Accelerating LU-Decomposition of Arbitrarily Sized Matrices on FPGAs

Thesis submitted in partial fulfilment
of the requirements for the degree of

Master of Science in
in
Electronics and Communication Engineering by Research

by

Krishna Kumar Maram
2018122003
krishna.maram@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
November 2023

Copyright © Krishna Kumar Maram, 2023
All Rights Reserved

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "*Accelerating LU-Decomposition of Arbitrarily Sized Matrices on FPGAs*" by *Krishna Kumar Maram*, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Adviser: Prof. Suresh Purini

This is dedicated to my Parents, Family and Friends

Acknowledgments

The work done in this thesis, has been a collective effort. It is the result of persistence, hard work and constant support from people around me. I would like to thank my advisor Dr.Suresh Purini for his help and guidance. He had allowed to do the project at my own pace. I would like to thank my advisor Dr. Suresh Purini for his invaluable guidance, support. I had learnt a lot from him, during these years. He had always believed in my capabilities. He was patient enough to allow me to learn and grow at my own pace. He gave me the freedom to pursue a problem, even if there wasn't much progress and patiently waited till excellent results came out. I would like to thank my parents for providing me positive environment both physically and mentally, especially during Covid-19 lockdown, to complete this project. I thank my PhD mentor Ziaul Choudry for inspiring me to pursue this project, and giving me a road-map to complete. I want to thank my seniors, Shashwat Shrivastava, and Shashwat Khandalwel for inspiring, guiding and supporting me, so that I could follow their footsteps. I thank my friends Rajashekar Reddy, Sai Siddharth, Abhinav Navnit, Krishna Charan, Shivani Chepuri, Kadiyala Abhiram, Susheel Voora, Sasi Kiran for giving me a memorable campus life.

Abstract

Scientific applications like aircraft-design and machine-learning algorithms involve solving a system of linear equations. LU decomposition is helpful if the system needs to be solved repeatedly for the same Coefficient matrix but for different Constants of a Linear Equation. It also aids in other matrix operations, such as computing the determinant and inverse of a matrix. This thesis proposes an architecture of 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

LU decomposition is a powerful mathematical tool with a wide range of real-world applications. Some of the most common applications include:

- **Solving linear systems of equations.** This is perhaps the most well-known application of LU decomposition. LU decomposition can be used to solve any system of linear equations, regardless of whether the coefficients are integers, real numbers, or complex numbers.
- **Calculating matrix inverses.** The inverse of a matrix is a matrix that, when multiplied by the original matrix, produces the identity matrix. LU decomposition can be used to efficiently calculate the inverse of a matrix.
- **Calculating determinants** The determinant of a matrix is a number that summarizes the properties of the matrix. LU decomposition can be used to efficiently calculate the determinant of a matrix.
- **Solving differential equations** LU decomposition can be used to solve differential equations numerically. This is a powerful tool for simulating physical systems and engineering applications.
- **Image processing.** LU decomposition can be used to perform a variety of image processing tasks, such as image restoration, image compression, and image segmentation.

Here are some specific examples of how LU decomposition is used in real-life applications:

- **Engineering:** LU decomposition is used to analyze the structural stability of buildings and bridges. For example, engineers use LU decomposition to solve the equations that describe the forces acting on a structure, such as a bridge or a building. This information can then be used to determine whether the structure is safe and to identify any potential weaknesses.

- **Physics:** LU decomposition is used to simulate the behaviour of fluids and solids. For example, physicists use LU decomposition to simulate the flow of fluids around an airplane wing or to model the behaviour of a crystal. This information can then be used to understand the physical properties of these systems and to design new products.
- **Chemistry:** LU decomposition is used to calculate the properties of molecules and materials. For example, chemists use LU decomposition to calculate the energy levels of atoms or the structure of molecules. This information can then be used to design new drugs or to understand the properties of materials.
- **Biology:** LU decomposition is used to analyze the genetic code and to model the spread of diseases. For example, biologists use LU decomposition to analyze the DNA sequence of a virus or to model the spread of a disease through a population. This information can then be used to develop new treatments for diseases or to prevent the spread of diseases.
- **Finance:** LU decomposition is used to price derivatives and to calculate the risk of financial portfolios. For example, financial analysts use LU decomposition to price options or to calculate the risk of a portfolio of stocks. This information can then be used to make informed investment decisions.
- **Computer Science:** LU decomposition is used to implement a variety of algorithms, such as the Gaussian elimination method and the Cholesky decomposition method. For example, computer scientists use LU decomposition to implement the Gaussian elimination method, which is a common algorithm for solving systems of linear equations. This algorithm can then be used to solve a variety of problems, such as image processing and machine learning.

The thesis proposed focuses on dense matrix LU decomposition against sparse matrices that require a different approach [4]. We compare our work against the relevant work on dense matrices. The IP for LU decomposition in Xilinx Vitis library [5] requires the maximum matrix dimension to be specified at the synthesis time. Once synthesized, it cannot process matrices beyond that dimension. Jaiswal et al. [6] 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. [7] 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 submatrices that arise in the 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 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.

1.2 Summary of Contributions

The main contributions of this thesis are explained in Chapter - 5 in detail.
In this work, we contribute to accelerating LU decomposition on FPGAs.

- We propose a simplified architecture with parallel PEs coupled with an efficient multi-banked memory organization.
- Unlike other works, our simple PE structure is able to process different types of operations in the block LU decomposition algorithm.
- We employ efficient double buffering techniques to cut down on the block access latency. Independent controllers to fetch block data from HBM, process the block and write the result block data back to HBM.
- We implement our design on the latest Alveo board using Vitis-HLS and provide it as an off-the-shelf implementation. The algorithm is scalable to multiple SLRs available in the latest FPGAs to reduce the latency.
- We report highest *Giga*-Floating Point Operations Per Second (GFLOPS) performance for LU decomposition in FPGA.

1.3 Thesis Organization

The thesis further divided into following Chapters:

- Chapter 2 : This Chapter 2, explains LU-Decomposition, explains the Gaussian Elimination method algorithm. Explains and proves Block LU-Decomposition method and gives the Crout's method of LU-Decomposition. It also discusses relevant work done till now in FPGA acceleration of LU-Decomposition.
- Chapter 3 : This Chapter 3 explains the Design Space Exploration done for acceleration of LU-Decomposition. Based on the Design-Space Exploration done, architecture for LU-Decomposition is proposed. This chapter explains the simulations for analysis of design space exploration. It explains the proposed architecture in detail.

- Chapter 5 : This Chapter 5 explains the Design Space Exploration done for acceleration of LU-Decomposition. Based on the Design-Space Exploration done, architecture for LU-Decomposition is proposed. This chapter explains the simulations for analysis of design space exploration. It explains the proposed architecture in detail.
- Chapter 6 : This Chapter 6 explains the experimental setup. Gives the acceleration platform details. It explains the experimental results.It compares the performance, resource consumption and energy efficiency of the design with CPU,GPU and benchmark FPGA design.
- Chapter 7 : This Chapter 7, concludes the thesis and discusses the main contributions of this paper. It ends by discussing the future scope of this problem.
- ***Bibliography*** contains all the referenced papers used in this thesis.

Chapter 2

HPC, FPGA Acceleration and Design Flow

High-performance computing (HPC) is the field of solving complex scientific, engineering and financial problems by applying highly parallel computing systems. Efficient parallel execution of Processing Elements (PEs) to accelerate the complex algorithms is the core part of the computer architecture involved in the design of HPC systems.

Traditionally current generation microprocessor is the processing element within the limitations of Von-Nuemann computer architecture. For the last ten years, HPC systems have had three choices for implementation, viz., CPU, GPU and FPGA. Combinations of CPU and GPU, CPU with FPGA acceleration is also becoming a popular combination. The selection of choice depends on the particular application.

