

Matrix Computation Acceleration using FPGAs

M.Tech Project Report I

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Aashish Tamrakar

(Roll No. 193079034)

Under the guidance of

Prof. Sachin Patkar



Department of Electrical Engineering
Indian Institute of Technology, Bombay
2021

Dissertation Approval

This dissertation entitled

Matrix Computation Acceleration using FPGAs

by

Aashish Tamrakar

(Roll No. : 193079034)

is approved for the degree of

Master of Technology in Integrated Circuit and Systems specialization

(Examiner)

(Examiner)

(Chairperson)

Prof. Sachin Patkar
Dept. of Electrical Engineering

(Supervisor)

Date:

Place:

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this report. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: August 31, 2021

Aashish Tamrakar
(193079034)

Acknowledgement

I would like to express my deep gratitude to Prof. Sachin Patkar for his kind guidance all through. I also extend thanks to Anurag Choudhury, Yogesh Mahajan, M Abhijna Anand, Mini K. Namboothiripad and Mandar Datar for their constant help and support.

Aashish Tamrakar
Electrical Engineering
IIT Bombay

Contents

List of Figures	vi
1 Introduction	1
1.1 Motivation	1
1.2 The Project	2
2 Literature Review	3
3 Preliminaries	5
3.1 Sparse Matrix Data Structures	5
3.2 Sparse LU Decomposition	6
3.2.1 AMD Ordering	7
3.3 Gilbert-Peierls' Algorithm	8
3.4 Symbolic Analysis	10
4 Parallelizing Sparse LU Decomposition	12
4.1 Level of Granularity	12
4.2 The Hardware Architecture	13
4.3 Pivoting	14
5 Hardware Design	15
5.1 Top Level Design	15
5.2 Hardware Pivoting	16
5.3 Block RAM Units	16
5.4 Processing Elements	17
5.4.1 Multiply and Accumulate Unit	18
5.4.2 Divider Unit	19
5.5 Connection Box	19
6 Scheduler Tool	20
6.1 Input Processing	20
6.2 Symbolic Analysis	20

6.3	Generating Computational Flow Graph	22
6.4	Memory Address Assignment	23
6.5	Scheduling	24
6.5.1	Finding Schedulable nodes	24
6.5.2	Scheduling Table	25
6.5.3	An Example of Scheduling Process	27
7	Future Work	30
	Bibliography	

List of Figures

1.1	The overall flow of the project	2
2.1	NOC based hardware configuration (in [1])	3
2.2	Hardware Architecture used by Nechma [2]	4
3.1	Storage formats for sparse matrices	6
3.2	Algorithms for Sparse LU Decomposition	7
3.3	Effect of AMD ordering on the factorization of the 135×135 Matrix . .	8
3.4	Naming conventions used in algorithm 1	9
3.5	Non-zero pattern in LU decomposition	10
4.1	Overview of the proposed hardware architecture	13
5.1	Hardware architecure	15
5.3	Timing diagram for understanding the operation of Quad port BRAM .	17
5.2	BRAM unit with 4 time multiplexed ports	17
5.4	Generation Clock and Select Signals	18
6.1	Non-zero pattern in LU decomposition	21
6.2	Example of Symbolic Analysis	22
6.3	Example showing multiple possible computation flows for the single MAC element	23
6.4	Computational flow graph for example matrix shown in figure 6.2a . . .	23
6.5	Resource allocation tables used by scheduler	26
6.6	Selecting write ports	27
6.7	Selecting operations	28
6.8	Final allocation	29

Abstract

Solving the sparse linear system is one of the most critical steps in many scientific applications such as circuit simulation, power system modeling, and computer vision. These operations, especially circuit simulations, are iterative in nature and require tens of thousands of sparse system solves. The circuit simulation consists of two phases per iteration: model evaluation followed by asymmetric sparse linear system solve. The model evaluation phase is highly parallelizable unlike the subsequent phase, which involves highly sparse and circuit dependent asymmetric matrices. The project presents a scalable FPGA based LU solver system geared towards the matrices that arise in circuit simulations. The project leverages the fact that the structure of the sparse system remain the same during the entire simulation and hence can be analyzed symbolically only once to generate an execution schedule and use it for further iterations. The project integrates static pivoting and symbolic analysis to generate task graph which is the scheduled using the priority list based ASAP strategy. The software toolchain also provides option to generate and synthesize the required hardware.

Chapter 1

Introduction

1.1 Motivation

Sparse LU decomposition has been widely used to solve sparse linear systems of equations found in many scientific and engineering applications, such as circuit simulation, power system modeling and computer vision. Applications such as circuit simulation typically requires several thousand iterations of solving the linear system at different simulation steps. Hence the solving the sparse linear system is one of the most critical steps in circuit simulation. However it is a computationally expensive and difficult to parallelize on the traditional processing systems due to the highly sparse and instance dependent nature of the data which causes sparse algorithms to spend significant time data handling.

Naturally, an extensive research has been conducted on expediting the sparse LU decomposition process for range of computing platforms including the FPGA based systems. With the recent advancements in the fields of FPGA many researchers have developed accelerated solutions for specific scientific applications where matrices are symmetric and are diagonally dominant. Such problems are relatively easy to solve and parallelize. But in many applications including non-linear time domain circuit simulation, matrices do not follow a particular patterns. However such applications can leverage the fact that the underlying matrix structure, that is the locations of the non-zero elements, remains same for many iterations and only the numerical values are different in each iteration hence the data manipulation steps also remains the same. Overall performance can be boosted by sharing this manipulation steps. Such methods are already being used in many specialized libraries, like KLU (a direct sparse solver for circuit simulation problems). These methods can leverage further more by parallelizing a large number computation tasks using an FPGA.

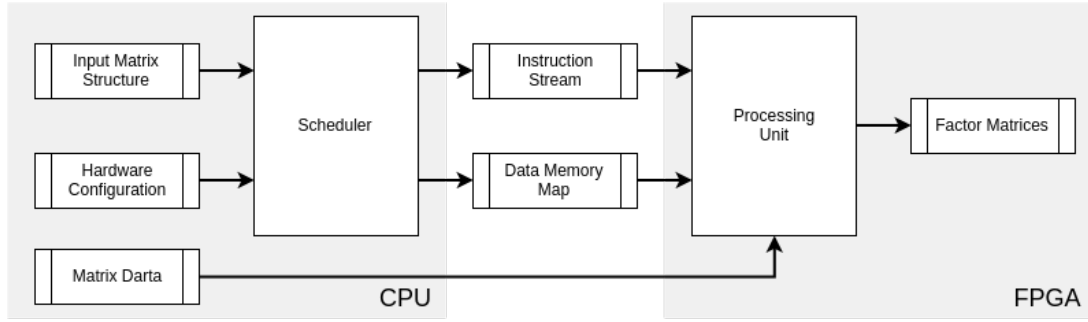


Figure 1.1: The overall flow of the project

1.2 The Project

The central goal of the project is to implement a scalable LU solver of the sparse matrices, especially the asymmetric matrices which arise in circuit simulation problems. The project is divided in two major sections: The scheduler and the hardware.

The scheduler C++ tool which accepts the target circuit matrix and generates the execution schedule based on the location of non-zero elements in the matrix. The schedule generated then can be converted into hardware instructions along with matrix data. Since the generated schedule does not depend on the actual values, it can be used to solve subsequent simulation stages.

The hardware processor is a set of deeply pipelined floating point multiply and accumulate (MAC), divider units and on-chip block memory units (BRAM). From here on the collection of MAC units and divider units is referred as Processing Elements (PE). The number of BRAMs and PEs is configurable and can be adjusted according to the need and available FPGA resources. The hardware accepts the data and instruction set generated from the scheduler tool to find the values of the factor matrices.

This report is divided in three major sections:

- Preliminaries and literature review: 2, 3
- Implementation
- Testing methodology and results

Chapter 2

Literature Review

A number of researchers have proposed various ideas for implementing an FPGA based LU solver. These methods vary largely with the degree of parallelism extracted from the problem and scheduling methods.

Kapre [1] *Kapre* (Figure 2.1) has proposed an FPGA accelerator for parallelizing the sparse matrix phase of the open-source Spice3f5 circuit simulator. The acceleration has been achieved by utilizing the symbolic analysis step of the KLU solver to generate the data flow graph. This graph is then mapped to the network of processing elements connected in Mesh grid. The mesh grid approach provides better scalability to the architecture. Also the hardware can be utilized for targeting other task graph based problems which can be scheduled statically.

