XILINX OPEN HARDWARE COMPETITION 2022: Acceleration of LU Decomposition on FPGAs

Yichen Zhang | s2130520@ed.ac.uk

30th June 2022

Team number: xohw22-006
Supervisor: Dr Danial Chitnis
Email of Supervisor: d.chitnis@ed.ac.uk
Github Link: https://github.com/danchitnis/LU-decomposition-FPGA.git
Video Link: https://youtu.be/b9Mz6KtWnw8

Summary

This project aims to accelerate the solving of linear systems by parallelizing the matrix decomposition on FPGAs. Solving large linear systems is part of everyday engineering design, including the integrated circuit simulation, where the majority of the matrices are sparse. Lower-upper (LU) decomposition is the most commonly used method to solve the sparse linear system. However, the sparse-matrix decomposition is hard to parallelize on regular processors due to the irregular structure of the input matrices. Modern FPGAs have the potential to compute these hard-to-parallelize problems more efficiently due to their reconfigurable structure, such as flexible memory access, loop flattening and unrolling. As a result, these FPGA implementations may lead to higher compute throughput and efficiency.

In this project, we proposed a parallel CPU+FPGA based architecture is proposed for the acceleration for SPICE engine. While the preprocessing and factorisation phases are implemented on the CPU, the solving phase is implemented on the FPGA. On the CPU side, the KLU Matrix Solve algorithm [1] is used and modified to suit the implementation on the FPGA.

LU decomposition is a mostly memory bound computation. It requires careful design on the data access. To accelerate the solving phase, preprocessing is used to explore the data structure of matrices and to generate the data access patterns. This preprocessing phase is done once typically for a matrix, so it is implemented on CPU. To accelerate the solving phase, vectorisation is used to allow parallelized solving of multiple right-hand side vectors. It shows about 1.2× speedup per right-hand side on solving phase of middle-size matrices. Considering the large frequency difference between Alveo U280 and Intel i7 8th CPU, it demonstrates it can be more advantageous in common cases while providing more power efficiency.

1 Introduction

1.1 Project Aim

- The aim of this project is to accelerate the solving of linear systems by parallelizing part of the matrix decomposition, especially the solving phase on FPGAs.
- Modify the KLU Matrix Solve algorithm to fit the implementation on the FPGA.

- Develop the parallel solution for the forward and backward substitution.

1.2 Contribution

- The contribution of the team is to design a sparse linear solver based on a CPU-FPGA architecture.
- Modified the Gilbert-Peierls' algorithm (left-looking algorithm) by extracting both symbolic and numeric analysis from numeric factorisation.
- Quantitative comparison between software KLU, software UMFPACK and our CPU+FPGA based implementation by a variety of circuit matrices from the University of Florida Sparse Matrix Collection [2]. Achieved 1.2× speedup on solving phase of middle-size matrices, compared with KLU and around 7× speedup compared with UMFPACK.
- Developed the forward and backward substitution to deploy parallelized solution.

Our responses to the Xilinx Open Hardware 2022 are given below:

- **Technical Complexity:** LU decomposition is a mostly memory bound algorithm. Solving this kind of problem requires careful design on the data access and communication, especially for the sparse LU decomposition, where the data storage is different from dense matrices. Therefore, graph theory is deployed to accelerate this process.
- **Implementation & Re-usability:** Our solution utilises HBM channels within Alveo U280 to increase the performance. In addition, the preprocessing, factorisation and solving phases are well separated, and detailed information is provided for compilation.
- **Marketability & Innovation:** During the previous two decades, the emergent development in Information and communications technology (ICT), riding on the internet backbone and increasing computation capabilities of computer technology, has revolutionised the commercial industry. It has also made it possible to deploy the accelerated linear system solver on the cloud-based FPGA. The accelerator is responsible for the acceleration of the matrix solver embedded in those circuit simulators. Accelerator developers can then merge several of these simulators and develop a cloud-based accelerator module. Finally, end-users may choose from a variety of easily accessible hardware accelerators or build their own to implement on the FaaS platform to accelerate their applications.

1.3 Sparse Linear System

Solving a linear system $Ax = b$ is relatively a basic algorithm problem and is closely related to the applications of almost every field, including engineering, physics, chemistry, etc. It is at the heart of various algorithms and applications, including computational geometry, circuit simulation and data science. A variety of approaches have been developed to solve the linear system, which can mainly be classified into two types: direct methods and iterative methods.

The direct methods solve the sparse linear system mainly by computing the determinant, inversion or factorisation of a matrix. For example, Cramer's rule calculates the solution in terms of the determinants of the coefficient matrix and the right-hand-side vector. Gaussian elimination finds the solution by calculating the inverse of the system. LU decomposition solves the system by factorising the system into a lower triangular matrix L and an upper triangular matrix U.