FPGAs are composed of a large array of configurable logic blocks (CLBs), Digital Signal Processing blocks (DSPs), Block RAM (BRAM), and Input/Output Blocks (IOBs). CLBs and DSPs, similar to a processor's Arithmetic Logic Unit (ALU), can be programmed to perform Arithmetic and Logic operations like Add, Multiply, Subtract, Compare and so on. Unlike a processor, in which the architecture of the ALU is fixed and designed in a general-purpose manner to execute various operations, the CLBs can be programmed with just the operations needed by the application. This results in increased computing efficiency.[8]

The FPGA architecture provides the flexibility to create a massive array of application-specific ALUs that enable both instruction and data-level parallelism. Because data flows between operators, there are no inefficiencies like processor cache misses; FPGA data can be streamed between operators. These operators can be configured to have point-to-point dedicated interconnects, thereby efficiently pipelining the execution of operators.[8]

The parallelism offered by FPGA architecture can be easily seen in a few examples of HPC-relevant parameters[8]:

- Internal bandwidth to move the operands and results of application-specific ALUs are in order of terabytes/sec (TB/s)

- Throughput on integer operations are in the order of Tera-operations/sec (TOPS)

- Throughput on floating point operations are in the order of gigaflops/sec (GFLOPS)

Over the years, FPGAs have gone through significant improvements. The initial days of FPGA consisted of a few hundred to thousands of programmable logic elements called configurable logic blocks (CLBs) and Block memory, which were used to implement glue logic in an electronic system. With the advances in semiconductor manufacturing, FPGAs also continuously offer larger capacity with high-speed interfaces such as PCI/PCIE and GBPS ethernet. FPGAs are also available with integrated multi-core RISC processors with high-speed DDR interface. Integrated High Memory Bandwidth memories, on-chip data converters, and high-speed URAMs are a few latest additions to FPGAs. New FPGAs are emerging with HBMs integrated into the same package. The floating point performance of the FPGAs also is enormously increasing with the increased number of DSPs available within FPGAs.

FPGAs tend to consume power in tens of watts, compared to other multicores and GPUs that tend to consume power in hundreds of watts. One primary reason for lower power consumption in FPGAs is that the applications typically operate between 100–300 MHz on FPGAs compared to applications on high-performance processors executing between 2–3 GHz[8].

FPGAs have unique advantages over traditional computing platforms such as CPU and GPU in High-Performance Computing, such as low latency, high energy efficiency, and high parallelism, making them a promising option for high-performance compute acceleration of certain algorithms which are compute-intensive [9]. For molecular modelling, FPGA dedicated communication structures, low-latency communication threads, and flexible algorithmic restructuring make it a preferred platform for modelling the iterative Newtonian interactions of atoms and molecules [10]. The financial world has made huge investments in FPGA technology for high-frequency trading applications specifically to take advantage of an FPGA's ability to handle electronic trade data with very low latency and minimal jitter.

Despite the availability of FPGA for more than four decades in the market. FPGA design requires specialized skills and different development environments, which typically fall outside the HPC developer environment. Due to this, a typical HPC developer prefers the CPU and GPU over the FPGA. Even then, the benefits of using the FPGA are quite attractive and developmental effort is worth investment for quite a few applications. Specific applications take advantage of fine-tuned architectures possible with FPGA-based Hardware implementation. For example, bioinformatics algorithms such Smith-Waterman, commonly used for DNA sequencing, is compatible with FPGAs spatial and temporal parallelism [11]. FPGAs superior energy efficiency makes it a competitive choice for implementing Linear algebra algorithms, a key library for scientific applications.

The low operating frequency of an FPGA clock compared to a high-end microprocessor is one of the limitations of the FPGA in an HPC environment. Implementation quality plays a substantial role in achieving 98% of the targeted applications. Even though FPGA accelerates the portion of the task, unless acceleration is preferable $50\times$, the user-acceptable speed up for the application cannot be achieved. Amdhal's Law demonstrates the limitation of accelerating the portion of the task. The advantage gained by FPGA speedup is shadowed by exorbitant development effort, lack of flexibility, portability and maintainability. Though the parallelisation using MPPs is a difficult task, the required expertise and

understanding are significantly high in the HPC development community. In an FPGA environment, it is efficient to implement algorithms involving one input and one output while saving several intermediary values in On-Chip registers. Due to the presence of dual-ported Block RAMs (BRAMs), and high flexibility of memory access, the memory bottleneck that may be present is eliminated.

2.1 High-level synthesis for FPGA acceleration

As discussed in the previous sections, one of the main difficulties in adopting FPGA in an HPC environment is the specialized skills required for FPGA implementations. Traditionally, Verilog HDL or VHDL are the languages adopted for modelling the algorithm for FPGA. Subsequently, FPGA synthesis and implementation tools are used to realize the hardware. In Verilog or VHDL, language is designed to easily model the concurrency. In the recent past, high-level synthesis concepts have evolved, where the algorithm is modelled using C/C++, and compiler pragmas are introduced to adopt concurrency in C. Bluespec verilog and High-level Synthesis (HLS) C from Xilinx are few choices available for this. In this section, some salient features and important pragmas typically used in Vitis HLS and Bluespec are discussed.

2.1.1 HLS-C

2.1.1.1 Introduction to Vitis HLS C

Vitis High-Level Synthesis (HLS) C is a powerful tool for designing and implementing hardware accelerators using high-level languages such as C or C++. It enables software engineers to describe algorithms and functionalities in a high-level language and automatically convert them into optimized hardware designs.

Vitis HLS C takes advantage of the inherent parallelism in algorithms by automatically identifying and exploiting parallel execution opportunities. By using pragmas and directives, software developers can guide the synthesis process and achieve the desired hardware implementation.

One of the key benefits of Vitis HLS C is its ability to abstract low-level hardware details, enabling software engineers to focus on algorithmic development rather than hardware-specific implementation. The tool generates RTL (Register Transfer Level) code that can be synthesized and targeted to a specific FPGA platform using the Xilinx Vivado Design Suite.

2.1.1.2 Key Features and Benefits of Vitis HLS C

Vitis HLS C offers several features and benefits that make it a popular choice for hardware acceleration:

Productivity: Vitis HLS C allows software engineers to leverage their existing skills and knowledge in high-level languages such as C/C++. This enables a higher level of productivity as developers can quickly prototype and iterate on designs, significantly reducing development time.

Portability: The high-level abstraction in Vitis HLS C enables designs to be portable across different FPGA platforms. By separating the algorithmic description from the hardware-specific details, the same high-level code can be retargeted to different FPGA devices without significant modifications.

Optimization: Vitis HLS C provides a wide range of optimization techniques to improve the performance and resource utilization of the generated hardware designs. Through pragmas and directives, developers can control loop unrolling, pipeline insertion, resource allocation, and array partitioning, among other optimizations.

Verification and Debugging: Vitis HLS C offers robust simulation and debugging capabilities to ensure functional correctness and facilitate design verification. Developers can use software-based simulations to validate the algorithmic behaviour before synthesizing the design for FPGA implementation.

System-Level Design Integration: Vitis HLS C seamlessly integrates with the Xilinx Vitis Unified Software Platform, allowing hardware accelerators to be seamlessly integrated into system-level designs. This integration enables co-design and co-optimization of hardware and software components, resulting in highly efficient and integrated systems.

In conclusion, Vitis HLS C provides software developers with a powerful toolset for designing and implementing hardware accelerators using high-level languages. By abstracting low-level hardware details and leveraging parallelism, Vitis HLS C enables rapid prototyping, optimization, and integration of hardware accelerators into larger systems. With its range of features and benefits, Vitis HLS C plays a crucial role in accelerating the development of FPGA-based applications in various domains.

High-Level Synthesis (HLS) is an automated design process that takes an abstract behavioural specification of a digital system and generates a register-transfer level structure that realizes the given behavior[12].

A typical flow using HLS has the following steps:

1. Write the algorithm at a high abstraction level using C/C++/SystemC with a given architecture in mind
2. Verify the functionality at the behavioural level
3. Use the HLS tool to generate the RTL for a given clock speed, input constraints
4. Verify the functionality of the generated RTL
5. Explore different architectures using the same input source code

HLS can enable the path of creating high-quality RTL rather quickly than manually writing error-free RTL. Vitis HLS can be used to develop and export Vivado IP to be integrated into hardware designs using the Vivado Design Suite.

Here are some key areas related to coding and synthesizing the C++ functions in HLS design:

- Hardware Interfaces:
  The arguments of the top-level function in a Vitis HLS design are synthesized into interfaces and ports that group multiple signals to define the communication protocol between the HLS design and components external to the design. Vitis HLS defines interfaces automatically, using industry standards to specify the protocol.

- Controlling the of HLS Design:
  The execution mode of an HLS design is specified by the block-level control protocol. The HLS design can have control signals to start/stop the execution, or it can be only driven when the data is available. [12]

- Task-Level Parallelism:

  To achieve high performance on the generated hardware, the HLS tool must infer parallelism from sequential code and exploit it to achieve greater performance. [12]

- Memory Architecture:

  The memory architecture is fixed in the CPU, but the developer can create their own architecture to optimize the memory accesses for running applications on FPGA. In C++ program, the arrays are fundamental data structures used to save or move the data around. In hardware, these arrays are implemented as memory or registers after synthesis. The memory can be implemented as local storage or global memory which is often DDR or HBM memory banks. Access to global memory has higher latency costs and can take many cycles, while access to local memory is often quick and only takes one or more cycles. [12].

