MatFactory: A Framework for High-Performance Matrix Factorization on FPGAs

Mingzhe Zhang[1], Xiaochen Hao[2], Hongbo Rong[3], Wenguang Chen[1]
[1]Tsinghua University, [2]Peking University, [3]Intel
zmz21@mails.tsinghua.edu.cn, xiaochen.hao@stu.pku.edu.cn, hongbo.rong@intel.com, cwg@tsinghua.edu.cn

ABSTRACT

Matrix factorization is a widely used powerful tool in signal processing, machine learning and high performance computing. For accelerating matrix factorization, FPGAs are suitable platforms, as they can build wide and deep pipelines with favorable power efficiency. Factorizing matrices on FPGAs is thus desirable; however, there is no infrastructure on FPGAs for matrix factorization so far, as it involves several challenges: applicability and scalability of the circuit, pipelining of irregular computing patterns, and effective data caching given the limited memory bandwidth.

We propose MatFactory, a novel framework that enables fast development of high-performance algorithms for factorizing matrices on FPGAs. We extract common key operators out of various factorization algorithms, and provide a convenient streaming interface that explicitly moves and manages data through the memory hierarchy. With the interface support, the operators can be easily reused as building blocks and composed together into diverse in-BRAM non-blocked factorization algorithms as well as in-DRAM blocked factorization algorithms. We evaluate MatFactory with three typical algorithms (Cholesky, LU and QR) on Intel A10 FPGA. Our non-blocked factorization achieves 4.0-10.7× speedup over Vitis Library on Xilinx Alveo U280 FPGA, and the blocked implementation further achieves 1.65-1.88× performance compared to the non-blocked version. This is the first framework that systematically designs and accommodates various matrix factorization algorithms for FPGAs, to the best of our knowledge, and it can be easily extended to support more LAPACK routines in general.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**.

ACM Reference Format:
Mingzhe Zhang, Xiaochen Hao, Hongbo Rong, Wenguang Chen. 2024. MatFactory: A Framework for High-Performance Matrix Factorization on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676780

1 INTRODUCTION

Matrix factorization is a fundamental technique in linear algebra that decomposes a matrix into a product of two or more matrices. This technique can reveal hidden pattern and relationship of data, making it essential for applications such as data mining, recommendation system, and signal processing. Each type of algorithm, including Cholesky, LU, QR, is suited to specific problems and data types, requiring huge efforts in implementation.

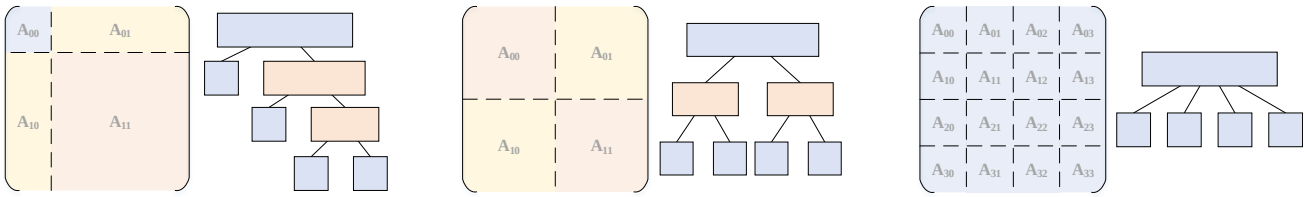
Matrix factorization is of great interest to accelerate. There has been well-established compute libraries on GPUs [25, 26], facilitating the development of HPC-related applications. In edge computing scenarios, FPGAs demonstrate a significant advantage in energy efficiency, and thus attract great attention as radar processing platforms [1, 16, 24, 29] and so on. However, developing a high performance matrix factorization library on FPGAs to tackle the varying requirements of diverse algorithms, matrix sizes, and data types is a non-trivial task due to the extremely-low productivity. Currently, only vanilla implementations of several algorithms on specific matrix sizes can be found on FPGAs.

We identify the major challenges in accelerating matrix factorization. First, matrix factorization algorithms involve triangular loop nests, where the trip count of an inner loop decreases over time. Unlike matrix multiplication that can be accelerated through regular systolic arrays, this type of loop structure often leads to uneven distribution of computations onto processing elements (PEs) and non-neighbor communication patterns between PEs, incurring huge difficulty in building a deeply-pipelined hardware structure for it. Second, the existing works usually target small matrices and assume that the entire matrix can be stored on-chip. However, these non-blocked implementations would rapidly reach the resource limitation, preventing their scalability. Only several efforts [17, 33, 36] demonstrate scalability, yet with relatively low performance for large matrices. Third, the existing implementations are tied to specific algorithms. They employ diverse approaches and use highly-customized circuits to exploit the properties and engineer performance for an algorithm [9, 13, 18, 21, 23, 35]. However, with the growing demands of different algorithms, these implementations suffer from limited applicability, and cannot be easily ported to other scenarios. Both non-blocked and blocked implementations fail to explore and utilize the similarity between different factorization algorithms.

To deal with these challenges, we propose MatFactory, a unified matrix factorization framework that achieves versatility and performance simultaneously through a systematic decomposition approach. Each factorization algorithm is decomposed into fine-grained hardware components, called *operators*. We can generalize the non-blocked factorization by breaking it down into elementary column operation (ECO) and dot product (DOT), while the blocked factorization is broken down into BLAS level-3 routines: triangular solve matrix (TRSM), symmetric rank-k update (SYRK) and general matrix multiply (GEMM). These vector and matrix operations benefit scalable implementations with regular compute patterns. We further note that there exist opportunities for reusing

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3676780



(a) Tail-recursive partitioning (b) Tree-based recursive partitioning (c) Uniform partitioning

Figure 1: Different matrix partitioning schemes used in dense linear algebra algorithms.

operators across different factorization algorithms. This allows us to share optimization techniques to tackle the similar triangular loop structures in different algorithms, achieving high performance through a uniform top-down strategy.