../ReviewLit/

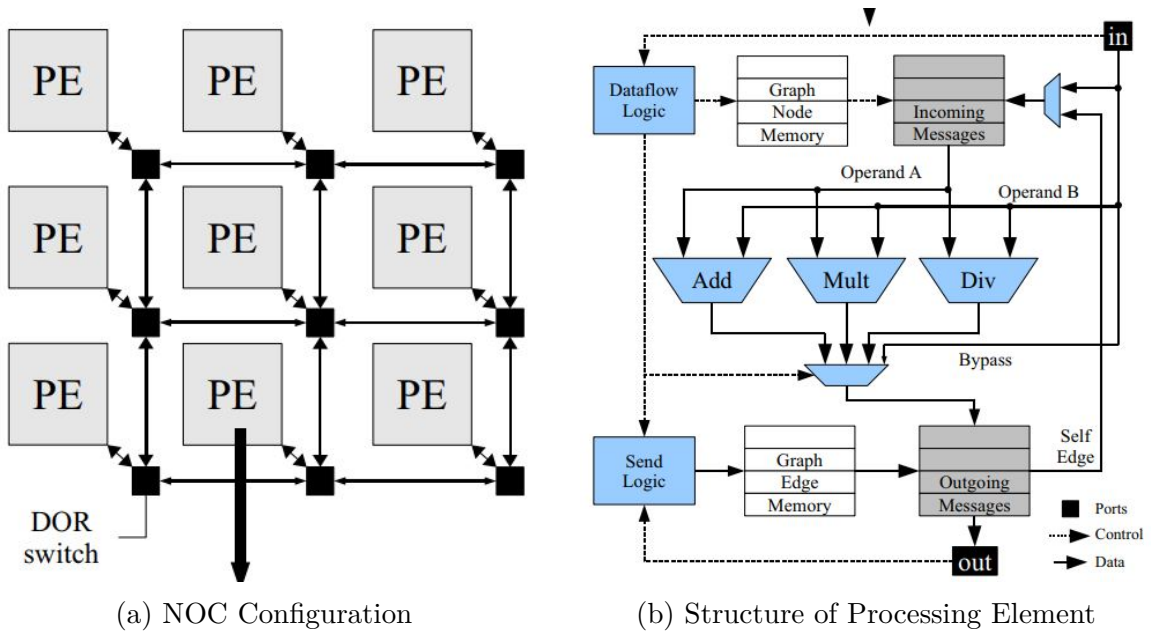


Figure 2.1: NOC based hardware configuration (in [1])

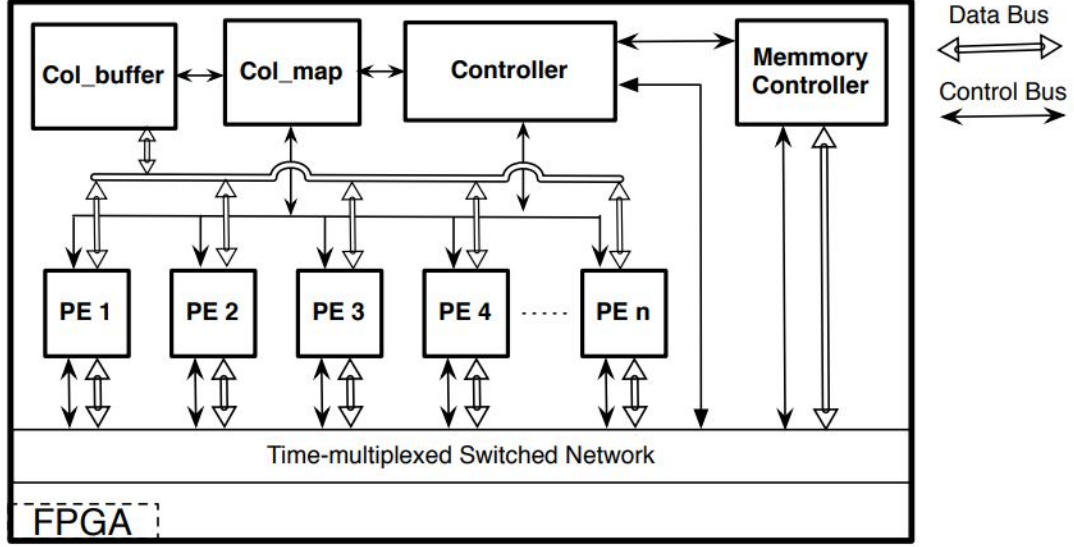


Figure 2.2: Hardware Architecture used by Nechma [2]

In [3], *Wu and Wei* have utilized the Gilbert-Peierls's algorithm to perform symbolic analysis to extract coarse-grain parallelism by tearing the matrices into small diagonal sub-problems. These sub-matrices are solved parallelly using an FPGA based shared-memory multiprocessor architecture called MPoPc. Each node consists of an Altera Nios processor attached to single-precision floating point unit. Reported acceleration are impressive but the benchmark matrices are smaller and results were not compared to previous FPGA implementations.

More recently, *Tarek Nechma and Mark Zwolinski* [2] have presented an approach to leverage medium-grain parallelism for LU decomposition based on the column based parallelism presented by the Gilbert-Peierls Algorithm. The architecture prepares a column execution schedule using the symbolic analysis and maps the computation of each column to processing element consisting a MAC unit, divider unit and BRAM. Individual column operations are scheduled using ASAP method. The columns are loaded from the main memory with the help of column buffers which is similar to the DMA engine. Each PE computes the column and then sends updates to the main memory. The average reported speedup over KLU is around 9x. However this does not include the average time spent on preprocessing in around 36x the factorization time.

The method used in this report is based on the approach similar to the Nechma's [2]. Our scheduler is geared to leverage the fine-grain parallelism using the set of deeply pipelined processing elements and low latency block RAMs available in FPGAs.

Chapter 3

Preliminaries

3.1 Sparse Matrix Data Structures

Sparse matrices are characterized by a relatively few non-zero elements. This property can be leveraged to store and operate these matrices efficiently. The three most popular sparse matrix storage formats are *Triplet Format*, *Compressed Column Sparse* and *Compressed Row Sparse*. Both time and memory space complexity of any sparse matrix operations are highly dependent on the underlying storage format and hence it is very important to carefully select the storage format depending on the operations to be performed.

Triplet Format

Triplet format is the simplest sparse matrix data structure consisting lists of all the non-zero entries in the matrix and corresponding row and Column indices.

Compressed Column Sparse

The CCS format consists of 3 arrays. 1) an array of non-zero entries with all the elements in the same column are listed one after the other 2) an array row indices corresponding to all the non-zero elements and 3) an array of the pointers where each column starts.

Compressed Row Sparse

CRS is similar to the CCS structure. It consists an array of the row pointers instead of the column pointers.

ELLPACK

The ELLPACK is a block based storage format and consists of two matrices, a data matrix and column index matrix. The size of both matrices is determined by the number of rows and the maximum number of non-zero elements in all the rows.

$\begin{bmatrix} 5 & 0 & -5 & 0 & 6 \\ 0 & 4 & 0 & -4 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 1 & -3 & 0 & -1 & 0 \\ 0 & 0 & -2 & 0 & 3 \end{bmatrix}$	<table><tr><td>Values</td><td>5</td><td>-5</td><td>6</td><td>4</td><td>-4</td><td>2</td><td>-</td><td>-3</td><td>-1</td><td>-2</td><td>3</td></tr><tr><td>Column Indices</td><td>0</td><td>2</td><td>4</td><td>1</td><td>3</td><td>0</td><td>0</td><td>1</td><td>3</td><td>2</td><td>4</td></tr><tr><td>Row Indices</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>2</td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td></tr></table>	Values	5	-5	6	4	-4	2	-	-3	-1	-2	3	Column Indices	0	2	4	1	3	0	0	1	3	2	4	Row Indices	0	0	0	1	1	2	3	3	3	4	4																																				
Values	5	-5	6	4	-4	2	-	-3	-1	-2	3																																																														
Column Indices	0	2	4	1	3	0	0	1	3	2	4																																																														
Row Indices	0	0	0	1	1	2	3	3	3	4	4																																																														
(a) Example Matrix	(b) Triplet format																																																																								
<table><tr><td>Values</td><td>5</td><td>2</td><td>1</td><td>4</td><td>-3</td><td>-5</td><td>-2</td><td>-4</td><td>-1</td><td>6</td><td>3</td></tr><tr><td>Row Indices</td><td>0</td><td>2</td><td>3</td><td>1</td><td>3</td><td>0</td><td>4</td><td>1</td><td>3</td><td>0</td><td>4</td></tr><tr><td>Column Pointers</td><td colspan="2">0</td><td colspan="2">3</td><td colspan="2">5</td><td colspan="2">7</td><td colspan="2">9</td><td></td></tr></table>	Values	5	2	1	4	-3	-5	-2	-4	-1	6	3	Row Indices	0	2	3	1	3	0	4	1	3	0	4	Column Pointers	0		3		5		7		9			<table><tr><td>Values</td><td>5</td><td>-5</td><td>6</td><td>4</td><td>-4</td><td>2</td><td>-</td><td>-3</td><td>-1</td><td>-2</td><td>3</td></tr><tr><td>Column Indices</td><td>0</td><td>2</td><td>4</td><td>1</td><td>3</td><td>0</td><td>0</td><td>1</td><td>3</td><td>2</td><td>4</td></tr><tr><td>Row Pointers</td><td colspan="3">0</td><td colspan="2">3</td><td colspan="2">5</td><td colspan="2">6</td><td colspan="2">9</td></tr></table>	Values	5	-5	6	4	-4	2	-	-3	-1	-2	3	Column Indices	0	2	4	1	3	0	0	1	3	2	4	Row Pointers	0			3		5		6		9	
Values	5	2	1	4	-3	-5	-2	-4	-1	6	3																																																														
Row Indices	0	2	3	1	3	0	4	1	3	0	4																																																														
Column Pointers	0		3		5		7		9																																																																
Values	5	-5	6	4	-4	2	-	-3	-1	-2	3																																																														
Column Indices	0	2	4	1	3	0	0	1	3	2	4																																																														
Row Pointers	0			3		5		6		9																																																															
(c) Compresses Column Sparse	(d) Compresses Row Sparse																																																																								

Figure 3.1: Storage formats for sparse matrices

Additional zeros are appended in a row to have uniform row length. This increases the storage overhead and can be eliminated with the advanced format such as sliced ELL-PACK format.