In Vitis High-Level Synthesis (HLS-C), pragmas are used to provide additional directives or instructions to the HLS tool to guide the synthesis process and achieve desired hardware implementations. Pragmas in HLS-C are similar to pragmas in C/C++ programming languages but with specific meanings and effects in the context of high-level synthesis.

Here are some commonly used pragmas in Vitis HLS-C:

1. HLS INTERFACE: This pragma is used to specify the interface properties of a function or module. It defines the ports and their characteristics, such as the data type, direction (input or output), and connection type (stream, array, or scalar).

2. HLS DATAFLOW: This pragma is used to enable the dataflow-style synthesis, where operations are pipelined and scheduled based on data dependencies rather than control flow. It allows the synthesis tool to explore parallelism and maximize performance.

3. HLS PIPELINE: This pragma is used to explicitly specify pipeline registers in the design. It instructs the tool to insert pipeline registers to increase the clock frequency and allow for better performance by exploiting parallelism.

4. HLS UNROLL: This pragma is used to unroll loops in the code. Loop unrolling duplicates loop iterations to expose more parallelism and reduce loop overhead. It can help improve performance but may increase resource utilization.

5. HLS RESOURCE: This pragma is used to control resource allocation and mapping. It allows specifying constraints on specific resources, such as DSP48 slices, BRAM, or LUTs, to guide the synthesis tool in optimizing the hardware implementation.

6. HLS INLINE: This pragma is used to force the inlining of functions or modules. Inlining eliminates the function call overhead by directly incorporating the called function's code at the call site. It can improve performance but may increase resource utilization.

7. HLS ARRAY PARTITION: This pragma is used to partition arrays into smaller partitions to enable parallelism. It allows specifying the partition factor and dimension(s) of the array to control how the data is distributed across memory banks or processing elements.

These are just a few examples of the pragmas commonly used in Vitis HLS-C. There are additional pragmas available that provide more fine-grained control over the synthesis process. The specific pragmas and their usage depend on the requirements of the design and the optimization goals.

Chapter 3

LU-Decomposition and Block LU-Decomposition

3.1 LU-Decomposition

LU decomposition is a method used to solve systems of linear equations. Consider a system of linear equations $A\mathbf{x} = \mathbf{b}$, where **A** is an $m \times n$ invertible matrix, **x** and **b** are of size **n** $\times$ 1. To obtain **x**, the below given augmented matrix is row-reduced until $[c_1, c_2, c_3, ..., c_n]$, is in an echelon form (the "forward" part of the row reduction process) and then use "back substitution" to complete the solution. A computer might proceed this way.

$[a_1, a_2, a_3, ..., a_n|b] \sim ... \sim [c_1, c_2, c_3, ..., c_n|d]$

Or continue row-reduction (the backward part of the row-reduction process) until the row-reduced echelon form arrives, from which the solutions can be read. Either way, in some applications, it is necessary to solve the system $A\mathbf{x} = \mathbf{b}$ many times (perhaps hundreds or thousands of times), always keeping **A** the same but changing **b**. Moreover, in these applications, **A** may also be a very large matrix. In such a situation, row reducing $[a_1a_2...a_i...a_n|b]$ each time is too inefficient, even with a computer doing the work. One way to avoid these repeated row reductions is to try to factor A

$A = LU$ where U is an echelon form of A and L is a square lower triangle matrix with 1's on the diagonal.
(U might not be square, but it only has 0 entries below leading entries in each row, so U resembles an Upper triangular matrix.)

A factorization $A = LU$ cannot always be found but if it can, the work involved does not take many more steps than it takes to solve $Ax = b$, just once by row reduction. And if we get this factorization $A = LU$, then it takes relatively few additional steps to find x each time a new b is chosen.
More specifically, this is an improvement because:

- Solving $Ly = b$, for y is simple since L is lower triangular.

- After finding y, it is easy to solve $Ux = y$ because U is in echelon form.
- Finally, if we change b, L and U remain the same.

3.1.1 Algorithm for LU-decomposition

This work is only the acceleration of only square-shaped matrices. Below is the pseudo-code of the Gauss-Jordan Elimination method to decompose a square matrix into a Lower-Triangular and Upper-Triangular Matrix.

Algorithm 1 Gauss-Jordan Elimination Method

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

Upon looking at Algorithm 1 it is clear that it has a substantial amount of parallelism available for each of the steps inside Loop1, Loop2, and Loop3. But its implementation requires storing the entire input matrix in the internal memory in order to get the best performance. Therefore, the available on-chip Memory is a significant limitation for processing large or arbitrary-sized matrices. 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 work, 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.

3.1.2 Other Works on LU-Decomposition

3.1.2.1 Kumar etal.,2012 [1]

In their paper Investigation of the performance of LU decomposition method using CUDA, Kumar and Singhal (2012) investigated the performance of LU decomposition on CPUs and GPUs using the CUDA programming platform. They implemented two different algorithms for LU decomposition on the GPU: a block-based algorithm and a row-based algorithm. The block-based algorithm divides the matrix into blocks of a fixed size and processes each block in parallel. The row-based algorithm divides the matrix into rows and processes each row in parallel.

Kumar and Singhal compared the performance of the CPU and GPU implementations of LU decomposition on a set of synthetic and real-world matrices. They found that the GPU implementations of LU decomposition were significantly faster than the CPU implementation, especially for large matrices.

On GPU they got, $3.5\times$ speedup over CPU.

3.1.2.2 Petrov etal.,2015 [2]

The paper "A Performance Comparison of Computing LU Decomposition of Matrices on the CPU and the GPU" by Petrov and Marinov (2015) compares the performance of different implementations of LU decomposition on the CPU and the GPU. The authors consider five implementations:

1. Eigen on the CPU
2. Intel MKL on the CPU
3. MATLAB on the CPU
4. MATLAB's Parallel Computing Toolbox on the GPU
5. CUDA/CULA on the GPU

The authors find that the CUDA/CULA GPU implementation is the fastest among all considered CPU and GPU implementations, with the exception of the largest considered matrices, which could not be computed on the GPU, due to the lack of memory. The GPU got a speed-up of $16\times$ over CPU.

3.1.2.3 Ino etal.,2004 [3]

The paper "Performance Study of LU Decomposition on the Programmable GPU" by Ino and Takeda (2004) presents a performance study of LU decomposition on the programmable GPU (graphics processing unit). The authors compare the performance of their GPU implementation to that of a CPU implementation on a variety of matrices.

The authors' GPU implementation of LU decomposition is based on the following steps:

1. The matrix is partitioned into blocks.
2. Each block is decomposed using a standard LU decomposition algorithm.
3. The blocks are recombined to form the LU decomposed matrix.

The authors' GPU implementation uses a variety of techniques to improve performance, including:

- **Parallelism:** The GPU can process multiple threads in parallel, which allows the authors to parallelize the LU decomposition algorithm.
- **Shared memory:** The GPU has shared memory that can be used to store intermediate results, which reduces the need to access global memory.
- **Coalesced memory access**: The GPU can access memory more efficiently when the accesses are coalesced. The authors use a variety of techniques to coalesce memory accesses in their implementation.

The authors evaluate the performance of their GPU implementation on a variety of matrices, including dense matrices, sparse matrices, and banded matrices. They find that their GPU implementation is significantly faster than the CPU implementation on all types of matrices.

Here are some additional details about the performance of the authors' GPU implementation:

- On a dense matrix of size 1024x1024, the authors' GPU implementation is 2.4 times faster than the CPU implementation.
- On a sparse matrix of size 1024x1024, the authors' GPU implementation is 4.7 times faster than the CPU implementation.
- On a banded matrix of size 1024x1024, the authors' GPU implementation is 5.3 times faster than the CPU implementation.

The authors conclude that their GPU implementation of LU decomposition is a viable alternative to CPU implementations for a variety of applications.

3.2 Block LU Decomposition

The Block LU-Decomposition is an extension of the LU-Decomposition technique for matrices, where the original matrix is decomposed into the product of two triangular matrices: One Lower Triangular Matrix (L) and another Upper Triangular Matrix (U). The Block LU-Decomposition extends this idea to decompose a large matrix into smaller blocks.The Block LU-Decomposition is particularly useful when dealing with large matrices that exhibit a block structure. By decomposing the matrix into smaller blocks, we can efficiently perform various matrix operations and solve linear systems.

Suppose we have a square matrix A of size N × N, and we want to decompose it into lower triangular blocks (L) and upper triangular blocks (U) of smaller sizes. We assume that each block is also a square matrix.