We also propose a data streaming interface to facilitate the recomposition of operators. Ideally, a large matrix should be stored in the device DRAM to avoid frequent data transfers between host and device memory. Matrix factorization can then be performed in place, meaning that the outputs are written into the same location as the inputs within DRAM. This process entails handling complex data access patterns, which is the primary obstacle in realizing a high performance design. We provide users with a unified streaming interface to direct data movement across three memory levels: DRAM, BRAM, and registers. This interface conceals the complexities of optimizations by automatically specializing it into an on-chip data loader/unloader to accommodate common data movement patterns. This approach enables us to separate the concerns of algorithm decomposition and data orchestration, improving the productivity with a simplified implementation flow.

We evaluated MatFactory on Intel Arria 10 GX1150 FPGA with three typical algorithms (Cholesky, LU and QR) in LAPACK routines[1]. Our non-blocked factorization reaches an average of 10.7×, 4.0× and 10.5× performance compared with the corresponding Vitis Library routines evaluated on Xilinx Alveo U280 FPGA; the blocked factorization further achieves 1.65-1.88× speed up compared to the best performance of the non-blocked design. This is the first attempt to accommodate various matrix factorization algorithms for FPGAs.

In summary, our contributions are as follows:

- We propose an operator-level abstraction that standardizes and unifies the designs of matrix factorization on FPGAs, for non-blocked and blocked algorithms.
- We develop a user-friendly streaming interface using C++ templates. It supports diverse data movement patterns across three levels of memory hierarchy.
- We productively develop a high-performance library of LAPACK routines, achieving performance comparable to, or better than, the state-of-the-arts.

2 BACKGROUND AND MOTIVATION

In this section, we use an illustrating example to show typical matrix factorization algorithms, and then briefly discuss the current status of linear algebra implementations on FPGAs.

[1]Their corresponding routines are potrf, getrf_nopivot and geqrf.

2.1 General Structure of Matrix Factorization

In this subsection, we introduce general matrix factorization algorithms, including non-blocked algorithms and blocked algorithms. We use in-place LU factorization as an example in which we would like to decompose a square matrix into a lower triangular matrix and an upper unitriangular matrix:

$$A = \begin{bmatrix} l_{00} & & 0 \\ \vdots & \ddots & \\ l_{n0} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} 1 & \cdots & u_{0n} \\ & \ddots & \vdots \\ 0 & & 1 \end{bmatrix} = LU \tag{1}$$

2.1.1 Non-blocked matrix factorization. Gaussian elimination is a well-known method for solving non-blocked LU factorization. However, it operates directly on the original matrix and involves many read-modify-write cycles per element due to elementary row operations. Instead, we follow the Crout's method [3], in which **each computed value is written only once**, as is shown in Equation (2) and (3). Although the count of basic arithmetic operations remains the same, the dominant computation pattern shifts from elementary row operations to dot products for better scalability.

$$l_{ij} = a_{ij} - \Sigma_{k=0}^{j-1} l_{ik} u_{kj} \tag{2}$$

$$u_{ji} = (a_{ji} - \Sigma_{k=0}^{j-1} l_{jk} u_{ki}) \div l_{jj} \tag{3}$$

2.1.2 Blocked matrix factorization. Blocked factorization algorithms shift most of the workload to the high-performance matrix-matrix operations. In Equation (4), all matrices are divided into four blocks. We perform the matrix multiplication blockwise, yielding the submatrices in the resulting matrices L and U with the following equations:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{bmatrix} = LU \tag{4}$$

$$L_{00}U_{00} = A_{00} \tag{5}$$

$$U_{01} = L_{00}^{-1} A_{01} \tag{6}$$

$$L_{10} = A_{10} U_{00}^{-1} \tag{7}$$

$$L_{11}U_{11} = A_{11} - L_{10}U_{01} \tag{8}$$

As we can see, the left-hand side of Equation (5) and (8) are two smaller sub-matrix factorization tasks. Equation (6) and (7) are TRSM and the right-hand side of Equation (8) is a GEMM.

Note that in Equation (4), A_{00} and A_{11} are square submatrices of arbitrary size. Based on the blocked algorithm, two partitioning schemes of matrix A can be derived [28]. Tail-recursive scheme

in Figure 1(a) traverse the input with **a fixed step-size**. It partitions a matrix into "panel"s and processes the panels iteratively. At each step of the algorithm, a small submatrix of fixed size (A_{00}) is factorized in a non-blocked way, and then followed by an update of the trailing matrix through matrix-matrix operations, leaving a shrinking large submatrix (A_{11}) to be factorized for the following steps. According to the call graph, we could easily flatten it into a loop-based implementation. Tree-based recursive scheme in Figure 1(b) splits the input matrix A evenly in half. To factorize A_{00} and A_{11}, it then recursively splits these submatrices. The recursion stops when the submatrix size falls below **a certain threshold** and the call graph is a balanced binary tree. We refer to the step-size or threshold as the *block size*. The block size is determined at compile time. We can directly factorize matrices that are equal to or smaller than the block size.

2.2 BLAS and LAPACK Implementations

It is natural to use LAPACK to accelerate matrix factorization tasks. Since most LAPACK routines are constructed in terms of calls to BLAS, we first discuss several BLAS solutions on FPGAs, which may offer some applicable insights in the development of LAPACK hardware libraries.

The blocked optimization of matrix multiplication on systolic array architectures has become well-developed and standardized [6, 20]. Matrices are evenly tiled in 2D (Figure 1(c)), transforming a large matrix multiplication into many isomorphic submatrix multiplications. A systolic array can process these tiles one by one: the elements of a tile are streamed from DRAM, buffered on BRAM, and then forwarded to the boundaries of the first row and column of PEs. This approach is applicable to other BLAS level 3 routines. FBLAS [5] and Lasa [8] are two representative works that develop BLAS libraries on FPGAs. FBLAS has realized all generic BLAS routines by composing HLS modules written in OpenCL but does not take advantage of matrix properties such as symmetry and triangularity. Consequently, the computational overhead of TRMM and SYRK is nearly equivalent to that of GEMM. Lasa provides a compiler-assisted approach to generate BLAS routines. To transform compute-intensive loops into a systolic array, developers describe the algorithm using a Domain Specific Language (DSL), thereby delegating spatial mapping and optimizations to a compiler, which substantially boosts performance as well as productivity.