On the other hand, the iterative method is more modern. It attempts to solve the linear system based on an initial condition and successive approximates to the final solution, such as spectral sparsificaiton of graphs and division-free inversion. However, the behaviour and its stability of iterative methods is

heavily based on the convergence[1] of the input matrices. On the contrary, the operation cost of behaviour methods is closely related to the cost of the matrix multiplication, making it preferred to the randomised and iterative methods in real applications as the directed methods are more robust and predictable.

LU decomposition is a direct method that can solve a large sparse linear systems multiple times, with various applications in circuit simulation, structure analysis, power networks, etc. A variety of parallel sparse linear solvers have adopted LU decomposition and have been running on massively parallel supercomputers. For example, Cray XE6 [3], which is a type of distributed-memory machines, has implemented SuperLU_DIST [4] solver for LU decomposition.

With the continuous development of IC industry, the size of FPGAs has grown to the extent that intense and heavy floating-point operations can be now accommodated. After a decade of research, it has been proved that the accelerating algorithms on FPGAs is a promising research avenue. Modern FPGAs have made it possible to fit a large computation kernel and establish parallel computation machines. In this paper, a sparse linear solver is build based on a CPU-FPGA architecture. While the pre-processing is made on CPU, the FPGA performs the numeric factorisation and solving of the matrices.

2 Foundation of LU Decomposition

Suppose $A \in \mathbb{R}^{n\times n}$. An LU decomposition of A refers to the factorisation of A into a lower triangular matrix L and an upper triangular matrix U, with proper row and/or column permutations. Here, permutation is to change the order of the row or column of a matrix according to a row permutation matrix P and a column permutation matrix Q, and each row and column of the permutation matrix contains a single 1 with 0s everywhere else.

A sparse matrix refers to a matrix that has few nonzeros in it. Although the quantification of "few" is not defined, typically $\mathcal{O}(n)$ elements are in a sparse matrix, where n is the order of the matrix. Sparse matrices are omnipresent in scientific computation when modelling systems with numbers of elements with restricted couplings. In this project, KLU is used to solve the sparse linear system. We modified it to fit the static feature of the FPGA. It first performs the symbolic analysis (subsection 2.1, subsection 2.2) to explore the structure of the matrix, and then performs the numeric factorisation.

2.1 Block Triangular Form (BTF)

Block matrix is a a matrix that is broken into sections, called sub-matrices. Block triangular form is a special block matrix that either upper or lower part of the diagonal section is all zeros. Equation 1 shows an example of an upper block triangular matrix and the submatrix A_{11} of it is a 2×2 submatrix. It is obtained by a row permutation matrix P and a column permutation matrix Q, such that $BTF = PAQ$.

Permuting a sparse matrix can save computational work as well as the intermediate storage for various sparse matrix algorithms, including the linear least squares problem and LU decomposition [5]. In LU decomposition, by converting the input sparse matrix to BTF can help

1. The lower triangular part of the matrix below the block diagonal requires no factorisation effort at all.

2. The nonzeros in the off-diagonal region will not contribute to any fill-in.

[1] For an $n \times n$ matrix A, A is convergent if:

$$\forall i, j \in [1, n], i, j \in \mathbb{N} \Rightarrow \lim_{k\to\infty} \left(A^k\right)_{ij} = 0$$

3. The diagonal blocks are independent of each other. Therefore, submatrices of the diagonal regions can be factorised parallelly. For example, submatrix A_{11} and A_{22} can be factorised at the same time.

$$\begin{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} & \mathbf{A_{12}} & \mathbf{A_{13}} \\ \mathbf{0} & \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} & \mathbf{A_{23}} \\ \mathbf{0} & \mathbf{0} & \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \end{bmatrix} \tag{1}$$

Sargent and Westerberg [6] proposed the initial algorithm to compute BTF of a matrix. Their algorithm is based on the fact that all the nodes in a cycle of a graph must lie in the same strong component. However, this involves a time complexity of $\mathcal{O}(n^2)$. Tarjan [7] modified this algorithm in 1972, and reduced the time complexity to $\mathcal{O}(n + \tau)$, where τ is the number of the off-diagonal nonzeros.

Tarjan's algorithm focuses on the avoidance of the potentially expensive node collapsing steps. It deploys a stack to hold the nodes that are on the correct *path* or from which a backtrack has occurred, and indicates that the required strong components can be read successively from the top of the stack.