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. Algorithm 2 describes the Block LU decomposition method of solving the algorithm 1. k and i loops represents outermost iterations in Block LU decomposition. The computations in the first outermost iteration corresponding to the *k-loop and j-loop* from Algorithm 2 is summarized in the following steps.There are four different cases for processing of the block depending on the index of k and i. These steps in block LU decomposion can be proven by induction method. Below are the steps when number of blocks is equal to 3. The same steps can be extended to decompose matrix in any number of blocks of size B.

Case 1: $A_{11} = L_{11}U_{11}$

Case 2: $U_{12} = L_{11}^{-1}A_{12}$ and $U_{13} = L_{11}^{-1}A_{13}$

Case 3: $L_{21} = A_{21}U_{11}^{-1}$ and $L_{31} = A_{31}U_{11}^{-1}$

Case 4: For $1 < i, j \leq 3$, $A_{ij} = A_{ij} - L_{i1} \times U_{1j}$

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 a 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

Algorithm 3 shows the algorithm for decomposition of A_{12}

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 shows the algorithm for decomposition of A_{21}

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

Algorithm 4 shows the algorithm for decomposition of A_{22}

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

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. 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.

Chapter 4

FPGA Acceleration of LU-Decomposition - Literature survey

In this chapter, a survey of the various implementation reported in literature for FPGA acceleration of LU-Decomposition. Literature selected was specific to LU-Decomposition for dense matrices.

1. Work done by Viktor. K Prassana et al. [7]

   This work focused on implementing the block LU decomposition algorithms on FPGA. Two sets of Processing Engines (PE)s are proposed to implement the various types of blocks of block LU decomposition. One set of PEs for implementing algorithms 1,3 and 4 and another set of PEs for matrix multiplication as in algorithm 5 and a single PE for matrix subtraction. Operations involved in algorithms 1,3, 4 and 5 are identified as OPLU, OPL, OPU and OPMMS respectively. The implementation proposed stacked matrices for storing the $b \times b$ sized sub-matrices of the input matrix. The number of stacked matrices, s is more than the combined latency of the floating-point compute elements in the PEs, in order to resolve the data dependencies. Stacked zero matrices are proposed in cases where data dependencies could not be resolved. In this architecture, pipelining depth of the floating-point units decides the number of zero matrix stacking and the limit of bxb matrix stack size s. An FSM implemented the overall scheduling between the stacked matrices and two sets of PEs for various operations.

   This work also proposes a formula to assess the latency in terms of cycles based on block size b and stack size s. A qualitative proof for the proposed formula is also provided. The domain-specific modelling approach is adopted to estimate the candidate performance in terms of energy, latency and area for various block sizes. The area is based on the actual implementation, latency and power estimations were obtained from the tools and from low-level simulations.

   The technology used for low-level simulation results, the component designs were coded in VHDL, implemented using Xilinx ISE 5.2i on the Xilinx Virtex-IIPro XC2VP125 device. The areas were obtained from the PAR report and power was obtained using ModelSim 5.7 and XPower. Maximum matrix size was restricted to 1000 x 1000 and matrix block size was limited to 16 due to the limitations in selected FPGA. The performance of single block LU decomposition and depth

of pipelining were analyzed for various block sizes. The analysis infers that a smaller block size and smaller stack size are energy efficient, due to zero matrix stacking effect. matrix LU decomposition results were compared with the implementation in the Pentium processor and reported a $23\times$ improvement for a 1000×1000 matrix.

2. Work done by Manish Kumar et al. [6]

   This work focused on Block LU Decomposition implementation on FPGA and looks into the scalability of the implementation across multiple FPGAs for large size of matrices. The Block LU decomposition algorithm implemented is the same as the one reported in the previous chapter. Architecture proposes different block sizes for different algorithms of block LU decomposition. In algorithm-2, for case-1 , $b \times b$ size matrix, for case-2, $b \times (n-b)$, for case-3 $(n-b) \times b$ size and for case 4, $(n-b) \times (n-b)$ sizes are proposed. For blocks of algorithms 3,4 and 5, computations are handled by loading the segment of matrix size M, which is larger than b, but smaller than n. A double-buffered approach was used to address the memory latency. A separate compute module is used for the $b \times b$ matrix of case-1, and a separate compute module to calculate the inverse matrix required for case-2 and case-3. A matrix multiplication compute element is used for matrix multiplication required in case-2, case-3, and case-4. A subtraction unit is additionally used for case-4. Overall 4 different types of compute units were proposed to implement the Block LU decomposition. Work claims to restrict the block size to 16, due to the restrictions in the computation of inverse matrix calculation. The proposal also didn't give the details of the inverse matrix computation module. The M size is fixed to 256 based on the size of the block ram which can accommodate the number of double-precision floating-point elements.

   The design was implemented in Verilog language and targeted to the Vertex-5 device. The work claim to achieve 250 MHz in post-Place and Route for Vertex-5 FPGA. This work also further made an analysis of the scalability of this architecture to multiple FPGAs to reduce the latency for large-size matrix blocks. The authors studied one-dimensional scaling and two-dimensional scaling approaches for scaling across multiple FPGAs and proposed the strategy for implementation in each FPGA and intercommunication between FPGAs. Suitable assumptions are made for intercommunication between FPGAs and based on single FPGA latency calculations, estimates were derived for larger size matrices using interconnected FPGAs.

3. Guiming Wu, Yong Dou et.al[13]

   The work Blocking LU Decomposition for FPGAs proposes a single-type Processing Engine for all types of blocks in block LU decomposition in comparison to other works listed above. For the convenience of implementation, block iterations are mapped to space, whereas the block positions in block lu-decomposition are mapped to time. PEs are mapped to various types of blocks in each iteration and block data moves from PE to another PE.

4. Vitis Library Solver [5]

This work does not implement Block LU-Decomposition. It only implements the Gaussian Elimination method (1). It is limited to process matrices up to sizes 512×512. The architecture has only 128 Processing Elements.128 elements of a row are processed simultaneously. The matrix to be decomposed is stored inside, FPGA's internal memory as a 128×128, three-dimensional array, in which only 128 elements are processed simultaneously. For a Matrix of size 512×512, four layers of 128×128 are organized in three dimensions. The LU decomposition algorithm is modeled in HLS-C .

Chapter 5

Proposed FPGA acceleration of Block LU-decomposition

5.1 Design Space exploration

The architecture design of the block LU decomposition is influenced by the available resources in the FPGA, including memory, compute resources and global memory bandwidth. Two critical factors for the block LU decomposition are selecting the block size and determining the number of processing elements (PEs) to use. The number of PEs is decided by the FPGA's available DSPs and LUT resources. For instance, Alveo U50 has 270 MACs in one SLR region. FPGA routing congestion can limit the full utilization of available DSP and LUT resources, typically allowing only around 60% to 70% of DSP resources to be used for seamless FPGA implementation. For example, a double-precision multiplier and subtractor require 11 DSPs, and 1200 LUT storage elements are required. Due to limited LUTs available per SLR (Slice Logic Region) and routing congestion, practically only 128 PEs can be used. Therefore, the number of blocks in the block LU decomposition can be decided based on the available resources.

Increasing the number of PEs aims to achieve concurrent execution of PEs for any block size. A straightforward choice is to select the block size (B) equal to the number of PEs in the given platform, assuming there are no memory constraints. However, in modern FPGAs, internal memory is not a performance bottleneck but rather the available resources for Block LU Decomposition. For example, the Alveo U50 FPGA SLR has 232Mb of internal memory, accommodating 3.6M double words, equivalent to matrix block sizes up to 20K. However, the number of PEs cannot exceed 128 per SLR. Therefore, the number of available PEs dictates the block size rather than the internal memory available. Other choices like $B < No.of PEs$ could also be considered.

In this research work, these choices were analyzed as they influence the accelerator's performance and the complexity of the architecture. The goal is to balance the available FPGA resources, block size, and the number of PEs to design an efficient and effective block LU decomposition architecture. In order to achieve optimum design, three different choices were analysed.

5.1.1 Choice-1: $B < No.of PE$:

To achieve simultaneous computation of partial sums of multiple rows in a single clock cycle in algorithms 1 to 6, 256 PEs of Multiplier and Subtractor blocks of the FPGA would be utilized. If block size (B) in these algorithms is set to 16, then 16 rows (256) of partial sums could be computed simultaneously. To perform this step, 256 memory banks will be required to access all partial sums simultaneously. For each iteration of algorithms 1, 3, 4, 5, and 6, 256 partial sums would be computed in the first clock cycle, and partial sums of the second iteration would be computed in the next clock cycle. The computation of partial sums at the present iteration uses the output of partial sums computed in the previous cycle. The latency of PE for one data processing would be one cycle.