The triplet format is simple to create but difficult to use in most sparse matrix algorithms because of the absence of ordering of the elements. CCS is the most widely used format because of many sparse algorithms uses column frontend to proceed. Many sparse matrix libraries requires the row/column indices to appear in a ascending order for the ease of access and simplicity, although it is non necessary.

3.2 Sparse LU Decomposition

The matrix multiplication equation $A = LU$, with L unit lower triangular U upper triangular, and can be rewritten to give the expressions for L and U as follows:

$$U_{(i,j)} = A_{(i,j)} - \sum_{k=1}^{i-1} L_{(i,k)} U_{(k,j)} \quad (3.1a)$$

$$L_{(i,j)} = \frac{A_{(i,j)} - \sum_{k=1}^{j-1} L_{(i,k)} U_{(k,j)}}{U_{(j,j)}} \quad (3.1b)$$

The same Gaussian elimination can be used in a factorization of sparse matrices. Since the column vectors are sparse, some of the inner products reduces to zero and hence can be excluded for the update rule. However, since it may be necessary to pivot based on previously updated elements, a sparse Gaussian elimination algorithm cannot know the exact nonzero structure of these vectors in advance of all numerical computation.

Direct methods of sparse LU decomposition are widely divided into three categories. Left-looking, Right-Looking and Crout [4]. These methods are characterized by the fac-

tor update (Multiply and Accumulate) and normalization (Division) rules.

Left-Looking

Left-looking algorithms factorizes the matrix in column-by-column manner. Factors of current column are updated using the previously computed columns with equation $A_{(i,j)} = A_{(i,j)} - L_{(i,k)} \times U_{(k,j)}$, where $k = [1, \dots, \min(i, j)]$. Updated factors are then normalized with the pivot value.

Right-Looking

The Right looking algorithm first factorizes a column from the lower triangular part of a matrix, then uses the resulting non-zero elements of that column to update the affected components in the rest of the matrix by using the equation $A_{(i,k)} = A_{(i,k)} - L_{(i,j)} \times U_{(j,k)}$, where $k = [j + 1, \dots, N]$, j is the index of current factored column, and N is the column dimension of matrix A

Crout

Similar to the Left-looking algorithm, the Crout method performs updates with previously factored elements before normalizing a given vector. The difference is that the Crout method operates both on columns and rows, while the Left-looking algorithm only operates on columns. (See figure 3.2)

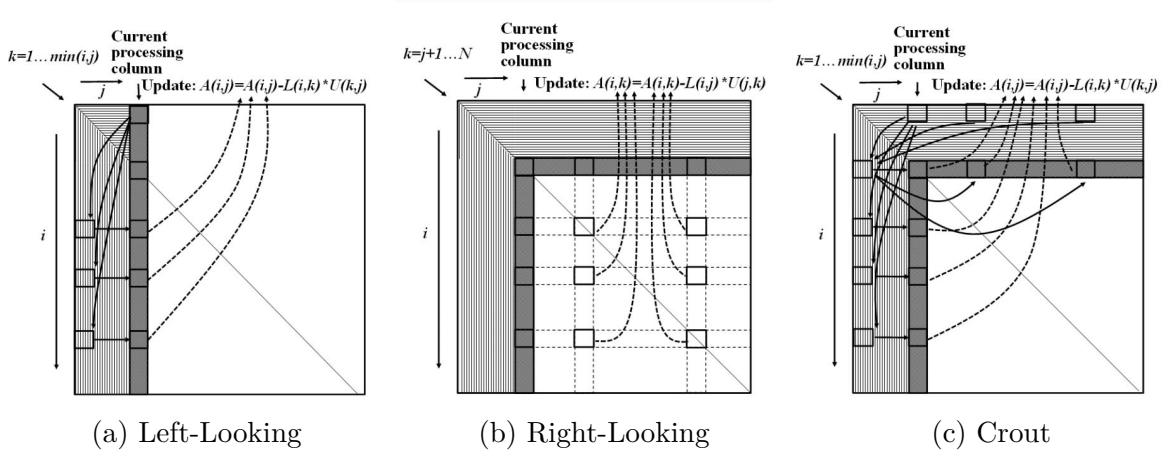


Figure 3.2: Algorithms for Sparse LU Decomposition

3.2.1 AMD Ordering

An Approximate Minimum Degree ordering algorithm (AMD) for pre-ordering a symmetric sparse matrix prior to numerical factorization is presented. The Cholesky factorization of ordered symmetric matrices tends to be sparser than that of the original ones. This method can be extended to asymmetric matrix, say A , by applying the AMD on $A + A'$.

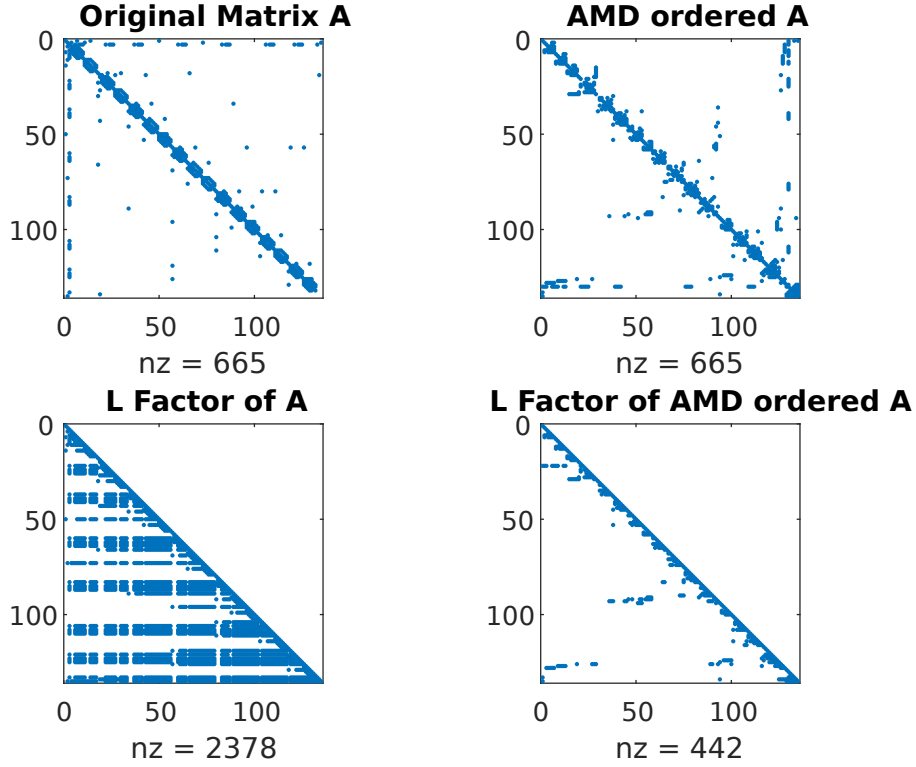


Figure 3.3: Effect of AMD ordering on the factorization of the 135×135 Matrix

Pre-ordering the matrices reduces the number of fill-ins in the factors and hence reduces the number of required floating point operations dramatically.

The figure 3.3 demonstrates the effect of AMD ordering on the *Rajat11*, a matrix of order 135×135 . As reduction in the *NNZ* increases significantly as the order increases.