In contrast, blocked matrix factorization does not merely break down into smaller decomposition tasks after partitioning. Various submatrix operations, including non-blocked factorization and matrix-matrix operations, pose a huge challenge to systolic array architectures. Effective implementation requires careful consideration not only of the non-blocked part but also of synchronization and data movement across other parts of the design. The compiler-assisted approach exhibits limitations when dealing with blocked matrix factorization, as loop tiling cannot be directly applied to the triangular loop nest. Therefore, the scalability of the generated array may be compromised. For example, AutoSA [32] handles LU factorization up to a maximum size of 24×24 (an 24×24 systolc array). Vitis Library [35] provides the non-blocked version of potrf, getrf_nopivot and geqrf. These LAPACK routines are synthesizable only when the matrix is small enough to resolve the routing congestion and fit in the on-chip BRAM.

2.3 Operator-level Abstraction

As we have shown in Section 2.1, the blocked factorization consists of non-blocked factorizations and matrix-matrix operations in BLAS level 3 routines. We further generalize non-blocked factorization by abstracting vector operations from Cholesky-Crout algorithm and QR-MGS (Modified Gram Schmidt) algorithm. Table 1 summarizes the operators in MatFactory. To adapt these algorithms to the hardware, we make use of built-in hardened dot-product intellectual property (IP) cores. DOT and ECO constitute non-blocked factorization. TRSM, SYRK and GEMM contribute to the trailing matrix update. Moreover, GEMM desgin can be reused for SYRK. By leveraging the symmetry, we iterate only the lower triangle of the input/output matrices to save half of computation.

Table 1: Abstracted operators in matrix factorization.

	DOT	ECO	TRSM	SYRK	GEMM
Non-blocked Cholesky	✓				
Non-blocked LU	✓				
Non-blocked QR	✓	✓			
Blocked Cholesky	✓		✓	✓	
Blocked LU	✓		✓		✓
Blocked QR	✓	✓			✓

3 FRAMEWORK OVERVIEW

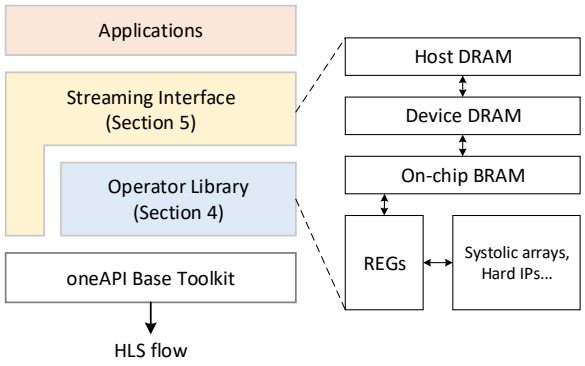


Figure 2: The overall workflow of MatFactory.

In this section, we introduce our framework, as well as the programming, execution and memory models.

Figure 2 shows an overview of MatFactory. It consists of an operator library and a streaming interface, which is used to build matrix factorization algorithms. The operator library offers cross-language compatibility[2], supporting Verilog, OpenCL, and DPC++. For example, developers are able to construct operators from RTL sources and integrate them into DPC++ designs. The C++ template-based streaming interface is used for data management and movement across three levels of a memory hierarchy.

In our memory hierarchy, the host has only a DRAM, whereas the device has a DRAM, on-chip BRAMs, and registers. Data are

[2] `fpga_crossgen` and `fpga_libtool` commands create a single library archive file.

transferred over a PCIe bus, a device DRAM bus, or on-chip interconnects (i.e, FIFOs) depending on their locations. For host/device DRAM, there are two memory access modes in oneAPI: buffer and accessor mode and Unified Shared Memory mode (USM). A buffer acts as a container for data that can be accessed from both host and device side, while an accessor indicates when and where the data is needed, and the actual data movement and synchronization are done by the SYCL runtime transparently. USM allows reading and writing of data with conventional pointers. It offers two pointer allocation APIs: (1) Shared pointer, which functions similarly to the buffer and accessor, with the runtime automatically moving data; (2) Device pointer, which can only be accessed from the device and therefore require explicit movement of data between host and device. We choose USM with explicit device pointers since it gives us full control over data movement and synchronization, enabling on-demand movement of any granularity.

We assume that multiple pipelines may be running on the same device. Each of them consists of a set of kernels. Specifically, the first kernel reads data from DRAM, the last kernel writes the results back to DRAM, and other kernels perform the computational tasks. The core of execution model is defined by how the kernels execute. When a kernel is submitted for execution, the kernel and the values associated with the arguments to the kernel define a kernel-instance. An event associated with a particular kernel-instance is used to constrain the order of execution among related kernel-instances. This mechanism enables synchronization across the pipelines.

MatFactory is built upon oneAPI [12] with HLS design flow. It currently targets Intel FPGAs; however, the methodology could be adapted to cover Xilinx FPGAs while maintaining the same high-level interface. OneAPI uses Data Parallel C++ (DPC++), which is based on modern C++ and incorporates SYCL [7]. The DPC++ Compiler conforms to C++17 language standard by default[3]. In contrast, OpenCL is based on C99, which is a C standard more than two decades old and has more language limitations. We found that code written in OpenCL can be migrated to DPC++ without performance degradation. Furthermore, we can leverage new language features in DPC++ to make our framework more flexible and productive.

4 COMPUTE OPTIMIZATIONS

In this section, we discuss the detailed design of operators and the pipelining of computations.

4.1 Operator Design

In an HLS program, operators are implemented with unrolled loops. Every expression within an unrolled statement will be instantiated as hardware, and the unrolling factor must be a compile-time constant. The parallelism on spatial architectures is achieved by either horizontal unrolling (SIMD vectorization) or vertical unrolling (pipeline parallelism) [4]. Below we describe how to write a GEMM.

