# FPGA Implementation of LU Decomposition Algorithm on Crossbar Network

## M.Tech Project Report

Submitted in partial fulfillment of the requirements
of the degree of

**Master of Technology**
**Microelectronics**

by

**Anurag Choudhury**
**(Roll No: 183070032)**

under the guidance of
**Prof. Sachin Patkar**


**Department of Electrical Engineering**
**INDIAN INSTITUTE OF TECHNOLOGY BOMBAY**
**Powai, Mumbai - 400076**
**June 2020**

# Dissertation Approval

The dissertation entitled

## FPGA Implementation of LU Decomposition Algorithm on Crossbar Network

by

**Anurag Choudhury**
(Roll No. : 183070032)

is approved for the degree of
Master of Technology in Electrical Engineering

| | |
|---|---|
| (Examiner) | (Examiner) |
| (Chairperson) | Prof. Sachin Patkar<br>Dept. of Electrical Engineering<br>(Supervisor) |

Date: 30th June, 2020
Place: IIT, Bombay.

# 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 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 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.

**Anurag Choudhury**
**Roll no. 183070032**
IIT Bombay

June 30, 2020

# Acknowledgements

I would like to express my deep gratitude to Prof. Sachin Patkar for his kind guidance all through. I also extend thanks to Mandar Datar and Niraj Sharma for their constant help and support.

**Anurag Choudhury**

# Contents

# Abstract

Solving linear system of equation is the most critical step in many engineering applications like circuit simulation, training of neural networks, computer vision etc. These are also the applications where the linear system of equations has to be solved for thousands of iterations. In such scenarios it becomes important to come up with a more efficient way to solve the linear system of equations rather than using traditional techniques like gaussian elimination. Solving the linear system of equations using LU decomposition helps in significantly reducing the compute time. This paper focuses on FPGA acceleration of sparse linear system of equations which arise during circuit simulation. LU decomposition of a sparse matrix has significant amount of parallelism that can be exploited by FPGA and is difficult to do so using traditional processors. The LU decomposition approach specified in this project has three main parts. The first part does symbolic analysis of the matrix and generates a directed flow graph where each node corresponds to an arithmetic operation to be performed for computing the LU decomposition of the matrix. The second part takes this directed flow graph and hardware features such as number of arithmetic units and BRAMs as inputs and generates a static schedule for cross-bar type network. The final third part is hardware implementation of the cross-bar network.

# List of figures

# CHAPTER 1

# Introduction

## 1.1 Motivation

Solving a linear system of equation of the form $Ax = B$ is the most time-consuming step in many engineering applications like circuit simulation, training of neural networks, computer vision etc. If for a particular application, we need to solve the system of linear equation only once to arrive at the result then it might not be that big of an issue and can be neglected. But for applications mentioned above, where we need to solve the system of linear equation thousands of time to arrive at the result, developing an efficient algorithm to solve the linear system of equation becomes necessary. This is where LU decomposition comes into picture. By decomposing matrix A into product of two matrices i.e. lower triangular matrix L and upper triangular matrix U, we can get a significant improvement in time required to solve the linear system of equation $Ax = B$. This can be shown mathematically that the cost of factorizing matrix A into L and U is $O(N^3)$. Once the factorization is done, the cost of solving $LUx = b$ is $O(N^2)$. So, if we want to do the simulation for K timesteps, the total cost becomes $O(N^3 + kN^2)$. On the other hand, if we solve the linear equation using Gaussian Elimination, the total cost becomes $O(kN^3)$ which is much larger than $O(N^3 + kN^2)$. If matrix A is sparse, the time required to compute L and U can be further reduced by extracting the parallelism that emerges from this sparseness. However, it is computationally expensive to parallelize the LU decomposition algorithm on traditional processing systems due to interdependent nature of data as a result of which significant portion of time is spent on moving around the data rather than actual computation. Hence this project focuses on accelerating LU decomposition of sparse matrix using FPGA where the parallelism can he properly exploited.

## 1.2 Organization of thesis

The goal of the project is to implement LU decomposition of sparse matrix that arise from circuit simulation. The project is mainly divided into three parts. In the first part, the symbolic analysis of the matrix whose LU decomposition is to be calculated is done. Based on the symbolic analysis a directed flow graph is generated. In the second part, we use the directed flow graph along with hardware features such as arithmetic unit count and BRAM count to generate a static schedule for a cross bar network. The third part involves the hardware implementation of cross-bar network and interfacing it with Zynq processing system, so that it can be tested on FPGA. Apart from these

three major parts there is one additional part which is pre-processing of the matrix whose LU decomposition is to be calculated. This is mainly done to ensure that there are no zeros in the diagonal of the matrix and also to reduce the number of non-zero elements in the matrix. The pre-processing step is not a major focus of this thesis and hence will be discussed in brief. At the end I have discussed the results and compared the performance of the scheduler with similar implementation from other paper[2]. The summary of the thesis is given below: -

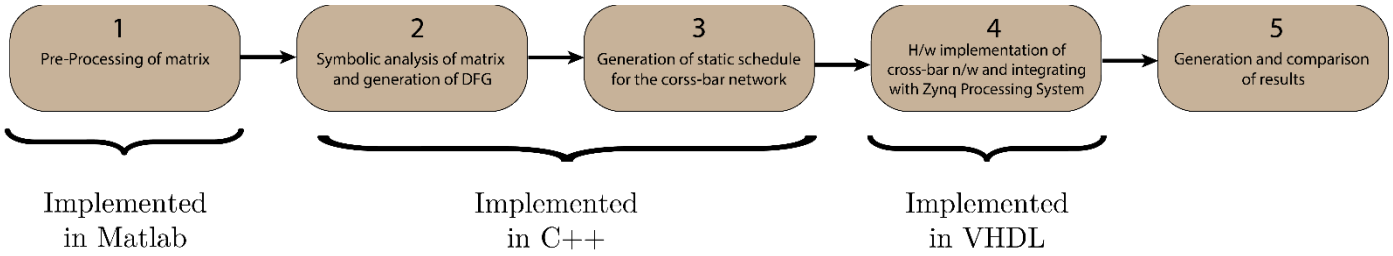



Fig 1.2(a): Flow of thesis

The 2nd, 3rd, 4th and 5th stage of the project will be discussed in detail in the upcoming sections.

# CHAPTER 2

# Literature Review

Various ideas have been proposed by the researchers around the world on accelerating the LU decomposition algorithm using FPGA. Some of the proposed methods focus on exploiting the coarse-grained parallelism, while the other methods focus on extracting the fine-grained parallelism.

In [1], *Kapre* has proposed a method where he accelerated the LU decomposition of sparse matrices on FPGA by using a mesh type architecture in NOC. The major advantage of his approach is that it's scalable. In his approach, first he did the symbolic analysis of the KLU solver and generated a data flow graph. The graph is then divided and distributed among the processing elements which are connected in mesh topology. The graphs are entirely stored in on-chip memory of the FPGA. As the partial graphs are locally stored within the PEs, the PEs can make independent local decisions. This allows the architecture to exploit fine-grained parallelism. This particular architecture has obtained a speedup of 1.2-64x using a 250 MHz Xilinx Virtex-5 FPGA as compared to Intel Core i7 965 processor. The mesh architecture and the structure of processing elements in shown in the below two figures: -

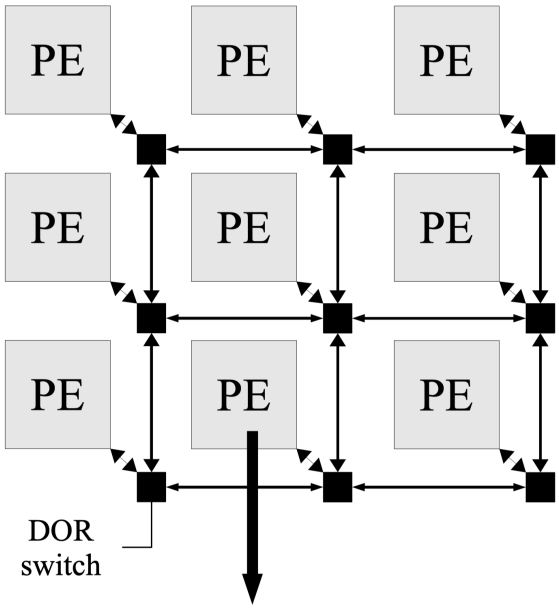



Fig 2(a): PEs connected in mesh topology

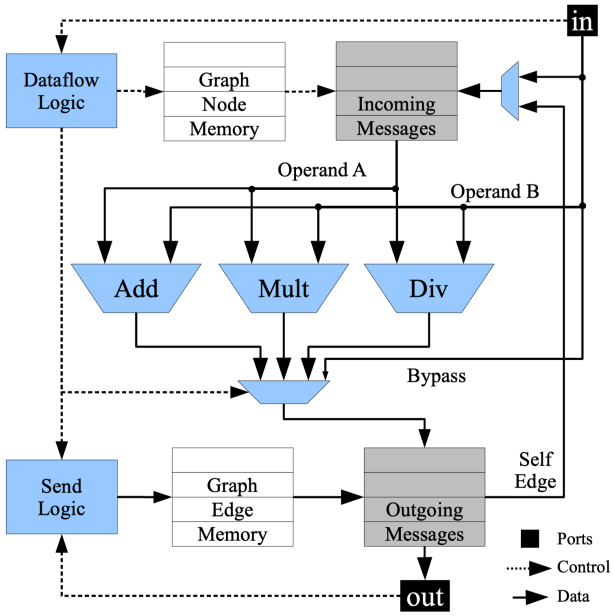



Fig 2(b): Internal structure of PEs

In [3], Wei Wu and Yi Shan proposed an architecture that exploited the coarse-grained parallelism observed in LU decomposition. A modified version of Gilbert-Peierls's algorithm was used to achieve the same. In this approach, every processing element processed a column of the matrix independently. Since coarse-grained parallelism was targeted, the reported speed gain was in the range of 0.5-5.36x which is not much impressive. The architecture used by Wei is shown in the figure below: -

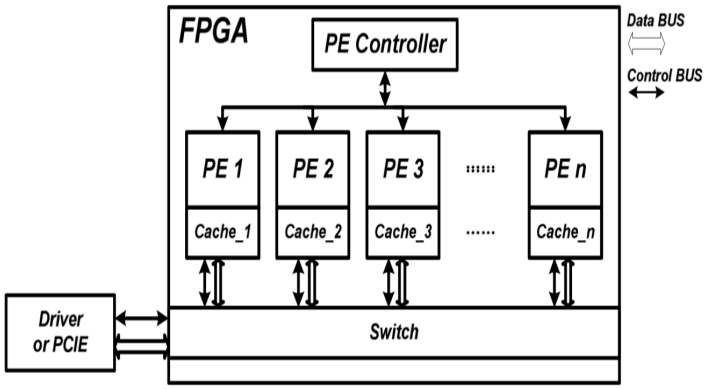



Fig 2(c): FPGA architecture proposed by Wei

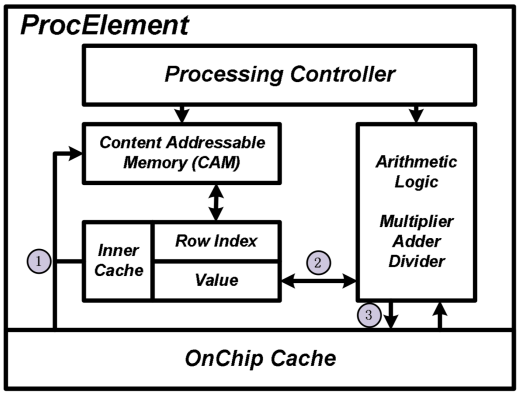



Fig 2(d): Internal Structure of PEs

In [2], Nechma proposed an architecture that exploited medium-grained parallelism observed in LU decomposition. He used the left-looking Gilbert-Peierls's algorithm to achieve the same. In his approach, first he did the symbolic analysis to generate the data flow graph, then he used the data flow graph to generate a column execution schedule. Each of the column is then assigned to a processing element consisting of a multiplier, a subtractor, a divider and a local BRAM. Individual column operations are scheduled using ASAP scheduling. An average speed up of 9.65x was reported using a 250 MHz Xilinx Virtex-5 FPGA as compared to 6-core Intel Xeon 2.6 GHz processor with 6 GB RAM. The architecture used by Nechma is shown in the figure below: -

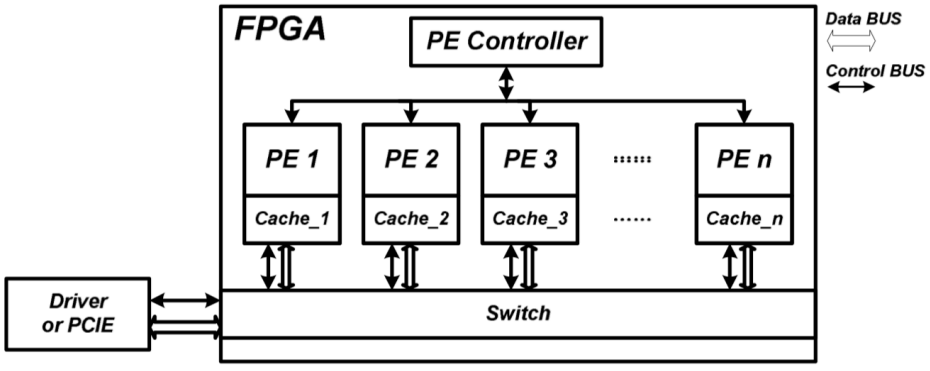



Fig 2(e): FPGA architecture proposed by Nechma

# CHAPTER 3

# Prerequisites

## 3.1 Sparse Matrix Data Structure

Sparse matrices are characterized by very few non-zero elements. For sparse matrices it's not desired to store the entire matrix as it would lead to waste of storage space. Hence there are special data structures developed to store a sparse matrix. One of such data structures is the compressed column sparse data structure. Three 1 dimensional arrays are required to store a sparse matrix in CCS format.

An example of a sparse matrix and its corresponding storage in CCS format is shown below: -

$$A = \begin{bmatrix} 5 & 0 & -5 & 0 & 6 \\ 0 & 4 & 0 & -4 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 1 & -3 & 0 & -1 & 0 \\ 0 & 0 & -2 & 0 & 3 \end{bmatrix}$$

Fig 3.1(a): Test matrix

| 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 | | 11 |

Fig 3.1(b): Test matrix CCS format

The pseudo code to get the actual row and column indices from CCS format is given below: -

```
for i := 0 to size(colPtr)-1 do
    for j := colPtr_i to colPtr_{i+1} do
        actual column := i
        actual row := rowInd_j
        actual value := val_j
```

In the above pseudo code, 'Values' have been represented as 'val', 'Row Indices' have been represented as 'rowInd' and 'Column Pointers' have been represented as 'colPtr'.

## 3.2 Left-Looking Algorithm for LU Decomposition

If we solve the equation $A = LU$, where L is the lower triangular matrix and U is the upper triangular matrix, the equations we will get for each element of L and U is given below: -

$$U_{(i,j)} = A_{(i,j)} - \sum_{k=1}^{i-1} L_{(i,k)} U_{(k,j)} \tag{3.2a}$$

$$L_{(i,j)} = \frac{A_{(i,j)} - \sum_{k=1}^{j-1} L_{(i,k)} U_{(k,j)}}{U_{(j,j)}} \tag{3.2b}$$

Left-looking algorithm factorizes the matrix in column by column manner. Factors of current column are computed with the help of previously computed columns. To understand it better, let's consider a 5x5 matrix:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & A_{1,5} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} & A_{2,5} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & A_{3,5} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} & A_{4,5} \\ A_{5,1} & A_{5,2} & A_{5,3} & A_{5,4} & A_{5,5} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ L_{2,1} & 1 & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 1 & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 1 & 0 \\ L_{5,1} & L_{5,2} & L_{5,3} & L_{5,4} & 1 \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} & U_{1,4} & U_{1,5} \\ 0 & U_{2,2} & U_{2,3} & U_{2,4} & U_{2,5} \\ 0 & 0 & U_{3,3} & U_{3,4} & U_{3,5} \\ 0 & 0 & 0 & U_{4,4} & U_{4,5} \\ 0 & 0 & 0 & 0 & U_{5,5} \end{bmatrix}$$

As per left-looking algorithm, the order in which elements are calculated is as follows: -

1st we calculate the column 1 elements i.e. $U_{1,1}$, $L_{2,1}$, $L_{3,1}$, $L_{4,1}$, $L_{5,1}$.