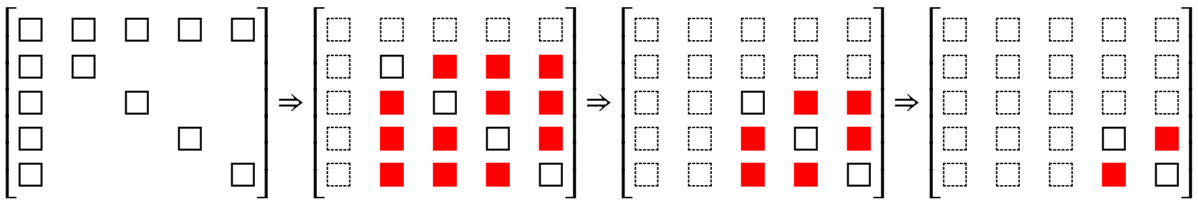
The main steps of Tarjan's algorithm are

1. Do DFS (Depth First Search) from the unvisited nodes in the graph and preordering the nodes, stored in an array called $dfsn[n]$
2. If a new vertex is visited, push it to the stack.
3. Let the array $low[n]$ represent the smallest preordering value among its descendants and itself.
4. If a vertex has finished its searching, update its *low* value. If $low[p] = dfsn[p]$, it means that vertex p is one of the roots of the strongly connect components and its descendants are reachable to each other. Meanwhile, they all cannot reach to the ancestor of vertex p and other unvisited paths of ancestor. Hence pop elements out from the stack until vertex p is popped out.
5. Continue searching until all vertices are visited.

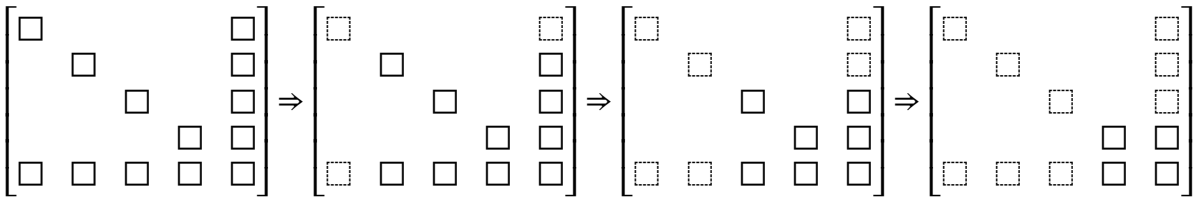
2.2 Fill-reducing Ordering

After the BTF is obtained, we should try to reduce the fill-ins for each of the blocks. As we have discussed in the last section that the structure pattern of the matrix would have a large impact on the number of nonzeros of the factors and may lead to large number of *fill-ins*. A fill-in is defined as a nonzero in position (i, j) of the factorisation result, where it is originally zero in the input matrix. Hence, to limit or reduce the number of fill-ins, it is a widely used practice to reorder the rows or columns of the input matrix prior to LU decomposition. Figure 1 shows the influence of the ordering during the decomposition procedures.

Mathematically speaking, the reordering procedure is to find a row permutation matrix P and a column permutation matrix Q, such that by computing Equation 2, it creates much less fill-ins than the original matrix A. This equation is for pattern symmetric matrix. If it is not, then the input matrix $A + A^T$ can be used, known as *symmetrisation*. The reordering calculation only take the structure of the input matrix into consideration and the numeric value is ignored. However, Rose and Tarjan [8] have proved that the optimised ordering for symmetric pattern matrices which derived minimum fill-ins is an *NP-complete* (Nondeterministic Polynomial-Time Complete) problem, which means it cannot be solved in polynomial time and require exponential time, i.e., $\mathcal{O}(2^n)$ and above. Yannakakis [9] have proved that



(a) Without ordering



(b) With ordering

Initial nonzero Factorised nonzero where initial value is nonzero fill-in

Figure 1: Comparison of Ordering

the same is for asymmetric pattern matrices. Accordingly, allowing certain number of fill-ins would be computationally cheaper than finding the optimised permutation matrices, and *heuristic* algorithms are widely deployed in an attempt to reduce fill-ins.

$$A' = PAQ \tag{2}$$

The minimum degree algorithm [10] is widely used heuristic algorithm to find a row permutation matrix P instead of two permutation matrices. By doing permutation given in Equation 3, it tries to achieve fewer nonzeros. Here, the *degree* refers to the number of edges that are incident to the vertex, i.e., the edges are coming into the vertex.

$$A' = PAP^T \tag{3}$$

The basic idea of the minimum degree algorithm is to select the vertex p as the k^{th} pivot such that the degree of vertex p, $t_p \equiv \left|adj_{G^{k-1}}(p)\right|$, is minimum. Here $|\ldots|$ means the size of a set. However, computing the degrees of nodes can be relatively time-consuming and it is the costliest part of the algorithm. Approximate Minimum Degree (AMD) [11] algorithm, however, tries to avoid the re-computation. Rather than keep track of the exact degree, AMD tries to find an upper bound of the degree which is easier to calculate. For vertices with least degrees, this bound can be very tight. With the help of appropriation instead of exact degree, AMD tends to save both time and memory usage, particularly for matrices with irregularly structure. Meanwhile, it also has on impact on the quality of reordering.