Matrix addition in GEMM is trivial. Therefore, we simply parallelize it in SIMD manner. The matrix multiplication code is shown in Figure 3. Line 1-4 replicates the computational unit multiple times via vertical unrolling, so that data can be processed in a pipelined fashion. By introducing vectorization (Line 6-7), input vectors from

[3]Newer versions can be selected by using the `-std` flag.

```
1  #pragma unroll
2  for (int i = 0; i < rows; i++) {
3    #pragma unroll
4    for (int j = 0; j < cols; j++) {
5      sum = init ? 0 : c[i][j][0];
6      #pragma unroll
7      for (int k = 0; k < vlen; k++) {
8        a[i][k] = (j == 0) ? tile_a[i][k] : a[i][k];
9        b[j][k] = (i == 0) ? tile_b[j][k] : b[j][k];
10       sum += a[i][k] * b[j][k];
11     }
12     c[i][j][0] = sum;
13   }
14 }
```

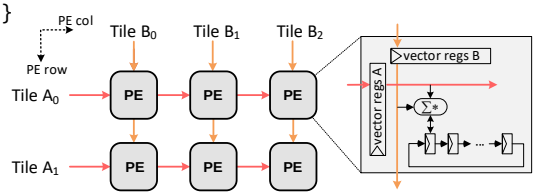


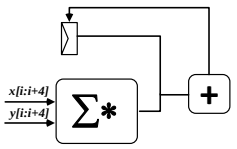
Figure 3: Implementation of MatMul Systolic Array.

the boundaries of the systolic array are propagated between computational units (Line 8-9). Line 10 shows a typical reduction pattern on vectors; `sum` is initialized with 0 if a reduction just starts; otherwise, it gets the value from the previous partial sum (Line 5). With the help of streaming interface, the systolic array is able to process input matrices tile by tile. We will discuss it in Section 5.

4.2 Pipeline-Enabling Transformations

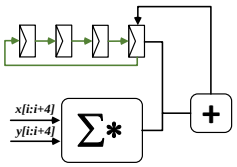
Unrolling only provides the potential degree of parallelism in the architecture. However, whether a design is well-pipelined is determined by the loops enclosing these operators. An important property of the outer loops is initiation interval (II), which refers to the number of hardware clock cycles for which the pipeline must wait before it can process the next iteration. II can often be considered the inverse throughput of the pipeline, and a perfect pipeline is expected to have II=1.

4.2.1 Accumulation Interleaving. Dot product is a commonly used compute pattern. As shown in Figure 4(a), we have a DOT operator that can only processes input vectors of length 4. We enclose DOT with an outer reduction loop to support dot products of arbitrary size. The addition on Line 2 requires the result of the addition in the previous iteration of the loop. There is a self-dependence across the outer loop iterations. For single precision, effective pipelining is not hindered as each addition operation takes only one cycle. However, this loop-carried dependency is particularly relevant when targeting double-precision, because the DSP block does not support double precision addition natively. Assume that a 64-bit floating point addition takes four cycles, the pipeline would stall until the previous one completes. This issue can be resolved by interleaving nested loops, as illustrated in Figure 4(b). Rather than a single register, we keep partial sums in four registers. Each location is only updated every 4 cycles and thus the previous addition has sufficient cycles to complete. Accumulation interleaving is often accompanied by shifting/rotating registers (Line 2), which we will introduce in Section 5.




```
for (int i = 0; i < N; i+=4)
  sum+=DOT4(x[i:i+4],y[i:i+4]);
return sum;
```

(a) Single precision

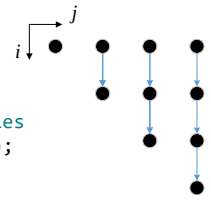



```
for (int i = 0; i < N; i+=4) {
  Rotate(sum[0:4]);
  sum[0]+=DOT4(x[i:i+4],y[i:i+4]);
}
return sum[0]+sum[1]+sum[2]+sum[3];
```

(b) Double precision

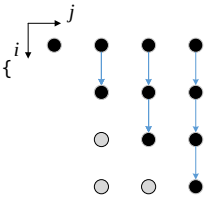
Figure 4: Interleave accumulations to resolve loop-carried dependency on sum.

4.2.2 Triangular Loop Transformation. Triangular loop nests, where the trip count of an inner loop decreases over time, appear in most non-blocked matrix factorization algorithms. Figure 5 shows such a loop nest. The loops are usually not well-pipelined by the HLS tools because of the time-varying trip count of the inner loop. Ideally, a new iteration of the loops is expected to be launched every cycle (II=1). However, as the trip count of the inner loop shrinks below a threshold, there will be insufficient cycles for operator `foo` to complete and generate its output before the following outer iteration starts and reads that output. The HLS compiler has to conservatively increase the II to relax the dependency between two calls.

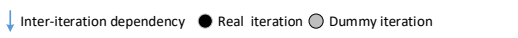



```
for (int i = 0; i < 4; i++)
  for (int j = i; j < 4; j++)
    // operator foo takes 3 cycles
    buf[i][j] = foo(buf[i-1][j]);
```

(a) Before transformation




```
#pragma ivdep safelen(3)
for (int ij = 0; ij < 13; ij++) {
  buf[i][j] = foo(buf[i-1][j]);
  i = (j==3) ? i+1 : i;
  j = (j==3) ? min(i,1) : j+1;
}
```



(b) After transformation

Figure 5: A triangular loop nest and its dependence graph.

We introduce triangular loop transformation in our framework. This transformation can be divided into two steps: (1) Inserting dummy iterations into an inner loop when the inner loop's trip count is less than a threshold `M`, and (2) flattening the triangular loop nest and adding a pragma "`ivdep safelen(M)`" before the flattened loop [14]. These dummy iterations can be viewed as "bubbles" in the pipeline, providing extra cycles for completing the computations. In the mean time, the pragma indicates to the compiler that there are a maximum of `M` iterations before a loop-carried dependence might be introduced. For the example in Figure 5, `M` is set to 3, allowing enough time to execute operator `foo` even in the worst case (i=3, j=4). As a result, we are able to achieve a perfect pipeline with II=1.

5 MEMORY OPTIMIZATIONS

In this section, we discuss the details of our streaming interface design, which is used to recompose operators as the entire matrix factorization algorithms.

