolic array and grouped systolic array design

instead. We further optimize the computation schedule so that PEs within the same PEG share one FSM. In this way, we reduce the resource cost of interconnects between PEs and simplify the complexity of control logic. Alternative grouping strategies, such as organizing PEs into a rectangle PEG, lead to a more complicated resource sharing schedule.

When computing C and U blocks, only PEs on and below the diagonal line are involved in the calculation. Rest PEs merely forward the input to the output as shown in Figure 6a and 6b, where input data is only fed from the left input ports. As shown in Figure 6c and 6d, when it comes to L and T blocks, all the PEs execute the same operations. Data is fed from both the left and top input ports. Detailed computation schedule is covered in Section III-C.

Overall, FLUD consists of B PEGs and each PEG contains B PEs, where B^2 is the size of a square LUD block. PEGs process the iteration-level parallelism, while PEs within each PEG perform column-level parallelism. As shown in Figure 5b, each PEG has three FIFO ports, one left input port, one top input port, and one right output port. Both left and right ports are coalesced with B elements and the top input port is one element wide. For each block, the data transfer controller streams in the input data from off-chip memory and forwards data to the left input port of the first PEG. The corner and upper perimeter blocks will be forwarded through the top input port when computing the left perimeter and trailing blocks, respectively. The output of the last PEG will be streamed out to off-chip memory.

C. Computation Schedule for Resource Sharing

In this paper, we assume the memory layout of the input matrix is column-major: we transfer each block column by column and process multiple blocks row by row. FLUD can also accommodate to a row-major matrix by transferring each block row by row and processing blocks column by column.

In each round of block LUD, we first compute the corner state and upper perimeter state for the first row of blocks, then process the rest rows of blocks by repeating the lower perimeter state and trailing state sequentially.

To optimize data locality, we process blocks row by row and buffers the results of the first block of each row, where each PEG buffers one column of the first block. In other words, the results of blocks C and L will be buffered for the computation of the rest blocks in the same row.

Assuming the original input matrix size is $N \times N$ and the block size is $B \times B$, the matrix for the current round can be divided into $b \times b$ blocks, with b initially set as $b = N/B$. Block C and T are both square where they consist of 1 and $(b-1) \times (b-1)$ blocks, respectively. The upper perimeter consists of $1 \times (b-1)$ blocks and the lower perimeter consists of $(b-1) \times 1$ blocks.

Figure 6 shows the computation of j^{th} columns within one row of blocks in PEGs at various locations where $0 < j < bB$. We denote the PEG number with p where $0 < p \leq B$.

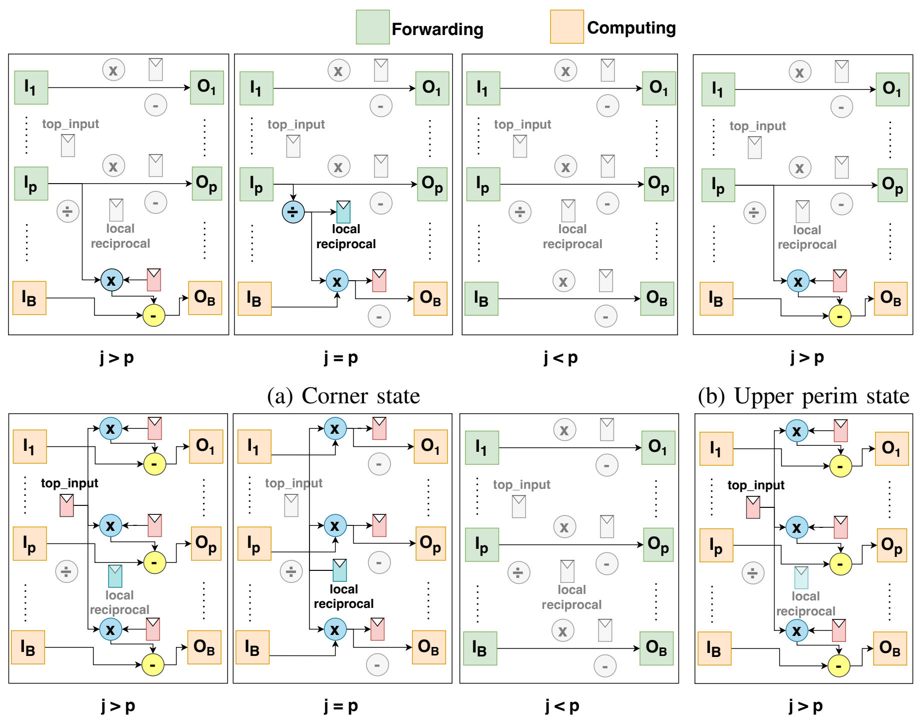
1) Corner State: In the corner state, when processing j^{th} column in p^{th} PEG, only elements below p^{th} row will be processed. As shown in Figure 3a and Figure 6a, j^{th} column first flows into PEGs where $j > p$, and each PEG updates part of j^{th} column by $C[p{+}1 : B][j] = C[p{+}1 : B][j] - C[p{+}1 : B][p] * C[p][j]$. When $j = p$, the p^{th} PEG applies division to elements below the diagonal line by multiplying the reciprocal, where $C[p{+}1 : B][j] = C[p{+}1 : B][j]/C[p][p]$. Note only p^{th} PE contains a divider unit. Both results and reciprocal value will be buffered in this PEG for the computation of subsequent columns and lower perimeter, respectively. When $j < p$, those PEGs simply forward input to output.