Next, we calculate the 2nd column elements i.e. $U_{1,2}$, $U_{2,2}$, $L_{3,2}$, $L_{4,2}$, $L_{5,2}$.

Next, we calculate the 3rd column elements i.e. $U_{1,3}$, $U_{2,3}$, $U_{3,3}$, $L_{4,3}$, $L_{5,3}$ and so on.

Apart from left-looking algorithm, there is right-looking and crout algorithm. The only difference between them is the order in which elements of L and U are calculated.

## 3.3 Gilbert-Peierls' Algorithm

Gilbert-Peierls[4] proposed an algorithm for LU decomposition of a matrix with partial pivoting which has a time complexity proportional to the number of floating-point operations. His algorithm is based on left-looking algorithm discussed above. Gilbert-Peierls' Algorithm[4] as mentioned in Nechma's paper[2] is given below: -

**Algorithm 1** Gilbert-Peierls LU factorization of a $n$-by-$n$ asymmetric matrix $A$

1: $L = I$
2: **for** $k = 1$ to $n$ **do**
3: $\quad b = (A : k)$
4: $\quad$ solve the lower triangular system $L_k x = b$
5: $\quad$ do partial pivoting on $x$
6: $\quad U(1 : k, k) = x(1 : k)$
7: $\quad L(k : n, k) = x(k : n)/U(k, k)$
8: **end for**

In line 1, L is first initialized to identity matrix. Note that for L matrix, the diagonal elements are always 1. In line 3, b is a column vector which is assigned $k^{th}$ column of matrix A. In line 4, we are solving for x. Here x is nothing but the right-hand side expression of equation 3.2a or the numerator of right-hand side expression of equation 3.2b. So, one may think what's the difference between Gilbert-Peierls algorithm and left-looking algorithm. The answer is in step 4. In step 4, along with solving for x, he does the symbolic analysis. In symbolic analysis, he neglects all those multiplication terms from equation 3.2a and 3.2b that involve multiplication with 0, and also neglects all those U and L terms which equate to 0(because all the multiplication terms are 0 and $A_{(i,j)}$ is also 0). Now if we need to find LU decomposition of matrix A 10,000 times, which is normal for circuit simulation, the symbolic analysis has to be done only once. This is because in applications like circuit simulation, the position of non-zero elements remain fixed, only their value changes.

# Chapter 4

# LU Decomposition Implementation

As discussed in the earlier section, I have divided the implementation of LU decomposition into 4 parts i.e. pre-processing, symbolic analysis, schedule generation and h/w implementation. The pre-processing stage would be discussed briefly, whereas the other 3 stages will be discussed in depth.

## 4.1 Pre-processing of Matrix

The pre-processing stage is done entirely in MATLAB using inbuilt functions of MATLAB. Pre-processing of matrix ensures better performance and also ensures that that there is no divide by zero errors when we compute the LU decomposition of A. Divide by 0 can occur if $U_{j,j}$ becomes zero(Refer equation 3.2b). Apart from this we also use this stage to calculate the correct L and U matrices so that we can later verify the results for correctness. All the sub stages in pre-processing stage is shown in the figure below: -

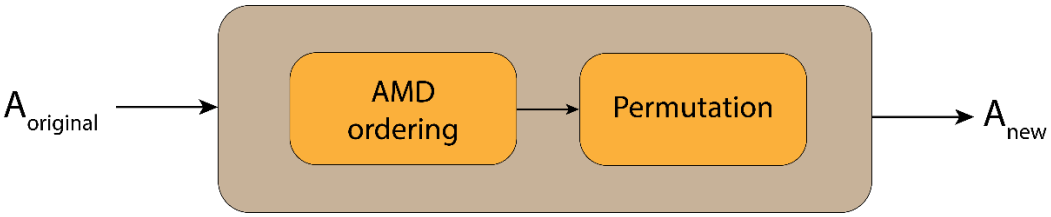



Fig 4.1(a): Pre-Processing Stage

### 4.1.1 AMD Ordering

Approximate minimum degree ordering is done on matrix A to reduces the number of non-zero elements in L and U. This in turn decreases the number of floating-point operations which reduces the compute time. To illustrate the effect of AMD ordering on matrix A, consider the fpga_dcop_01 matrix before and after AMD ordering.

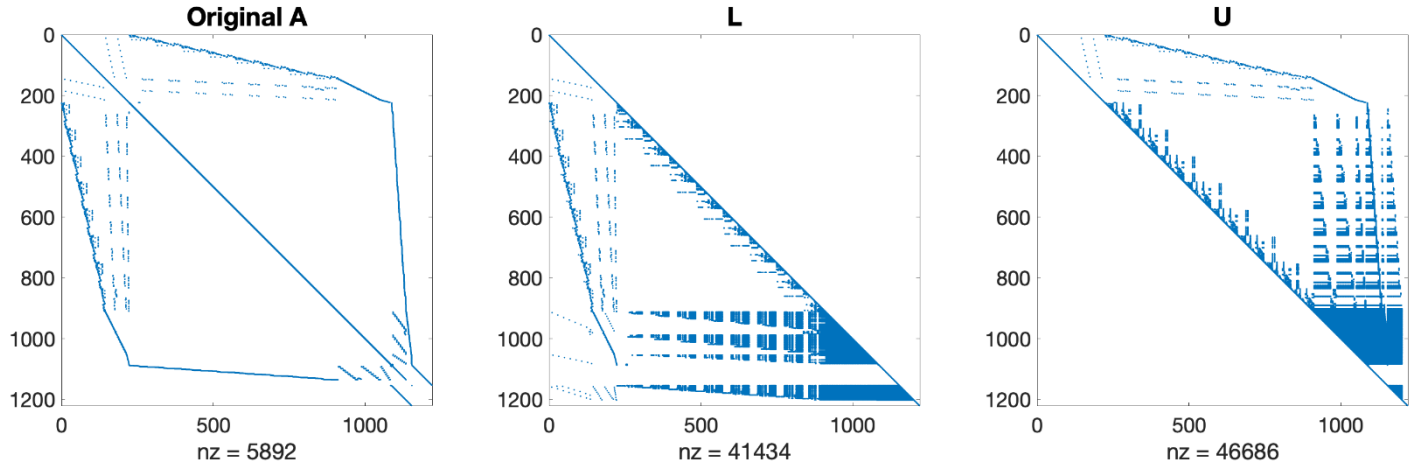



Fig 4.1.1(a): Non-zero elements of L and U with original matrix A

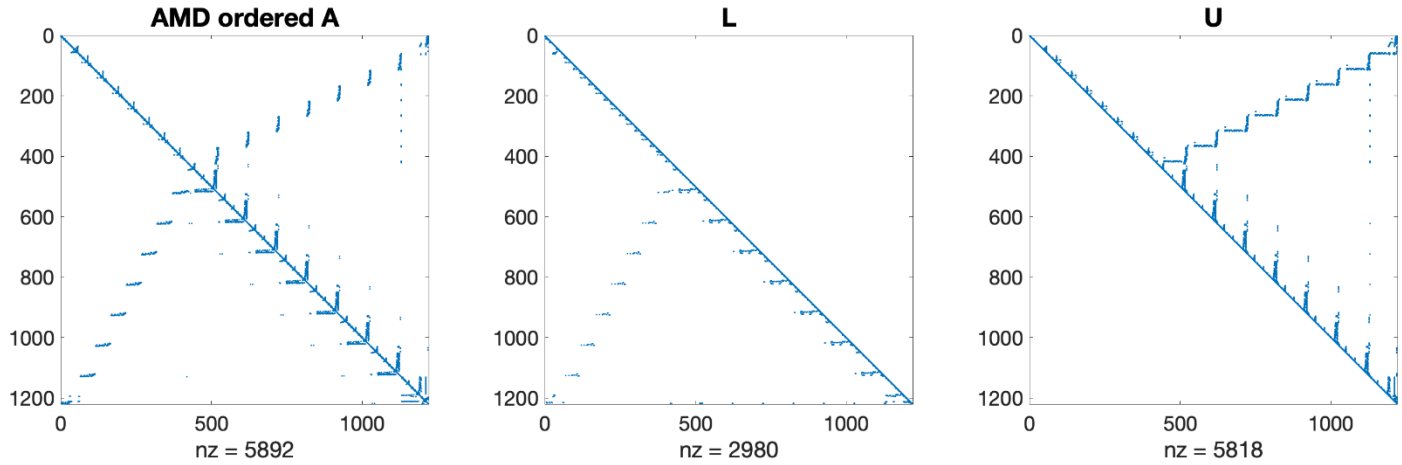



Fig 4.1.1(b): Non-zero elements of L and U with AMD ordered matrix A

From the above 2 figures it's clear that if we do AMD ordering of A, the number of non-zero elements in L reduced by a factor of 14 and number of non-zero elements in U reduced by a factor of 8.

## 4.1.2 Permutation of A

Pivoting is done to ensure that none of the diagonal elements of U become zero. This is necessary to avoid divide by zero errors which can occur in equation 3.2b. Hence the actual equation becomes: -

$$L * U = P * AMD(Original\ A)$$

Here P is the permutation matrix. Hence $A_{new}$ can be written as: -

$$New\ A\ = P * AMD(Original\ A)$$

In the subsequent stages, all the operations will be done on $A_{new}$.

## 4.2 Symbolic Analysis and Dataflow Graph generation

This stage is one of the most important stage in LU decomposition of the matrix. The purpose of this stage is the elimination of unnecessary arithmetic operations encountered while computing L and U matrices. These are the operations like multiplication with zeros or subtracting zero from a number etc. The substages of symbolic analysis stage are shown in the figure below: -

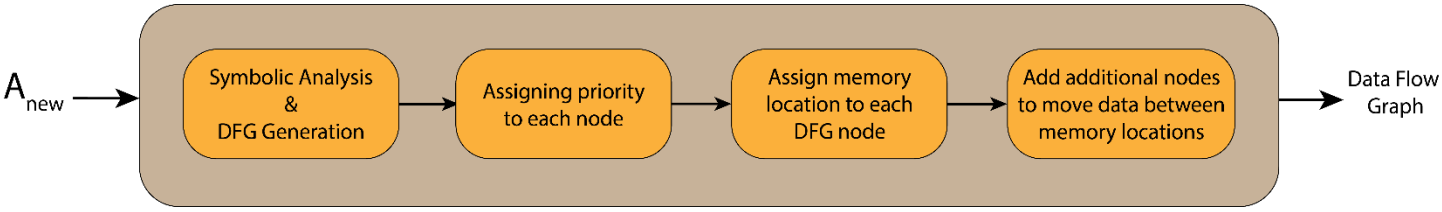



Fig 4.2(a): Symbolic Analysis & DFG Generation Stage

### 4.2.1 Symbolic Analysis and DFG Generation

In this section first I will propose the formal algorithm for symbolic analysis mentioned in Nechma's paper[2]. Then I will graphically explain how is symbolic analysis implemented in c++ code. This will give a better understanding of the symbolic analysis stage.

The symbolic analysis algorithm mentioned in Nechma's paper[2] is given below: -

**Algorithm 2** Sparse forward substitution

1: $x = b$
2: **for** *each* $j \in \mathcal{X}$ **do**
3: **for** *each* $i > j$ for which $l_{ij} \neq 0$ **do**
4: $x_i = x_i - l_{ij}x_j$
5: **end for**
6: **end for**

The above 6 lines can be considered as expanded form of step 4 of **Algorithm 1** discussed in section 3.3.

In order to have a better intuitive understanding of the symbolic analysis stage, I have graphically explained the procedure using matrix A mentioned in section 3.1. As Gilbert-Peierls' algorithm is inherently a left-looking algorithm (discussed in section 3.2), I go column by column to explain the symbolic analysis stage. Consider only the non-zero places in matrix A(filled with red color) in the figure below: -

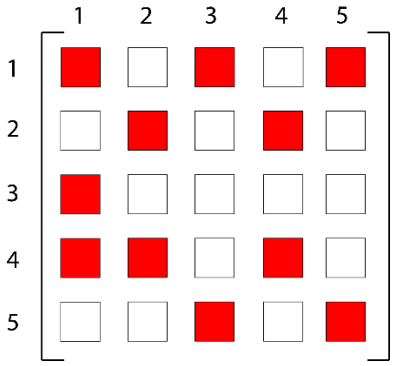
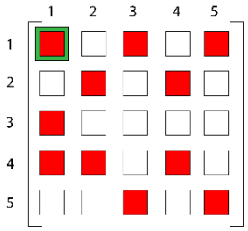



Fig 4.2.1(a): A matrix non-zero location

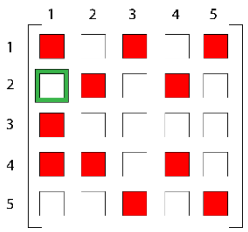
Let's call the above matrix 'fill-in-tracking' matrix. Once the initial non-zero locations have been identified, the fill-in-tracking matrix has nothing to do with A matrix. The new non-zero locations in the fill-in-matrix will be marked with yellow color as we move from one column to the next. For every column, I will go through each element of the column from top to bottom. The selected column element will be highlighted with a green halo. At the end I will compare the original equations with the equations obtained after symbolic analysis.

***Column 1: -***

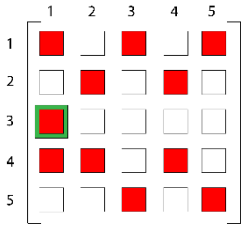
It's the only column that is guaranteed to have no other additional non-zero elements.



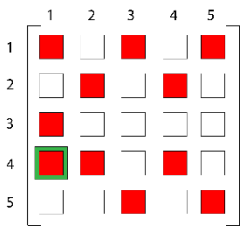
According to equation 3.2a,
$U_{1,1} = A_{1,1}$



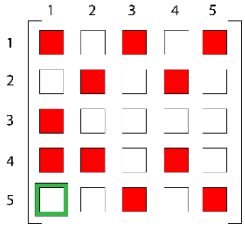
According to equation 3.2a,
$L_{2,1} = A_{2,1}/U_{1,1} = 0$
Since $A_{2,1}$ is 0, the entire $L_{2,1}$ equation will be neglected.



According to equation 3.2b,
$L_{3,1} = A_{3,1}/U_{1,1}$



According to equation 3.2b,
$L_{4,1} = A_{4,1}/U_{1,1}$



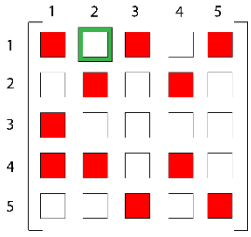
According to equation 3.2b,

$L_{5,1} = A_{5,1}/U_{1,1} = 0$

Since $A_{5,1}$ is 0, the entire $L_{5,1}$ equation will be neglected.

***Column 2: -***

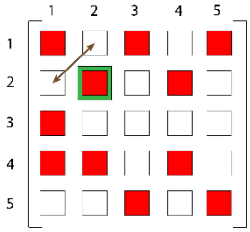
Column 2 onwards there can be additional non-zero locations, and MAC operations will also come into play from here on.



According to equation 3.2a,

$U_{1,2} = A_{1,2} = 0$

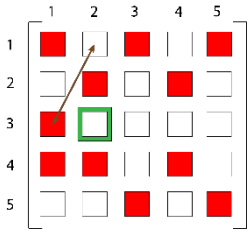
Since $A_{1,2}$ is 0, $U_{1,2}$ will be neglected.



According to equation 3.2a,

$U_{2,2} = A_{2,2} - L_{2,1}*U_{1,2} = A_{2,2}$

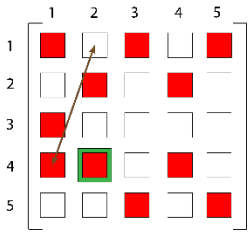
Since both $L_{2,1}$ & $U_{1,2}$ terms are 0, the multiplication operation can be neglected.