To summarize, with 256 processing elements (PEs) and a block size of 16, we can simultaneously compute 16 rows of partial sums in a single clock cycle. To ensure seamless execution between iterations, each PE must have one cycle of latency for data processing. This concurrent computation significantly improves the performance and efficiency of the block LU decomposition on the FPGA architecture.

For efficient implementation, we utilize the Floating-point Multiply-Accumulate (MAC) Intellectual Property (IP) available from Xilinx [14]. This IP supports both single-precision and double-precision data types. We configure the IP for double precision to suit our requirements. The IP implementation is internally pipelined and can operate with various latencies and frequencies. For instance, setting the IP to have one cycle latency reduces the operating frequency to 50 MHz. Conversely, with a latency of 20 cycles, the operating frequency can exceed 300 MHz. In our implementation, we configure the IP to have a latency of 12 cycles and an operating frequency above 250 MHz, making it suitable for our needs. The pipelining within the FMAC IP ensures a throughput of one cycle throughout its execution.

When the block size is set to 16×16, and 256 PEs are used, the IP is configured with a 12-cycle latency, and the operating frequency is set at 250 MHz. Consequently, each iteration of the k-loop takes 12 cycles, so that partial sums of present loop are available for next loop. However, this approach increases the latency of processing each block, and subsequently, the overall LU decomposition latency increases as well. Main limitation of this approach is inability to utilize pipelining available in FMAC IP.

Another possible configuration involves using the FMAC IP with one cycle latency and a frequency of 50 MHz. This setup results in each k-loop iteration having a latency of one cycle. However, the overall LU decomposition latency increases due to the reduced operating frequency.

An alternative approach is to use a block size (B) less than the number of available PEs and processing of multiple $B \times B$ blocks concurrently. In the case under consideration, we process one row of 16 blocks simultaneously. While this architecture poses no constraints on the PE's internal pipeline within the FMAC IP, it demands the simultaneous fetching of 16 blocks to internal memory. Additionally, it requires duplication of control flow logic, leading to increased LUTs and routing constraints. Storing many matrices in the FPGA's internal memory can also result in inefficient utilization of the block memories available in the FPGA.

5.1.2 Choice-2: $B > No.ofPE$:

The second choice is to have a block size greater than the number of available processing elements (PEs). Choice-2 allows for concurrent usage of the PEs but results in a more complex memory organization and scheduler. The *Vitis solver library* [5] implements a non-blocking architecture that supports matrix sizes greater than the number of PEs. The Vitis library is benchmarked on Alveo U50 to explore the design space to determine the latencies for different matrix sizes using 128 PEs. The matrix is organized in a three-dimensional memory, enabling all PEs to access data concurrently without any resource conflicts.

5.1.3 Choice-3: $B = No.ofPE$:

The third choice is to select a block size (B) equal to the number of available processing elements (PEs). With this choice, memory organization and data access for processing PEs become simpler without any resource constraints. The latency of double precision Floating-point Multiply-Accumulate (FMAC) is much less than B, so it does not constrain the block execution. Consequently, the operating frequency for the FMAC IP can be set to the highest possible value in the FPGA. This choice also results in minimal degradation in sustainable Giga-Floating Point Operations Per Second (GFLOPS) compared to peak GFLOPS.

Table 5.1 Table compares PE efficiency and latency between choices of taking block sizes as 16 and 256. It also compares the architecture with a block size of 16 and assumes the operational latency of floating point unit as one

Parameters	**Choice-1**	**Choice-1 option-2**	**Choice -3**
Matrix Size	4096	4096	4096
Block Size	16	16	256
No. Pes	256	256	256
No. of Blocks	5625216	5625216	1496
Latency obtained per block (us)	**4.27**	**0.32**	**71.68**
Sustained GFLOPS	**3.84**	**25.6**	**122.2**
Operating Frequency	300 MHz	50MHz	300MHz
Total latency(s)	**12.00**	**1.8**	**0.377**

We modelled the three choices mentioned above using Vitis-HLS C and recorded the simulated results for each choice in Table 5.1. The matrix size selected for the modelling is 4096×4096. For choice-1, we employed 256 processing elements with a block size of 16 for the Block LU decomposition model. The FMAC IP used in the model had a latency of 20 cycles, an internal pipeline, and an operating frequency of 300 MHz. Based on the Vitis-HLS C simulation, we estimated the latency and calculated

the sustained GFLOPS using the latency and operating frequency. The experimental results are tabulated in column 2 of Table 5.1.

For choice-1 in the design space exploration, we conducted an alternative experiment (option-2) with a different configuration of the FMAC IP. In this experiment, the FMAC IP was configured for a one-cycle latency and an operating frequency of 50 MHz. The block size remained at 16, and the number of PEs was set to 256. Based on the simulation results, we recorded the latency and estimated the sustained GFLOPS, which are shown in the third column of Table 5.1.

We also developed another model for choice-3, where both the block size and the number of PEs were set to 256. This model configured the FMAC IP with a 20-cycle latency and an operating frequency of 300 MHz. The simulated latency and sustained GFLOPs for a matrix size of 4096 are tabulated in the fourth column of Table 5.1.

It is observed that the latency in choice-3 is considerably lower than latency in both options of choice-1. In choice-1, the number of blocks to be processed is significantly higher than in choice-3. Moreover, the processing time for each block is not proportionately reduced either due to data dependencies or the lower operating frequency. As a result, both options of choice-1 exhibit higher latency than choice-3. Due to the lower latency, the sustained GFLOPS is higher for choice-3 than choice-1, even with the same operating frequency.

Considering the simplicity, lower latency, and better efficiency, we have selected the choice-3, where the block size (B) is equal to the number of PEs for our implementation.

Table 5.2 Table compares PE efficiency and latency with choice-3 i.e block size equal to number of PEs with varied block size

Parameters a	**Option 1**	**Option 2**	**Option 3**
Matrix Size	4096	4096	4096
Block Size	32	64	128
No. Pes	32	64	128
Blocks/row	128	64	32
No. of Blocks to be computed	707000	89400	11400
Operating Frequency (In MHz)	300	300	300
Sustained GFLOPS	9.6	19.2	38.4
latency in Secs.	**5.43**	**1.98**	**0.82**

In Table 5.2, we compared the performance of Block LU decomposition for a matrix size of 4096 for various block sizes. Three different block sizes are considered namely 32, 64 and 128. The frequency of operation was 300 MHz. The block size was equal to the number of PEs for each option. The Vitis-HLS model is simulated for various block sizes for the same matrix size of 4096, and results are tabulated.

Energy for different block sizes is also estimated and observed that larger block size is also equally energy efficient.

5.1.4 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 memory blocks. Thus the whole PE array can perform the core vector computation $\vec{v} = \vec{v_1} \pm \vec{v_2} \odot \vec{v_3}$ in parallel without any bottleneck, provided $|v| \leq B$, which will be the case in our application. Figure 5.3 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 $B \times B$.

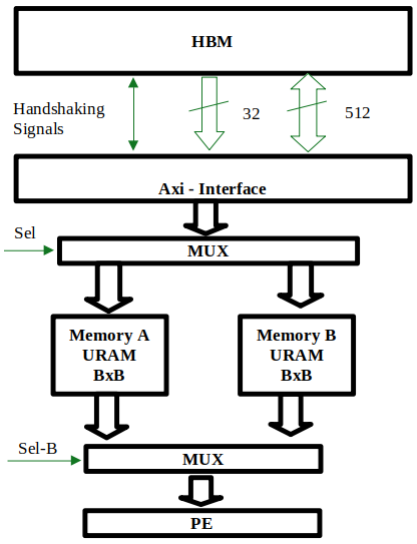

Figure 5.1 Figure shows the memory interface to HBM and connections to PE

The double buffering method is used for external memory access so that memory access time does not influence the latency. Double buffering is used for UMEM and RMEM. We configure the AXI data bus for maximum bit width of 512 bits and fetch Block data from HBM. With double buffering for

UMEM and RMEM, the total internal memory requirement is $5 \times B \times B$. Memory sharing and AXI interface to HBM are shown in Fig 5.1.

The controller used for block fetch, process and write is shown in Fig 5.2. Three instances of the same controller are executed concurrently with required handshake signals between the controllers. The read controller, which fetches the data from HBM. The process controller processes the block data as described in the flow chart. And the write controller, which writes back the data back into HBM. BR and BC represent the indices for blocks in rows and columns. For each outer iteration of k, BR=k and BC=k block is fetched. While processing this block, the next block (BC=K+2) in the first row of the outer iteration is fetched from HBM. This way, one by one, the block in the first row of the outer iteration is fetched internally and processed. After the completion of all blocks in the first row of the outer iteration, blocks in subsequent rows are fetched and processed. The read controller generates the off-chip memory address of the block being fetched.