A linear problem $Ax = b$ can be solved by solving the reordered problem where P is the AMD permutation matrix.

$$(P^T AP)(P^T x) = P^T b \quad (3.2)$$

3.3 Gilbert-Peierls' Algorithm

Gilbert-Peierls' algorithm [5] (1) provides a method to achieve LU decomposition with partial pivoting in time proportional to $O(flops(LU))$, i.e. number of floating point operations [5]. It is a left looking algorithm because it computes the k^{th} column of L and U using the previously computed $(k - 1)$ columns of L matrix. The notations used in the algorithm are as follows: j is the index of the column of L and U being compute. Unprimed variables mean the section of the matrix or vector above the row with index j . Primed variables represent the section of matrix or vector below the row index j including

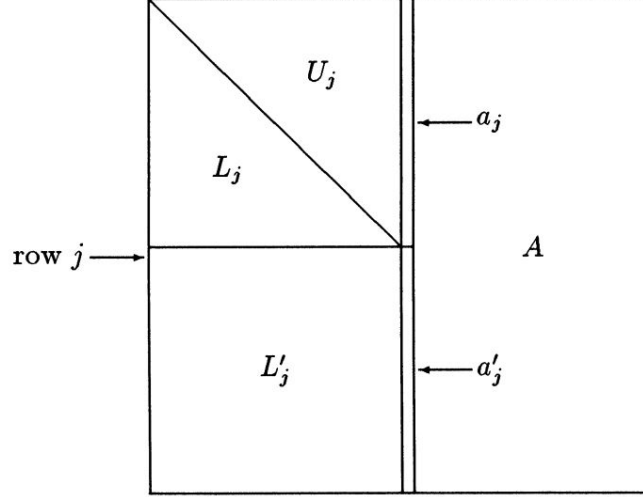


Figure 3.4: Naming conventions used in algorithm 1

the j^{th} row. thus vector $a_j = (a_{1,j}, a_{2,j}, \dots, a_{j-1,j})^T$ and vector $a'_j = (a_{j,j}, \dots, a_{n,j})^T$,

$$L_j = \begin{bmatrix} l_{1,1} & \dots & o \\ \vdots & \ddots & \vdots \\ l_{j,1} & \dots & l_{j-1,j-1} \end{bmatrix} \quad L'_j = \begin{bmatrix} l_{1,j} & \dots & l_{j,j-1} \\ \vdots & \ddots & \vdots \\ l_{n,1} & \dots & l_{n,j-1} \end{bmatrix}$$

The naming conventions in general are explained the figure 3.4.

Algorithm 1 Gilbert-Peierls Algorithm: A Column-Oriented LU Factorization

Precondition: A , a $n \times n$ asymmetric matrix

```

1  $L := I$ 
2 for  $j := 1$  to  $n$  do                                      $\triangleright$  Compute  $j^{th}$  column of  $L$  and  $U$ 
3   Solve  $L_j u_j = a_j$  for  $u_j$ 
4    $b'_j := a'_j - L'_j u_j$ 
5   Do Partial Pivoting on  $b'_j$ 
6    $u_{jj} := b_{jj}$ 
7    $l'_j := b'_j / u_{jj}$ 

```

The central idea of the Gilbert-Peierls' algorithm [5] is to solve the lower triangular system $Lx = b$ where L is a lower triangular matrix and x, b are sparse column vectors (Algorithm 3). This method requires to pre-compute the location of all the non-zero elements in the sparse column vector x , which is referred as *Symbolic Analysis*. This step has to repeated for all the n column of the matrix A .

3.4 Symbolic Analysis

Symbolic analysis is a process to determine the set of non-zero locations (χ) in solving the lower triangular system $L_j x = b$, where J_j is a unit diagonal lower triangular matrix representing only first $(j - 1)$ columns. Listing all the non-zero locations χ gives the numerical computation time proportional to the number floating point operations i.e. $O(f)$,

Algorithm 2 Gilbert-Peierls Algorithm: Solving a Dense Triangular System $L_j x = b$

Precondition: L_j is a dense lower triangular matrix, x, b are sparse column vectors

```

1  $x := b$ 
2 for  $k = 1$  to  $j - 1$  do
3    $x := x - x_i(l_{1,i}, l_{2,i}, \dots, l_{j-1,i})$ 

```

The algorithm 2 represents the process of solving a dense lower triangular system. In sparse systems some of the b entries and L_j entries are zero resulting in no change and hence can be eliminated. Algorithm 3 represents the sparse version of the algorithm

Algorithm 3 Gilbert-Peierls Algorithm: Solving Triangular System $L_j x = b$

Precondition: L_j is a lower triangular matrix, x, b are sparse column vectors

```

1  $x := b$ 
2 for each  $j \in \chi$  do
3   for each  $i > j$  for which  $l_{ij} \neq 0$  do
4      $x_i := x_i - l_{ij}x_j$ 

```

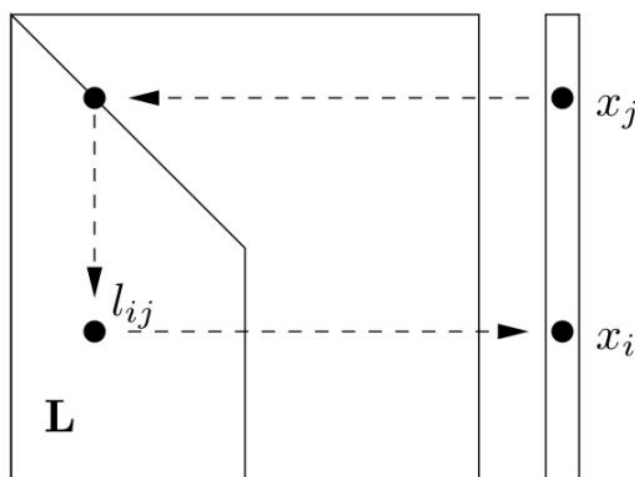


Figure 3.5: Non-zero pattern in LU decomposition

Lines 1 and 3 of the algorithm 2 suggests that the element of the result vector (x_{ij}) can become non-zero only if the corresponding element in the vector b (b_i) is non-zero or there exists a non-zero element $l_{i,j}$ where j is less than i and x_j is non zero.

$$(b_i \neq 0) \implies (x_i \neq 0) \tag{3.3a}$$

$$(x_j \neq 0) \text{ and } \exists i (l_{ij} \neq 0) \implies (x_i \neq 0) \tag{3.3b}$$

These two implications can be visualized using the figure 3.5. In the column factorization algorithms like Gilbert-Peierls, we know the locations of all the non-zero elements for the columns with indices lower than j we can determine the non-zero locations (χ) before solving for that column and for the entire lower triangular system.

Chapter 4

Parallelizing Sparse LU Decomposition

4.1 Level of Granularity

One of the most important steps in designing a parallel algorithm is to determine the level of granularity of the operations to be scheduled. The LU decomposition of a sparse matrix naturally has a fine-grain parallelism between individual arithmetic operations because of large number of zero elements. This granularity can be exploited effectively using a set using a set of small processing elements operating in parallel. Since the operations to be scheduled are large in number, the latency of the storage unit and throughput of PEs become the major contributing factors towards the performance of the entire system.

A medium grain parallelism can also be defined over the LU factorization process by dividing arithmetic operations into the sets of operations required to compute one column. Gilbert-Peierls algorithm provides the information about the columns required in computation of certain column. So we can process multiple columns in parallel. A directed acyclic graph (DAG) can be used to define dependency among the columns i.e. each node in the DAG represents a column and its parents represent the column required to compute the given column. The scheduler can schedule the column computation operations according to the graph. Such kind of parallelism is easy to extract using the Gilbert-Peierls algorithm [1]. The degree of parallelism increases with the number of columns in the matrix as more and more independent sets of columns can be found. However, this may cause load imbalance in cases where only a few floating point operations are required for an entire column operation.

We can also extract coarse-grained parallelism by dividing the columns into disjoint sets of dependent column and each set can be computed independently. This project utilizes the first approach of extracting fine-grain parallelism by scheduling individual MAC and divide operations. this kind of approach is ideal for network like systems where each node

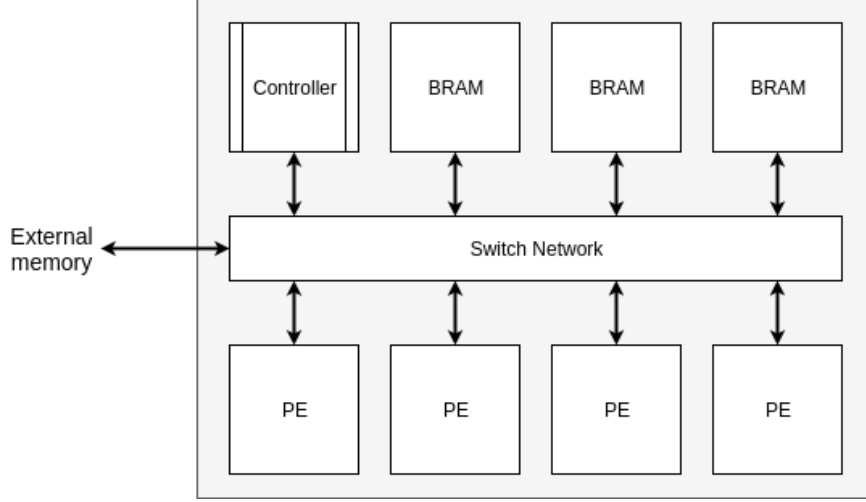


Figure 4.1: Overview of the proposed hardware architecture

has enough computational resources to compute set of dependent columns.

Our implantation is geared towards extracting fine grain parallelism. Modern FPGAs can support a considerable number of pipelined MAC and divide units along with the low latency on-chip BRAMs of quick data access. So naturally, FPGAs are well equipped for handling large number of basic tasks. The latency of arithmetic operations depends on the pipeline depth of processing elements and hence is fixed and known to the scheduler. The scheduler can leverage this knowledge to optimize memory operations.