2) Upper Perimeter State: In the upper perimeter state, when a column flows into p^{th} PEG, elements below p^{th} row are updated using buffered value as shown in Figure 3b and Figure 6b. The j^{th} column, $U[:][j]$, is processed in p^{th} PEG:

$$U[p+1:B][j] = U[p+1:B][j] - U[p][j] * C[p+1:B][p] \quad (4)$$

3) Lower Perimeter State: The operations of the lower perimeter state are similar to that of the corner state but all elements are updated when $j \geq p$. Process on j^{th} column of a $B \times B$ lower perimeter block, $L[:][j]$, can be presented as:

$$L[:][j] = \frac{L[:][j] - \sum_{p=1}^{j-1} L[:][p] * C[p][j]}{C[j][j]} \quad (5)$$



(a) Corner state (b) Upper perim state

(c) Lower perimeter state (d) Trailing state

Fig. 6: Computation schedule of j^{th} column in p^{th} PEG. Units colored with grey are idling during the process of j^{th} column. In corner and upper perimeter states, only PEs below the diagonal line are active. In lower perimeter and trailing states, all PEs are operational when $j > p$.

When $j > p$, each PEG will process one step of $L[:][j] - \sum_{p=1}^{j-1} L[:][p] * C[p][j]$ as shown in Figure 3c and Figure 6c. Each $C[p][j]$ is drained from `top_input` port. When $j = p$, the p^{th} PEG processes division with the buffered reciprocal value, $C[j][j]$. The results will be stored in registers for the computation of trailing blocks on the same row. When $j < p$, the rest PEs simply forward j^{th} column.

4) Trailing State: After the lower perimeter state, blocks of trailing matrix on the same row will be arranged. In this state, $(b-1)$ blocks in the same row are fed to the systolic array continuously along with corresponding values of the upper perimeter block, which is drained from `top_input`:

$$T[:][j] = T[:][j] - \sum_{p=1}^{B} L[:][p] * U[:][j] \tag{6}$$

Each PEG processes one step of the above matrix multiplication operations as shown in Figure 3d and Figure 6d.

In the proposed data schedule, the initial delay between PEGs is not decided by block size but only by the latency of one MAC operation and division. Additionally, the matrix size is dynamically configurable at run-time by passing it as a parameter to the kernel, allowing for flexible adaptation.

D. Throughput Modeling and Analysis

Assuming the original input matrix size is $N \times N$, the block size and the number of PEs are both $B \times B$, and the current number of blocks is $b \times b$ where $1 \leq b \leq \frac{N}{B}$. We first present the latency of each row of blocks separately:

The first row of blocks consists of the corner state and the upper perimeter state. The p^{th} PEG first applies division to p^{th} column, then computes MAC operations on the rest columns. Let L_{DIV} be the latency of division, L_{MUL} be the latency of multiplication, and L_{MAC} be the latency of MAC operation. The first PEG first computes the division by reciprocal value on the first column, and applies MAC to the second column, then forwards the results to subsequent PEG. Delay between PEGs can be presented as $(L_{DIV} + L_{MUL} + L_{MAC})$. The last PEG starts its own MAC operation pipeline after $(L_{DIV} + L_{MUL} + L_{MAC})B$ cycles. Rest $B(b-1)$ columns within the same row of blocks are computed continuously. The total latency to finish the last PEG can be presented as:

$$L_{row1} = (L_{DIV} + L_{MUL} + L_{MAC})B + B(b-1) \tag{7}$$

For the rest of the rows, each consists of one lower perimeter block and $b-1$ trailing blocks. As each PEG already buffered the reciprocal value, the latency of each row can be similarly presented as:

$$L_{restrow} = (L_{MUL} + L_{MAC})B + B(b-1) \tag{8}$$

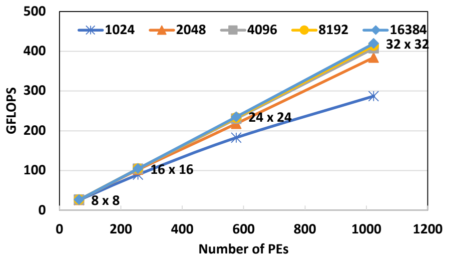
The latency of each iteration can be presented as:

$$L_{iter} = L_{row1} + (b-1)L_{restrow} \tag{9}$$

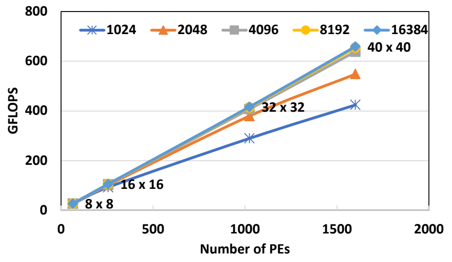
And the overall latency is the sum of all iterations:

$$L = \sum_{iter=1}^{N/B} L_{iter} \tag{10}$$

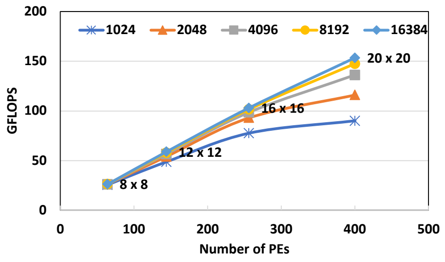
The total number of LUD operations needs $\frac{N^3}{3}$ multiplication and $\frac{N^3}{3}$ subtraction, which are approximately $\frac{2N^3}{3}$ operations in total. Assuming the frequency is f, the throughput of the proposed design can be expressed as:



(a) FLUD performance under float



(b) FLUD performance under float16



(c) FLUD performance under double

Fig. 7: FLUD performance across various data types, number of PEs, and input matrix sizes

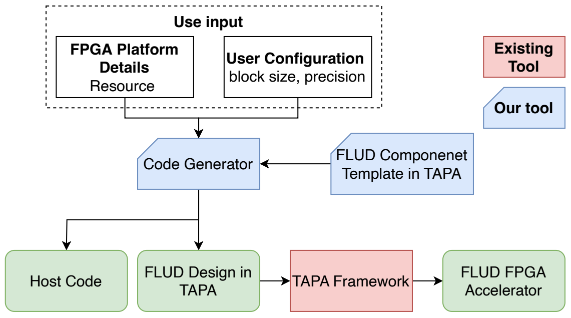


Fig. 8: FLUD automation flow

$$T = 2N^3 f / 3L \tag{11}$$

Based on Equations 7 to 11, when the matrix dimension N is large enough, the throughput $T(N)$ is proportional to B^2, which is also the total number of PEs:

$$\lim_{N\to\infty} T(N) = 2fB^2 \propto B^2 \tag{12}$$

E. Design Automation Tool

To enable fast design space exploration, we also develop an automation tool, as shown in Figure 8, to generate the highly-optimized FLUD accelerator on a given FPGA. The input of our automation tool includes 1) the FPGA platform information such as available resources, 2) user-specified configuration such as the block size and data precision (default is float). Based on these inputs and our FLUD PE and PEG templates pre-designed in Vitis HLS and TAPA [19], our code generator will automatically generate both the host code and the FLUD design in TAPA, which will be finally built into the FPGA bitstream. TAPA [19] is a programming framework that provides coarse-grained floorplanning and pipelining optimizations that improve the timing closure for task-parallel HLS designs on modern multi-die FPGAs. When a user omits the block size, our tool will automatically choose the maximum block size (i.e., the maximum number of PEs) that can fit on the target FPGA, based on our performance models presented in Section III-D and the resource estimation.