2.3 Left-looking Algorithm

The aim of a good sparse LU decomposition algorithm is to solve a linear system in time and space proportional to $\mathcal{O}(n) + \mathcal{O}(nnz)$ [12]. However, it is not easy to realise due to the various structure of the matrix in the course of decomposition. As the matrices are sparse, it is not easy to know the exact nonzero structure of these vectors in advance of the numerical calculation. Some nonzeros will be added to the results where they are zeros in the original matrix and this can be even worse with pivoting.

To deal with this issue, Gilbert and Peierls (G-P) proposed a left-looking algorithm with partial pivoting, in time proportional to the number of floating-point operations [13]. Here, "left-looking" means calculation of the k^{th} column of L and U is only based on the already computed column from 1^{st} to $(k\text{-}1)^{th}$.

G-P's algorithm consists of two stages. The first is the symbolic analysis stage which computes the nonzero pattern of the kth column and the second stage consists a numeric factorisation stage which solves the lower triangular system $L\boldsymbol{y} = \boldsymbol{b}$.

2.3.1 Symbolic Analysis

It is possible to predict the nonzero pattern of the factors given an input matrices if the partial pivoting is not applied. However, this is not the case when partial pivoting is applied. Consider the implementation of forward and backward substitution and the entire algorithm is shown in Algorithm 1.

Algorithm 1 Symbolic G-P's Algorithm

```
1: x=A(:,k)                                   ▷ The k^th column to perform
2: for j = 1 : n do
3:     if x(j) ≠ 0 then
4:         x(j + 1 : n) = x(j + 1 : n) − L(j + 1 : n, j)x(j)
5:     end if
6: end for
```

The algorithm above, especially line 8, would take about $\mathcal{O}(n)$ to complete. This $\mathcal{O}(n)$ term seems short and harmless, but the outer *for* loop would perform n times during the factorisation, resulting in an unacceptable $\mathcal{O}(n^2)$ time complexity.

Hence, to avoid the n^2 term, the algorithm must be modified to fit the implementation of sparse matrices. Let us say that we are now calculating the kth column of L and U, we can create a list χ of the index j of x. The index j is ascending order and should satisfy the following condition.

$$\chi = \{j | x_j \neq 0\} \tag{4}$$

Algorithm 2 Modified Symbolic G-P's Algorithm

```
1: x=A(:,k)                                   ▷ The k^th column to perform
2: for j ∈ χ do
3:     for i = j + 1 : n do
4:         if L(i, j) ≠ 0 then
5:             x(i) = x(i) − L(i, j)x(j)
6:         end if
7:     end for
8: end for
```

This can decrease the single loop term $\mathcal{O}(n)$ to $\mathcal{O}(\eta(b))$, where $\eta(b)$ is the number of nonzeros in the kth column of A. Accordingly, to solve the triangular system, it is necessary to determine the nonzero pattern of the corresponding column, i.e., χ, before the computation of the column.

From Algorithm 2, it is easy to find that nonzeros can come from only two places: line 4 and line 8. If we ignore the numerical cancellation, these two lines can be expressed as Equation 5 and Equation 6 respectively.

$$\text{line 4: } [b_i \neq 0 \Rightarrow x_i \neq 0] \tag{5}$$

$$\text{line 8: } [x_j \neq 0 \wedge (\exists i, l_{ij} \neq 0) \Rightarrow x_i \neq 0] \tag{6}$$

These two equations can be converted to a graph-traversal problem.

Define $G(L_k)$ as the directed graph of the lower triangular matrix L. $G(L_k) = (V, E)$ where V are the vertices $V = \{1, 2, \dots, n\}$ and E are the edges $E = \{(j, i)|l_{ij} \neq 0\}$. Equation 5 refers to the nonzeros in vector $\boldsymbol{b}$ that are to be marked as the vertices in $G(L_k)$. Equation 6 suggests that if a vertex j is marked and there exist continuous edges that connect vertex j to vertex i, then vertex i is also marked.

Let say that we have a set $\mathcal{B} = \{i|b_i \neq 0\}$ which represents the nonzero pattern of b. Then the nonzero pattern of x, namely χ, can be calculated by determining the vertices which can be reached from the vertices in set $\mathcal{B}$. This can be denoted as

$$\chi = Reach_{G(L)}(\mathcal{B}) \tag{7}$$

This can be called as a reachability problem. Reachability problem can be solved typically by a depth-first search across the $G(L_k)$ from the vertex set $\mathcal{B}$. The elimination graph can be used to derive the nonzero pattern. However, the order that the elimination graph obtains is in row direction. G-P's algorithm, on the contrary, factorises the matrix from left to right in column direction. Hence, the elimination graph cannot be deployed to get the nonzero pattern on time.

2.3.2 Numeric Factorisation

Numeric factorisation is to find the solution to the sparse triangular system. As the topological order, i.e., χ, has been computed in the symbolic analysis, the unknown vector x can be easily computed. This is obvious when we write the equations consisting of a lower triangular solve. Now we can write the entire Gilbert-Peierl's algorithm starting with an identity lower triangular matrix L and a full-zero upper triangular matrix U, shown in Algorithm 3.