5.1 Data Access Patterns

Blocked matrix factorization on FPGAs requires a more complicated I/O network. In most FPGA-accelerated scenarios, input data can span CPU host and FPGA device. Both the host and the device have their own DRAM. To improve the utilization of the off-chip memory bandwidth, data has been serialized on the host before sending them to device over a PCIe bus, then device can load them sequentially from DRAM. However, it is impractical to assume contiguous reads and writes in device DRAM, especially for the blocked algorithms, where in-place updates are performed many times and input submatrices are generated on-the-fly. For example, the resulting matrix generated by non-blocked factorization is written back to its original location in DRAM. Subsequently TRSM needs to read the lower (or upper) triangular matrix from this location. The data may be fragmented in device DRAM since it hasn't been serialized. Besides, pipelines that access data at the same address must be synchronized. These complicated data access patterns have motivated us to introduce the following memory optimizations: (1) a simplified data movement interface that allows us to move data between host and device, and across three memory levels; (2) several carefully designed mechanisms that help us mitigate the non-continuous accesses in both DRAM and BRAM.

5.2 Explicit Data Movement

We summarize commonly used data movement patterns in Table 2. The specific implementation details have been concealed through a user-friendly streaming interface provided by our framework. `MemCopy` explicitly transfers data between host and device. As we have discussed in previous subsection, it may be required to bring the data that was computed by a kernel on the device to the host and do some operation on it and send it back to the device. The cost is quite high, so it is important to avoid frequent data transfers between host and device.

We offer interfaces for accessing device memory, which use two types of Load-Store Units (LSUs) according to the access pattern: (1) a burst-coalesced LSU, which buffers DRAM requests until the largest possible burst can be made; (2) a prefetching LSU, which prefetches large blocks from DRAM to keep in a FIFO based on the previous address. To maximize bandwidth utilization, every cycle, N elements are loaded/stored simultaneously via LSU and sent/received using a single-producer single-consumer queue, referred to as *pipe*. BRAM is a crucial level for data caching as the banks keep data belonging to it from DRAM pipe and distribute data to multiple computational units. A pipelined never-stall LSU

is connected to BRAM, so multiple BRAM requests can be in flight at a time. This provides a range of optimization opportunities for on-chip data management and reuse in FPGAs.

Table 2: Data Movement Interfaces.

Function	Description
`MemCopy(host_ptr, dev_ptr)`	Copy data from host DRAM to device DRAM over PCIe.
`MemCopy(dev_ptr, host_ptr)`	Copy data from device DRAM to host DRAM over PCIe.
`DRAMToPipe(dev_ptr, N)`	Load data from DRAM and write into a pipe, N elements at a time.
`PipeToDRAM(dev_ptr, N)`	Read data from a pipe and store to DRAM, N elements at a time.
`BRAMToPipe(bram_ptr, N)`	Read data from N banks and write into a pipe.
`PipeToBRAM(bram_ptr, N)`	Read data from a pipe and distribute into N banks.
`Shift/Rotate(regs)`	Move data from one register to its neighbor.

5.2.1 Double buffering. A double buffer can be viewed as a combination of `PipeToBRAM` and `BRAMToPipe`. It consists of a read and a write buffer. Incoming data from a pipe is written into the write buffer, while output data is simultaneously transferred from the read buffer to another pipe. Once they have completed their tasks, the roles of two buffers are switched.

5.2.2 Shift/Rotate registers. We have already demonstrated shift register pattern in Section 4. Each operator is associated with shift registers that temporarily store intermediate results. These registers are cascaded and "shifted" every cycle, right before we perform a computate. Thus, the live values can be accessed from the registers with constant indices (Line 5 in Figure 3 and Line 3 in Figure 4(b)). Besides, shift registers are well-suited to spatial architectures as they can be efficiently implemented. We provide primitives (`Shift` and `Rotate`) to hide the implementation of these registers.

5.3 Skewed Matrix Storage

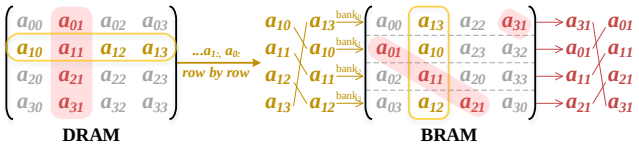
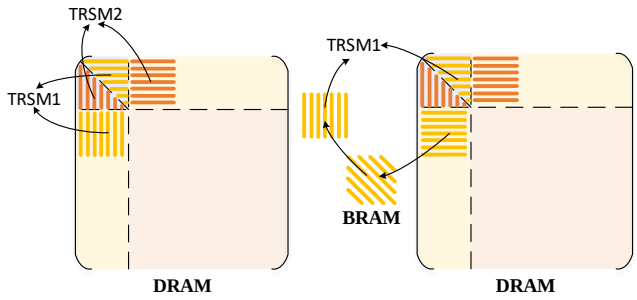


Figure 6: Memory skewing in BRAM.

We observe a requirement for fetching both rows and columns from a matrix, as exemplified in Equation (2). Specifically, the dot product requires a row vector from matrix L and a column vector from matrix U. As L and U gradually overwrite the original matrix A, they actually point to the same memory space. Memory skewing [2] is a mechanism to improve data accessibility in parallel memory systems (e.g., BRAM). Deviating from traditional row-major or column-major formats, skewed storage allows accessing rows, columns, and diagonals of a two-dimensional data at the full memory bandwidth. An example of 4×4 matrix is shown in Figure 6. A matrix that is stored in row-major order in DRAM is loaded row by row, and each row is skewed before being stored into BRAM: the n-th row is shifted by $n-1$ times, and then distributed to different BRAM banks. When reading from BRAM, a reordering circuit is required to restore the correct order of the data. The columns in the matrix can be fetched in parallel by accessing the diagonals of the skewed matrix. The shaded red areas in Figure 6 illustrate how the second column in the original matrix become a diagonal. Memory skewing also reduces on-chip resources; otherwise, we have to duplicate BRAM storage to store the matrix separately in row-major and column-major order.

5.3.1 On-chip matrix transposition. The skewed matrix layout facilitates efficient matrix transposition. For example, in Figure 7(a), there are two TRSM whose access patterns are completely different. For TRSM1, the yellow-colored upper triangular matrix and rectangular matrix are accessed in row major and column major, respectively. Assume the entire matrix is stored in row-major order. Reads from the rectangular matrix are non-contiguous. However, non-contiguous accesses of DRAM lead to inefficient LSUs and thus decrease bandwidth utilization. To address this efficiency issue, we allow the rectangular matrix to be stored and accessed in row major as usual, but then stored into BRAM with the skewed matrix layout, enabling efficient column-by-column access, as can be seen from Figure 7(b). To further improve efficiency, we implement memory skewing within a double buffer: both write buffer and read buffer are arranged in the skewed format. For TRSM2, the orange-colored rectangular matrix is accessed in-order efficiently.