According to equation 3.2b,

$L_{3,2} = (A_{3,2} - L_{3,1}*U_{1,2})/U_{2,2} = 0$

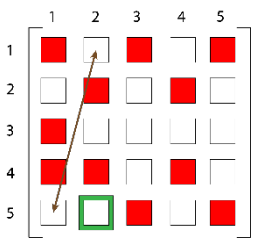
Since $A_{3,2}$ & $U_{1,2}$ are 0, the entire $L_{3,2}$ equation is neglected.



According to equation 3.2b,

$L_{4,2} = (A_{4,2} - L_{4,1}*U_{1,2})/U_{2,2} = A_{4,2}/U_{2,2}$

Since $U_{1,2}$ is 0, the multiplication operation can be neglected.

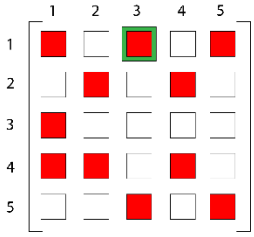


According to equation 3.2b,

$L_{5,2} = (A_{5,2} - L_{5,1}*U_{1,2})/U_{2,2} = 0$

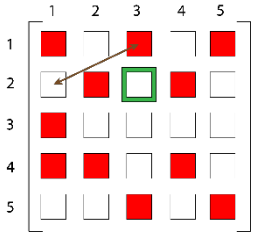
Since $A_{5,2}$ & $U_{1,2}$ are 0, the entire $L_{5,2}$ equation is neglected.

***Column 3: -***



According to equation 3.2a,

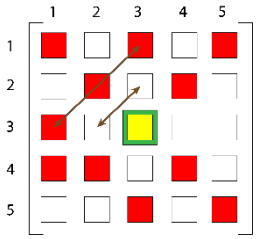
$U_{1,3} = A_{1,3}$



According to equation 3.2a,

$U_{2,3} = A_{2,3} - L_{2,1}*U_{1,3} = 0$

Since $A_{2,3}$ & $L_{2,1} = 0$, the entire $U_{2,3}$ equation can be neglected.

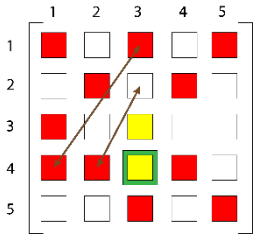


According to equation 3.2a,

$U_{3,3} = A_{3,3} - L_{3,1}*U_{1,3} - L_{3,2}*U_{2,3} = - L_{3,1}*U_{1,3}$

Since both $L_{3,2}$ & $U_{2,3}$ are zero, the second multiplication operation is neglected.

Also note that, this is the first element in fill-in-matrix that converted from zero to non-zero element. Hence marked with yellow color.

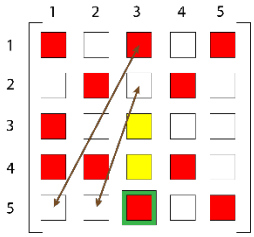


According to equation 3.2b,

$L_{4,3} = (A_{4,3} - L_{4,1}*U_{1,3} - L_{4,2}*U_{2,3})/U_{3,3} = (- L_{4,1}*U_{1,3})/U_{3,3}$

Since $U_{2,3}$ is zero, the second multiplication operation is neglected.

Note that this element in the fill-in-matrix is converted to a non-zero element. Hence marked with yellow color.

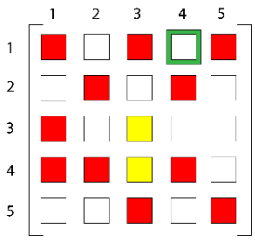


According to equation 3.2b,

$L_{5,3} = (A_{5,3} - L_{5,1}*U_{1,3} - L_{5,2}*U_{2,3})/U_{3,3} = A_{5,3}/U_{3,3}$

Since $L_{5,1}$ is 0 the 1st multiplication operation is neglected. Since both $L_{5,2}$ & $U_{2,3}$ are zeros, the second multiplication operation is also neglected.

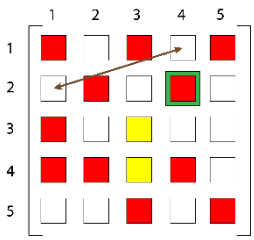
***Column 4: -***



According to equation 3.2a,

$U_{1,4} = A_{1,4} = 0$

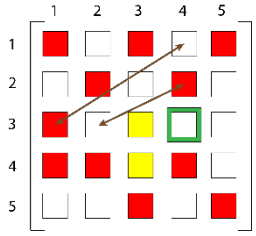
Since $A_{1,4}$ is 0, $U_{1,4}$ will be neglected.



According to equation 3.2a,

$U_{2,4} = A_{2,4} - L_{2,1}*U_{1,4} = A_{2,4}$

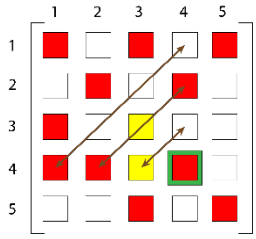
Since both $L_{2,1}$ & $U_{1,4}$ is 0, the multiplication term can be neglected.



According to equation 3.2a,

$U_{3,4} = A_{3,4} - L_{3,1}*U_{1,4} - L_{3,2}*U_{2,4} = 0$

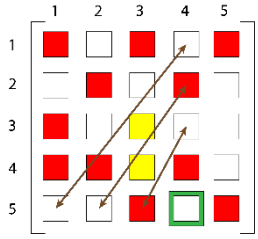
Since $A_{3,4}$, $U_{1,4}$ & $L_{3,2}$ are zeros, the entire $U_{3,4}$ equation can be neglected.



According to equation 3.2a,

$U_{4,4} = A_{4,4} - L_{4,1}*U_{1,4} - L_{4,2}*U_{2,4} - L_{4,3}*U_{3,4} = A_{4,4} - L_{4,2}*U_{2,4}$

Since $U_{1,4}$ is 0, the 1st multiplication term is neglected and since $U_{3,4}$ is 0, the 3rd multiplication term is neglected.

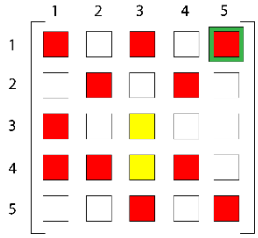


According to equation 3.2b,

$L_{5,4} = (A_{5,4} - L_{5,1}*U_{1,4} - L_{5,2}*U_{2,4} - L_{5,3}*U_{3,4})/U_{4,4} = 0$

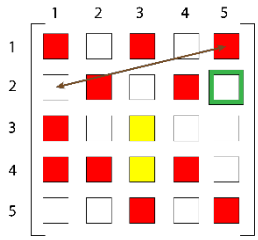
Sine $A_{5,4}$ is 0 and one or both the operands is 0 for all the three multiplication terms, the entire $L_{5,4}$ equation is neglected.

***Column 5: -***



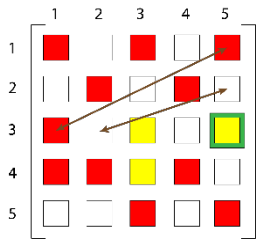
According to equation 3.2a,

$U_{1,5} = A_{1,5}$



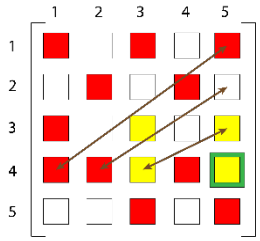
According to equation 3.2a,

$U_{2,5} = A_{2,5} - L_{2,1}*U_{1,5} = 0$



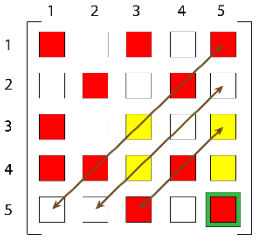
According to equation 3.2a,

$U_{3,5} = A_{3,5} - L_{3,1}*U_{1,5} - L_{3,2}*U_{2,5} = - L_{3,1}*U_{1,5}$



According to equation 3.2a,

$U_{4,5} = A_{4,5} - L_{4,1}*U_{1,5} - L_{4,2}*U_{2,5} - L_{4,3}*U_{3,5} = - L_{4,1}*U_{1,5} - L_{4,3}*U_{3,5}$




According to equation 3.2a,

$U_{5,5} = A_{5,5} - L_{5,1}*U_{1,5} - L_{5,2}*U_{2,5} - L_{5,3}*U_{3,5} = A_{5,5} - L_{5,3}*U_{3,5}$

Now comparing all the equations before and after symbolic analysis: -

| Col. No. | Equations before symbolic analysis | MAC ops | DIV ops | Equations after symbolic analysis | MAC ops | DIV ops |
|---|---|---|---|---|---|---|
| 1 | $U_{1,1} = A_{1,1}$ | 0 | 0 | $U_{1,1} = A_{1,1}$ | 0 | 0 |
| | $L_{2,1} = A_{2,1}/U_{1,1}$ | 0 | 1 | - | 0 | 0 |
| | $L_{3,1} = A_{3,1}/U_{1,1}$ | 0 | 1 | $L_{3,1} = A_{3,1}/U_{1,1}$ | 0 | 1 |
| | $L_{4,1} = A_{4,1}/U_{1,1}$ | 0 | 1 | $L_{4,1} = A_{4,1}/U_{1,1}$ | 0 | 1 |
| | $L_{5,1} = A_{5,1}/U_{1,1}$ | 0 | 1 | - | 0 | 0 |
| 2 | $U_{1,2} = A_{1,2}$ | 0 | 0 | - | 0 | 0 |
| | $U_{2,2} = A_{2,2} - L_{2,1}*U_{1,2}$ | 1 | 0 | $U_{2,2} = A_{2,2}$ | 0 | 0 |
| | $L_{3,2} = (A_{3,2} - L_{3,1}*U_{1,2})/U_{2,2}$ | 1 | 1 | - | 0 | 0 |
| | $L_{4,2} = (A_{4,2} - L_{4,1}*U_{1,2})/U_{2,2}$ | 1 | 1 | $L_{4,2} = A_{4,2}/U_{2,2}$ | 0 | 1 |
| | $L_{5,2} = (A_{5,2} - L_{5,1}*U_{1,2})/U_{2,2}$ | 1 | 1 | - | 0 | 0 |
| 3 | $U_{1,3} = A_{1,3}$ | 0 | 0 | $U_{1,3} = A_{1,3}$ | 0 | 0 |
| | $U_{2,3} = A_{2,3} - L_{2,1}*U_{1,3}$ | 1 | 0 | - | 0 | 0 |
| | $U_{3,3} = A_{3,3} - L_{3,1}*U_{1,3} - L_{3,2}*U_{2,3}$ | 2 | 0 | $U_{3,3} = - L_{3,1}*U_{1,3}$ | 1 | 0 |
| | $L_{4,3} = (A_{4,3} - L_{4,1}*U_{1,3} - L_{4,2}*U_{2,3})/U_{3,3}$ | 2 | 1 | $L_{4,3} = (- L_{4,1}*U_{1,3})/U_{3,3}$ | 1 | 1 |
| | $L_{5,3} = (A_{5,3} - L_{5,1}*U_{1,3} - L_{5,2}*U_{2,3})/U_{3,3}$ | 2 | 1 | $L_{5,3} = A_{5,3}/U_{3,3}$ | 0 | 1 |
| 4 | $U_{1,4} = A_{1,4}$ | 0 | 0 | - | 0 | 0 |
| | $U_{2,4} = A_{2,4} - L_{2,1}*U_{1,4}$ | 1 | 0 | $U_{2,4} = A_{2,4}$ | 0 | 0 |
| | $U_{3,4} = A_{3,4} - L_{3,1}*U_{1,4} - L_{3,2}*U_{2,4}$ | 2 | 0 | - | 0 | 0 |
| | $U_{4,4} = A_{4,4} - L_{4,1}*U_{1,4} - L_{4,2}*U_{2,4} - L_{4,3}*U_{3,4}$ | 3 | 0 | $U_{4,4} = A_{4,4} - L_{4,2}*U_{2,4}$ | 1 | 0 |
| | $L_{5,4} = (A_{5,4} - L_{5,1}*U_{1,4} - L_{5,2}*U_{2,4} - L_{5,3}*U_{3,4})/U_{4,4}$ | 3 | 1 | - | 0 | 0 |
| 5 | $U_{1,5} = A_{1,5}$ | 0 | 0 | $U_{1,5} = A_{1,5}$ | 0 | 0 |
| | $U_{2,5} = A_{2,5} - L_{2,1}*U_{1,5}$ | 1 | 0 | - | 0 | 0 |
| | $U_{3,5} = A_{3,5} - L_{3,1}*U_{1,5} - L_{3,2}*U_{2,5}$ | 2 | 0 | $U_{3,5} = - L_{3,1}*U_{1,5}$ | 1 | 0 |
| | $U_{4,5} = A_{4,5} - L_{4,1}*U_{1,5} - L_{4,2}*U_{2,5} - L_{4,3}*U_{3,5}$ | 3 | 0 | $U_{4,5} = - L_{4,1}*U_{1,5} - L_{4,3}*U_{3,5}$ | 2 | 0 |
| | $U_{5,5} = A_{5,5} - L_{5,1}*U_{1,5} - L_{5,2}*U_{2,5} - L_{5,3}*U_{3,5}$ | 3 | 0 | $U_{5,5} = A_{5,5} - L_{5,3}*U_{3,5}$ | 1 | 0 |
| **Total** | | 29 | 10 | **Total** | 7 | 5 |

It's evident from the table that because of symbolic analysis, the number of MAC operations reduced by almost 76% and number of DIV operations reduced by 50%.

It's also important to note that the set of equations obtained after symbolic analysis is only valid for those matrices whose non-zero locations is same as the non-zero locations of A matrix mentioned in section 3.1.

Now we can use the equations obtained after symbolic analysis to generate a data flow graph. The data flow graph shown in the below figure uses a matrix naming convention according to which matrix index starts from 0 instead of 1. Hence the term $L_{4,1}$ according to the above table is $L_{3,0}$ in the data flow graph.