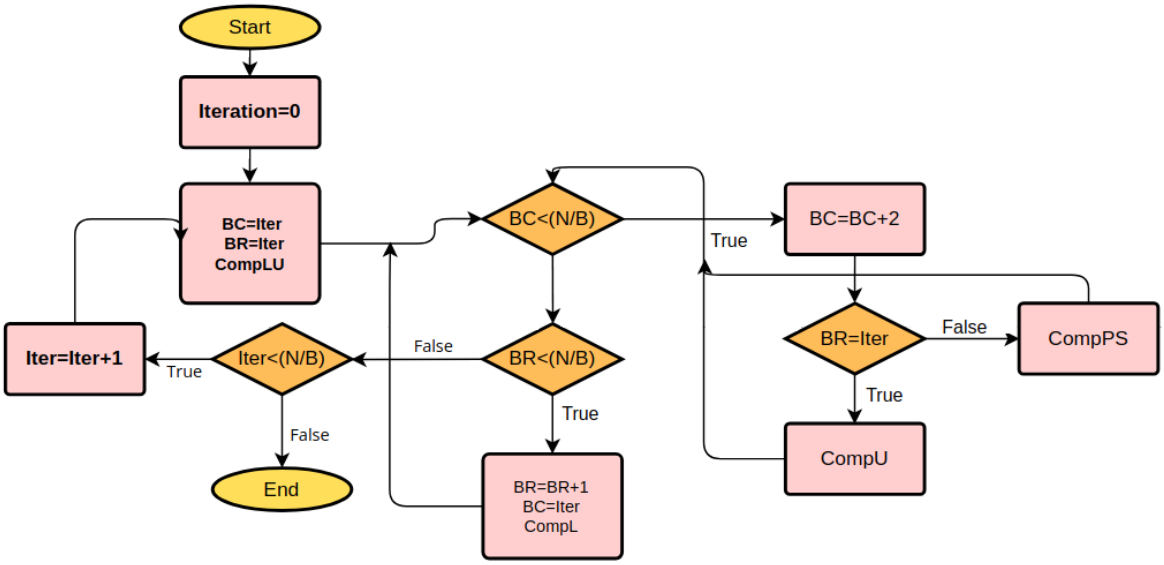

Figure 5.2 The flow chart showing how Block LU decomposition was implemented. The flow chart shows Block Read, Write and Execute steps

The process controller handles the data movement between the internal memories and PEs. The write controller generates the addresses for writing back the result of the block after processing. After completion of the first outer iteration, LU decomposition for blocks in the first row and first columns are computed. All three controllers repeat the above-described operations for N/B iterations.

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.

The PE array processes a $B \times B$ block in $B - 1$ iterations. The linear PE array along with memory interfaces implement Algorithms-1, 3,4,5 for $B \times B$ matrices. 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 $\vec{v} = \vec{v_1} \pm \vec{v_2} \odot \vec{v_3}$ 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 3.2. 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.

The `UMEM`, `LMEM`, and `RMEM` memory units are of size $B \times B$. Each is split across B banks to facilitate interfacing with the B number of PEs in the PE array. Data stored in B memory banks in this sequence makes it easy to perform concurrent multipliers and subtractors in every cycle of block processing. Each bank has an exclusive data port to access the PEs simplifies the scheduling. The total internal memory required is $5 \times B \times B$ with `UMEM` and `RMEM` being double-buffered. Figure 5.3 shows the organization of different memories, PE and interfaces.

As we discussed in the previous sections, Block LU decomposition execution is limited by the number of PEs available in an FPGA. Simultaneous processing of multiple blocks in different FPGAs reduces the latency. But this requires high-speed interconnects between the multiple FPGAs and the host system.

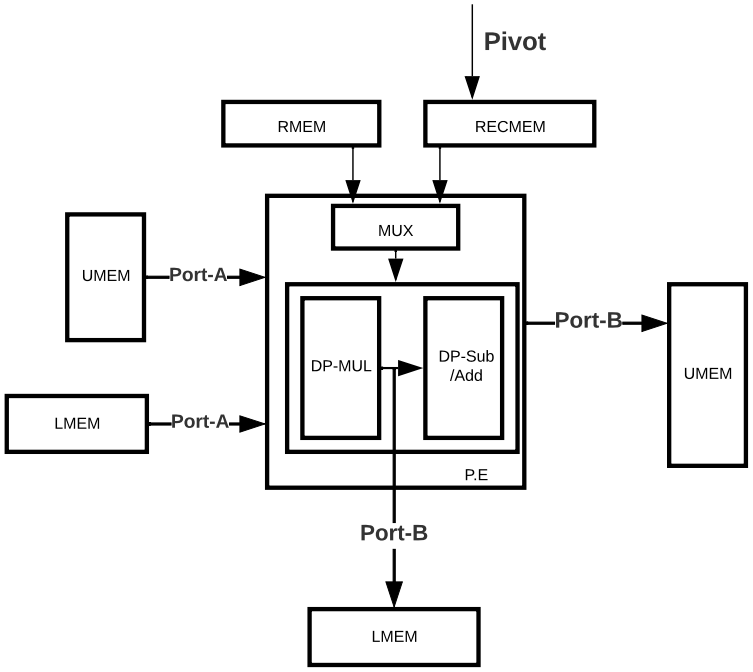

Figure 5.3 Accelerator architecture consisting of Processing Engine (PE) array and distributed memory banks.

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. Both SLRs can access different data blocks with dedicated HBM interfaces available to each SLR. 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.

In our example above, A_{11} is fetched from global HBM Memory and stored in the UMEM of both the SLRs. L_{11} and U_{11} is computed in both the SLRs. Subsequently, A_{12} is fetched to SLR0 and A_{13} is fetched to SLR1 simultaneously. U_{12} and U_{13} are processed concurrently using L_{11} in both SLRs. This way of concurrent processing can happen to A_{22} and A_{23} and so on. For a large number of blocks, in every row of blocks, only one block processing is duplicated. For example, if the number of blocks in a row of blocks is equal to 128, with one block duplication per row, duplication is less than 0.75%.

Chapter 6

Implementation and Experimental Results

6.1 Alveo U50 Platform Details

The Alveo U50/U50 LV accelerator card uses an UltraScale+™ FPGA containing a PCIE4C block. The PCIE4C block is compliant with the PCI Express Base Specification v3.1 supporting up to 8.0 GT/s (Gen3 x16) and compatible with PCI Express Base Specification v4.0 supporting up to 16.0 GT/s (Gen4 x8). The PCIE4C block is also compliance with CCIX Base Specification Revision 1.0 v0.9, supporting speeds up to 16.0 GT/s. This accelerator card is equipped with 8 GB of high-bandwidth memory (HBM2), and Ethernet networking capability [15].

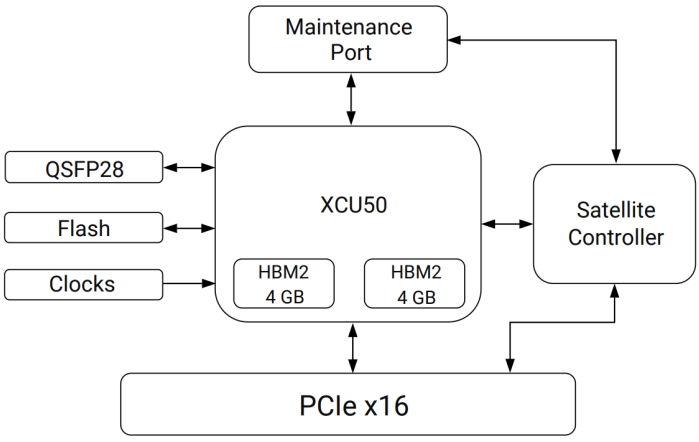

Figure 6.1 Figure represents the memory organization of a block,in Block LU decomposition

Alveo U50 data centre acceleration card connected to a host Xeon CPU through a PCIe 3.0 x16 interface. The Alveo U50/U50 LV card features the XCU50 FPGA, which uses Xilinx stacked silicon interconnect (SSI) technology to deliver breakthrough FPGA capacity, bandwidth, and power efficiency. This technology allows for increased density by combining multiple Super-Logic Regions (SLRs).

Table 6.1 Table shows available resources of Alveo-U50 per SLR

Site Type	SLR0	SLR1
CLB LUT	440000	424000
Block RAM	261	264
URAM	362	349
DSP	2150	2098

The XCU50 FPGA on the U50 card comprises two Super Logic Regions (SLRs). Each SLR region contains DSP units greater than 2000, BRAM greater than 250 blocks, URAM (288kb) blocks greater than 320, and LUTs as shown in table6.1. The Xilinx Alveo U50/U50 LV accelerator card is a custom-built UltraScale+ FPGA that runs optimally (and exclusively) on Alveo architecture [15].