(a) Direct access order **(b) Access with memory skewing**

Figure 7: Utilize skewed matrix layout to achieve on-chip matrix transposition.

Note that the accesses to the triangular matrices also introduce an efficiency problem: two triangular matrices are produced by a previous non-block LU factorization, and the orange-colored lower triangular is accessed in column major. Our solution will be covered in next subsection.

5.4 Optimizations Among Multiple Pipelines

Different pipelines accessing data from the same global memory address may lead to underutilized memory bandwidth. For instance, a non-blocked LU pipeline reads a block from DRAM, factorizes it in-BRAM and writes the result back to the same location in DRAM. TRSM pipelines then read the factorized triangular matrices from this location. Explicit synchronization is required between LU and

TRSM pipelines. The main efficiency constraint arises from the frequent and non-contiguous accesses of DRAM for small triangular matrices.

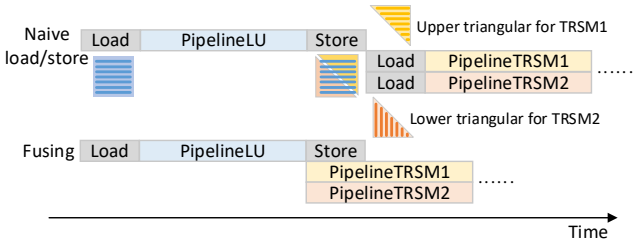


Figure 8: Fusing LU and TRSM using on-chip BRAM.

5.4.1 On-chip fusion. Since the triangular matrices are generated by previous pipeline on chip, they can be directly passed to the subsequent pipeline as inputs without re-read from DRAM. Therefore, to reduce the frequent and non-contiguous read operations of the triangular matrices, we fuse the pipelines of LU and TRSM during each step of factorization as is shown in Figure 8. On-chip fusion can significantly increase the off-chip DRAM bandwidth utilization.

6 EVALUATION

In this section, we evaluate MatFactory on Intel DevCloud [10]. We use oneAPI Toolkit 2023.2 and Quartus Prime 19.2 for implementing and synthesizing our designs, and target Arria 10 GX1150 [11] board. We also use Board Management Controller [15] to measure the power dissipation at runtime. All matrices are using single-precision floating-point format (FP32).

6.1 Non-blocked Factorization

First, we discuss the performance of non-blocked in-BRAM factorization. The total arithmetic operation count for Cholesky, LU, QR is $\frac{1}{3}n^3$, $\frac{2}{3}n^3$, $2n^3$, respectively, where n represents the size of the square matrix. We set the input matrix size to 128×128, run each factorization 8192 times and calculate the average as the final GOPS. The results and resource usage have been shown in Table 3.

Table 3: Non-blocked matrix factorization (128×128) on A10.

	MHz	DSP	LUT	BRAM	GOPS	Power
Cholesky	282	9%	23%	22%	17.4	23.0 W
LU	237	18%	26%	26%	29.7	25.0 W
QR	227	18%	25%	30%	91.4	27.0 W

6.1.1 Comparison with Vitis Library. Figure 9 compares MatFactory with several HLS and RTL implementations on FPGAs. Vitis Library [35] is the most comprehensive work we have found that covers all the three algorithms with all the matrix sizes. We use Vitis Library 2023.2 as the baseline for the non-blocked evaluation. Since Vitis Library is primarily designed for Xilinx FPGAs, we test it on Xilinx Alveo U280 [34]. Table 4 lists the hardware parameters of two FPGA platforms. We tune the NCU (number of compute units) parameter in these routines to the maximum that could be synthesized, and achieve a geometric mean speedup of 10.7×, 4.0× and 10.5× over potrf, getrf_nopivot and geqrf routines in Vitis Solver Library respectively. Despite many years of development, Vitis Library still provides limited performance of matrix factorization, even on high-end FPGAs.

Table 4: Specifications of two FPGAs used in evaluation.

Platforms	Intel Arria 10 GX1150	Xilinx Alveo U280
Technology Node	Intel 20nm	TSMC 16nm
Computing Units	1518 DSPs	9024 DSPs
DRAM	8GB DDR	8GB HBM & 32GB DDR
Bandwidth	34 GB/s	460 & 38 GB/s
BRAM	65.7MB	41MB

6.1.2 Comparison with specialized designs on FPGAs. As most of designs are highly-customized for specific factorization algorithms, we also compare our framework with specialized implementations to evaluate the trade-off between performance and flexibility. AutoSA [32] is a polyhedral-based compilation framework that maps LU factorization to a 2D systolic array in a triangular shape. Kumar et al. [19] develop a linear array for computing LU factorization. MatFactory gains 1.8× and 1.4× speedup over their designs respectively. For Cholesky and QR, we demonstrate performance on par with Intel Reference design [13]. Their performance slightly surpasses ours because it isolated and precomputed all the control signals to reduce fan-out. Langhammer et al. [21] have implemented a QR circuit with a frequency of 430 MHz. Their hand-tuned RTL design achieves a much higher frequency compared to ours (288 MHz). Nevertheless, MatFactory still retains 65% of their performance while reducing the implementation effort. The results prove that our designs are of high flexibility, productivity, and comparable to expert-written HLS code.

6.1.3 Comparison with works on CPUs and GPUs. We also compare our work with the state-of-the-art implementations on many-core processors [22] and GPUs [26]. Considering that their works focus on double precision, we report the throughput as well as the efficiency, the measured throughput divided by the peak[4]. The CPU design runs on a core group. A core group of the SW26010Pro processor has a peak performance of 2.3 TFLOPS, while V100 has a peak performance of 7 TFLOPS. As shown in Table 5, we achieve much higher efficiency and absolute performance with favorable power consumption (~30W). Moreover, matrix sizes of 10^1~10^3 are representative of the typical complexity for signal processing applications, such as Kalman filter [31] and space-time adaptive processing (STAP) [24]. The results demonstrate that our FPGA designs are more suitable for small matrix factorization in real-time and low-power scenarios, with relatively low overhead.