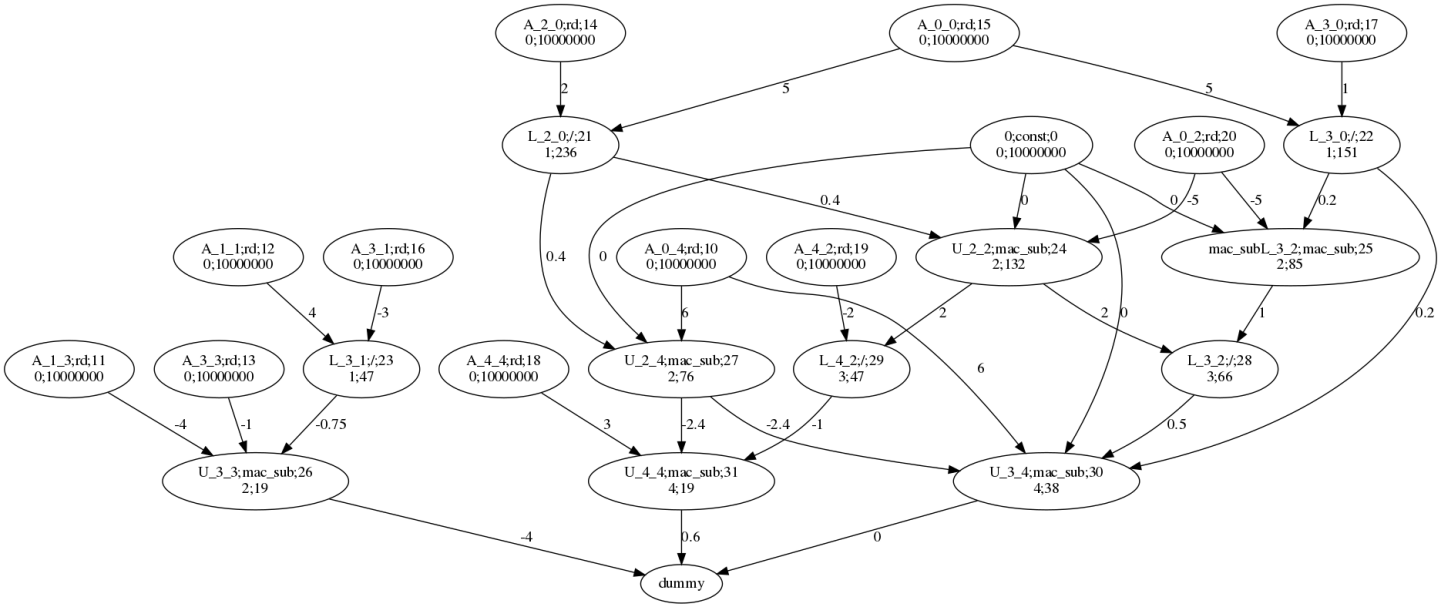



Fig 4.2.1(b): Data flow graph for computing LUD of matrix A mentioned in section 3.1

While generating the data flow graph, in case of an element of L matrix, I have separated the MAC operation node from DIV operation node. The separated MAC operation node is named as "mac_subL_x_y". This can be seen for $L_{3,2}$ in the above graph.
The meaning of terms inside each node of the data flow graph is described in the figure below: -

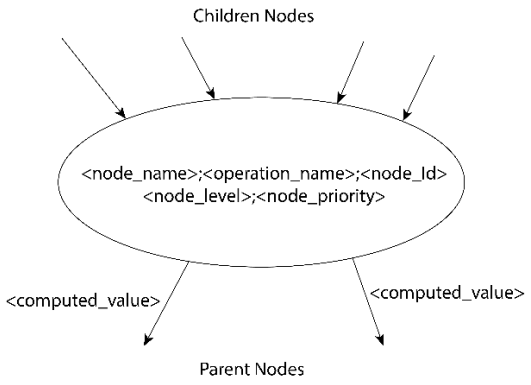



Fig 4.2.1(c): A single node depicting the terms associated with the node

Note the convention used by children nodes and parent nodes in this project. Nodes which are providing a value to a given nodes are **children** of the given node. And the nodes which are accepting value from a given node are **parents** to the given node.
<node_name>: Indicates the name associated with the node. The node name is of the form "L_x_y", "U_x_y", "mac_subL_x_y" etc. The "node_name" is unique.

<operation_name>: Indicates the operation performed by the node. The operations can be "rd"(memory read), "wr"(memory write), "mac_sub"(MAC), "/"(DIV) or "const"(indicates that the node has a constant value of 0).

<node_Id>: Indicates a unique Id associated with each node.

<node_level>: Indicates the level of the node i.e. the worst-case distance from a leaf node.

<node_priority>: It indicates the priority level of the node. This parameter is extremely useful while scheduling the graph.

<computed_value>: This parameter indicates the value computed by the node. Ex. $U_{2,4}$ = -2.4, $U_{3,3}$ = -4 etc.

### 4.2.2 Assigning Priority to each Node

Proper priority assignment is very useful for optimal schedule generation. In case of contention in scheduling, nodes with the highest priority are scheduled first. Priority assignment starts from root nodes and we gradually move towards leaf node. It's important to note that all the leaf nodes are of type "rd"(memory read) and has been hardcoded a priority of 10000000. This should not be an issue as nodes of type "rd" are simple memory read operations and are never assigned to arithmetic units(MAC or DIV) for scheduling.

The priority calculation formula for node of type "/"(DIV) is given below: -

$$\text{"/" node priority} = \sum \textit{Parent Node priority} + \textit{DIV latency}$$

The priority calculation formula for node of type "mac_sub"(MAC) is given below: -

$$\text{"mac\_sub" node priority} = \sum \textit{Parent Node priority} + \textit{No. of MAC operations} * \textit{MAC latency}$$

Let's consider an example of each for the data flow graph mentioned in section 4.2.1. The data flow graph was generated for a **DIV Latency of 28 cycles** and **MAC Latency of 19 cycles**.

Consider the "/" node "L_3_0". Its priority is calculated as follows: -

$$\begin{aligned}\text{Priority of L\_3\_0} &= \sum \textit{Parent Node priority} + \text{DIV latency} \\ &= (85 + 38) + 28 \\ &= 151\end{aligned}$$

Consider the "mac_sub" node "U_3_4". Its priority is calculated as follows: -

$$\begin{aligned}\text{Priority of U\_3\_4} &= \sum \textit{Parent Node priority} + \textit{No. of MAC operations} * \textit{MAC LAtency} \\ &= (0) + 2*19 \\ &= 38\end{aligned}$$

### 4.2.3 Assigning Memory Location to each Graph Node

In this project the memory location assignment is done randomly. Nodes of type "rd" are already stored in memory as this node represents the elements of A matrix whose LU decomposition has to be done. The nodes with name "mac_subL_x_y" have only one parent i.e. "L_x_y" and it's memory location is also same as "L_x_y". This is so because we don't require the value computed by the node "mac_subL_x_y".

### 4.2.4 Adding Extra Nodes to move Data b/w Memory Locations

In our design we have used two types of arithmetic units namely MAC and DIV. MAC requires three operands whereas DIV requires two. Since as per our hardware design, an operand fetched from BRAM is available for only 1 cycle, it's important that all the operands for a particular arithmetic operation can be fetched in the same cycle. Hence when we assign the memory location to nodes randomly, it's possible that all the operands to MAC operation are assigned to the same BRAM. Hence considering the worst case, we need to use al least 3 port BRAM in our design. Unfortunately, Xilinx IP provide BRAMS up to dual port. In such scenarios, it's necessary to temporarily move one or more of the operands to a separate BRAM. This is why the additional nodes of type "wr" are added which move data from one BRAM location to another.

The algorithm to add "wr" node before a DIV node is much easier than MAC node. The only time a DIV node will require a "wr" node is when we use single port BRAMS. And the only time a MAC node will require a "wr" node is when we use single or double port BRAMS.

The algorithm for adding "wr" node before a DIV node is given below: -

**Algorithm 3** Adding "wr" node befor selected DIV nodes

```
for each DIV Node in DataFlowGraph:
    BRAMPortUsageList = zeros(Total BRAM count)
    Populate "BRAMPortUsageList" with information about BRAM port usage while fetching the operands for the DIV node.
    if (Any entry in "BRAMPortUsageList" > number of ports available with the BRAM):
        overusedBRAMId = Index of "BRAMPortUsageList"  for which the above condition held true
        possibleWrNodeBRAMList = [ ]
        Populate "possibleWrNodeBRAMList" with BRAM Ids other than "overusedBRAMId"
        if (Any of the 2 children of the DIV node have a parent "wr" node with BRAMId which also belongs to the list
        "possibleWrNodeBRAMList"):
            selectedChild = The child for which above condition held true
            selectedWRNode = The "wr" node for which the above condition held true
            Break the connection between the "selectedChild" and DIV node and attach the "selectedWRNode" as a child to the
            DIV node
        else:
            newWrNode = Create a "wr" Node with BRAMId from "possibleWrNodeBRAMList" which is the most empty
            Break the connection between the 1st child and DIV node and attach the "newWrNode" as a parent to the 1st child
            and as a child to the DIV node.
        end if
    end if
end for
```

In the above algorithm, "BRAMPortUsageList" will have a size equal to total number of BRAMs available. Each index in that list corresponds to a BRAMId. So, if there are 4 BRAMs and the list looks like [0,0,1,1], it indicates only 1 port of BRAM with BRAMId = 2 and 3 are required to fetch the DIV operands.

The algorithm for adding "wr" node before a MAC node is given below: -

**Algorithm 4** Adding "wr" node befor selected MAC nodes

```
1:  for each MAC Node in DataFlowGraph:
2:      for each MUL Pair in MAC Node:
3:          for i = 1 to 2 do:
4:              BRAMPortUsageList_1 = zeros(Total BRAM count)
5:              BRAMPortUsageList_2 = zeros(Total BRAM count)
6:              Populate "BRAMPortUsageList_1" with information about BRAM port usage, considering the MUL pair and MAC
                node itself as children
7:              Populate "BRAMPortUsageList_2" with information about BRAM port usage, considering the MUL pair and A
                (suppose MAC operation is like A-BC-DE-FG) as children
8:              if (Any entry in "BRAMPortUsageList_1" > number of ports available with the BRAM):
9:                  overusedBRAMId = Index of "BRAMPortUsageList_1"  for which the above condition held true
10:                 possibleWrNodeBRAMList = [ ]
11:                 Populate "possibleWrNodeBRAMList" with BRAM Ids which have free ports available according to both
                    "BRAMPortUsageList_1" and "BRAMPortUsageList_2"
12:             else if (Any entry in "BRAMPortUsageList_2" > number of ports available with the BRAM):
13:                 overusedBRAMId = Index of "BRAMPortUsageList_2"  for which the above condition held true
14:                 possibleWrNodeBRAMList = [ ]
15:                 Populate "possibleWrNodeBRAMList" with BRAM Ids which have free ports available according to both
                    "BRAMPortUsageList_1" and "BRAMPortUsageList_2"
16:             end if
17:             if ("overusedBRAMId" == BRAMId of 1st multiplicand operand &&
                "overusedBRAMId" == BRAMId of 2nd multiplicand operand):
18:                 if (Any of the 2 operands of MUL pair have a parent "wr" node with BRAMId which also belongs to the list
                    "possibleWrNodeBRAMList"):
19:                     selectedChild = The operand of MUL pair for which above condition held true
20:                     selectedWRNode = The "wr" node for which the above condition held true
21:                     Break the connection between the "selectedChild" and MAC node and attach the "selectedWRNode" as a
                        child to the MAC node
22:                 else:
23:                     newWrNode = Create a "wr" Node with BRAMId from "possibleWrNodeBRAMList" which is most empty
24:                     Break the connection between the 1st operand of MUL pair and MAC node and attach the "newWrNode"
                        as a parent to the 1st operand of MUL pair and as a child to the MAC node.
25:                 end if
26:             else:
27:                 selectedChild = The operand of MUL pair whose BRAMId matches with "overusedBRAMId"
28:                 if ("selectedChild" has a parent "wr" node with BRAMId which also belongs to the list
                    "possibleWrNodeBRAMList"):
29:                     selectedWRNode = The "wr" node for which the above condition held true
30:                     Break the connection between the "selectedChild" and MAC node and attach the "selectedWRNode" as a
                        child to the MAC node
31:                 else:
32:                     newWrNode = Create a "wr" Node with BRAMId from "possibleWrNodeBRAMList" which is most empty
33:                     Break the connection between the "selectedChild" and MAC node and attach the "newWrNode" as a
                        parent to the "selectedChild" and as a child to the MAC node.
34:                 end if
35:             end if
36:         end for
37:     end for
38: end for
```

According to the above algorithm, I have added the "wr" nodes only before the MUL pairs in MAC node. So, if a MAC node operation is of the form A-BC-DE-FG, then "wr" node will never be added before A.

There are few important points that I would like to summarize. Algorithm 3 comes into action only when using single port BRAMs. Algorithm 4 comes into action only when using single and dual port BRAMs. For the above 2 algorithms to work, it's necessary that we have at least 4 ports available in total, i.e. either 4 single port BRAMs or 2 dual port BRAMs.

To understand algorithm 3 intuitively, let's take an example of a DIV node. I have considered **4 single port BRAMs** for this example. Consider the DIV node shown below. The node of interest is highlighted in green: -

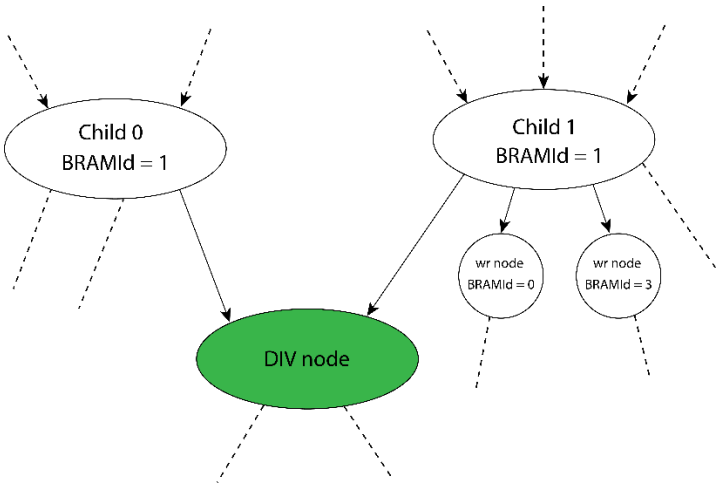



Fig 4.2.4(a): Isolated DIV node for demonstrating addition of "wr" node – PART I

For the above DIV node, "BRAMPortUsageList", is given below: -

**BRAMPortUsageList = [0,2,0,0]**

From the above list, it's clear that **overusedBRAMId = 1**

Hence, "possibleWrNodeBRAMList" will be: -

**possibleWrNodeBRAMList = [0,2,3]**

Now, Child 1 already has 2 "wr" nodes with BRAMIds which also belongs to the list "possibleWrNodeBRAMList" i.e. "wr" node with BRAMId = 0 & "wr" node with BRAMId = 3. Assume that "wr" node with BRAMID = 3 is most empty. So, we will make necessary changes in the connection between the Nodes. The new graph will look as given below: -

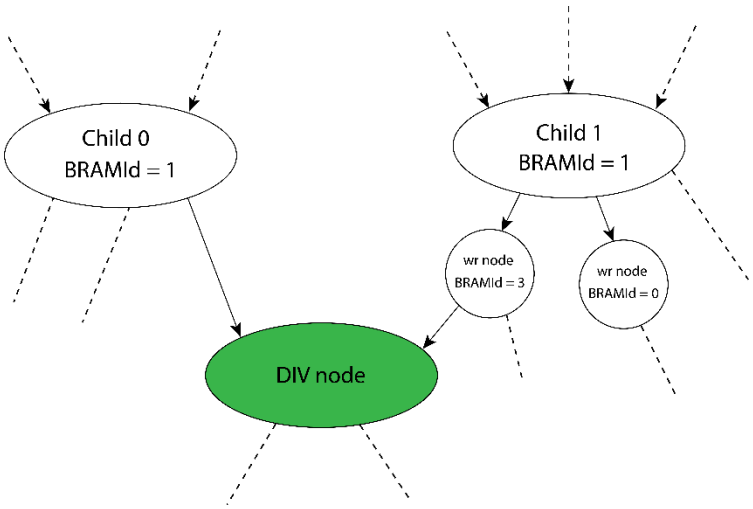



Fig 4.2.4(b): Isolated DIV node for demonstrating addition of "wr" node – PART II

Next, in order to understand algorithm 4 intuitively, let's take an example of a MAC node. I have considered **4 single port BRAMs** for this example. Consider the MAC node shown below. The node of interest is highlighted in green.: -

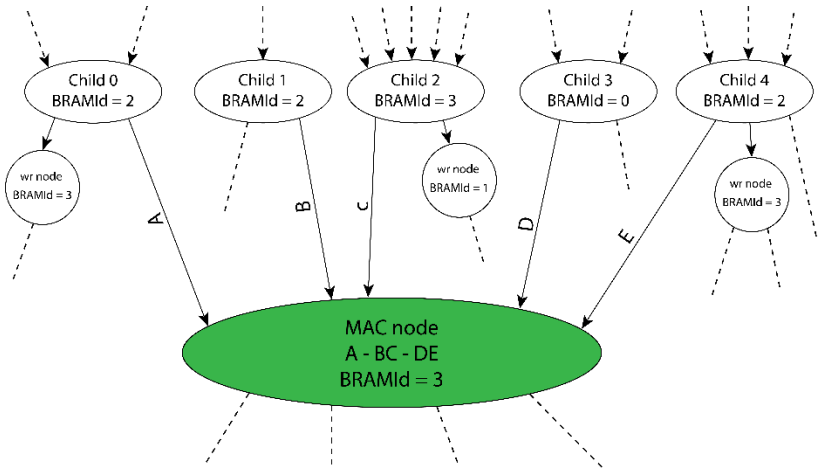



Fig 4.2.4(c): Fig: Isolated MAC node for demonstrating addition of "wr" node – PART I

It is important to note that for MAC node the BRAMId of MAC node itself is also used for deciding whether "wr" node should be added or not. This is necessary because a MAC node can have more than 1 MAC operation. So, if for some reason, the subsequent MAC operation belonging to the same MAC node cannot be scheduled, the intermediate result is stored in the BRAM corresponding to the BRAMId of the MAC node. In the above figure if A-BC is scheduled first, A will be read from BRAMID = 2 (Child 0). But suppose A-DE (assume $A_{new}$ = A-DE) was scheduled first, and when the result was available, $A_{new}$-BC could not be scheduled. So $A_{new}$ will be stored in BRAM with BRAMId = 3. Later, when $A_{new}$-BC will be scheduled, $A_{new}$ will be read from BRAM with BRAMId = 3. Hence, we can see s that when scheduling MAC operation with B & C, A can be fetched from either BRAMId = 2 or 3. This holds true for all the MAC operations present in a single MAC node. This is also the reason why I need minimum 4 ports (either 4 single port BRAM or 2 dual port BRAMS) for algorithm 4 to work.