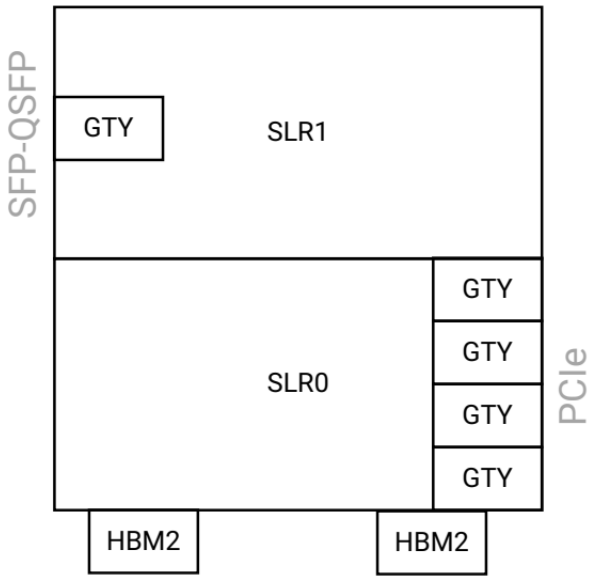

Figure 6.2 Figure showing the block diagram of SLRs and HBM of Alveo-U50

The XCU50's bottom SLR (SLR0) integrates an HBM2 controller to interface with the adjacent 8 GB HBM2 memory.

6.2 Block LU decomposition implementation in Alveo-U-50

The proposed hardware accelerator is synthesized at 250 MHz with 128 PEs per SLR using Vitis-HLS version 2021.1. Block Diagrams for HBMs with SLRs are shown in Fig.6.2. In the block diagram, the first SLR is named as `SLR0` and the second SLR is named as `SLR1`. The number of PEs per SLR is restricted due to the limitation of available CLBs, even though more DSPs are available in FPGA.

The Vitis generated utilization report Fig.6.3 shows resources utilized per SLR. In the Vitis generated utilization report Fig.6.3 it can be seen that 72% of `SLR0` and 68% and `SLR1` is being utilized. 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.

Site Type	SLR0	SLR1	SLR0 %	SLR1 %
CLB	39912	36848	72.62	68.24
CLBL	20987	19458	71.68	66.45
CLBM	18925	17390	73.70	70.35
CLB LUTs	186897	187122	42.51	43.32
LUT as Logic	177859	179131	40.45	41.47
using O5 output only	3103	1216	0.71	0.28
using O6 output only	145987	150536	33.20	34.85
using O5 and O6	28769	27379	6.54	6.34
LUT as Memory	9038	7991	4.40	4.04
LUT as Distributed RAM	2324	3845	1.13	1.94
using O5 output only	0	0	0.00	0.00
using O6 output only	164	375	0.08	0.19
using O5 and O6	2160	3470	1.05	1.75
LUT as Shift Register	6714	4146	3.27	2.10
using O5 output only	0	0	0.00	0.00
using O6 output only	4958	3356	2.41	1.70
using O5 and O6	1756	790	0.85	0.40
CLB Registers	280766	237709	31.93	27.51
CARRY8	6548	6348	11.91	11.76
F7 Muxes	3978	3868	1.81	1.79
F8 Muxes	1191	1292	1.08	1.20
F9 Muxes	0	0	0.00	0.00
Block RAM Tile	261	245	38.84	36.46
RAMB36/FIFO	256	245	38.10	36.46
RAMB36E2 only	256	245	38.10	36.46
RAMB18	10	0	0.74	0.00
URAM	128	128	40.00	40.00
DSPs	1435	1431	49.83	46.58
PLL	0	0	0.00	0.00
MMCM	0	0	0.00	0.00
Unique Control Sets	6115	4254	5.56	3.94

Figure 6.3 Vitis generated report of the utilization of resources per SLR in Alveo U-50 for the proposed design

The accelerator's internal memory is banked; therefore, all eight input words can be written in a single cycle. We replicate the accelerator's compute units across two SLRs. Both compute units process two blocks of data concurrently. One SLR computes even-numbered blocks (`Kernel_getrf_nopivot_0_e` in SLR0), and another SLR (`Kernel_getrf_nopivot_0_o` in SLR1) computes the odd-numbered blocks. Both the kernels HLS code is similar, except that one executes even number of blocks and

another executes odd number of blocks. Host C-code implements simultaneous execution of both the kernels in both the SLRs. Fig.6.4 shows the organization of two kernels in to two different SLRs.

The content `LMEM` memory block, which is required for both even and odd numbered blocks, is processed in both SLRs. The block duplication causes a minor latency penalty ($\approx 5\%$). However, this reduces the need for tight synchronisation and favours independent read, execute and write operations. The accelerator writes the output back to the HBM, from where it is read back by the host. Matrix size N is a run-time parameter from the host while executing the hardware.

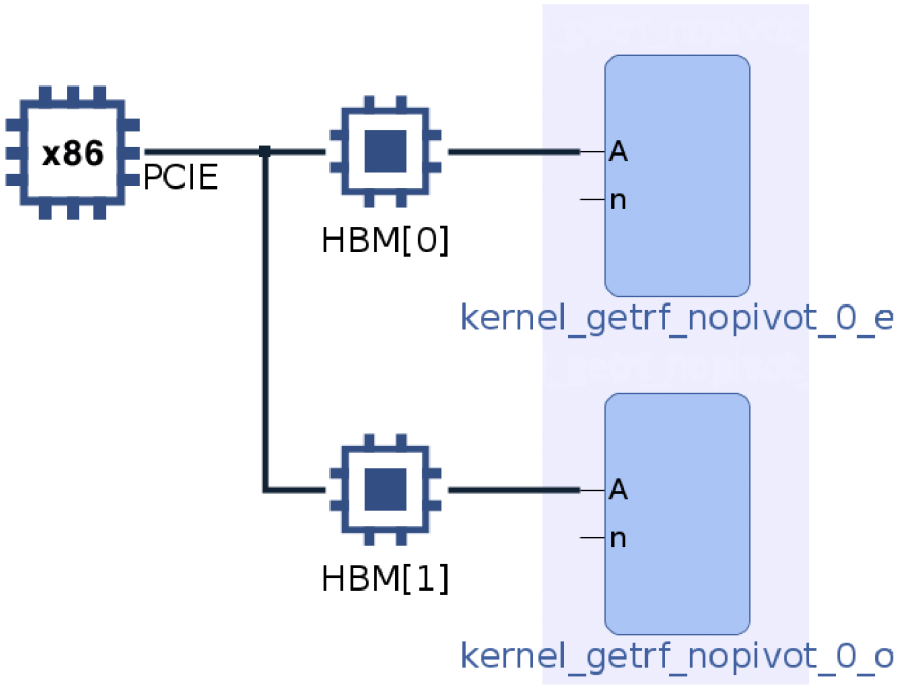

Figure 6.4 Vitis generated diagram of Block LU decomposition in Alveo U-50

6.3 Experimental Results

We compare the performance of our accelerator with a four-core (8 threads) Intel Xeon E2224G CPU running at 3.5 GHz and 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 [16], and GPU is bench-marked using the PyTorch library[17] and Cuda-Dense library [18].

Table 6.2 Table for comparison of latencies of Block-LU Decomposition for different matrix sizes ($N \times N$) for proposed FPGA design, CPU, and GPU

Matrix Size N	**Proposed FPGA design latency (ms)**	**Intel MKL Multi-core Xeon CPU latency (ms)**	**Torch.lu multi-core GPU latency(ms)**	**Cuda-Dense multi-core GPU latency (ms)**	**FPGA over CPU**
256	0.21	50	0.93	2.53	234.74
1024	8.80	120	12.92	19.53	13.31
Matrix Size N	**Proposed FPGA design latency (s)**	**Intel MKL Multi-core Xeon CPU latency (s)**	**Torch.lu multi-core GPU latency(s)**	**Cuda-Dense GPU latency (s)**	**FPGA over CPU**
2048	0.06	1.71	0.04	0.05	27.52
4096	0.48	13.82	0.18	0.20	28.68
8192	3.81	96.34	1.18	0.85	25.29
16384	29.40	695.78	11.77	5.74	23.67
32768	243.2	3500	45.3	10.50	14.39

Table 6.2 shows that the proposed FPGA design outperforms the Intel MKL CPU's library. The latency improvement is more than $20\times$ over the CPU for most matrix sizes. For smaller matrix sizes, our accelerator's operational latency is lower than the GPU's. For larger matrix sizes, the torch.lu's latency is $2-3$ times and $4-5$ times lower than FPGA latency. This is because the theoretical peak performance of Alveo U-50 FPGA is 1488 GFLOPS, Which is much lower the GPU's 16.3 TFLOPs performance. 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.