Table 5: Performance of non-blocked Cholesky/LU/QR (256×256) on SW26010Pro, V100 GPU and Arria10 FPGA.

	GOPS	Efficiency
Ma et al. [22] (CPU)	5.5 / 6.2 / 12.0	0.24% / 0.27% / 0.52%
cuSolverDn (GPU)	19.6 / 9.1 / 16.8	0.28% / 0.13% / 0.24%
MatFactory (FPGA)	42.1 / 71.6 / 271.1	5.31% /10.6% / 31.1%

[4]For FPGAs, the peak throughput is defined as *the total number of DSPs*×*frequency*×2.

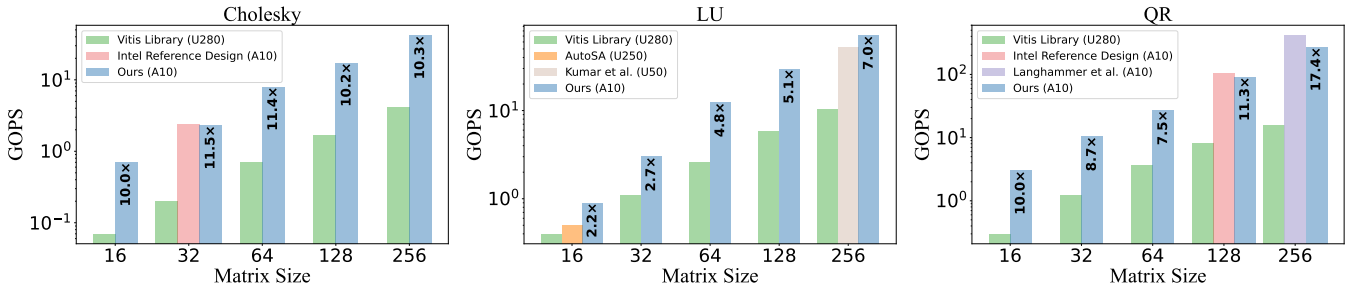


Figure 9: Comparison of non-blocked matrix factorization on FPGAs. The largest matrix size is up to 256×256 due to resource constraints.

6.2 Blocked Factorization

For the blocked factorization, we adopt the partitioning scheme shown in Figure 1(a) and convert tail recursion to a loop, because the loop-based implementation provides a better control on kernel launching and finishing. We test input matrices ranging from 1024 to 16384, and manually search for the best block size from 16 to 256. Table 6 shows the best results we can achieve on A10. The blocked factorization designs achieve 1.65-1.88× speed up compared to the best performance of non-blocked designs. Currently, we cannot support the blocked QR factorization algorithm because it involves tall-and-skinny QR factorization (TSQR) after partitioning. TSQR is challenging to fit into BRAM. Some solutions that further partition TSQR have been proposed [27, 30]. We will leave it as part of our future work.

Table 6: Blocked matrix factorization on A10.

	MHz	DSP	LUT	BRAM	GOPS	Power
Cholesky	239	60%	52%	79%	79.2	29.5 W
LU	232	66%	57%	58%	118	30.0 W

6.2.1 Effect of on-chip fusion. Table 7 show the execution time breakdown of blocked LU factorization. A performance bottleneck lies in the TRSM pipeline as it repeatedly reads triangular matrices from non-continuous addresses in DRAM. That is memory bound: the required load bandwidth is 62 GB/s, which far exceeds 34 GB/s bandwidth that A10 can provide. On-chip fusion fuses two pipelines, non-blocked LU and the subsequent TRSM. This optimization reduces the load bandwidth requirement down to 46 GB/s and achieves 6.6× performance improvement.

Table 7: The execution time (ms) breakdown of blocked LU factorization (4096×4096).

	Non-blocked LU	TRSM	GEMM	GOPS
Naive load/store	2.8	2078	492	17.8
On-chip fusion	76.6		312	118

6.2.2 Performance analysis. On-chip fusion only partially mitigates the pressure of the memory bandwidth. The design remains memory bound as long as multiple pipelines access data from the same global memory address. Currently, GEMM almost dominates the blocked factorization, as it occupies a significant portion of the total execution time. We create a 8×8×8 (height×width×vector length) systolic array for GEMM. The systolic array can achieve 205 GOPS if we test it independently. However, there will be a performance degradation if we integrate it into our blocked factorization design. A profile of the performance of GEMM in each iteration is shown in Figure 10. We have identified two main reasons for the degradation: (1) a standalone GEMM is not limited by memory bandwidth and can consume data at the maximum rate required by the systolic array; (2) pipeline parallelism cannot be fully utilized, because GEMM processes tall-and-skinny input matrices (A_{01} and A_{10} in Figure 1(a)) and the input matrices' sizes gradually decrease over time. Consequently, the performance of GEMM declines significantly after the 25th iteration. Our blocked factorization design does not compare favorably with GPUs in terms of absolute performance (e.g., 118 GOPS vs. 921 GOPS for 4096×4096 matrix). This is due to the huge gap in the hardwares' capacities between V100 GPU and A10 FPGA: the memory bandwidths are 900 GB/s vs. 34 GB/s, and the computing resources are 5120 CUDA cores & 640 tensor cores vs. 1518 DSPs.

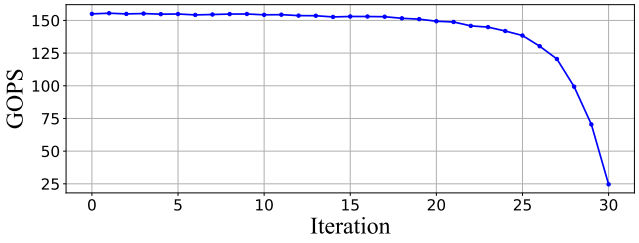


Figure 10: The performance of GEMM in each iteration of the blocked LU factorization (matrix size is 4096, block size is 128).