Let's proceed with the example. First, consider the MAC operation A-BC. For this MAC operation, "BRAMPortUsageList_1" & "BRAMPortUsageList_2" as discussed in algorithm 4 is given below: -

**BRAMPortUsageList_1 = [0,0,1,2]**

**BRAMPortUsageList_2 = [0,0,2,1]**

From "BRAMPortUsageList_1" itself we can see that **overusedBRAMId = 3**, hence no longer need to check "BRAMPortUsageList_2".

Hence, "possibleWrNodeBRAMList" will be: -

**possibleWrNodeBRAMList = [0,1]**

Note that only those BRAMIds are put in the above list which have available ports as per both "BRAMPortUsageList_1" & "BRAMPortUsageList_2".

Now since overusedBRAMId = 3 which is the BRAMId of Child 2, **selectedChild = Child 2.**

Since Child 2 already has a "wr" node with BRAMId which matches with one of the BRAMIds in "possibleWrNodeBRAMList", no need to create a separate "wr" node. So, we will make necessary changes in the connection between the Nodes. The new graph will look as given below: -

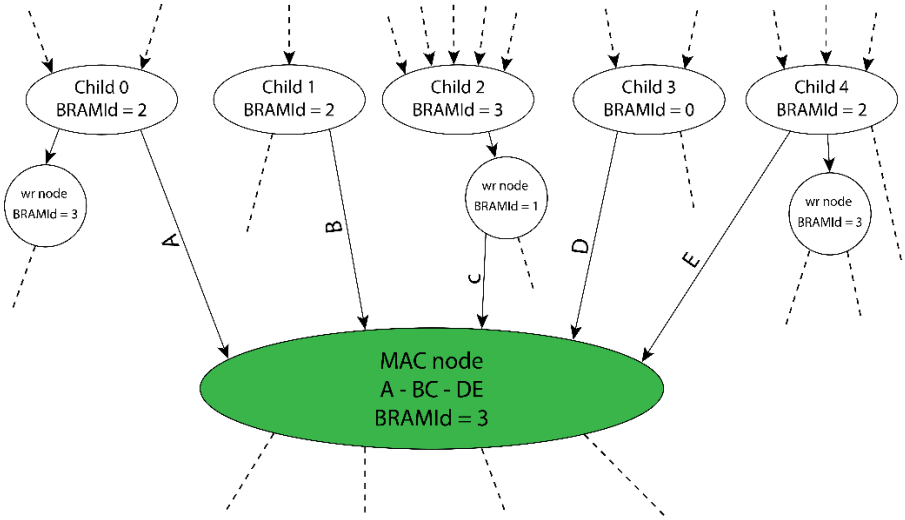



Fig 4.2.4(d): Isolated MAC node for demonstrating addition of "wr" node – PART II

As for each MAC operation we have to check the BRAMPortUsageList twice, "BRAMPortUsageList_1" & "BRAMPortUsageList_2" for MAC operation A-BC according to the new graph will be as follows: -

$$\textbf{BRAMPortUsageList\_1} = [0,1,1,1]$$
$$\textbf{BRAMPortUsageList\_2} = [0,1,2,0]$$

Now from "BRAMPortUsageList_2" we can see that **overusedBRAMId = 2**
Hence, "possibleWrNodeBRAMList" will be: -

$$\textbf{possibleWrNodeBRAMList} = [0]$$

Now since overusedBRAMId = 2 which is the BRAMId of Child 1, **selectedChild = Child 1.** In this case we need to create a "wr" node with BRAMId = 0. So, we will make necessary changes in the connection between the Nodes. The new graph will look as given below: -

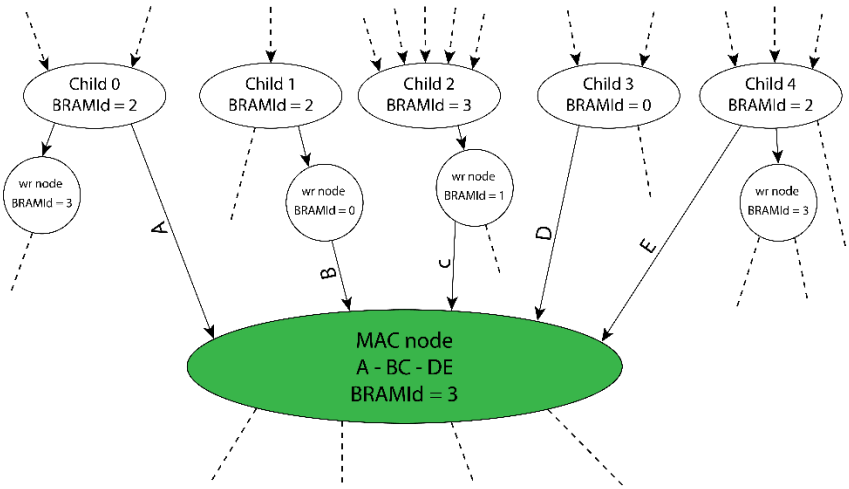



Fig 4.2.4(e): Isolated MAC node for demonstrating addition of "wr" node – PART III

Now we move on to the next MAC operation in the MAC node, i.e. A-DE. For this MAC operation, "BRAMPortUsageList_1" & "BRAMPortUsageList_2" is as follows: -

**BRAMPortUsageList_1 = [1,0,1,1]**

**BRAMPortUsageList_2 = [1,0,2,0]**

Now from "BRAMPortUsageList_2" we can see that **overusedBRAMId = 2**
Hence, "possibleWrNodeBRAMList" will be: -

**possibleWrNodeBRAMList = [1]**

Now since overusedBRAMId = 2 which is the BRAMId of Child 4, **selectedChild = Child 4.** In this case we need to create a "wr" node with BRAMId = 1. So, we will make necessary changes in the connection between the Nodes. The new graph will look as given below: -

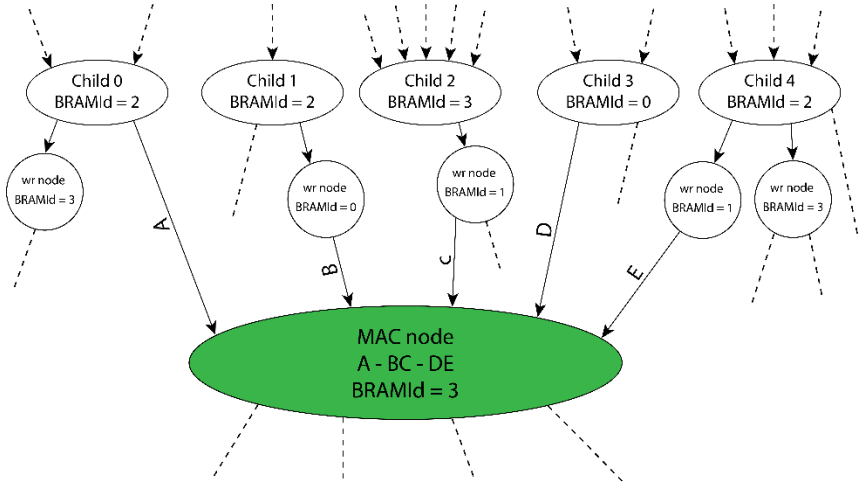



Fig 4.2.4(f): Isolated MAC node for demonstrating addition of "wr" node – PART IV

As for each MAC operation we have to check the BRAMPortUsageList twice, "BRAMPortUsageList_1" & "BRAMPortUsageList_2" for MAC operation A-DE according to the new graph will be as follows: -

**BRAMPortUsageList_1 = [1,1,0,1]**

**BRAMPortUsageList_2 = [1,1,1,0]**

As there is no overusedBRAMId according to both the above lists, and no other MAC operations to be checked in the highlighted MAC node, the previous graph is the final graph.

With this we finish the Symbolic Analysis & DFG Generation Stage of this project.

## 4.3 Generation of Static Schedule for Crossbar Network

The scheduling part of the project is the most complicated part. Like the previous 2 stages, this stage does not have any other sub parts. The scheduling algorithm used for this project is **list scheduling**.

The basic idea behind list scheduling is that operations are scheduled continuously based on availability of operands and resources (number of adders, multipliers etc.). When enough resources are not available, operations are scheduled based on priority.

In order to explain list scheduling, I have taken an example. Then I will explain how the list scheduling used in this project is slightly different from this example.

Consider the dummy graph given below: -

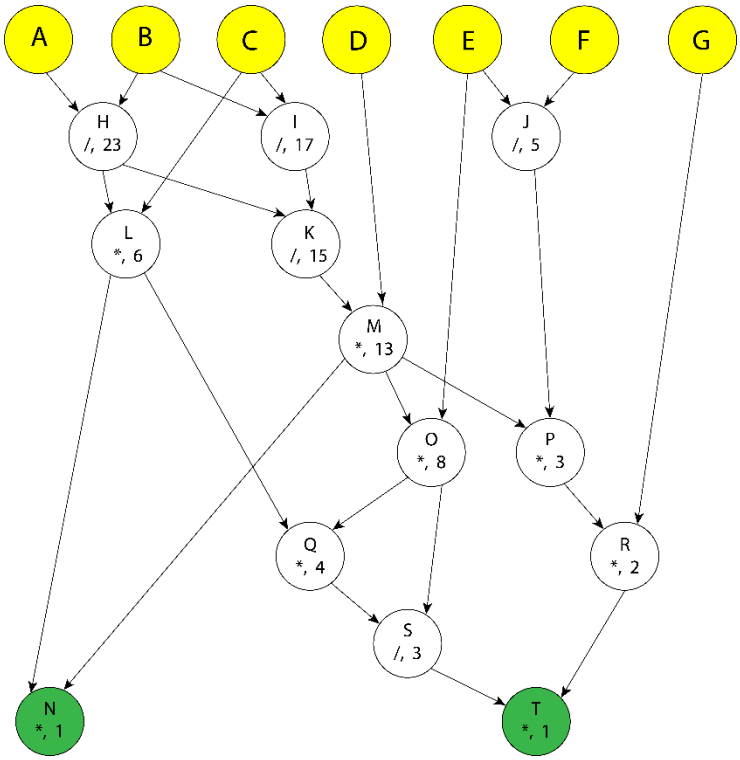



Fig 4.3(a): Dummy graph for explaining list scheduling

In the above graph, leaf nodes are marked with yellow color and root nodes are marked with green color. Two types of arithmetic operations are used, DIV("/") and MUL("*"). **Latency of DIV is 2** and **latency of MUL is 1**. The number inside each of these nodes mark it's priority and the "/" node priority formula mentioned in section 4.2.1 is used to calculate the priority of every node except leaf nodes.

In terms of resources assume that we have **2 MUL** units and **2 DIV units** which are **pipelined** i.e. new operations can be assigned to the arithmetic units every cycle irrespective of their latency.

For demonstrating list scheduling with this graph, I have considered 4 lists "schedulable_list", "execution_list" & "cyclesLeft_list". The last two lists go hand in hand. The contents of these lists in each cycle is given below: -

***Time = 0***

initial schedulable_list = | H | I | J |

execution_list = | H | I |

cyclesLeft_list = | 2 | 2 |

updated schedulable_list = | J |

***Time = 1***

initial schedulable_list = | J |

execution_list = | H | I | J |

cyclesLeft_list = | 1 | 1 | 2 |

updated schedulable_list = Null

***Time = 2***

initial schedulable_list = | L | K |

execution_list = | J | L | K |

cyclesLeft_list = | 1 | 1 | 2 |

updated schedulable_list = Null

***Time = 3***

initial schedulable_list = Null

execution_list = | K |

cyclesLeft_list = | 1 |

updated schedulable_list = Null

***Time = 4***

initial schedulable_list = | M |

execution_list = | M |

cyclesLeft_list = | 1 |

updated schedulable_list = Null

***Time = 5***

initial schedulable_list = | N | O | P |

execution_list = | O | P |

cyclesLeft_list = | 1 | 1 |

updated schedulable_list = | N |

***Time = 6***

initial schedulable_list = | N | Q | R |

execution_list = | Q | R |

cyclesLeft_list = | 1 | 1 |

updated schedulable_list = | N |

***Time = 7***

initial schedulable_list = | N | S |

execution_list = | N | S |

cyclesLeft_list = | 1 | 2 |

updated schedulable_list = Null

***Time = 8***

initial schedulable_list = Null

execution_list = | S |

cyclesLeft_list = | 1 |

updated schedulable_list = Null

***Time = 9***

initial schedulable_list = | T |

execution_list = | T |

cyclesLeft_list = | 1 |

updated schedulable_list = Null

***Time = 10***

initial schedulable_list = Null

execution_list = Null

cyclesLeft_list = Null

updated schedulable_list = Null

Now, the concept behind list scheduling used in this project is same as the above example with 2 main differences: -

1. The operands have to be fetched from memory, and the results have to be stored in memory.
2. The MAC type nodes used in this project can internally have multiple MAC operations. So, whichever MAC operation is ready in a MAC node, will be scheduled

Among these two points, the first point is the bigger hurdle in writing an efficient scheduler. This is because the memory reads have to be very strategically scheduled. I will discuss about this later in this problem section.

The scheduler implemented in this project takes the following inputs: -

- Number of BRAM blocks
- Read & write latency of BRAM blocks
- Number of ports per BRAM block
- Number of MAC and DIV units
- Latencies of MAC and DIV units

According to me, the best way to explain the scheduling algorithm is to first mention the critical lists and variables used in the scheduling algorithm and then explain the scheduling algorithm with a flow chart.

**Lists and Variables used in the Scheduling Algorithm**

All the list names that ends with "_cycles_left", stores the number of cycles required for the corresponding operation to end. A particular operation ends when the value at that particular index of "_cycles_left" list that corresponds to that particular operation becomes 0. The operation can be memory read, memory write or arithmetic operation.

The important lists used in scheduling which will be also be used to explain the flow chart are mentioned below: -

a) **ready_list**
   This is the only list which does not retain its state as we move from one clock cycle to the next, i.e. this list is empty at the beginning of each clock cycle. Whenever we finish writing a node in memory, that node is also written to the "ready_list".

b) **schedulable_list & schedulable_list_cycles_left**
   "schedulable_list" contains those nodes which can be scheduled after their operands have been read from memory. "ready_list" is used to update the "schedulable_list" after which "ready list" is cleared off. "schedulable_list_cycles_left" and schedulable_list go hand in hand. When a node has not been planned to be scheduled, the corresponding value in "schedulable_list_cycles_left" is -1. When a node in "schedulable_list" has been planned to be scheduled, first its operands have to be read. At that cycle, the corresponding index in "schedulable_list_cycles_left" is written with a value = read_latency of BRAM. When this value turns 0 and that node has still not been scheduled, the value is reset to -1. And if the

node has been scheduled, it's cleared from the "schedulable_list" and corresponding index in "schedulable_list_cycles_left" is also cleared off.

c) **execution_list & execution_list_cycles_left**
A node is put into this list when it is assigned to an arithmetic unit. "execution_list" and "execution_list_cycles_left" go hand in hand. "execution_list_cycles_left" is used to keep a track of how many more cycles are required to finish a particular arithmetic operation. When a node is put in "execution_list", the corresponding index in "execution_list_cycles_left" is assigned a value = latency of that arithmetic unit.

d) **mem_read_list & mem_read_list_cycles_left**
A node is put in this list when its value has to be read from memory. "mem_read_list" and "mem_read_list_cycles_left" go hand in hand. "mem_read_list_cycles_left" is used to keep a track of how many more cycles are required for the node to be read from memory. When a node is put in "mem_read_list", the corresponding index in "mem_read_list_cycles_left" is assigned a value = read_latency of BRAM.

e) **mem_write_list & mem_write_list_cycles_left**
These two nodes are used to track the write status of nodes that are being written to memory. They perform the task similar to "mem_read_list" and "mem_read_list_cycles_left".