Algorithm 3 Numeric Factorisation of GP

```
1:  L = speye(n)                                   ▷ Sparse identity matrix of order n
2:  U = sparse(n,n)                                ▷ All zero sparse matrix
3:  for k = 1 : n do
4:      x=A(:,k)                                   ▷ The k^th column to perform
5:      for j ∈ χ do
6:          for i = j + 1 : n do
7:              if L(i, j) ≠ 0 then
8:                  x(i) = x(i) − L(i, j) ∗ x(j)
9:              end if
10:         end for
11:     end for
12:     U(1 : k, k) = x(1 : k)                     ▷ Upper triangular
13:     L(k : n, k) = x(k : n)/U(k, k)             ▷ Lower triangular
14: end for
```

2.4 Forward and Backward Substitution

One of the largest advantages of LU decomposition is that once the decomposition result L and U is found, only the triangular systems are needed to solve to get the unknown vector x, by applying forward and backward substitution.

For a linear system and its LU decomposition

$$A\boldsymbol{x} = \boldsymbol{b} \tag{8}$$

$$A = LU \tag{9}$$

$$LU\boldsymbol{x} = \boldsymbol{b} \tag{10}$$

we can first substitute $\boldsymbol{U}\boldsymbol{x}$ with a vector $\boldsymbol{y}$

$$\boldsymbol{L}\boldsymbol{y} = \boldsymbol{b} \tag{11}$$

Here, L is the lower triangular matrix. Hence the forward substitution can be applied to solve this system for $\boldsymbol{y}$. Once $\boldsymbol{y}$ is obtained, the backward substitution can be implemented to solve for $\boldsymbol{x}$.

$$\boldsymbol{U}\boldsymbol{x} = \boldsymbol{y} \tag{12}$$

Then, if the right-hand side $\boldsymbol{b}$ changes, only one lower triangular system and one upper triangular system are required to compute to obtain the new solution. Unlike Gaussian elimination, there is no need to compute the main LU decomposition again.

To further illustrate, for the lower triangular system given in Equation 11, we can write them as

$$\begin{cases} l_{11}y_1 & & & = b_1 \\ l_{21}y_1 + l_{22}y_2 & & & = b_2 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1}y_1 + l_{n2}y_2 + \cdots + l_{nn}y_n & & & = b_n \end{cases} \tag{13}$$

To solve this lower triangular system, Equation 14 can be used, which solves $\boldsymbol{y}$ from y_1 to y_n. Here, L is the unit lower triangular, i.e., $l_{ii} = 1$, so it requires no division. Similarly, for backward substitution, Equation 15 is used, which solves in the reverse order of forward substitution from y_n to y_1. However, backward substitution requires the division operation on the diagonal element.

$$\begin{cases} y_1 = b_1 \\ y_i = b_i - \sum_{j=1}^{i-1} l_{ij}y_j \end{cases} \tag{14}$$

$$\begin{cases} x_n = \dfrac{y_n}{u_{nn}} \\ x_i = \dfrac{y_i - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}} \end{cases} \tag{15}$$

For the time complexity, both forward and backward substitution require n divisions, $\frac{n^2-n}{2}$ summations and $\frac{n^2-n}{2}$ multiplications. Therefore, the total number of operations is $2n^2 - n$ and the time complexity is $\mathcal{O}(n^2)$.

3 Implementation

The factorisation phase of the LU decomposition is implemented on the CPU and the solving phase is implemented in the FPGA. In this project, the kernel is implemented in the Alveo card U280, taking the advantages of HBM to increase the performance.

3.1 HLS Configuration

- *# pragma HLS PIPELINE II=1*
  Pipelining is a technique for increasing instruction level parallelism in an algorithm's hardware

implementation by overlapping distinct phases of operations and functions. Pipelining can be enabled by the *PIPELINE* pragma. It reduces the initial interval (II) for a loop or even function by allowing the concurrent execution of the different operations. A pipelined loop can start processing new inputs every II clock cycles.

- *# pragma HLS DEPENDENCE variable = X type = inter dependent = false*
  Pipelining can be prevented by the dependencies. There are mainly two types of dependencies: *loop-independent dependence* and *loop-carried dependence*.

- *# pragma HLS ARRAY_PARTITION variable = Xwork type = block factor = 32 dim = 2*
  An array that is implemented in the BRAM may subject to the limit port access, which prevent the parallel access to the same array. A dual-port RAM can also allow two simultaneous access in one clock cycle.Therefore, arrays which are implemented as memory or memory ports can frequently create performance bottlenecks.

  With array partition, it is easier to implement the vectorisation optimisation, which typically requires parallel access to the same array.

- *# pragma HLS UNROLL factor = 32*
  Loop unrolling can create multiple separate operations instead of a single set of operations. The *UNROLL* pragma changes loops by duplicating the loop body in the RTL design, allowing part of or entire loop iterations to run in parallel. By default, loops in C/C++ functions are kept rolled. The *UNROLL* pragma can be used to increase data access and throughput.

