2023 International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA/VLSI-DAT) | 979-8-3503-3416-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/VLSI-TSA/VLSI-DAT57221.2023.10134072

Accelerating LU-Decomposition of Arbitrarily Sized Matrices on FPGAs

Maram Krishna Kumar, Ziaul Choudry, Dr. Suresh Purini
International Institute of information Technology, Hyderabad
{krishna.maram,ziaul.c}@research.iiit.ac.in, suresh.purini@iiit.ac.in

***Abstract*—In this paper, we design and develop a hardware accelerator for computing the LU decomposition of an input matrix. Our accelerator consists of two simple linear arrays of Processing Engines (PEs), one on each of the two SLR regions of the FPGA. All the computations arising from the block LU decomposition are simplified and scheduled on these two PE arrays. On an Alveo U50 FPGA, our design achieves a peak floating-point performance of 128 GLOPS/s and an average performance of 95 GFLOPS/s. We achieve $\approx 15\times$ speedup on latency compared to an Intel MKL implementation on a 4-core Intel Xeon CPU.**

***Index Terms*—LU decomposition, FPGAs**

I. Introduction

Scientific applications like aircraft-design and machine-learning algorithms involve solving a system of linear equations $A\vec{x} = \vec{b}$. LU decomposition is helpful if the system needs to be solved repeatedly for the same matrix A but for different right-hand side vectors $\vec{b}$. It also aids in other matrix operations, such as computing the determinant and inverse of a matrix. Algorithm 1 shows the well-known Gaussian elimination based method for the LU decomposition of a matrix. The sequential time complexity of the algorithm is $\Theta(n^3)$. The inner j and l loops are marked with the keywords **par for**, indicating that those loops can be executed in parallel. Thus in the k^{th} iteration of the outermost for-loop, $(N-k)\times(N-k+1)$ Multiply-and-Accumulate (MAC) operations can be performed in parallel, demonstrating ample available parallelism. However, the operational intensity, which is defined as the number of operations per byte fetched, is 0.25 (one multiplication and addition per two floating-point operands), which is very low. This indicates that memory bandwidth would constitute the bottleneck, especially in FPGAs where it is possible to perform a large number of operations in parallel using the available DSPs. Across the whole algorithm, the operational intensity goes up to $N/8$ if only the whole matrix fits into the device cache or on-chip FPGA memory, alleviating the memory bottleneck. For all practical purposes, this will not be the case as the matrix dimension N is usually large. Hence, we use the Block LU decomposition algorithm, refer Algorithm 2, to improve the operational intensity and thereby exploit the available data parallelism. In this paper, we propose a hardware accelerator on FPGAs, using which we can perform LU decomposition of matrices of arbitrary dimension. The input matrix is transferred to the HBM memory of the FPGA and the output factor matrices L and U are available in the HBM memory after the computation is over.

Algorithm 1: Gaussian elimination method for matrix LU decomposition. Small letters denote scalar matrix entries.

```
input : $A_{N\times N}$ matrix.
output: a lower triangular matrix $L_{N\times N}$ and an upper
        triangular matrix $U_{N\times N}$ where $A = LU$.
for $k \leftarrow 1$ to $N-1$ do
    par for $j \leftarrow k+1$ to $N$ do
        $l_{jk} = a_{jk}/a_{kk}$
        par for $l \leftarrow k$ to $N$ do
            $a_{jl} = a_{jl} - l_{jk} \times a_{kl}$;
        end
    end
end
$U \leftarrow A$
```

This paper focuses on dense matrix LU decomposition against sparse matrices that require a different approach [5]. We compare our work against the relevant work on dense matrices. The IP for LU decomposition in Xilinx Vitis library [6] requires the maximum matrix dimension to be specified at the synthesis time. Once synthesized, it cannot process matrices beyond that dimension. Jaiswal et al. [3] uses a linear array of Processing Engines (PEs) followed by an adder tree for matrix multiplication. As the PE array size increases, the adder tree size and the corresponding resource utilization also increase. Further, the intermediate inverse matrices are also explicitly computed. These factors limit their approach's scalability and hence use multiple FPGAs to process larger matrices with better latency. Govindu et al. [1] uses a circular array of PEs for LU decomposition and matrix multiplication. This results in long latency; hence, they show the application of their architecture on streaming input matrices to reduce the effective latency. The same architecture is reused to stream blocks of sub-matrices that arise in block LU decomposition algorithm. For a $b \times b$ block, each PE requires $2b$ storage, limiting the system's scalability. In this paper, we instantiate a linear array of PEs on each of the two available SLR regions on the FPGAs. These two linear arrays operate independently, providing task-level parallelism. All the computations in the Block LU decomposition Algorithm 2 are scheduled on the PE arrays. We avoid direct dot product computations in matrix multiplications by loop reordering. This eliminates the need