7 CONCLUSIONS

We propose a productive and high-performance framework, MatFactory, for factorizing matrices on FPGAs. Our abstraction is sufficiently simple and effective to accommodate various algorithms on spatial architectures, ensuring versatility across different matrix factorization scenarios. We have developed key LAPACK routines using this framework and achieved performance comparable with or better than existing hardware libraries and expert-written code.

REFERENCES

[1] Ray Andraka and Andrew Berkun. 1999. FPGAs make a radar signal processor on a chip a reality. In *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers (Cat. No. CH37020)*, Vol. 1. IEEE, 559–563.

[2] Paul Budnik and Davis J Kuck. 1971. The organization and use of parallel memories. *IEEE transactions on computers* 100, 12 (1971), 1566–1569.

[3] Prescott D Crout. 1941. A short method for evaluating determinants and solving systems of linear equations with real or complex coefficients. *Electrical Engineering* 60, 12 (1941), 1235–1240.

[4] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2020. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1014–1029.

[5] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. 2020. FBLAS: Streaming linear algebra on FPGA. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[6] Yong Dou, Stamatis Vassiliadis, Georgi Krasimirov Kuzmanov, and Georgi Nedeltchev Gaydadjiev. 2005. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. 86–95.

[7] The Khronos Group. 2024. SYCL. https://www.khronos.org/sycl

[8] Xiaochen Hao, Mingzhe Zhang, Ce Sun, Zhuofu Tao, Hongbo Rong, Yu Zhang, Lei He, Eric Petit, Wenguang Chen, and Yun Liang. 2023. Lasa: Abstraction and Specialization for Productive and Performant Linear Algebra on FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 34–40.

[9] Ali Ibrahim and Maurizio Valle. 2018. Real-time embedded machine learning for tensorial tactile data processing. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 11 (2018), 3897–3906.

[10] Intel. 2024. Developer Clouds for Accelerated Computing. https://devcloud.intel.com

[11] Intel. 2024. Intel Arria 10 FPGA and SoC Product Table. https://cdrdv2-public.intel.com/714167/arria-10-product-table.pdf

[12] Intel. 2024. oneAPI Programming Model. https://www.oneapi.io

[13] Intel. 2024. oneAPI Samples. https://github.com/oneapi-src/oneAPI-samples

[14] Intel. 2024. Removing Loop-Carried Dependencies Caused by Accesses to Memory Arrays. https://www.intel.com/content/www/us/en/docs/programmable/683521/21-4/removing-loop-carried-dependencies-caused.html

[15] Intel. 2024. Thermal and Power Guidelines. https://www.intel.com/content/www/us/en/docs/programmable/683795/current/introduction.html

[16] Raafia Irfan, Waqas Ahmed TOOR, et al. 2017. FPGA-based Low Latency Inverse QRD Architecture for Adaptive Beamforming in Phased Array Radars. *radioengineering* 26, 3 (2017).

[17] Manish Kumar Jaiswal and Nitin Chandrachoodan. 2011. FPGA-based high-performance and scalable block LU decomposition architecture. *IEEE Trans. Comput.* 61, 1 (2011), 60–72.

[18] Jeremy Johnson, Timothy Chagnon, Petya Vachranukunkiet, Prawat Nagvajara, and Chika Nwankpa. 2008. Sparse LU decomposition using FPGA. In *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*. 1–12.

[19] Maram Krishna Kumar, Ziaul Choudry, and Suresh Purini. 2023. Accelerating LU-decomposition of arbitrarily sized matrices on FPGAs. In *2023 International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA/VLSI-DAT)*. IEEE, 1–4.

[20] Hsiang-Tsung Kung. 1982. Why systolic architectures? *Computer* 15, 1 (1982), 37–46.

[21] Martin Langhammer and Bogdan Pasca. 2018. High-performance QR decomposition for FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 183–188.

[22] Wenjing Ma, Fangfang Liu, Daokun Chen, Qinglin Lu, Yi Hu, Hongsen Wang, and Xinhui Yuan. 2023. An Optimized Framework for Matrix Factorization on the New Sunway Many-core Platform. *ACM Transactions on Architecture and Code Optimization* 20, 2 (2023), 1–24.

[23] Alexander Maltsev, Vladimir Pestretsov, Roman Maslennikov, and Alexey Khoryaev. 2006. Triangular systolic array with reduced latency for QR-decomposition of complex matrices. In *2006 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 4–pp.

[24] Volker Mauer and Michael Parker. 2011. Floating point STAP implementation on FPGAs. In *2011 IEEE RadarCon (RADAR)*. IEEE, 901–904.

[25] NVIDIA. 2024. Basic Linear Algebra on GPUs. https://developer.nvidia.com/cublas

[26] NVIDIA. 2024. Direct Linear Solvers on NVIDIA GPUs. https://developer.nvidia.com/cusolver

[27] Hiroyuki Ootomo and Rio Yokota. 2019. TSQR on TensorCores. In *SC19 research poster*.

[28] E Peise and P Bientinesi. 2016. Recursive Algorithms for Dense Linear Algebra: The ReLAPACK Collection.(2016).

[29] Jimmy Pettersson and Ian Wainwright. 2010. Radar signal processing with graphics processors (GPUs).

[30] Abid Rafique, Nachiket Kapre, and George A Constantinides. 2012. Enhancing performance of Tall-Skinny QR factorization using FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 443–450.

[31] Jeremy Soh and Xiaofeng Wu. 2016. An fpga-based unscented kalman filter for system-on-chip applications. *IEEE Transactions on Circuits and Systems II: Express Briefs* 64, 4 (2016), 447–451.

[32] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.

[33] Guiming Wu, Yong Dou, Junqing Sun, and Gregory D Peterson. 2010. A high performance and memory efficient LU decomposer on FPGAs. *IEEE transactions on computers* 61, 3 (2010), 366–378.

[34] Xilinx. 2024. AMD Alveo U280 Product Brief. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-u280-product-brief.pdf

[35] Xilinx. 2024. Vitis Accelerated Libraries. https://www.xilinx.com/products/design-tools/vitis/vitis-libraries.html

[36] Wei Zhang, Vaughn Betz, and Jonathan Rose. 2012. Portable and scalable FPGA-based acceleration of a direct linear system solver. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 5, 1 (2012), 1–26.