3.2 Vectorisation

Vectorisation is the process that converts an algorithm from a scalar implementation, in which each pair of operands is processed separately, to a vector computation, in which a single instruction can refer to a vector. Vectorisation tries to optimise the throughput of a program and utilise a target device to its maximum or near-maximum potential. This is typically realised by coalescing access to memory as well as using multiple banks concurrently [14]. Vectorising the memory access can be achieved to read multiple array indices as a single and wider data word. For applications which are memory-bound, this can help improve the memory bandwidth and improve the throughput for the overall application.

3.2.1 SIMD

Modern CPUs have introduced direct support for vector operations through SIMD (Single Instruction and Multiple Data) [15]. A CPU with 512-bit registers, for example, can retain 16 32-bit single precision numbers (*float* type in C/C++) and do a single computation 16 times faster than performing a single instruction. In the example shown in Figure 2, the 64-bit data can carry just one 64-bit double precision floating point number or two 32-bit single precision floating point numbers, but the standard add instructions can only add one pair of numbers. In contrast, the AVX2 (Advanced Vector Extensions 2) instruction set, which was announced in 2013 and is currently widely used on commodity hardware, introduces a 256-bit vector register. This enables for the simultaneous addition of four double precision floating point numbers or eight single precision floating point values. It also emphasises another contrast with the scalar register, which will simply squander the extra space by doing single addition at a time.

To further exploit the vectorisation, data structure alignment should be done to ensure the vectorisation speed-up. Alignment is a property of a memory address, expressed as the numeric address modulo of powers of two. It is recommended to align any data type to a power of a and minimum 32 bits. Misaligned memory access can possibly incur large performance losses.

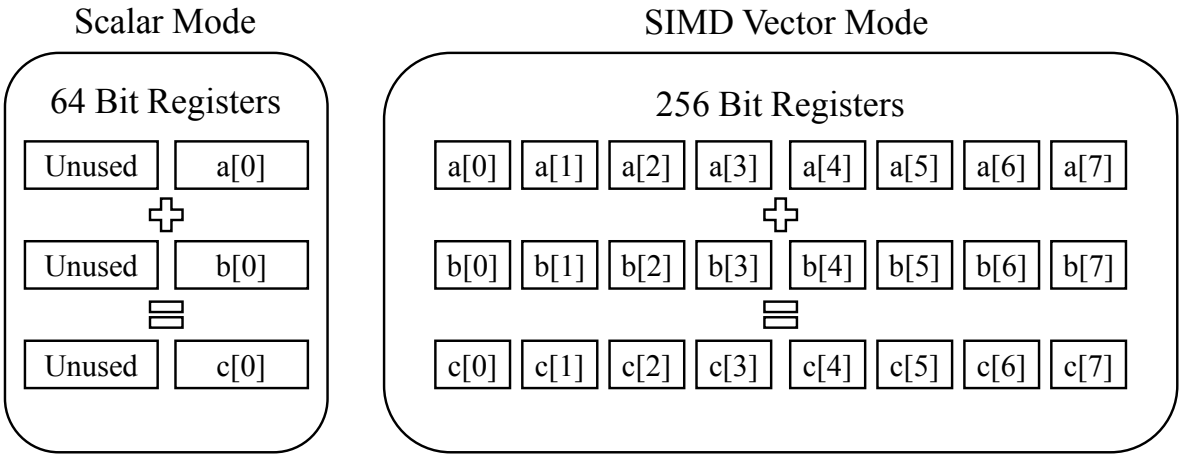


Figure 2: Illustration of SIMD

3.2.2 Loop Unrolling

Vectorisation is a well explored optimisation technique for performance improvement and has been explored for FPGAs as well. Vectorisation on FPGAs is similar to, but not exactly the same as that in CPUs. On FPGAs, Vectorisation mainly refers to both pipelining and unroll.

Loop unrolling can create multiple separate operations instead of a single set of operations. The *UNROLL* pragma changes loops by duplicating the loop body in the RTL design, allowing part of or entire loop iterations to run in parallel. By default, loops in C/C++ functions are kept rolled. Synthesis builds the logic for one iteration of the loop when loops are rolled, and the RTL design executes this logic in order for each iteration of the loop. The loop induction variable specifies the number of iterations for which the loop should be run. Logic inside the loop body, such as break conditions or changes to a loop exit variable, can also affect the number of iterations. The *UNROLL* pragma can be used to increase data access and throughput.

The *UNROLL* pragma allows the loop to be unrolled completely or partly. For each loop iteration, completely unrolling a loop makes a copy of the loop body in the RTL, allowing the complete loop to be performed concurrently. On the contrary, the partial loop unrolling is specified by a factor *N*, resulting in N copies of the loop body and a reduction of loop iterations.