4.2 The Hardware Architecture

The hardware should consists of a set of basic arithmetic processing units which can access local low latency memory in parallel to fully utilize the capabilities of an FPGA. At the same time it should be easily scalable according th requirements and available resources with. The on chip BRAMs have limited memory and may not be sufficient hence the hardware should have access to the external memory. Figure 4.1 shows the overview of proposed architecture. The switch network provides simple and easy access to all the BRAM ports and PE ports allowing to send operands from one source to multiple destinations simultaneously. Also it is easy to add new BRAMs and PEs to the system by extending the number of connections of the switch bar. The entire system can be controlled with very basic information such as switch box connection an operations to be performed at each BRAM and PE port and hence. The design philosophy is similar to the Very Long Instruction World (VLIW) processor architecture.

4.3 Pivoting

Numerical stability is also an important factor in solving linear systems. Pivoting method is used to handle the numerical instability that may arise because of numerical cancellation of diagonal element. There is a trade off between the stability and sparsity requirements for pivot selection are often contradictory as pivot selection changes the matrix structure and may increase number of fill-ins. Since our method expects that the matrix structure is remains the same, we can not implements the actual pivoting step in our design. This problem can be solved easily by replacing any tiny pivots with $\sqrt{\epsilon}\|A\|$ where ϵ is a machine precision and $\|A\|$ is the norm of the matrix A . (refer [2])

Chapter 5

Hardware Design

The general hardware design was discussed in the previous chapter. This chapter provides the detailed overview of the hardware design and implementation.

5.1 Top Level Design

The figure 5.1 shows the top level design of the proposed hardware architecture. The hardware consists of a set of pipelined MAC and divider units. The connection box controls the inputs and outputs of each unit using the instruction bitstream fed through the input FIFO. The entire bitstream need not be stored on the FPGA and the available memory can be used only for data. The factors are stored in the BRAMs. Also the data from the matrix A is required only once, hence it can be streamed too. This will save the data loading time of the matrix.

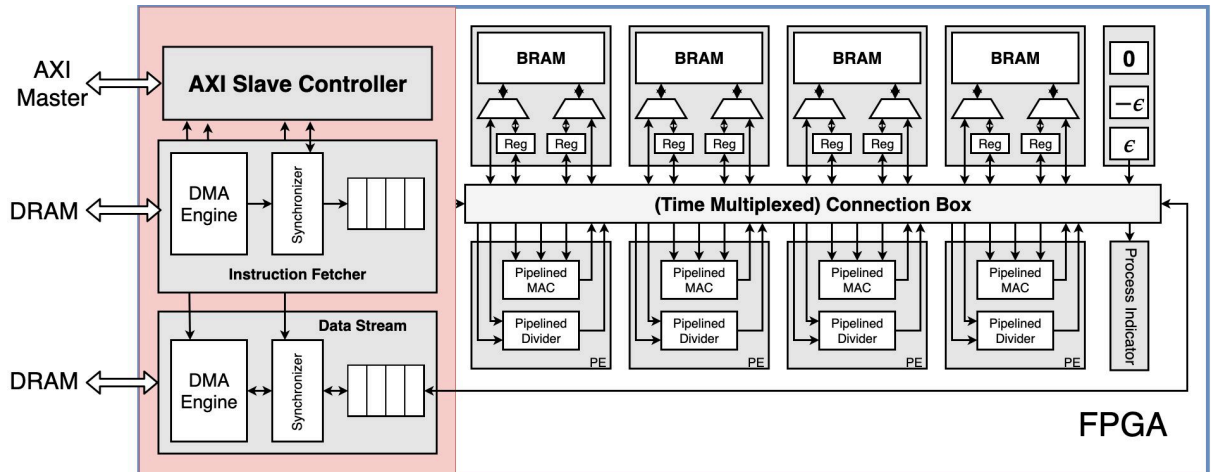


Figure 5.1: Hardware architecture

5.2 Hardware Pivoting

As discussed in earlier chapter, pivoting is very importing for the numerical stability of the solution. But it is not feasible to regenerate the schedule for each matrix as the pivoting depends on the actual value. In order to ensure the numerical stability, we have to replace the tiny pivots $|a_{ii}| < \sqrt{\epsilon}||A|| = \epsilon_0$ with the $\sqrt{\epsilon}$, where $||A||$ is the norm of the matrix A . The ϵ is called as machine precision. (e.g. 2^{-24} and 2^{-53} for single precision and double precision IEEE 754 formats). This is acceptable in practical terms as the SPICE linear system solution is used as part of Newton-Raphson iteration, and an occasional small error during the iterative process does not affect the integrity of the final solution. [6]. Norm of the matrix can be calculated in the symbolic analysis step.

The pivoting block in the hardware is controlled by the instruction stream too. The result of pivots are compared with the ϵ_0 and will be replaced at the output of the PE. This block is only required at the outputs of the MAC units. To further simplify the design we can ignore the comparison of mantissa. This allows us to use normal integer comparator to compare the magnitudes. This scheduler should generate an instruction bit suggesting the block to enable checking for the particular result.

5.3 Block RAM Units

Xilinx's Zynq 700 FPGA has true double port BRAMs, i.e. each BRAM can handle two read/write requests simultaneously provided there are is no write conflict in a single cycle. These blocks can operate at much higher clock frequency (roughly 2X) than the Processing Elements. So we can use time multiplexing the two available ports to emulate additional ports. Figure 5.2 shows the structure of a single BRAM unit used in the implementation. Two available ports are generated with the help of two clock with frequency ratio 2:1. These synchronous clocks are generated using the Xilinx's Clocking Wizard IP which uses on-chip PLLs.

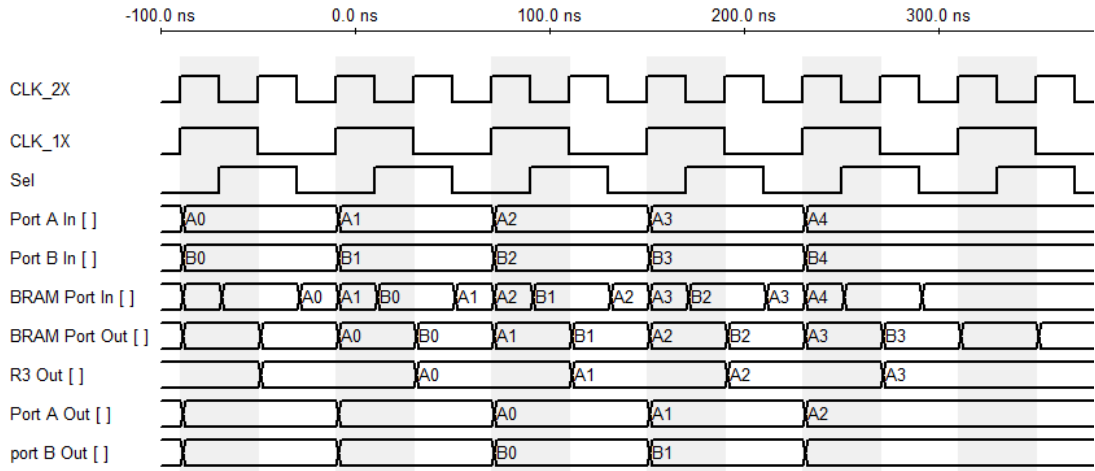


Figure 5.3: Timing diagram for understanding the operation of Quad port BRAM

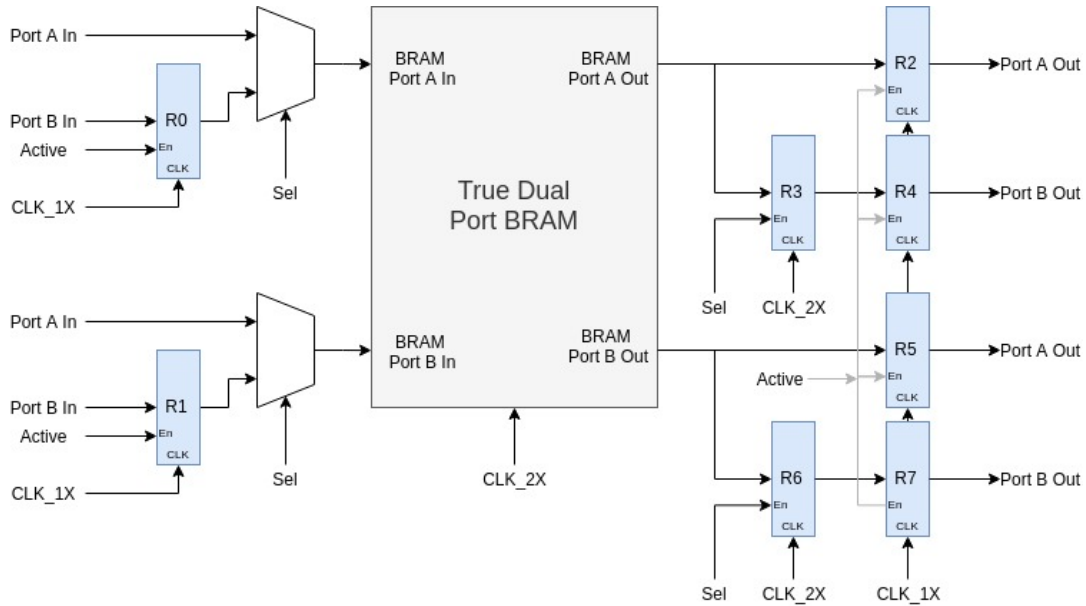


Figure 5.2: BRAM unit with 4 time multiplexed ports

The operation of the quad port BRAM used in the project is shown in the figure 5.3. Please note that the waveform is not time accurate and is just for explaining the operation process. The Sel signal is generated by sampling the slow clock at the falling edge of fast clock. The figure 5.4 shows the generation of select signals and associated clocks along with reset and active. The additional Active signal is generated to gate the clock signals for PEs to stall them when the next instruction is not available.