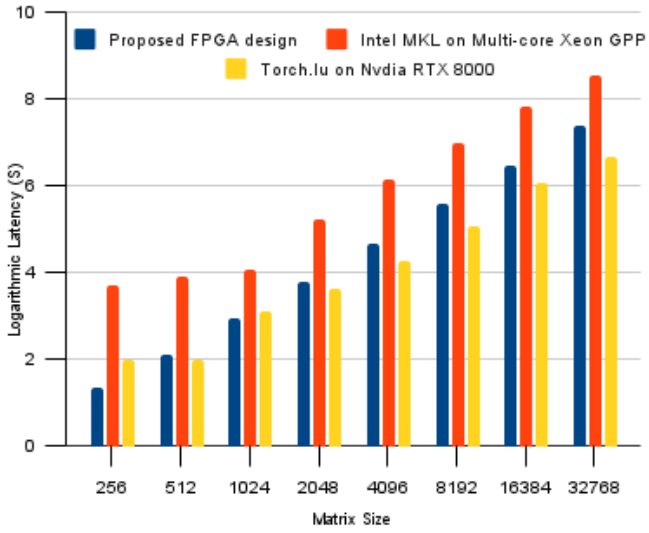

Figure 6.5 Graph comparing the latencies of the proposed design, Intel MKL implementation on a CPU and Torch.lu implementation on a multi-core GPU

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 6.5 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. Table6.3 shows the sustainable GFLOPS for FPGA acceleration.

Table 6.3 Table showing the Sustained GFLOPS performance and efficiency for different matrix sizes with 128 No. of PEs in each SLR, operating at 250 MHz of the proposed architecture

Matrix size	**Sustained GFLOPS**	**Efficiency**
256	98.38	0.77
512	100.88	0.79
1024	97.19	0.76
2048	100.88	0.79
4096	99.61	0.78
8192	98.38	0.77
16384	100.88	0.79

Table6.4 tabulates the energy for different block sizes for a fixed matrix size of 4096. From this, we can conclude that energy efficiency increases with an increase in matrix size.

Table 6.4 Table showing how energy efficiency increases with increase in block size

Block Size	**Energy in Joules**
16	9.043
32	8.23
64	7.2
128	6.48

Table 6.5 Performance of the proposed design on FPGA with the benchmark programs of CPU and GPU in GFLOPS

Matrix Size	Proposed FPGA Design Performance	Torch GPU Library	Cuda-Dense GPU Library	Intel MKL CPU Library
256	98.38	11.19	4.12	2.34
1024	97.19	51.61	34.14	18.33
2048	100.88	123.92	97.54	10.03
4096	99.61	232.95	217.74	9.94
8192	98.38	288.77	402.49	12.10
16384	100.88	231.91	475.72	16.30
32768	96.44	482.23	2080.24	3500.00

Energy is computed from the average power consumption for the $N \times N$ matrix LU decomposition and operational latency of the design for all three platforms. Energy efficiency (number of GFLOPS per joule) for all three platforms is compared in Fig.6.6.

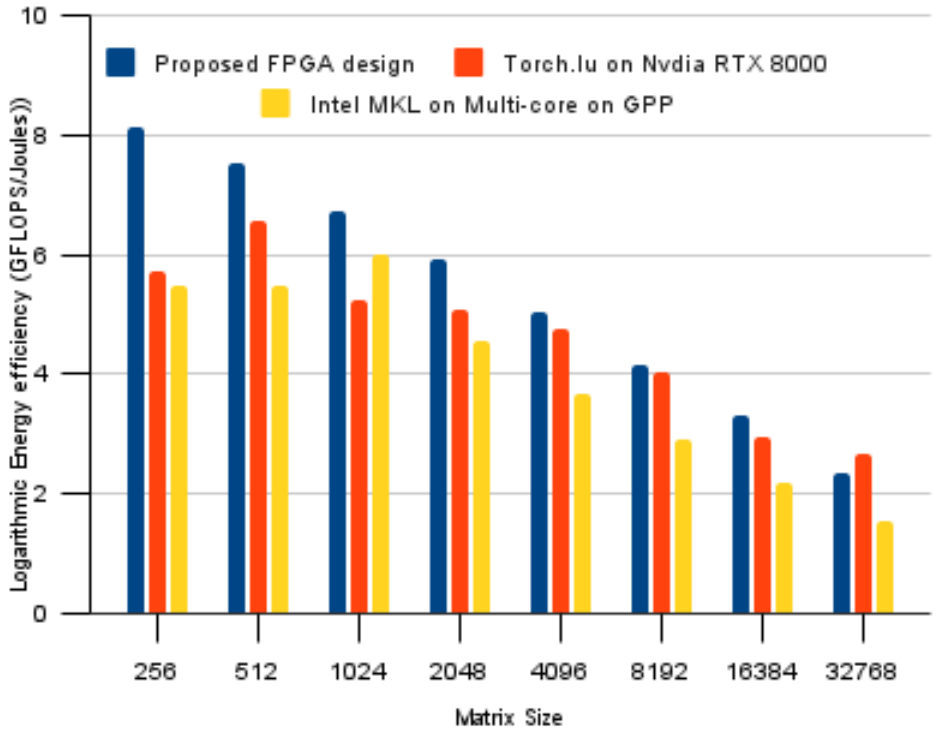

Figure 6.6 Graph comparing the Energy Efficiency of the proposed design, Intel MKL implementation on a CPU and Torch.lu implementation on a multi-core GPU

The Proposed design, because of its simple architecture, is compute-efficient. If we compare our per unit LUT consumption with other implementations, We consume less and perform far better than their designs. For example, refer to [6]'s design, whose LUT/PEs consumption is 3432. The proposed design's per unit LUT consumption (LUT/PE) is 2910. The design by Manish Kumar Jaiswal ([6]) was on implemented on multiple FPGAs. The proposed design performs 3 times better than the [6]'s design, with less LUT consumption.

Chapter 7

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 process matrices of arbitrary dimension. Our work on Block LU decomposition demonstrated the viability of FPGA-based acceleration for data centres for solving linear equations. FPGA resources are customisable and provide good performance even for double-precision floating-point mathematical computations. FPGA-based acceleration achieved speedup over the benchmark program on an Intel Xeon CPU. The performance of our implementation was comparable to the benchmark program of LU decomposition on GPU. Our work scales up with multiple dies on FPGA, and implementation on FPGA with more dies, GFLOPS performance and latency can be better than GPU.

Bibliography

[1] Caner Ozcan and Baha Sen. Investigation of the performance of lu decomposition method using cuda. volume 1, page 50–54, 12 2012.

[2] Vladimir Petrov and Marin Marinov. A performance comparison of computing lu decomposition of matrices on the cpu and the gpu. *Proceedings of the International Conference on Electrical and Electronics Engineering (ICEST)*, pages 26–30, 2015.

[3] Fumihiko Ino, Manabu Matsui, Keigo Goda, and Kenichi Hagihara. Performance study of lu decomposition on the programmable gpu. pages 83–94, 12 2005.

[4] 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.

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

[6] 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.

[7] 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.

[8] Prasanna Sundararajan. High Performance Computing Using FPGAs. In *white paper, Xilinx WP375 (v1.0), September 10,2010*. 2010.

[9] O. Williams S. Kestur, J.D. Davis. BLAS Comparison on FPGA, CPU and GPU. In *2010 IEEE Computer Society Symposium on VLSI*.

[10] B. Sukhwani, M. Chiu, Md. A. Khan, and M.C. Herbordt. Effective Floating Point Applications on FPGAs: Examples from Molecular Modeling. In *High Performance Embedded Computing (HPEC), IEEE 2009*. 2009.

[11] Jeff Allred, Jack Coyne, William Lynch, Vincent Natoli, Joseph Grecco, and Joel Morrissette. Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–4, 2009.

[12] Xilinx. VITIS Highlevel Synthesis User Guide. `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS`, 2023.

[13] Guiming Wu, Yong Dou, and Gregory D. Peterson. Blocking lu decomposition for fpgas. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–112, 2010.

[14] Xilinx. Floating-Point Operator v7.1. `https://docs.xilinx.com/v/u/en-US/pg060-floating-point`, 2020.

[15] Xilinx. Alveo U50 Data Center Accelerator Card Data Sheet. `https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds965-u50.pdf`, 2020.

[16] 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.

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

[18] Nvdia. cuSOLVER API Reference. `https://docs.nvidia.com/cuda/cusolver/index.html#cusolverdn-dense-lapack`, 2022.