IV. Experimental Results

A. Experimental Setup

FLUD is built for the AMD/Xilinx Alveo U280 datacenter FPGA using Vitis and Vitis HLS 2021.2, together with the TAPA framework [19] to improve its floorplanning. The host code is written using XRT 2021.2 APIs, which runs on the Intel Xeon Sliver 4212 CPU with Ubuntu 18.04 OS. We measure the on-board performance of FLUD on different precisions (half, single, and double-precision floating-point) across a range of matrix sizes (1024×1024 to $16,384 \times 16,384$).

The FPGA power consumption is measured using the vendor *xbutil* tool and the resource utilization is extracted from the post place and route report. Last, we also compare the performance and power efficiency of FLUD with the widely used LAPACK library [15] on the 12-core Xeon Sliver 4212 CPU with g++ -O2 optimization and 24 hyper-threads.

B. FLUD Performance

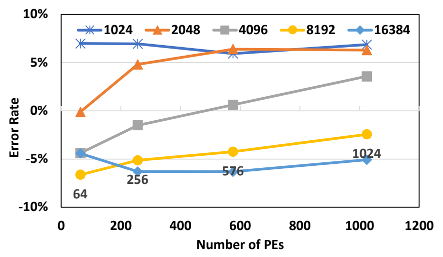
We evaluate the throughput of our designs with multiple data precisions, including `float`, `float16`, and `double`, as shown in Figure 7a to Figure 7c. In each sub-figure, on the x-axis, we sweep the number of PEs from 8×8 to the maximum number of PEs that can pass the timing on the Alveo U280 FPGA. Each line shows the throughput under a different matrix size from 1024×1024 to $16,384 \times 16,384$.

As shown in Figure 7a, FLUD can reach a peak throughput of 427.95 GFLOPS with 32×32 PEs (32 PEGs) on `float` data type. The throughput increases as the number of PEs grows, as analyzed in Section III-D. When the input matrix size increases, the throughput of FLUD also increases and progressively approaches its upper limit, because the fixed overhead in Equations 7 and 8 (L_{DIV}, L_{MUL}, and L_{MAC}) plays a less weight. In fact, with 32 PEGs, where each PEG includes 31 PEs that perform on a MAC operation and 1 PE that performs a MAC and DIV operation, the theoretical maximum throughput is $32\times(31\times2+3)$ = 2,080 FLOPs/cycle. As HBM reaches its maximum bandwidth at 225 MHz in our configurations (512-bit streaming width) [20] and FLUD operates in a streaming way, the theoretical maximum throughput is 2,080 FLOPs/cycle $\times$ 225 MHz = 468 GFLOPS. Our design achieves more than 91% of the theoretical maximum throughput; the small gap is due to the resource underutilization in C, L, and U blocks.

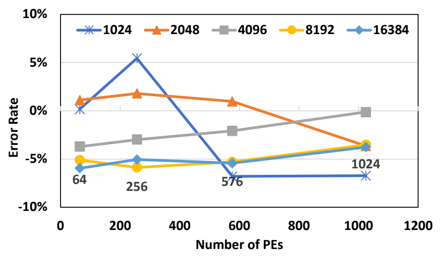
The throughput for `float16` and `double` present a similar trend to `float`. Due to the resource consumption difference, `float16` can accommodate more PEs while `double` allows for fewer PEs. `float16` achieves a peak throughput of 659.48 GFLOPS with 40×40 PEs, while `double` reaches a peak throughput of 153.46 GFLOPS with 20×20 PEs.