Algorithm 2: Block LU-decomposition algorithm. We use capital letters to denote block sub-matrices.

```
input : $A_{N\times N}$ matrix.
output: a lower triangular matrix $L_{N\times N}$ and an upper triangular matrix
        $U_{N\times N}$ where $A = LU$.
x‘ // $B$ is block size. $P$ is number of partitions.
$P = N/B$
for $k \leftarrow 1$ to $P$ do
    for $i \leftarrow k$ to $P$ do
        for $j \leftarrow k$ to $P$ do
            // Case 1
            if $i == k$ and $j == k$ then
                // Using Algorithm 1
                Compute $A_{kk} = L_{kk} \times U_{kk}$
            end
            // Case 2
            if $i == k$ and $j > k$ then
                $U_{kj} = L_{kk}^{-1} \times A_{ij}$
            end
            // Case 3
            if $i > k$ and $j == k$ then
                $L_{ik} = A_{ij} \times U_{kk}^{-1}$
            end
            // Case 4
            if $i > k$ and $j > k$ then
                $A_{ij} = A_{ij} - L_{ik} \times U_{kj}$
            end
        end
    end
end
```

for accumulation structures such as adder tree circuits. Further, we exploit problem structure to eliminate the need for explicit matrix inverse computations. All these provide for a scalable architecture with low resource utilization.

II. Architecture

As our hardware accelerator is based on the Block LU decomposition algorithm, we provide its details in the following section.

A. Block LU Decomposition

For the sake of simplicity and without loss of generality, consider a matrix $A_{N\times N}$ where $N = 3B$. The following is a block partitioned representation of the matrix where each entry A_{ij} is a $B \times B$ block sub-matrix of A. L_{ij} and U_{ij} denote lower and upper triangular $B \times B$ block sub-matrices respectively.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

The LU decomposition can be computed in three outermost iterations. The computations in the first outermost iteration corresponding to the *k-loop* from Algorithm 2 is summarized in the following steps.

1) **Case 1:** $A_{11} = L_{11}U_{11}$
2) **Case 2:** $U_{12} = L_{11}^{-1}A_{12}$ and $U_{13} = L_{11}^{-1}A_{13}$
3) **Case 3:** $L_{21} = A_{21}U_{11}^{-1}$ and $L_{31} = A_{31}U_{11}^{-1}$
4) **Case 4:** For $1 < i, j \leq 3$, $A_{ij} = A_{ij} - L_{i1} \times U_{1j}$

Algorithm 3: Algorithm for Case-2

```
input : Matrices $A_{B\times B}$ and $L_{B\times B}$.
output: $U_{B\times B} = L^{-1} \times A$
for $k \leftarrow 1$ to $B$ do
    par for $j \leftarrow k+1$ to $B$ do
        par for $l \leftarrow 1$ to $B$ do
            $a_{jl} = a_{jl} - l_{jk} \times a_{kl}$;
        end
    end
end
$U \leftarrow A$
```

Algorithm 4: Algorithm for Case-3

```
input : Matrices $A_{B\times B}$ and $U_{B\times B}$.
output: $L_{B\times B} = A \times U^{-1}$
for $k \leftarrow 1$ to $B$ do
    par for $j \leftarrow 1$ to $B$ do
        $l_{jk} = a_{jk}/u_{kk}$
        par for $l \leftarrow k+1$ to $B$ do
            $a_{jl} = a_{jl} - l_{jk} \times u_{kl}$;
        end
    end
end
```

Case 1 computation is performed using the LU decomposition method from Algorithm 1. Cases 2, 3, and 4 are handled by the Algorithms 3, 4, and 5, respectively. We leave proof of correctness for these algorithms involving basic linear algebra to the reader due to lack of space. Cases 2 and 3 involve matrix inverse and multiplication operations. Case 4 involves matrix multiplication and subtraction. It can be noticed that matrix inverses are not explicitly computed. Also, matrix multiplication is performed without using dot product formulation. We exploit the fact that the computational structure of Algorithms 1, 3, 4 and 5 is exactly the same. The innermost loop in these algorithms computes $\vec{v} = \vec{v_1} \pm C \cdot \vec{v_2}$ where $\vec{v}, \vec{v_1}, \vec{v_2}$ and C are vectors of same dimension.