5.4 Processing Elements

All the processing elements are pipelined to achieve higher throughput. Pipelining allows scheduler to assign independent operations in multiple clock cycles provided the operands

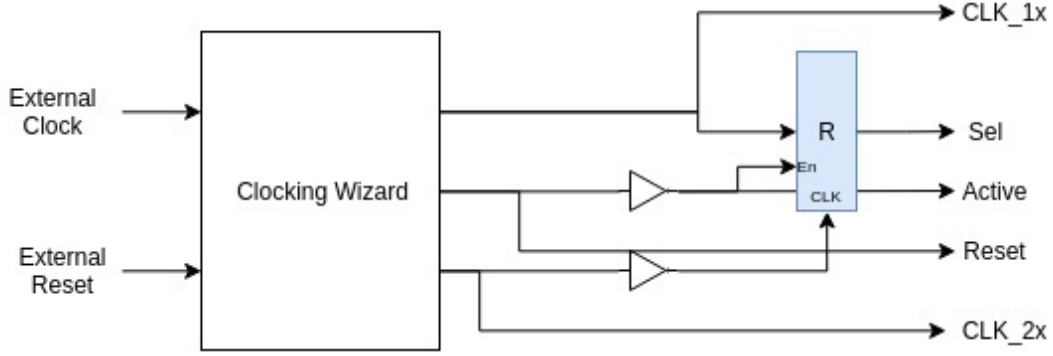


Figure 5.4: Generation Clock and Select Signals

are available. The depth of pipeline for PEs is dependent on the operating frequency and the number of resources available on the target FPGA. In general lower latency PEs should decrease the required number of clock cycles for the entire factorization process. But at the same time shallow pipelines have higher path delays and hence require slower clocks.

Xilinx provides readily available IPs for floating point operations. These IPs provide AXI Stream (AXIS) interface for both operands and result ports and can operate in both blocking and non-blocking modes. For the scope of this project, the schedule generated by the scheduler algorithm is cycle accurate and valid data is synchronized removing the need of blocking mode. This allows to reduce operation latency by reducing pipeline depth and hardware resources.

5.4.1 Multiply and Accumulate Unit

The LU factorization process requires only a floating point multiply and subtract operations ($Result = C - AB$). So we can remove the addition feature to save some of the resources. The Xilinx's floating point IP supports operations of format $Result = AB \pm C$. this issue can be easily resolved by inverting the sign bit of one of the multiplicands and using the multiply nad add unit. The Xilinx's MAC units can utilize on-chip DSP Slices to achieve higher performance.

$$C - AB \rightarrow (-A.B) + C$$

For the scope of this project all the MAC units are set to utilize maximum number of DSP slices since they are not being utilized in any other unit.

5.4.2 Divider Unit

The Xilinx Floating Point Divider IP utilizes some variant of the radix-2 SRT division algorithm and hence can not use DSP slices. These units are bulkier and hence have to be deeply pipelined to operate. The speed grade of the target FPGA is one of the most dominating factors in determining the depth of pipeline.

5.5 Connection Box

The connection box should allow connection from any output port to any input port for all the BRAMs and PEs. This can be achieved with multiplexers. All the ports of both BRAMs and PEs form the set of input signals for these multiplexers. The select signals are provided by the instruction register. The number of the multiplexers depends on the number of PEs, number of BRAMs and number of ports at each BRAM. These multiplexers can be pipelined to achieve desirable operating frequency. This allows some scalability to the system. The additional latency can be modeled as the delays of the PEs without affecting the scheduling algorithm. Since the location of all the elements is predetermined there is no need to transfer data from one BRAM to the other and hence we can eliminate connections from BRAM outputs to BRAM inputs.

Chapter 6

Scheduler Tool

The scheduler is a tool which accepts the information about the hardware configuration and matrix structure to generate a instruction stream and BRAM memory map for all the elements of input matrix as well as the generated factors. For a given circuit this tool has to be used only once. After that generated information can be used for subsequent iterations. The operation of the scheduler tool is divided onto three major phases:

1. Input processing, Symbolic analysis and Memory Allocation
2. Scheduling
3. Intersection stream and Memory map generation

6.1 Input Processing

The scheduler tool accepts the input matrix data in the Compressed Column Sparse (CCS) format. Also it requires the hardware configuration information as follows:

- Number of BRAM blocks
- Latency of all the units
- Number of MAC units
- Number of ports available at each BRAM
- Number of divider units
- Address depth of BRAMs

The preprocessing process ensures that the on-chip memory is sufficient to perform the operation.

6.2 Symbolic Analysis

The aim of symbolic analysis is to find the set of non-zero locations (χ). This can be achieved using the second part of Gilbert-Peierls' algorithm (Algorithm 3). The non-zero

elements in column i can be found using the following equations:

$$(b_i \neq 0) \implies (x_i \neq 0) \quad (6.1a)$$

$$(x_j \neq 0) \text{ and } \exists i (l_{ij} \neq 0) \implies (x_i \neq 0) \quad (6.1b)$$

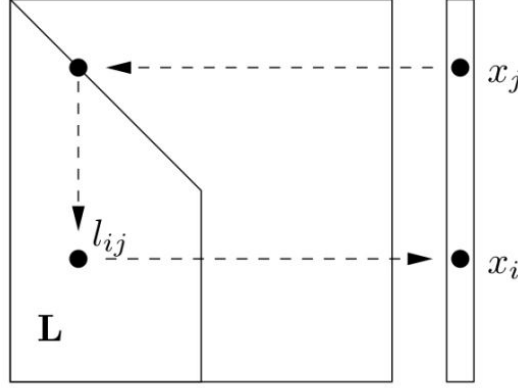


Figure 6.1: Non-zero pattern in LU decomposition

These two implications can be modelled as a traversal of a DAG representing the column dependance of the lower triangular section of graph A [2]. The set (S) of non-zero elements in any column (j) is given by

$$s = \bigcup_{x_{ij} \in a_j} Reach(x_{ij}) \quad (6.2)$$

where a_j is a column vector of matrix A and the $Reach(x)$ is defined as the location of non-zero elements in the x^{th} column of the factor matrix which below the diagonal of the matrix. The reach of each column then has to be updated as fill ins may have been added to the column of factor matrix. The location of fill-ins can be found by removing the non-zero locations in a_{ij} from S as follows:

$$fillIn(j) = reach(j) - \{i | x_{ij} \neq 0\} \quad (6.3)$$

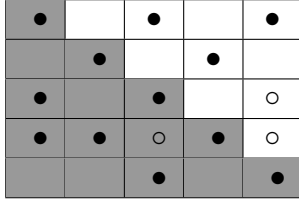
To illustrate the algorithm consider the matrix A shown in the following figure 6.2. The bullets (\bullet) represent non zero entries in the original matrix and the circles (\circ) represent the location of fill-ins. For example if we look at the column zero, it has non-zero elements at row indices 2 and 3. Therefore the $Reach(0) = \{2, 3\}$. It means that any column with column index greater than 0 having a non-zero element at row index 0 will have non-zero elements at the row indices 2 and 3. Since columns 2 and 4 satisfy the above condition

they have non-zero elements at row indices 2 and 3 in the factor matrix.

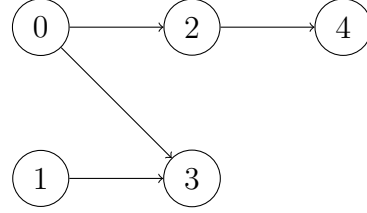
$$\begin{aligned}
NonZero(Col_2) &= Reach(0, 2, 5) \\
&= Reach(0) \cup Reach(2) \cup Reach(5) \\
&= \{0, 2, 3\} \cup \{2, 5\} \cup \{5\} \\
&= \{0, 2, 3, 5\}
\end{aligned}$$

and hence the new reach of column 2 is $\{2, 3, 5\}$ location of fill-ins in column 2 are

$$\begin{aligned}
fillIns(Col_2) &= Reach(2) - \{0, 2, 5\} \\
&= \{3\}
\end{aligned}$$



(a) Matrix with asymmetric non-zero pattern



(b) Column dependency graph

Figure 6.2: Example of Symbolic Analysis

6.3 Generating Computational Flow Graph

The computational flow graph is a directed acyclic graph of all the operations which are required to compute the factor matrix. Each node of the graph represents single arithmetic operation such as MAC and divide. The children nodes of a particular node represents the elements required to compute the give node and hence the children must be executed before executing the parent node. A calculation of single element may require multiple MAC operations and may or may not require divide operation (normalization) depending on the location of the element in the matrix. For example in the factorization of the example matrix (figure 6.2a) $U_{3,4} = A_{3,4} - U_{2,4}L_{3,0} - U_{0,4}L_{3,2}$. Since there are multiple atomic MAC operations and we can not execute them parallelly, the scheduling has to schedule one after the another. The scheduler can not determine which of the required multiplicands will be ready first using the flow graph. The task graph must represent all the possibilities for the execution of particular node (figure 6.3). This problem is solved by generating super node for all the AMC operations required for particular element.