The above-mentioned lists are part of the function "scheduleNoReg_exp" written in file "Schedule.cpp" of the accompanying project. Next, I would list the variables that are a part of the struct "Node" mentioned in file "Graph.h" of the accompanying project.

a) **readyBit**
This is a bool type variable. This variable indicates whether the final value of node is saved in memory.

b) **nodeEvalStatus**
This is a list. This list is exclusively used for MAC type nodes. The size of this list is same as the number of MAC operations in a MAC node. A single MAC node may be performing a long operation consisting of 2 or more MAC operation like (A-BC-DE). In order to keep a track of which all operations have been done, this variable is used. Every index location in this this list can hold 3 values, 0, 1 or 2. Status value 0 indicates that the particular MAC operation is not evaluated. Status value 1 indicates that the particular MAC operation has been evaluated. Status value 2 indicates that the particular MAC operation is currently being evaluated

c) **latestIntermediateResultAvailable**
If a MAC node has multiple MAC operations, the intermediate results might have to be saved in memory. This variable tells if the intermediate result is updated in memory to the latest evaluation. This variable can hold 3 values -1, 0 or 1. Status value -1 indicates that the evaluation of MAC node has still not started. This also means that the value of all the locations in the list "nodeEvalStatus" is 0. Status value 0 indicates that node is currently being evaluated or is in the process of being written to memory(i.e. the latest intermediate

result is not available in memory). Status value of 1 indicates that the latest intermediate result is saved in memory.

The lists and the variables mentioned above are the most important ones. Using them I will explain each stage of the scheduler flowchart. Every stage in the flowchart is dedicated to updating one or two specific lists which will be shown in the flowchart. But in reality, sometimes it's necessary to make some minor changes to another list which is not a major focus of that stage. Such granularities will be mentioned when I explain each of the stages. The scheduler flowchart is mentioned below: -

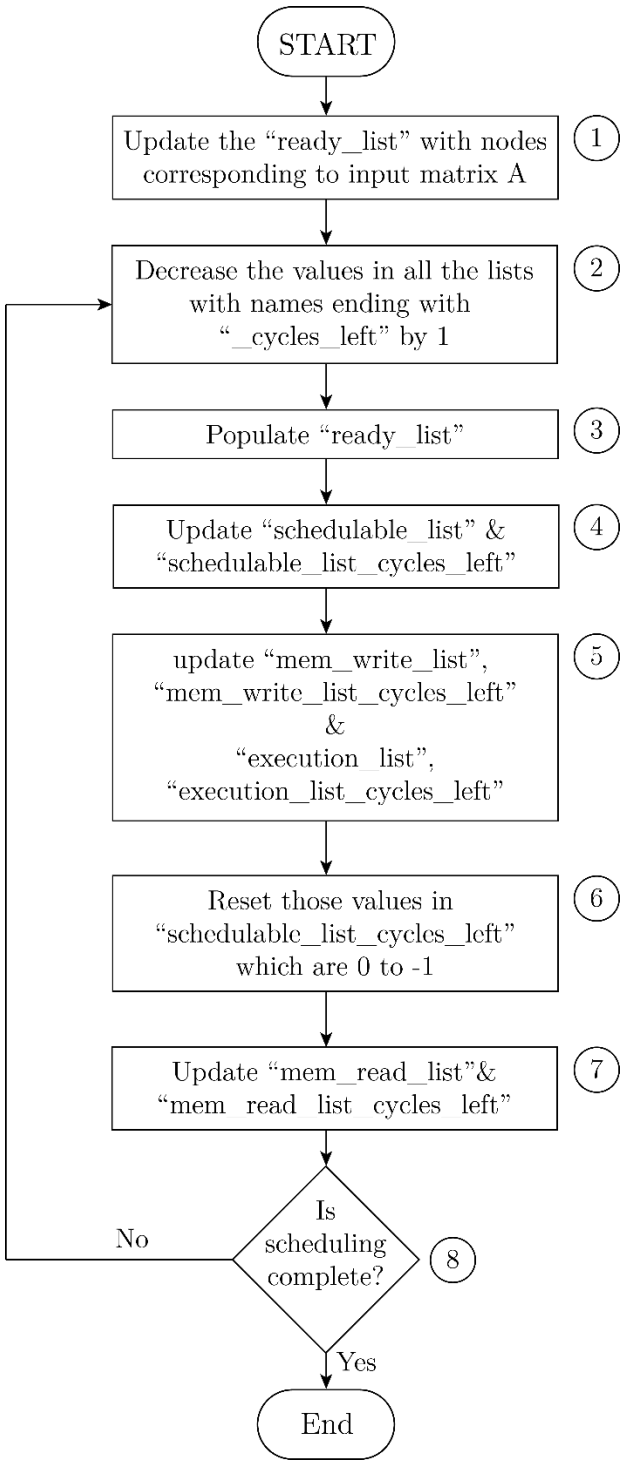



Fig 4.3(b): Flowchart of the scheduling algorithm

As seen from the above flowchart, there are total of 8 stages. The most complicated is the 7th stage followed by the 5th stage. The remaining stages are relatively much simpler.

***Stage 1: -***

The nodes in the graph which corresponds to input matrix A are already stored in BRAM. These are the nodes with type "rd" (i.e. memory read). For all these nodes the "readyBit" is set to true and "latestIntermediateResultAvailable" is set to 1. These nodes are also pushed to "ready_list". Apart from nodes of type "rd", there is one more node of type "const". This node corresponds to a constant value of 0. This is required when the MAC node operation is of type 0 – BC – DE. . . . It's important to note that there is only one node of type "const" in the entire graph. And for this node also "readyBit" is set to true and "latestIntermediateResultAvailable" is set to 1.

***Stage 2: -***

In this stage the values stored in all the lists with names ending with "_cycles_left" is decreased by 1. Note that these are the lists which keep a track of the number of cycles left to finish a particular operation. When a value stored in list becomes 0, it indicates that the operation has finished.

***Stage 3: -***

For the nodes whose memory write is complete (corresponding index in "mem_write_list_cycles_left" has become 0), their "latestIntermediateResultAvailable" is set to 1 and "readyBit" is also updated. For DIV nodes, "readyBit" is set to true whereas for MAC nodes "readyBit" is set to true only when all the MAC operations in that MAC node is complete. All these nodes are then pushed to "ready_list" and removed from "mem_write_list". The corresponding index position in the list "mem_write_list_cycles_left" is also removed.

***Stage 4: -***

Using the "ready_list" generated in stage 3, I update the "schedulable_list" & "schedulable_list_cycles_left". The value stored in the new entries of "schedulable_list_cycles_left" is -1. While adding new nodes to the "schedulable_list", it's made sure that the same node does not already exist in the list. This is possible with MAC nodes if multiple MAC operations belonging to the same MAC node becomes eligible for scheduling. At the end, "ready_list" is cleared off.

***Stage 5: -***

In this stage, first we create a temporary list called "live_list". This list is populated with those nodes from "mem_read_list" & "execution_list" which have finished operation (values in corresponding index of "_cycles_left" list becomes 0). For MAC nodes from "execution_list" which have finished operation, the element of the list "nodeEvalStatus" whose value is 2, is changed to 1. Then using the nodes present in "live_list", new nodes are found which can be scheduled next, and put it in another temporary list called "newExecutableNodes". **If arithmetic units are available in the current cycle and if there will be enough BRAMs available to store the result**

**when they will become ready, nodes present in "newExecutableNodes" are moved to "execution_list"** and the corresponding new indices of "execution_list_cycles_left" is assigned a value equal to latency of the corresponding arithmetic unit. At the same time when the nodes are moved to execution list, if they are present in "schedulable_list", they are removed. The corresponding index position of "schedulable_list_cycles_left" is also removed. If the node that is being moved to "execution_list" is a MAC node, depending on which MAC operation is scheduled, the corresponding index location of the list "nodeEvalStatus" is changed from 0 to 2 & "latestIntermediateResultAvailable" is changed to 0. Next, we need to find the appropriate nodes to be written back to memory. Note that we need not save all the nodes which have finished execution, either fully or partially (MAC nodes) to memory. In fact, I have specified in section 4.2.3 that we do not require the value computed by nodes with name "mac_subL_x1_y1" in the final result. Only the parent node of that node i.e. "L_x1_y1" makes use of the value computed by that node. To be specific there are 4 cases when a node is to be saved in memory. These 4 cases are mentioned below: -

- The result of a DIV node is always saved in memory as the values computed by these nodes corresponds to elements of L matrix.
- If the node is MAC node and further MAC operations are remaining to be scheduled for that node and it's not possible to put another MAC operation of the same node in execution list. In that case, the intermediate result is stored in memory.
- If the node is a MAC node which has completed all the MAC operations and this node is calculating an element of U matrix.
- If the node is a MAC node which has completed all the MAC operations and the name of the MAC node is of form "mac_subL_x1_y1" and it's not possible to put the node "L_x1_y1" in the execution list.

***Stage 6: -***

When we plan to schedule a node from "schedulable_list", first the operands have to be read from memory. Once the operands are fetched, before the node is moved to "execution_list", it's ensured that enough arithmetic units are available in the current cycle (the cycle in which the node will be moved to "execution_list") and once the execution of the node is actually done, enough BRAMs should also be available to save the node. Hence any one of the two checks can fail, although enough care is taken while fetching the operands from memory. In ideal scenario if an index location of the list "schedulable_list_cycles_left" is 0, then that corresponding node from "schedulable_list" would already have been removed in stage 5. If any index location of the list "schedulable_list_cycles_left" is still 0 at this stage, it means we have failed to schedule the node which was planned to be scheduled in the current cycle due to resource constraints. Hence, it's important to reset those values from the list "schedulable_list_cycles_left" to -1 which are still 0.

***Stage 7: -***

The "mem_read_list" is updated in this stage. But it is updated in two stages.

In **1st stage**, first I create a temporary list called "live_list" and populate it with nodes that will finish operation memory_read latency cycles after the current cycle. Then I find new nodes that can be put in "execution_list", memory_read latency cycles after the current cycle with the condition that at least one of the children of these nodes is a node from the live list. These nodes are put in the list "newExecutableNodes". Now, for each of the nodes in "newExecutableNodes", first I check if there are enough BRAM ports available in the current cycle to fetch the additional operands from memory. If the condition is satisfied, I check if there is enough arithmetic units available memory_read latency cycles after the current cycle so that the node from "newExecutableNodes" can be put in "execution_list". If this condition is also satisfied, I check if enough BRAM ports are available when the node from "newExecutableNodes" finishes its execution, so that this node can be saved in memory. If this condition also satisfies I put the additional nodes needed to schedule the node from "newExecutableNodes" to "mem_read_list". Corresponding new indices are added to the list "mem_read_list_cycles_left", and the value assigned to these new indices is equal to memory_read latency.

In **2nd stage,** I consider the nodes from "schedulable_list". For each node in the "schedulable_list", I perform all the 3 checks as I did for the 1st stage. If all the checks are successful, I put all the operands required to schedule the node from "schedulable_list" in "mem_read_list" & the corresponding index position of "schedulable_list_cycles_left" is set to value = memory_read latency. Corresponding new indices are added to the list "mem_read_list_cycles_left", and the value assigned to these new indices is equal to memory_read latency.

Note that the three checks I have mentioned, although they sound simple but are quite challenging to implement. A large part of the performance of the project depends on how well stage 7 has been implemented.

***Stage 8: -***

In this stage I just check if the "readyBit" of every node except those nodes whose names start with "mac_subL" is set to true. If the condition satisfies, it means scheduling is complete.

Hence, we can see that mainly because the operands have to be fetched from memory and the results have to be stored back, the complexity of scheduling increased by many times.

## 4.4 Hardware Implementation of the Crossbar Network

The hardware architecture that was chosen for this project is crossbar architecture. In this architecture, every unit is connected to every other unit. Such kind of connection does not pose any restriction when it comes to communication between any two units (such as a BRAM and a MAC), but it comes at a cost. As we increase the number of units, the number of interconnections keep on increasing exponentially. This is a disadvantage when it comes to scalability. Often it may happen that a moderately large crossbar network will fail to meet the timing requirements on lower end FPGAs although a large part of FPGA resources is still not being used. This is because the routing gets too complicated and there may not be enough switch boxes in the FPGA. The advantage of fully connected crossbar network is that it's simple to implement. The block diagram of the core LU decomposition hardware is shown below: -

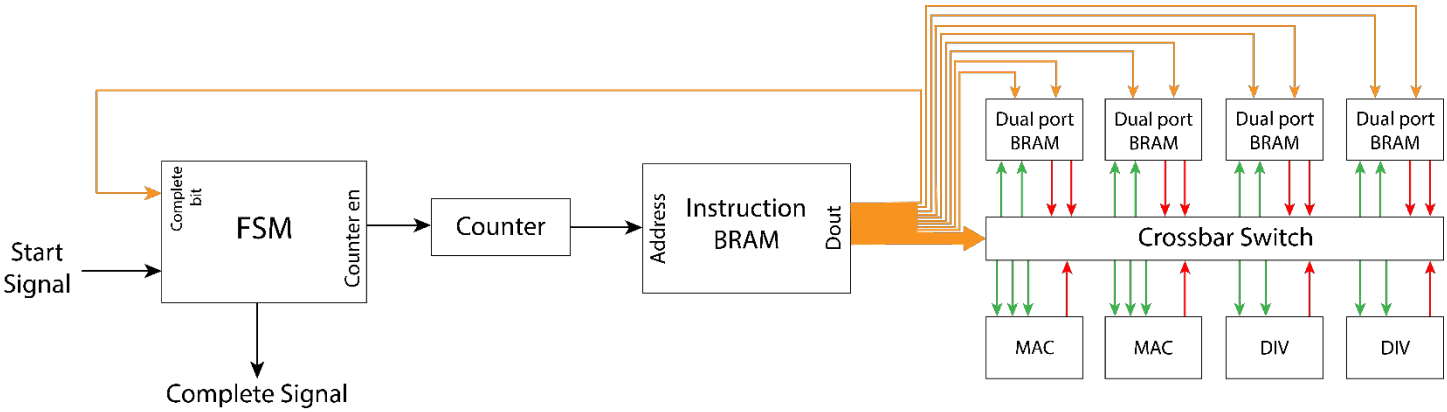



Fig 4.4(a): Block diagram of LU decomposition Crossbar Network

Note that the word size of instruction BRAM is quite large (~150 – 300 bits) and it depends on the number of MACs, DIVs and BRAMs we use in our design. The information contained in each word can be divided into 3 parts.

- The 1st part contains data that goes into the select pins of the muxes present at the input of each unit (BRAM or arithmetic). This is shown by the fat orange arrow going into the crossbar switch. The green arrows in the diagram are the inputs. According to the above figure there are 18 such muxes. The mux structure is shown below: -

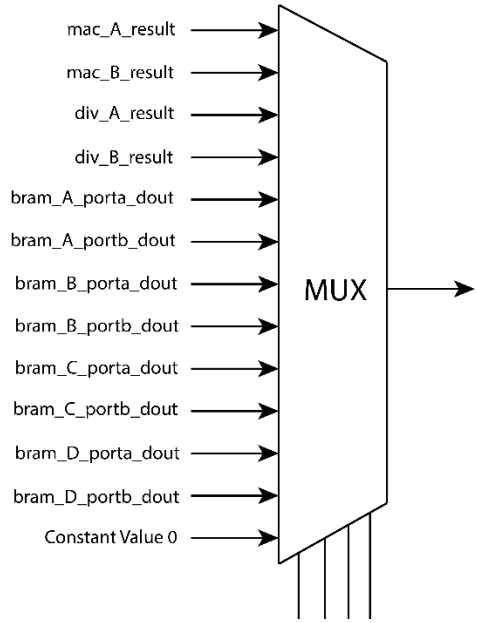



Fig 4.4(b): Mux at the input of each Unit

Note that the last input to the MUX is a constant value of 0. This is required when the MAC operation is of type 0-BC-DE…….