B. Architecture

The hardware architecture consists of a linear array of B independent Processing Engines (PEs). Each PE contains a double-precision Multiply and ADD/SUB compute units, which are synthesized using the available FPGA DSP blocks. Further, each PE is connected to one or more independent

Algorithm 5: Algorithm for Case-4

```
input : Matrices $A_{B\times B}$, $U_{B\times B}$ and $L_{B\times B}$.
output: $A = A - L \times U$
for $k \leftarrow 1$ to $B$ do
    par for $j \leftarrow 1$ to $B$ do
        par for $l \leftarrow 1$ to $B$ do
            $a_{jl} = a_{jl} - l_{jk} \times u_{kl}$;
        end
    end
end
```

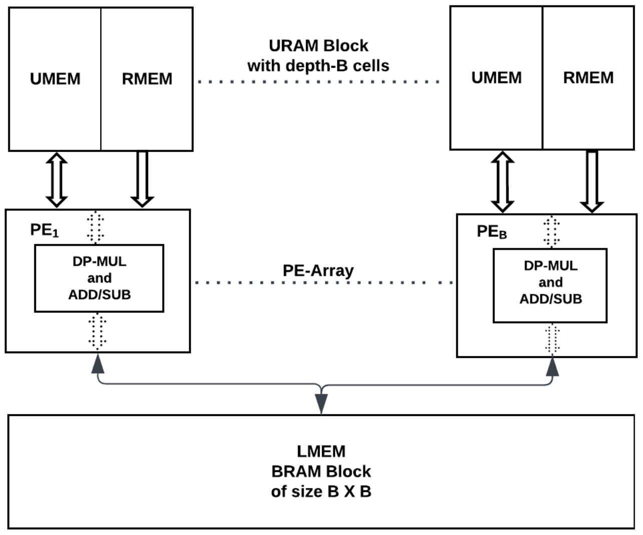

Fig. 1: Accelerator architecture consisting of Processing Engine (PE) array and distributed memory banks.

memory blocks. Thus the whole PE array can perform the core vector computation $\vec{v} = \vec{v_1} \pm C \cdot \vec{v_2}$ in parallel without any bottleneck, provided $|v| \leq B$, which will be the case in our application. Figure 1 shows the architecture of the PE array and Memory banks. The matrix to be decomposed is transferred from the CPU to the FPGA global High Bandwidth Memory (HBM). The matrix is then transferred from the global memory to the FPGA internal memory, made up of URAM and BRAM blocks, in units of $B \times B$ blocks to be processed by the PE array.

There are three logical memory units, namely UMEM, LMEM, and RMEM of size $B \times B$. UMEM and RMEM are split across B URAM banks, and each bank stores one column of the $B \times B$ data. URAM has a fixed width of 72-bits and supports dual-port configuration. Hence URAM is chosen for each bank of UMEM and RMEM. Each bank is directly connected to one unit of PE array. This facilitates concurrent multiplication and subtraction operations in every cycle of block processing. Since no parallel access to LMEM is required, it is mapped to BRAM blocks of size $B \times B$. The total internal memory required is $5 \times B \times B$ since UMEM and RMEM are double-buffered. Each PE consists of a double-precision Multiply and ADD/SUB compute unit instantiated from a pre-built Xilinx IP. The IP design is internally pipelined with a 12-cycle latency and an operating frequency of 250 MHz. Further, each IP consumes 11 DSPs and 1200 LUT registers. The DSP and LUT register resources available on the FPGA determine the maximum number of PEs and the block size.