C. Accuracy of Our Performance Model

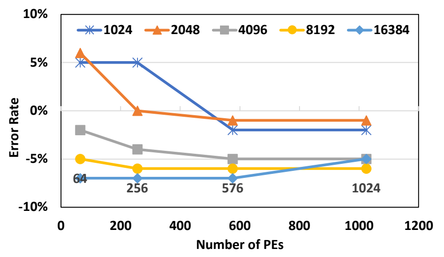
To evaluate the accuracy of our analytical performance model in Section III-D, we compare the measured performance



(a) FLUD accuracy under float



(b) FLUD accuracy under float16



(c) FLUD accuracy under double

Fig. 9: FLUD performance model accuracy across various data types, number of PEs, and input matrix sizes

TABLE II: Performance and energy efficiency comparison between FLUD, prior FPGA implementations, and CPU

Platform	Peak GFLOPS	GFLOPS / W	Frequency
CPU: LAPACK	90.72	1.07	NA
FPGA: [14]	142.15	3.64	156 MHz
FPGA: [13]	158.27*	NA	NA
FLUD (float)	427.95	10.97	255 MHz

of each configuration against the predicted one. Figure 9a to 9c presents the prediction error rates of our performance model across various matrix sizes and numbers of PEs, under float, float16, and double. For all configurations, the accuracy of our performance model lies within 7%.

D. Comparison to Prior FPGA Accelerators

In Table II, we compare FLUD to two recent FPGA designs that achieved the highest on-board performance for LUD. The approach in [14] evaluates their design on a cluster of four U280 FPGAs under the single-precision float, with a projected performance of 142 GFLOPS on a single U280. Our method achieves a significant speedup of 3x. The study in [13] provides one OpenCL benchmark on LUD (without detailed implementation details), evaluated on an Intel Arria 10 FPGA with a matrix size of 8,192$\times$8,192, achieving a throughput of 133.52 GFLOPS under float. Given that each DSP on Arria 10 can process one MAC operation, this performance can be extrapolated to U280, projecting a potential throughput of 158.27 GFLOPS. Our design is still 2.7x higher than the projected result of [13].

E. Comparison to CPU Library

Table II also compares the performance and energy-efficiency between FLUD and the widely used LAPACK [15] library on the CPU. Compared to LAPACK, FLUD achieves 4.71x higher GFLOPS and 10.25x better GFLOPS/watt.

F. Resource Utilization Comparison

Table III compares resource utilization between grouped PE design and non-grouped PE design (baseline 2D systolic array) on float data type. All resource utilization is collected post placement and routing except that the 32×32 non-grouped PE design is collected after synthesis. We compare the resource utilization from 8×8 PEs to 32×32 PEs on Alveo U280. The grouped PE design can fit at most 32×32 PEs while the non-grouped PE design can only fit 18×18 PEs.

On average, the grouped PE design can save 32.77% LUTs and 27.49% FFs on the Alveo U280 FPGA compared to the non-grouped PE design. Both grouped and non-grouped

TABLE III: Resource utilization comparison between grouped design and non-grouped design (baseline 2D systolic array)

Design Type	#PE	LUT	FF	BRAM	DSP
Non-Grouped Design	8×8	11.00%	6.00%	0.60%	3.10%
	16×16	31.72%	18.95%	0.60%	13.00%
	18×18	47.84%	30.77%	1.16%	14.61%
	32×32*	150.60%	100.90%	1.16%	60.30%
Grouped Design	8×8	7.30%	4.30%	0.60%	3.10%
	16×16	21.60%	13.90%	0.60%	13.00%
	18×18	27.70%	17.30%	4.10%	14.61%
	32×32	53.12%	37.92%	10.12%	60.30%
[14]		51%		49%	69%

designs consume the same amount of DSPs when the number of PEs is identical, as they equip the same computation units. Note when the number of PEs exceeds 16×16, the grouped PE design will implement the register array inside each PE with BRAMs to offload the usage of LUTs and FFs, consuming more BRAMs than the non-grouped PE design.

Finally, compared to the prior design [14], our design achieves better throughput while consuming fewer resources. Specifically, our streaming-based systolic array design requires significantly fewer BRAMs.

V. Conclusion

In this work, we have presented FLUD, a streaming-based systolic array design on the FPGA to effectively accelerate block LUD for arbitrary input sizes. FLUD exploits both column-level parallelism and iteration-level parallelism in the systolic array design and further shares the design to compute different LUD blocks with an optimized computation schedule. Moreover, FLUD groups a column of PEs together to replace their costly FIFO connections with lightweight registers and reduce multiple copies of local control logic inside each PE into a global one. Lastly, we also develop an automation tool to automatically generate the optimized FLUD design on a given FPGA platform with user configurations. Experimental results demonstrate that FLUD achieves a peak throughput of 427.95 GFLOPS for float, which is about 3x faster than state-of-the-art FPGA design and 4.71x faster than the widely used LAPACK library running on a 12-core Xeon Silver 4214 CPU. In future work, we plan to open source the design of FLUD.