- The 2nd part contains data about the address of each of these BRAMs. This is shown by orange arrows going into the BRAM.
- The 3rd part is the finish bit. It's is the last bit in every instruction word. This bit is used to indicate the last instruction. For the last instruction this bit is 1, and for every other instruction, this bit is 0.

The above block diagram does not show the additional circuitry used to load data into the BRAM and read data from the BRAM using the Zynq processing system. The additional circuitry is shown below: -

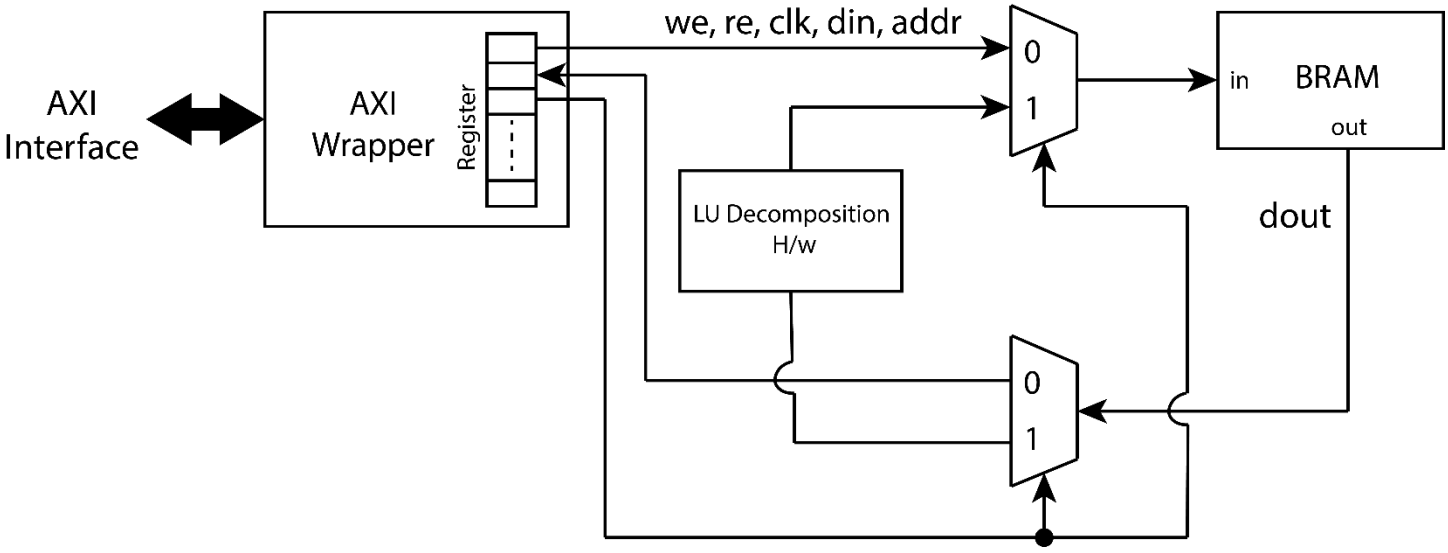



Fig 4.4(c): Additional circuitry to read/write data into BRAM from ZYNQ PS

The “Start Signal” & “Complete Signal” shown in the crossbar network block diagram are also interfaced with Zynq PS using AXI wrapper. This is omitted from the block diagram for simplicity purpose. The final block diagram obtained after interfacing the LU decomposition hardware accelerator to the Zynq PS is shown below: -

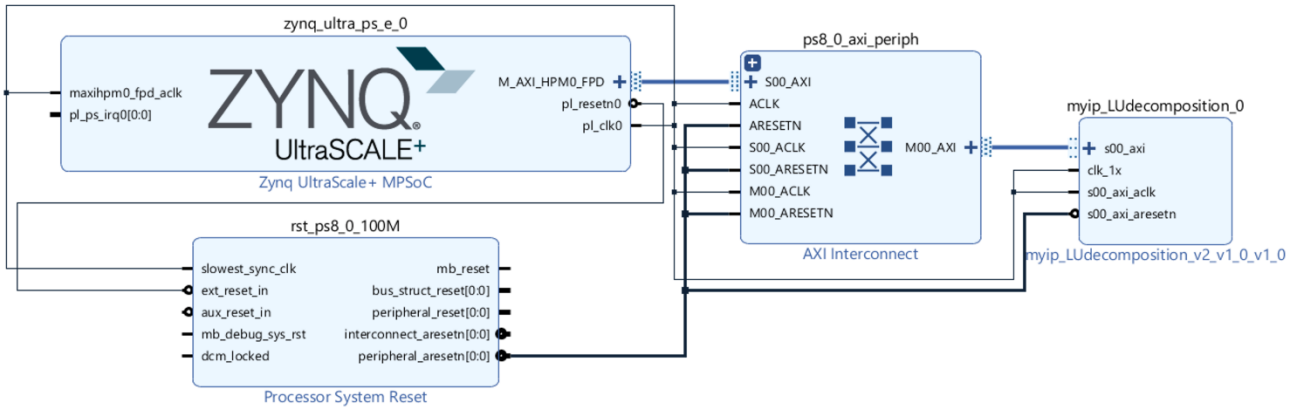



Fig 4.4(d): Interfacing LU decomposition IP with Zynq PS

## 4.4.1 Converting Dual Port to Quad Port BRAM

This project also discusses about the conversion of dual port BRAMs to quad port BRAMs using additional clock of twice the frequency. FPGA vendors generally provide BRAM IPs up to dual port BRAMs. So, if we want a higher port BRAM for any reason, we need to use 2 clocks. The basic idea is to encapsulate the dual port BRAM IP in a wrapper (referred to as QuadPort BRAM wrapper in this project) that converts it to quad port BRAM. Two types of QuadPort Bram wrappers have been discussed in this project: -

- Synchronous QuadPort BRAM wrapper
- Asynchronous QuadPort BRAM wrapper (BETA)

### 4.4.1.1 Synchronous QuadPort BRAM wrapper

The synchronous QuadPort BRAM wrapper is shown in the figure below: -

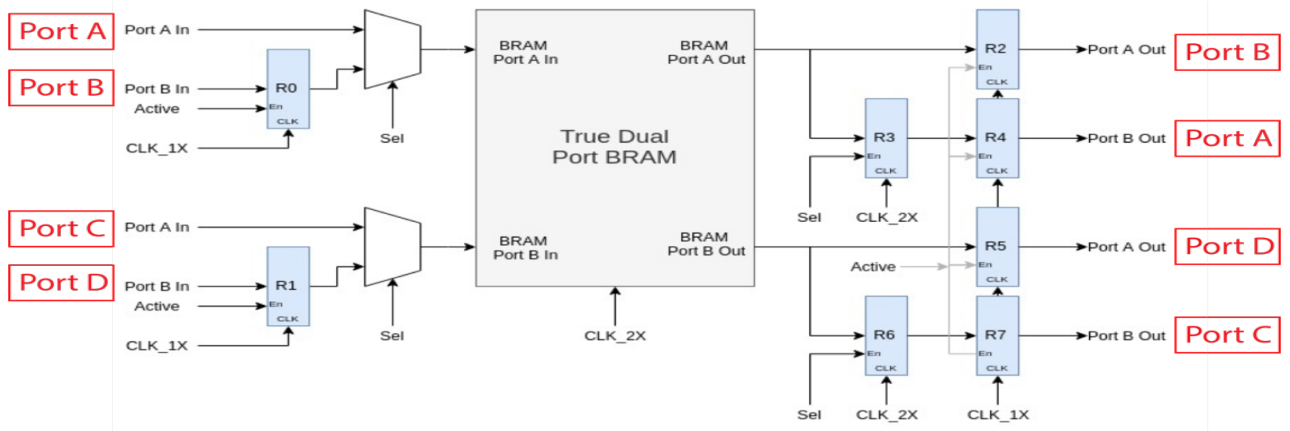



Fig 4.4.1.1(a): Synchronous QuadPort BRAM

The blue colored blocks in the above figure are D flipflops. The timing diagram for the above QuadPort BRAM wrapper is given below: -

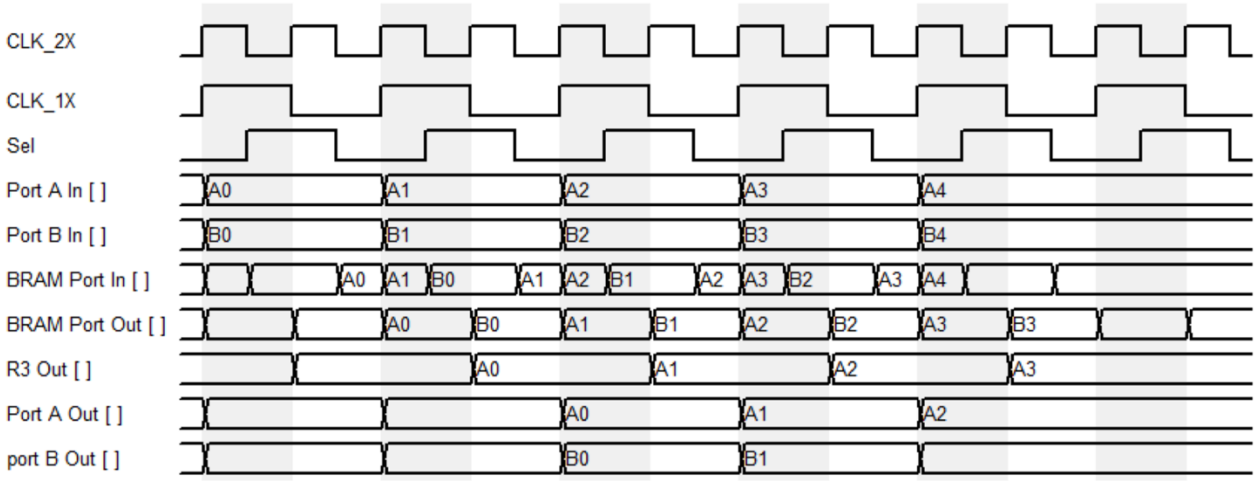



Fig 4.4.1.1(b): Timing diagram for synchronous QuadPort BRAM wrapper

The final block diagram obtained after interfacing the LU decomposition hardware accelerator to the Zynq PS if we use the above QuadPort BRAM wrapper is shown below: -

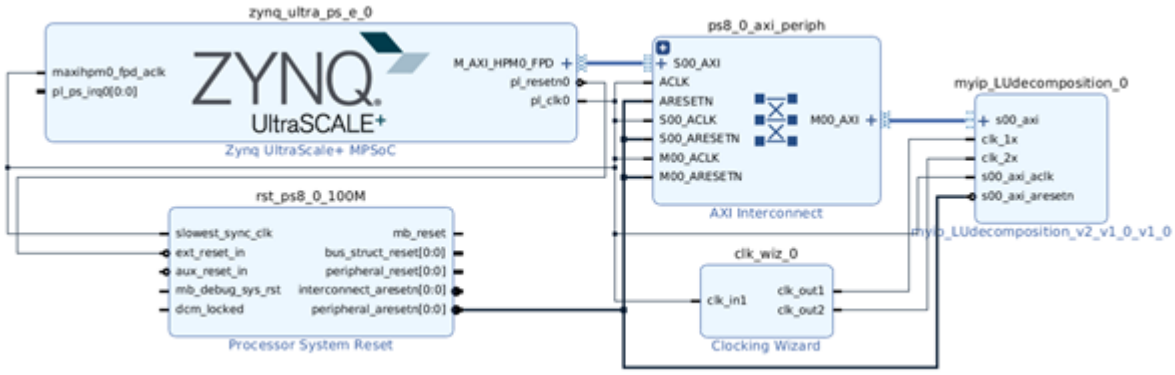



Fig 4.4.1.1(c): Interfacing LU decomposition IP(which uses QuadPort BRAM) with Zynq PS

## 4.4.1.2 Asynchronous QuadPort BRAM wrapper (BETA)

This wrapper uses latches instead of flipflops. This wrapper was designed to withstand a clock skew of (time period of clk_2x)/2. But in simulation it's performing even better than that. Full potential of this QuadPort BRAM wrapper is still not identified. Unfortunately, this wrapper is not synthesizable without generating timing violation. This is so because clk_1x and clk_2x (i.e. the two clock signals) have been used to generate some additional control signals (i.e. the clock signals have been passed through AND, OR, NOT gates). This is seen as clock gating by the synthesizer, and it's necessary to declare certain paths as false paths using constraints. Unfortunately, this part has not been done. So, the hypothesis is, if we specify all the false paths in the constraint file, the design will synthesize without timing violations. The asynchronous QuadPort BRAM wrapper is shown below: -

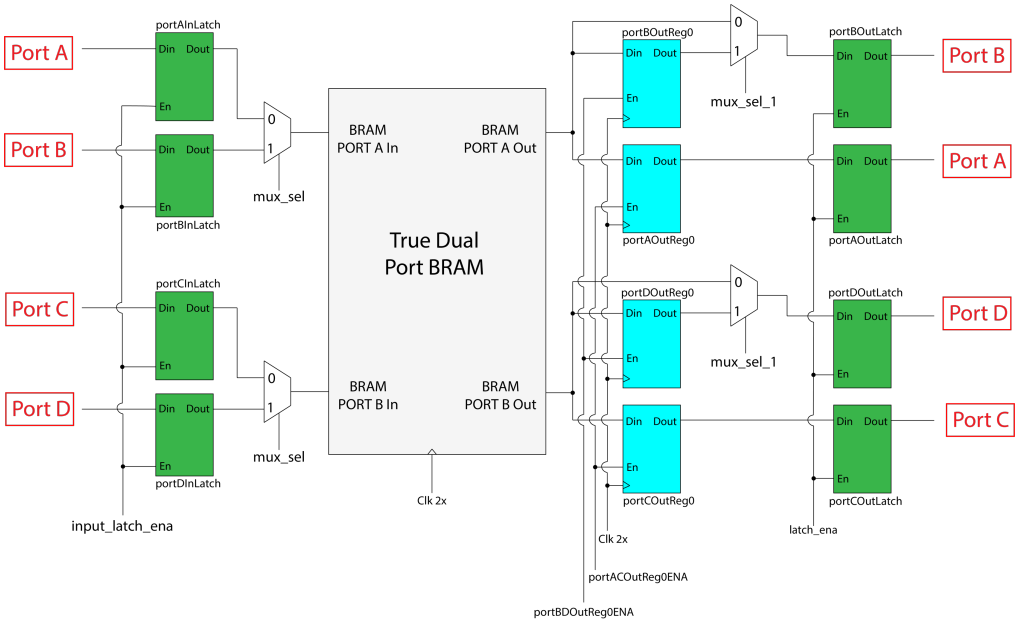



Fig 4.4.1.2(a): Asynchronous QuadPort BRAM

In the above figure, the light blue colored boxes are D flipflops and the dark green colored boxes are D latches. Clk_1x signal is not explicitly visible in the block diagram, but they are used to generate the control signals “input_latch_ena”, “mux_sel” etc.

# Chapter 5

# Results

After running the scheduler for various hardware configurations, I have decided to discuss on four observations as given below: -

1. Quad port vs Dual port BRAMs
2. Performance comparison with previous scheduler implementation
3. Performance comparison with Nechmas' implementation
4. Total error comparison while using 32-bit float as compared to 64-bit double

The conclusions for the above for observations will also be discussed in the same section.