As discussed previously, all the Algorithms 1, 3, 4 and 5 have the same computational structure. We now explain how the core computational steps in these algorithms of the form $\vec{v} = \vec{v_1} \pm C \cdot \vec{v_2}$ are scheduled on the accelerator. Each outermost iteration of the block calculates a pivot value using the first element of the respective pivot row. The pivot row is the first row containing its first non-zero element in the k^{th} column of the partially processed matrix. The pivot value is the reciprocal of the pivot element and is calculated inside the divider unit. The PE array has a single divider unit for pivot calculation. The pivot value calculation step is required only for Algorithm 1. The k^{th} column of the L values is calculated by multiplying the pivot element with the elements in the k^{th} column. The computed L values are stored in LMEM. L value calculation step is applicable only for Algorithms 1 and 4. After this, all the PEs work in unison to zero out the matrix's k^{th} column by processing one row at a time. The l^{th} PE reads in the pivot row's l^{th} element, multiplies it with the L value of the row and subtracts it from the l^{th} element of the row being updated. The processed rows are stored in UMEM, to be read in the next iteration. In the case of Algorithms 4 and 5, the pivot row is fetched from RMEM. In the, Algorithms 3 and 5 L values are fetched from the LMEM.

The orchestration of computations from the block LU decomposition Algorithm 2 onto the PE array is explained using the reference example from Section II-A. Initially, the block submatrix A_{11} is fetched from global HBM memory and stored in UMEM. Using the Agorithm 1, the factor matrices L_{11} and U_{11} are computed and stored in memory units LMEM and UMEM respectively. Further, the reciprocal of diagonal elements of A_{11} are computed and stored in the register-bank RECMEM of size B. Subsequently A_{12} is fetched inside and stored in UMEM and U_{12} is computed based on L_{11} already stored in LMEM. U_{12} computation follows Algorithm 3. Similarly U_{13} is computed. Now A_{21} is fetched internally and stored in UMEM and U_{11} is read to RMEM. L_{21} computation follows Algorithm 4. It is computed based on A_{21}, U_{11} and contents of RECMEM. The computed values of L_{21} are stored in LMEM and updated in global memory. In the next step A_{22} is read to UMEM and U_{12} to RMEM. The new value of A_{22} is computed again using the Algorithm 5 and updated in the global memory. Similarly the new value of A_{23} is updated. In similar lines L_{31}, A_{32} and A_{33} are updated. This will complete the outermost iteration 1 of Block LU decomposition. In outermost iteration 2, U_{22}, L_{22}, U_{23}, L_{32} and new value of A_{33} are computed. In outermost iteration 3 U_{33} and L_{33} are computed. UMEM and RMEM are double-buffered so that the next block is fetched while the PE is busy processing the current block.

Our architecture exploits multiple Super Logic Regions (SLRs) available within a single FPGA device. The computational unit consisting of PE array and memory blocks are duplicated in both the SLRs available. As the contents of LMEM block is common for all the blocks within a row, L matrix calculation of the first block in every row is processed in both the SLRs. The remaining blocks available within a row are uniformly distributed for processing across the two SLRs. This will reduce the latency because of the concurrent execution with minimal duplication penalty.

III. Experimental Results

We conducted our experiments using an Alveo U50 data center acceleration card connected to a host Xeon CPU through a PCIe 3.0 x16 interface. The XCU50 FPGA on U50 card comprises two SLRs. Each SLR region contains 2976 DSP units, 672 BRAM blocks, 320 URAM (288kb) blocks, and 436k LUTs. The proposed hardware accelerator is synthesized at 250 MHz with 128 PEs per SLR using Vitis-HLS version 2021.1. The number of PEs per SLR is restricted due to the limitation of available CLBs, even though more DSPs are available in FPGA. Even numbered blocks of a row are processed in SLR0, and odd-numbered blocks of a row are processed in SLR1. Our accelerator sustains an average throughput of 98 GFLOPs with a peak throughput of 128 GFLOPs.