In partial loop unrolling, it does not require N to be an integer which is a factor of the maximum loop iteration count. The Vitis compiler will automatically add an exit check to guarantee that partly unrolled loops are functionally equivalent to the original loop.

In this project, to realise the parallel solving for multiple right-hand side vectors, the forward and backward solving loop can be parallel.

4 Results

The final design is implemented and tested in the Alveo card U280. The CPU benchmark is performed on Intel i7-8850H.

Table 1 shows the properties of the tested circuit matrices and Table 2 lists the benchmark results on CPU. Here, we only list the factorisation time excluding the preprocessing analysis time, and the right-hand-solving time.

Figure 3a shows the benchmark results of the solving phase on CPU. The red marker shows the solving time takes per rhs when solving 10 rhs together. We can see that with SIMD applied, solving of multiple right-hand side vectors takes shorter time than proportional value. Figure 3b shows the solving

Table 1: Property of Matrices

Matrix	Order	NNZ	Sparsity	Pattern Symmetry	Numeric Symmetry
rajat11	135	665	3.65%	89.10%	63%
rajat14	180	1475	4.55%	100%	2.50%
rajat05	301	1250	1.38%	77%	70.60%
oscil_dcop_01	430	1544	0.84%	97.60%	69.80%
fpga_dcop_01	1220	5892	0.40%	81.80%	27.30%

Table 2: Benchmarking on CPU (Intel i7-8850H)

Matrix	Factorisation		Solve 1 rhs		Solve 10 rhs*	Speedup per rhs
	UMFPACK (µs)	KLU (µs)	UMFPACK (µs)	KLU (µs)	KLU (µs)	KLU
rajat11	127.5	40.9	22.48	2.8	13	2.15
rajat14	215.5	80.6	49.52	6.1	23.4	2.61
rajat05	263	119.7	61.5	10.2	27.4	3.72
oscil_dcop_01	765.3	164.3	64.76	11.7	41.6	2.81
fpga_dcop_01	1151.8	221.9	87.14	21	156.5	1.34

* Right hand side

time FPGA takes for per rhs when solving 10 rhs simultaneously (Time for data transfer time from global memory excluded). As the FPGA solves multiple right hand sides truly at the same time, the single right hand side solution is not displayed. We can see that for smaller matrices, FPGA tends to take longer time to solve per right hand side vectors than CPU. However, when the matrices becomes larger, FPGA tends to be faster than CPU, with a speedup of about 1.2×.

In total, when solving multiple right-hand sides vectors, FPGA can take advantages of its high throughput and lead to faster solving time per right-hand side. In addition, the power consumption of FPGAs is much smaller than CPU as the clock speed of FPGA is about 20 times slower than CPU, which decreases the cost of power supply and thermal management solutions in the system [16].

5 Conclusion and Future Work

5.1 Conclusion

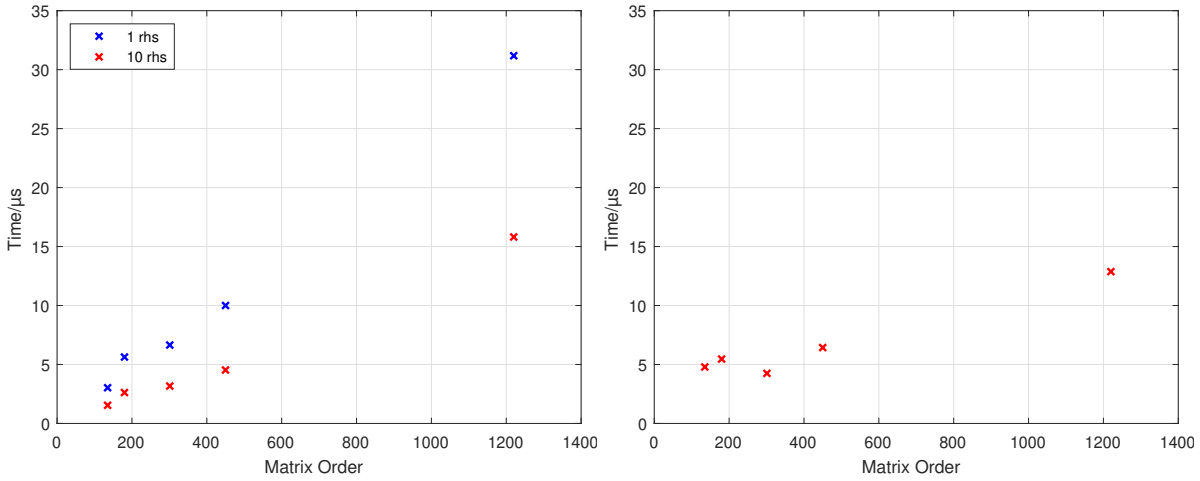
In this project, a parallel CPU+FPGA based architecture is proposed for the acceleration of LU decomposition for SPICE engine. While the preprocessing and analysis, the solving phase is implemented on FPGAs.