VI. Acknowledgement

This work is partly supported by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; CFI John R. Evans Leaders Fund; Huawei Canada, and AMD-Xilinx.

References

[1] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *Int. J. High Perform. Comput. Appl.*, vol. 35, no. 4, p. 344–369, jul 2021. [Online]. Available: https://doi.org/10.1177/10943420211003313

[2] A. A. Hussain, N. Tayem, M. O. Butt, A.-H. Soliman, A. Alhamed, and S. Alshebeili, "Fpga hardware implementation of doa estimation algorithm employing lu decomposition," *IEEE Access*, vol. 6, pp. 17 666–17 680, 2018.

[3] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU press, 2013.

[4] L. Trefethen and D. Bau, *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. [Online]. Available: https://books.google.ca/books?id=5Y1TPgAACAAJ

[5] D. Silvestre, J. Hespanha, and C. Silvestre, "A pagerank algorithm based on asynchronous gauss-seidel iterations," in *2018 Annual American Control Conference (ACC)*, 2018, pp. 484–489.

[6] A. Frank, D. Fabregat-Traver, and P. Bientinesi, "Large-scale linear regression: Development of high-performance routines," *Applied Mathematics and Computation*, vol. 275, pp. 411–421, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0096300315015805

[7] J. J. Dongarra, F. G. Gustavson, and A. Karp, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine," *SIAM Rev.*, vol. 26, no. 1, p. 91–112, jan 1984. [Online]. Available: https://doi.org/10.1137/1026003

[8] J. Dongarra, S. Hammarling, and D. Walker, "Key concepts for parallel out-of-core lu factorization," *Computers & Mathematics with Applications*, vol. 35, no. 7, pp. 13–31, 1998, advanced Computing on Intel Architectures. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0898122198000297

[9] G. Wu, Y. Dou, J. Sun, and G. D. Peterson, "A high performance and memory efficient lu decomposer on fpgas," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 366–378, 2012.

[10] M. K. Jaiswal and N. Chandrachoodan, "Fpga-based high-performance and scalable block lu decomposition architecture," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 60–72, 2012.

[11] M. K. Kumar, Z. Choudry, and S. Purini, "Accelerating lu-decomposition of arbitrarily sized matrices on fpgas," in *2023 International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA/VLSI-DAT)*, 2023, pp. 1–4.

[12] Q. Y. Tang, "Fpga based acceleration of matrix decomposition and clustering algorithm using high level synthesis," Master's thesis, University of Windsor, Windsor, Ontario, Canada, Jan. 2016. [Online]. Available: https://scholar.uwindsor.ca/etd/5669

[13] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 409–420.

[14] M. Meyer, T. Kenter, and C. Plessl, "Multi-fpga designs and scaling of hpc challenge benchmarks via mpi and circuit-switched inter-fpga networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 2, mar 2023. [Online]. Available: https://doi.org/10.1145/3576200

[15] "Lapack - linear algebra package," 2024, last accessed July 28, 2024. [Online]. Available: https://www.netlib.org/lapack/

[16] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 93–104. [Online]. Available: https://doi.org/10.1145/3431920.3439292

[17] G. Govindu, S. Choi, V. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar, "A high-performance and energy-efficient architecture for floating-point based lu decomposition on fpgas," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 149–.

[18] L. Zhuo and V. K. Prasanna, "High-performance and parameterized matrix factorization on fpgas," in *2006 International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.

[19] L. Guo, Y. Chi, J. Lau, L. Song, X. Tian, M. Khatti, W. Qiao, J. Wang, E. Ustun, Z. Fang, Z. Zhang, and J. Cong, "Tapa: A scalable task-parallel dataflow programming framework for modern fpgas with co-optimization of hls and physical design," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, dec 2023. [Online]. Available: https://doi.org/10.1145/3609335

[20] A. Lu, Z. Fang, and L. Shannon, "Demystifying the soft and hardened memory systems of modern fpgas for software programmers through microbenchmarking," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, jun 2022.