All the diagonal elements of the L factor matrix are zero and hence do not require memory storage and can be omitted from the process. Also the elements of the original matrix (A) are used only once and occupy the memory space corresponding non-zero locations in the factor matrix. There are many ways to allocate memory addresses to matrices. Following are the three simple ways to allocate address locations:

Linear Address Allocation : All the elements in the matrix are arranged linearly one after the other. This type of Allocation reduces the data gathering overhead after the execution has finished but hinders the performance as most of the times all the elements in one column are allocated in the same BRAM and cause read congestions.

Circular Address Allocation : In this allocation policy the elements from the same column are circularly distributed across all the BRAMs.

6.5 Scheduling

The scheduling algorithm has to generate an execution schedule considering the hardware constraints and data dependencies. Since the hardware does not have any dynamic problem handling capabilities the schedule generated must be cycle accurate and static. The scheduler uses ASAP strategy to assign the execution nodes. A priority list is used to select the nodes from the pool of ready nodes. The priority of the nodes is defined as:

$$Priority(n) = \sum_{i \in Parents(n)} Priority(i) + \sum_{x \in tasks(n)} Delay(x) \quad (6.4)$$

where $Parents(n)$ is the set of nodes which are dependent on the node n and $tasks(n)$ is the set of operations required to evaluate the node n . This definition of the priority ensures that the node which has more number of overall dependent nodes will be prioritized by the greedy scheduling algorithm. The important steps in the algorithm are as follows:

6.5.1 Finding Schedulable nodes

The scheduler maintains two boolean state variables corresponding namely, *Dirty* and *Done* for each of the nodes. The set states of these variables indicate the following:

Dirty : The memory data corresponding to the node is outdated and must not be read

Done : All the computations corresponding to node are done

The set state of *Done* variable does not indicate that the data in the memory is accurate. *Dirty* variable must be checked before accessing the data. These state variables ensure

Algorithm 4 Priority List based Scheduling Process

Precondition: G , a computation flow graph for LU decomposition

```
1  $scheduledNodes := 0$ 
2  $readyNodes := G.leaves()$ 
3 while  $scheduledNodes < G.size()$  do
4   Update status of nodes retired in previous cycle
5   Update  $scheduledNodes$ 
6   Update the ready nodes list
7   Assign memory port to retiring nodes
8   Calculate the free BRAM ports for reading and writing
9   Select the most prior set of ready nodes which can be schedules in current cycle
10  Assign the memory memory operations
```

that the data being read always correct and also helps scheduler to properly manage the dependencies. At the start of each cycle, the scheduler has to find the set of ready nodes which can potentially form the execution assignment set for the cycle. such nodes are called as ready nodes.

The node is called ready when:

MAC Node : (Augend is not dirty or live) and at least one of addend pair of multipliers is in done state.

Divide Node : Both the operands must be in the done state.

Since the ready state of nodes depends on the state of children nodes. This list is updates using the set of retiring nodes as they are fewer in number and anyway have to visited to generate write commands.

The scheduler also maintains a list called as **Live Variables**, which represents the set of nodes whose correct values are available on the output ports of the PEs or BRAMs and hence can be read in current cycle.

Not all the nodes in the ready nodes are schedulable in current cycle. Some of operands corresponding the ready nodes may be dirty and hence have to removed from the schedulable node list. The list of live variables and the dirty state of operands is checked to ensure the schedulability of individual the nodes.

6.5.2 Scheduling Table

The scheduling algorithm uses a set of three tables (figure 6.5), assignment table, retirement table and memory operation table to keep track of all the assigned operation and determine the next set assignments. The scheduler generates cycle accurate simulation

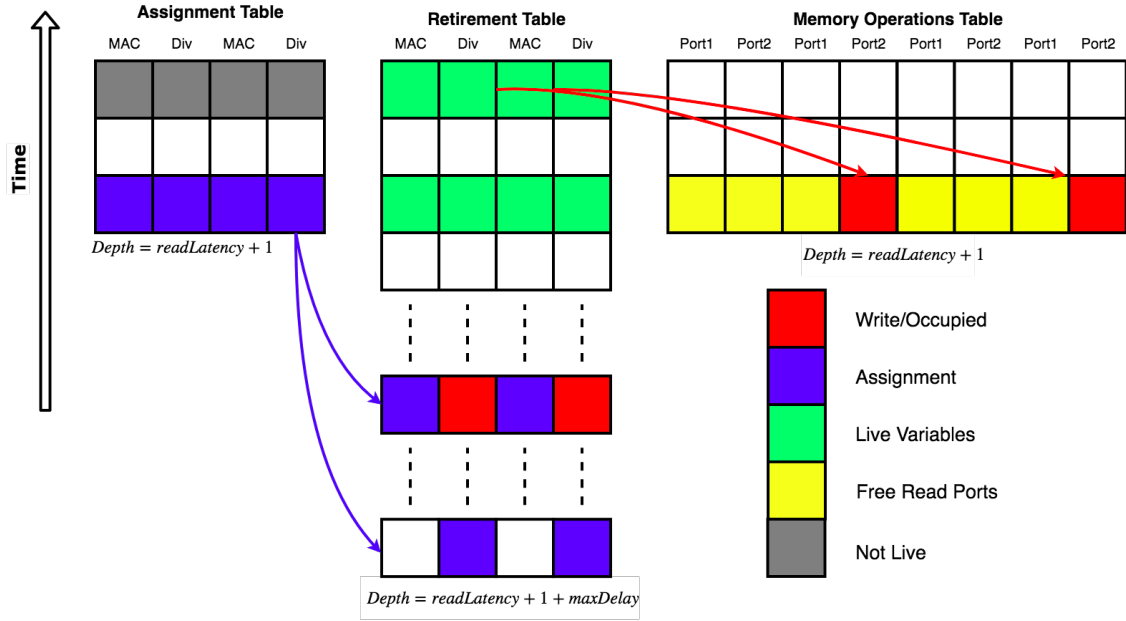


Figure 6.5: Resource allocation tables used by scheduler

and stores the assignments as a set of instructions for the hardware. The top row of each table represents the current assignments for each port. The tables are used to determine the future assignments.

Assignment Table:

The top row of the assignment table represents the operations to be assigned at current cycle. All the operation scheduled in a cycle must stored in the same cycle they retire because of unavailability of the output buffer. So the duplicate entries of the assigned operations are made in the corresponding retirement table cell. The top row of the table can be used to eliminate unnecessary writing of intermediate results.

Retirement Table:

The top row of the retirement represents the operations retiring in the current cycle i.e. operations whose results are available on the output ports of PEs. These results must be written to the memory except in the case of write cancellation.

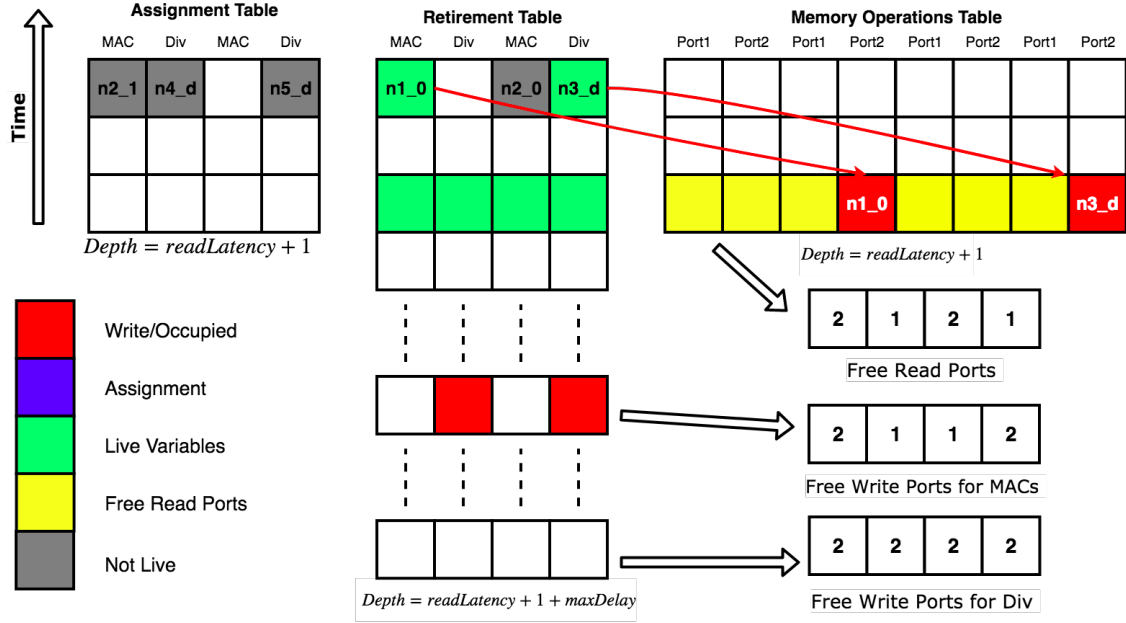


Figure 6.6: Selecting write ports

Memory Operation Table

Memory operation table maintains the record of available ports and assigned operations. Operations are assigned to particular BRAM from the 0th port (time multiplexed ports are numbered from 0 – (n – 1)) first order. The top row of the table indicates the memory operations in the current cycle.