For preprocessing, BTF method is implemented to permute the matrix into several block triangular matrices in the diagonal, which help reduce the total factorisation steps. AMD method is deployed to help reduce the fill-ins. For factorisation, the left-looking algorithm is implemented to reduce the total factorisation steps.

On FPGA side, pipelining technique is implemented in every loop which is possible. Vectorisation is deployed for the parallel solving of multiple right-hand-side vectors. To ensure a sufficient memory access to arrays, array partition technique to break an array into block sub-arrays, implemented in multiple small memories or multiple registers, which provide more ports. This can prevent II dependencies and improve latency from 8 cycles to only 1 cycle.

5.2 Future Work

There are a number of aspects that can be investigated further in regard to the points mentioned in this thesis.



(a) Results on CPU (b) Results on FPGA

Figure 3: Timing on CPU and FPGA (Time in per rhs solve)

- **Algorithms**: More parallelisation can be done to further achieve a better datapath, especially for those smaller matrices. Parallel access to the same arrays can be further improved.
- **Hardware**: Still a lot resources remains in the current FPGA and additional logic can be implemented in the current architecture. The global memory bandwidth has not been fully exploited.
- **Integration**: Currently, we only build a sparse linear system solver. There are still a lot needed to be done to integrate the sparse linear system solver into a real circuit simulator.
- **Comparison**: More work can be done to compare the performance of the FPGA solver against that of the multi-core processors and GPUs with an equivalent parallel solver.

6 References

[1] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: Klu, a Direct Sparse Solver for Circuit Simulation Problems," *ACM Trans. Math. Softw.*, vol. 37, no. 3, Sep. 2010, ISSN: 0098-3500, DOI: 10.1145/1824801.1824814.

[2] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, ISSN: 0098-3500, DOI: 10.1145/2049662.2049663.

[3] J. Kwack, G. Bauer and S. Koric, "Performance test of parallel linear equation solvers on Blue Waters–Cray XE6/XK7 system," English (US), 2016.

[4] X. S. Li and J. W. Demmel, "SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems," *ACM Trans. Math. Softw.*, vol. 29, no. 2, pp. 110–140, Jun. 2003, ISSN: 0098-3500, DOI: 10.1145/779359.779361.

[5] A. Pothen and C.-J. Fan, "Computing the Block Triangular Form of a Sparse Matrix," *ACM Trans. Math. Softw.*, vol. 16, no. 4, pp. 303–324, Dec. 1990, ISSN: 0098-3500, DOI: 10.1145/98267.98287.

[6] R. Sargent and A. Westerberg, "SPEED-UP in Chemical Engineering Design," *Transactions of the Institution of Chemical Engineers*, vol. 42, p. 190, Jan. 1964, DOI: 10.13140/2.1.2585.6001.

[7] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972, DOI: 10.1137/0201010, eprint: https://doi.org/10.1137/0201010.

[8] D. J. Rose and R. E. Tarjan, "Algorithmic Aspects of Vertex Elimination," in *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1975, pp. 245–254, ISBN: 9781450374194, DOI: 10.1145/800116.803775.

[9] M. Yannakakis, "Computing the Minimum Fill-In is NP-Complete," *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 1, pp. 77–79, 1981, DOI: 10.1137/0602010, eprint: https://doi.org/10.1137/0602010.

[10] A. George and J. W. Liu, "The Evolution of the Minimum Degree Ordering Algorithm," *SIAM Review*, vol. 31, no. 1, pp. 1–19, 1989, DOI: 10.1137/1031001, eprint: https://doi.org/10.1137/1031001.

[11] P. R. Amestoy, T. A. Davis and I. S. Duff, "Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm," *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 381–388, Sep. 2004, ISSN: 0098-3500, DOI: 10.1145/1024074.1024081.

[12] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006, DOI: 10.1137/1.9780898718881, eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718881.

[13] J. R. Gilbert and T. Peierls, "Sparse Partial Pivoting in Time Proportional to Arithmetic Operations," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 5, pp. 862–874, 1988, DOI: 10.1137/0909058, eprint: https://doi.org/10.1137/0909058.

[14] J. Kalomiros, S. W. Nabi and W. Vanderbauwhede, "Automatic Pipelining and Vectorization of Scientific Code for FPGAs," *International Journal of Reconfigurable Computing*, vol. 2019, p. 7 348 013, 2019, ISSN: 1687-7195, DOI: 10.1155/2019/7348013.

[15] Intel, "Intel® C++ Compiler Classic Developer Guide and Reference," Intel Corporation, Tech. Rep., version 2021.5, 21st Jun. 2021.

[16] K. Subramaniyam, "Proven Power Reduction with Xilinx UltraScale FPGAs," Xilinx, Inc, Tech. Rep. WP466, version 1.1, 15th Oct. 2015.