We compare the performance of our accelerator with a four-core (8 threads) Intel Xeon E2224G CPU running at 3.5 GHz and an Nvidia RTX 8000 GPU with 4608 CUDA operating cores and a peak performance of 16.3 TFLOPs. The CPU benchmarking is done using the LU function from the Intel MKL library [2], and GPU is bench-marked using the PyTorch library [4]. Table I shows that the proposed FPGA design outperforms the Intel MKL CPU's library. The latency improvement is more than 20X over the CPU for most matrix sizes. For smaller matrix sizes, our accelerator's operational latency is lower than that of the GPU. For larger matrix sizes, the GPU latency is 2 to 3 times lower than FPGA latency. Since we operate on 128×128 matrices using a PE array of size 128, in the ideal case, each outermost iteration should be completed in 128 clock cycles. However, due to pipe-lining overheads and a division operation involving pivot computation, our implementation takes 145 clock cycles. The latency for all iterations is thus $145 \times (128 - 1) = 18415$ cycles. The overall latency for complete LU-decomposition of a matrix is thus dependent on the number of blocks per matrix. As we can see, latency is very deterministic because of application-specific hardware implementation in FPGA. In contrast, latency can depend on many other factors in CPU and GPU implementations. The sustained GFLOPS performance for FPGA implementation is computed based on the total number of Floating-Point MAC operations for an $N \times N$ matrix LU decomposition divided by the latency from the experimental results. A similar approach calculates sustained GFLOPs performance for the CPU and GPU. Table II compares the performance of all three platforms. It can be noticed that the GPU implementations outperform our FPGA design on large matrix sizes, and our FPGA design outperforms CPUs.

TABLE I: Table comparing the latencies of block LU-decomposition using our FPGA design, CPU, and GPU. (ms) stands for milliseconds and (s) stands for seconds.

Matrix Size N	**Proposed FPGA design latency**	**Intel MKL Multi-core Xeon CPU latency**	**Torch.lu Multi-core GPU latency**	**FPGA over CPU (x times)**
256	0.21 ms	50 ms	0.93 ms	234.74
512	1.25 ms	80 ms	1.00 ms	61.58
1024	8.80 ms	120 ms	12.92 ms	13.31
2048	0.06 s	1.71 s	0.04 s	27.52
4096	0.48 s	13.82 s	0.18 s	28.68
8192	3.81 s	96.34 s	1.18 s	25.29
16384	29.4 s	695.7 s	11.7 s	23.67

TABLE II: Sustained GFLOPS performance comparison for FPGA, CPU and GPU.

Matrix Size	**Proposed FPGA Design Performance (GFLOPS)**	**Torch.lu Multi-core GPU Performance (GFLOPS)**	**Intel MKL Multi-core Xeon CPU (GFLOPS)**
256	51.59	11.19	2.34
512	70.96	83.19	3.48
1024	80.99	51.61	18.33
2048	91.87	123.92	10.03
4096	94.94	232.95	9.94
8192	96.14	288.77	12.1
16384	100	231.91	16.3

IV. Conclusion

In this work, we revisit the well-known block LU-decomposition algorithm and propose alternative ways to carry out the complex intermediate steps using a homogeneous computational structure. This facilitated the orchestration of all the computations on a hardware accelerator consisting of a Processing Engine array and a distributed collection of memory banks. This enables to processes matrices of arbitrary dimension.

References

[1] Gokul Govindu and Vikash Daga and Sridhar Gangadharpalli and V. Sridhar and Viktor K. Prasanna and . Efficient Floating-point Based Block LU Decomposition on FPGAs. In Toomas P. Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA'04, June 21-24, 2004, Las Vegas, Nevada, USA*, pages 276–279. CSREA Press, 2004.

[2] Intel. Developer Reference for Intel® oneAPI Math Kernel Library - C. https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html, 2022.

[3] Manish Kumar Jaiswal and Nitin Chandrachoodan. FPGA-Based High-Performance and Scalable Block LU Decomposition Architecture. *IEEE Transactions on Computers*, 61(1):60–72, 2012.

[4] Pytorch. Torch.lu Library Documentation. https://pytorch.org/docs/stable/generated/torch.lu.html, 2022.

[5] Wu, Guiming and Xie, Xianghui and Dou, Yong and Sun, Junqing and Wu, Dong and Li, Yuan. Parallelizing sparse LU decomposition on FPGAs. In *2012 International Conference on Field-Programmable Technology*, pages 352–359, 2012.

[6] Xilinx. Vitis Library Solver. https://xilinx.github.io/Vitis_Libraries/solver/2022.1/guide_L2/L2_api.html#getrf-nopivot, 2022.