Selecting the set of assignments

All the schedulable nodes are stored in the priority list in decreasing order. All combinations of nodes are checked for availability of BRAM ports for both reading and retirement. A valid combination with highest sum of priorities is selected as assignments for the current cycle. Since this is the most complex process (in terms of time complexity) and is in the critical path of the algorithm the size of the priority list is restricted to 100.

6.5.3 An Example of Scheduling Process

Lets say at some clock cycle the nodes $n1_0$, $n2_0$, $n3_d$ are retiring and the nodes $n2_1$, $n4_d$, $n5_d$ are being assigned to the PEs. The BRAMs have read latency of two. The schedulable list of nodes in the second next cycle can be calculated based on the current retiring nodes and contains nodes $n6_2$, $n7_d$, $n8_d$, $n9_0$, $n10_0$

Available Read Ports

2	1	2	1
---	---	---	---

Available Write Ports

2	1	1	2
---	---	---	---

2	2	2	2
---	---	---	---

Node Ids

n9_0		1	1	
n7_d	1			
n6_2		1	1	1
n10_0			1	1
n8_d	1			

Required Read Ports

Required Write Ports

Figure 6.7: Selecting operations

Assigning Write Ports

(Refer figure 6.6)

1. Retiring nodes: $n1_0, n2_0, n3_d$
2. Scheduled nodes: $n2_1, n4_d, n5_d$
3. Scheduled node $n2_1$ utilizes the intermediate result generated from operation $n2_0$, hence the write operation correspond to $n2_0$ can be eliminated. Hence the scheduler has to reserve only two write operations for remaining nodes $n1_0$ and $n3_d$.
4. Now we can calculate free BRAM ports for read operations.
5. Also we know the delay of each operation and hence determine calculate free PEs to assign operations.

Operation Selection

The figure 6.7 shows 5 ready nodes and available ports. The required number of read and write ports corresponding to each node for each BRAM are listed in the adjoining tables. A valid group of operations must have sum of ports required by selected nodes should be less than total available nodes. The set with maximum sum of priorities is assigned to PEs in current scheduling cycle. In this case $n7_d$, $n8_d$, $n9_0$ and $n10_0$ forms the valid group and hence will be assigned at the bottom of the assignment table confirming the assignment in the second next clock cycle.

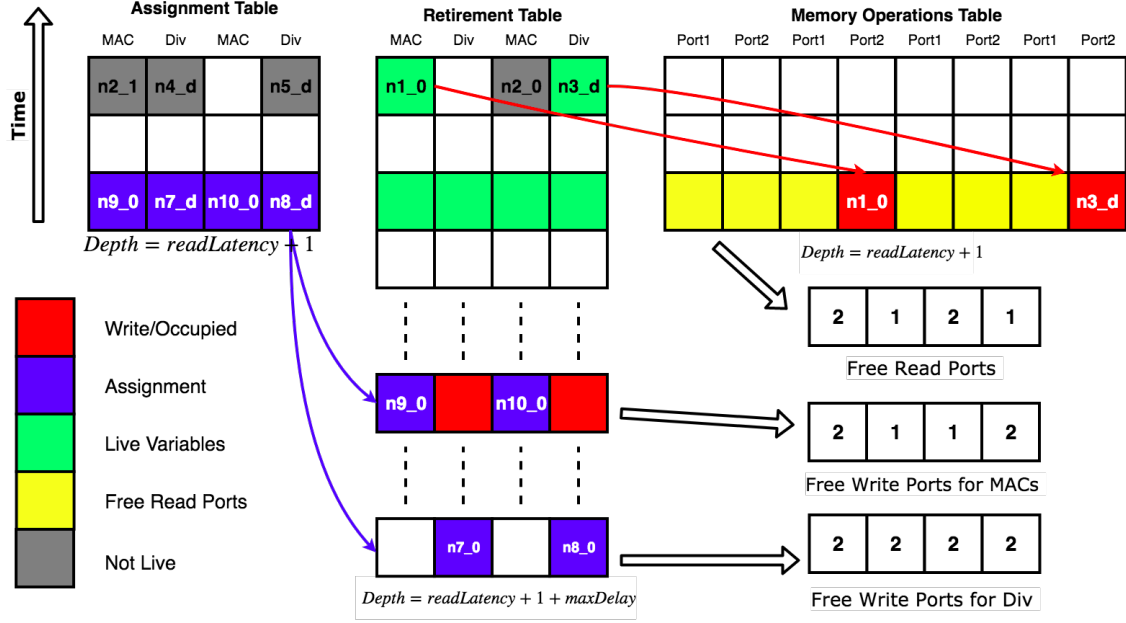


Figure 6.8: Final allocation

Memory Read Assignment

Once the assignments for the second next clock cycle are decided, the additional required data values are set to be read from the corresponding memory locations in the bottom row of the read allocation table.

Saving the Schedule

Figure 6.8 shows the final allocation of the resources. All the non-zero allocations in the top row of each table are saved in the schedule vector. The table entries are then cleared. The tables are stored as the two dimensional vector of structures. To avoid the huge amount of memory relocation due to addition and deletion of new entries, the table indices are accessed in rolling fashion i.e. the actual vector index for table at particular clock cycle is given by

$$Vector\ Index = (Table\ Index + Clock\ Cycle) \% Depth\ of\ Table$$

Chapter 7

Future Work

Synthesizable Quad Port BRAM

Though the design of the quad port BRAM is correct, the synthesis tool detects the false condition of hold violation between two registers triggered by two different clocks. This hold violation is then propagated to the implementation stage and the layout tool fails to meet timing requirements. Even if the hold condition is invalid the setup condition for the second register must be satisfied and hence we can not report the path as a false path. A proper solution must be found for this problem to be able to use time multiplexed BRAMs.

Integration with the On-Chip Processor

The current hardware implementation requires a wrapper testbench to feed the instruction and input data. Porting the scheduler tool chain to the on-chip ARM core of the Zynq SoC can eliminate the need to generate additional BRAM coefficient files inputs and instructions. Also the results can be redirected easily because of the shared memory scape of the processor and FPGA programable logic.

Column Group Based Schedule

Extracting fine grain parallelism provides significant gain over the coarse-grain solution but at the cost of pre-processing time. The level wise scheduling policy used by Nechma [2] is also interesting as it generates the schedule for each column one by one. We can define an intermediate approach where instead of looking at the entire graph we can generate a schedule for group of columns. The groups can be formed based on the level of the column node in the symbolic analysis.

Automatic Generation of the Time-Multiplexed Multiplexers

The current implementation generates multiplexers without adding intermediate registers to optimize the path delays. This is fine because of the smaller number of the processing elements used in this project. In order to increase number of processing units or BRAMs multiplexers must be pipelined. The path delay of the pipeline stage has to be determined based on the target FPGA.

Bibliography

- [1] N. Kapre and A. DeHon, “Parallelizing sparse matrix solve for spice circuit simulation using fpgas,” in *2009 International Conference on Field-Programmable Technology*, pp. 190–198, Dec 2009.
- [2] T. Nechma and M. Zwolinski, “Parallel sparse matrix solution for circuit simulation on fpgas,” *IEEE Transactions on Computers*, vol. 64, pp. 1090–1103, April 2015.
- [3] W. Wu, Y. Shan, X. Chen, Y. Wang, and H. Yang, “Fpga accelerated parallel sparse matrix factorization for circuit simulations,” in *Reconfigurable Computing: Architectures, Tools and Applications* (A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi, eds.), (Berlin, Heidelberg), pp. 302–315, Springer Berlin Heidelberg, 2011.
- [4] S. T. W.H. Press, “Crout’s method,” in *Numerical Recipes 3rd edition: The Art of Scientific Computing*, Cambridge Univ. Press, 2007.
- [5] J. 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.
- [6] X. S. Li and J. W. Demmel, “Making sparse gaussian elimination scalable by static pivoting,” in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pp. 34–34, Nov 1998.