## 5.1 Quad port vs Dual port BRAMs

All the graphs discussed in section 4.4.1 have been obtained after running the scheduler with 4 Quad port or 8 Dual port or 16 Single port BRAMs for various matrices. 4 MACs and 4 DIVs have been used with all the above 3 BRAM configurations

Consider the graph shown below where I have compared the number of cycles required by the scheduler for the 3 configurations. One using 4 Quad port BRAM, the other using 8 Dual port BRAMs and the last one using 16 Single port BRAMs.

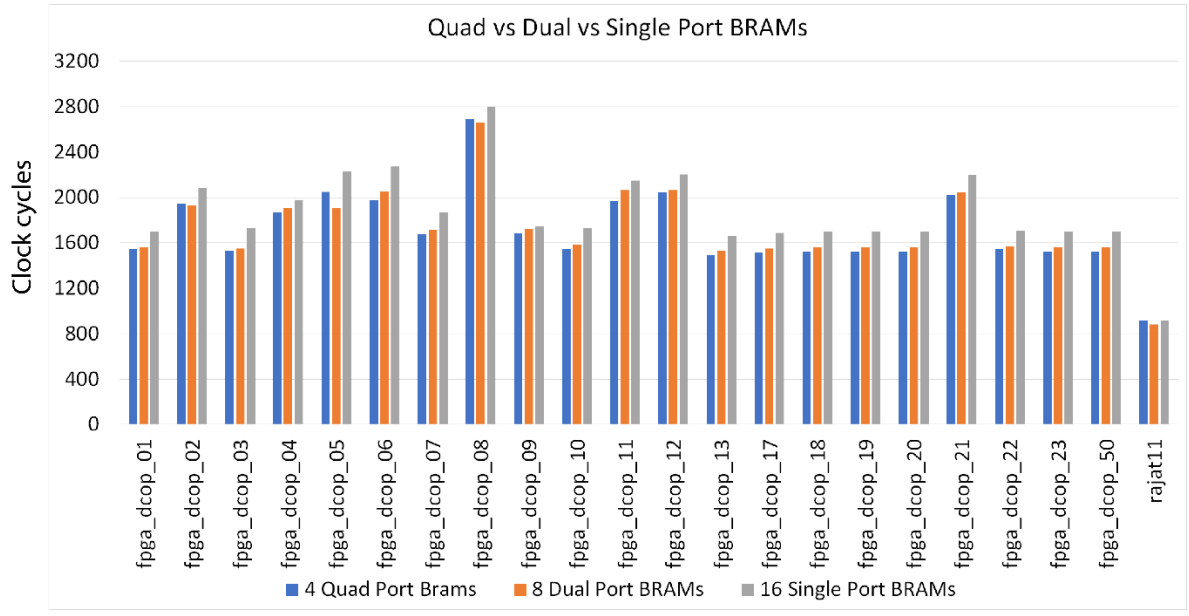



Fig 5.1(a): Quad vs Dual vs Single port BRAM cycles comparison

From the above graph it is clear that there is greater improvement as we move from Single port to Dual port BRAM as compared to when we move from Dual port to Quad Port BRAM. But to make the results better understandable consider the next graph where I have plotted the % improvement.

The % change in performance obtained while moving from Single port to Dual port and Dual port to Quad port is shown below: -

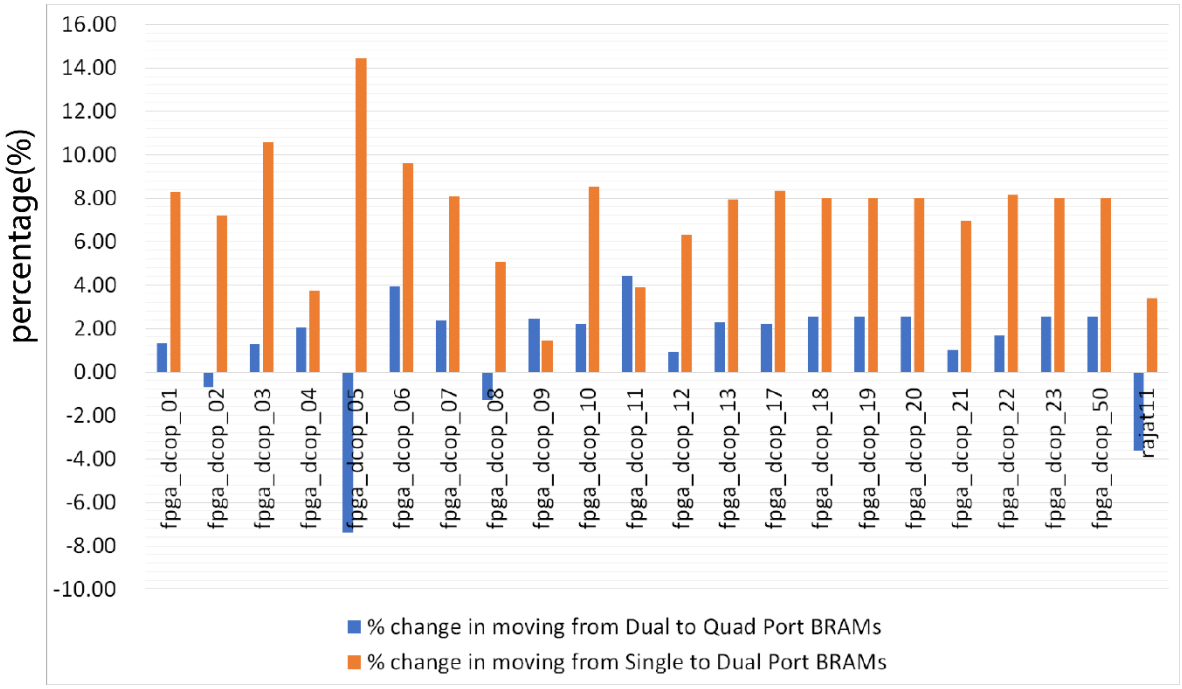



Fig 5.1(b): Quad vs Dual vs Single port BRAM % improvement comparison

From the above graph, it's quite clear that as we move from Single to Dual port BRAMs, there is performance improvement of around 8%, whereas when we move from Dual to Quad port BRAM, there is performance improvement of around 2%. For some matrices like fpga_dcop_02, fpga_dcop_05 & fpga_dcop_08, there is a drop in performance.
Now, one can argue that mere 8% increase in performance is not worth moving to Dual port BRAMs from Single Port BRAMs. But from our previous discussion, we know that if we are using Single or Dual port BRAMs, we have to add additional "wr" nodes, so that all the operands for any operation (MAC or DIV) can be fetched in same cycle. The next graph shows the percentage of "wr" nodes added while using Single and Dual port BRAMs.

Consider the graph which shows the % of “wr” nodes added while using Single and Dual port BRAMs.

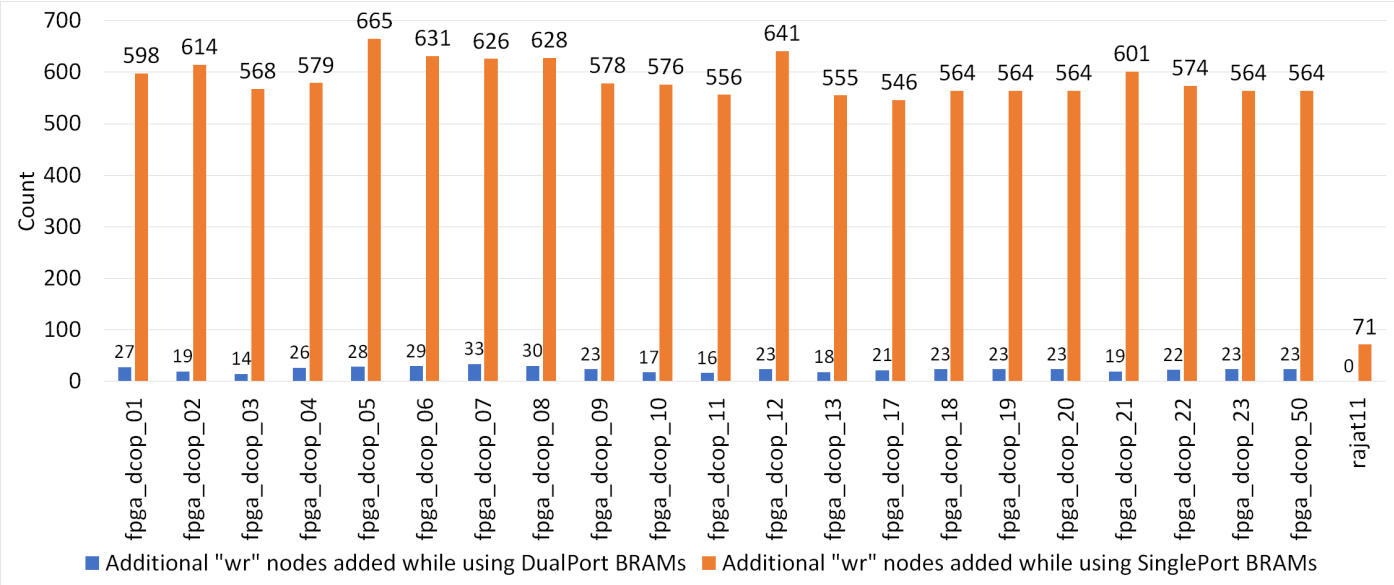



Fig 5.1(c): Comparison of additional “wr” nodes added

From the above graph, it’s clear that we need ~25x more “wr” nodes if we use Single port BRAMs as compared to Dual port BRAMs. Each “wr” node is using a space in BRAM. So, using less number of “wr” nodes helps in saving precious BRAM resources.

**Conclusion: -**

Hence the conclusion that can be obtained is that although moving from Single to Dual port BRAMs give only 8% improvement in performance, but if we compare it with the number of additional “wr” nodes that have to be added while using Single port BRAM, it’s worth using Dual port BRAMs.

## 5.2 Performance comparison with previous scheduler implementation

The previous scheduler that had been implemented worked with only Quad port BRAMs. It was also observed that the previous scheduler code gave runtime errors while working on matrices that were not of the family “fpga_dcop_”. Apart from this there were few other bugs. Hence, I implemented the scheduler again from scratch. The new scheduler works with any number of BRAM ports. The comparison has been done using 4 QuadPort BRAMs, 4 MACs & 4 Divs. The performance comparison of the new and old scheduler is shown below: -

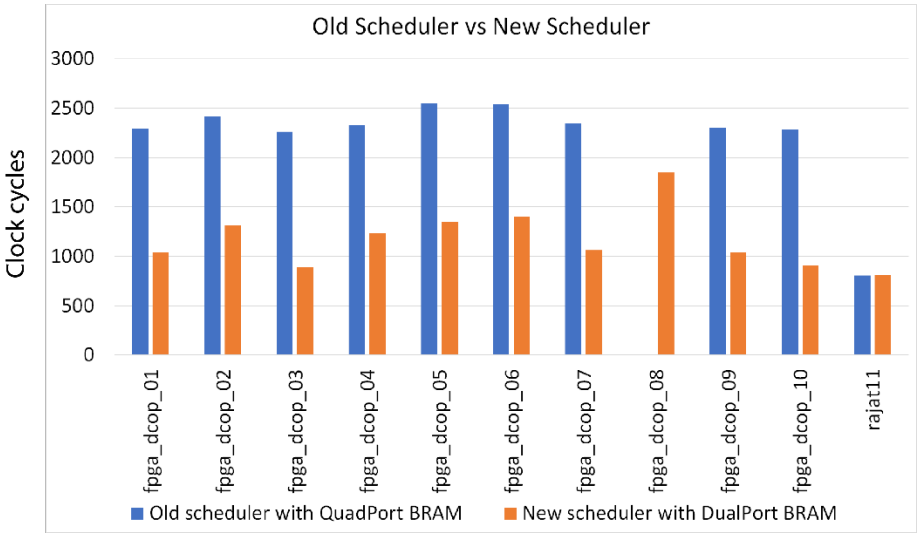



Fig 5.2(a): Comparison of Old and New scheduler

From the above graph it's clear that the new scheduler used 40-50% lesser clock cycles to schedule the same matrix. Also note that "fpga_dcop_08", the cycle count of old scheduler is not available because the old scheduler gave runtime error for that matrix.

**Conclusion: -**

The new scheduler implementation is better than old scheduler implementation as it schedules the matrices in almost half the cycles and can work with any number of BRAM port.

## 5.3 Performance comparison with Nechmas' implementation

Nechma's paper[2] is the main reference paper for this project. Hence, it's important to compare our results with his results. The graph mentioned below compares our results with his results for some of the matrices: -

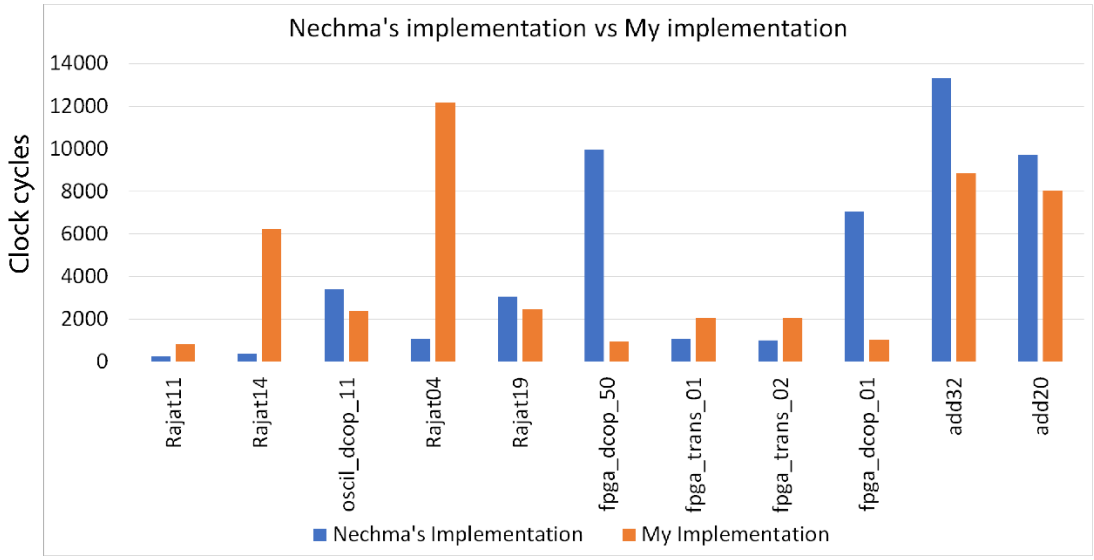



Fig 5.3(a): Comparing performance with Nechma's implementation

The above graph has been obtained with **16 dual port BRAMs, 16 MACs & 16 DIVs. BRAM rd/wr latency = 1, MAC latency = 19 & DIV latency = 28**. These values have been taken from Nechma's paper so that we can keep the comparison as close as possible.

We can see from the above graph that the performance is a mixed bag. For matrices like "Rajat14" and "Rajat04" my implementation is almost 10 times worse. Whereas for matrices like "fpga_dcop_50" & "fpga_dcop_01", my implementation is almost 10 times better. On an average for only 50% cases my implementation is better than Nechma's although this paper focuses on exploiting the fine-grained parallelism.

According to me, the hypothesis behind this performance issue is that we have used MAC arithmetic units instead of using MUL and ADD arithmetic units separately which Nechma[2] has done. Next I have tried to prove that this is the correct hypothesis. Consider the MAC node given below: -

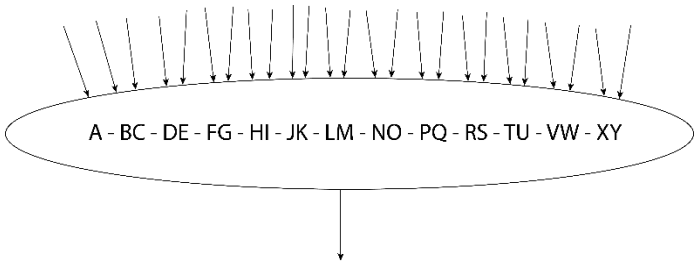



Fig 5.3(b): A MAC node

The above MAC node has 12 operations. We need to wait for MAC operation to finish before we can start another MAC operation from this node. Now, consider we have separate MUL and ADD arithmetic units. The new node will look as given below: -

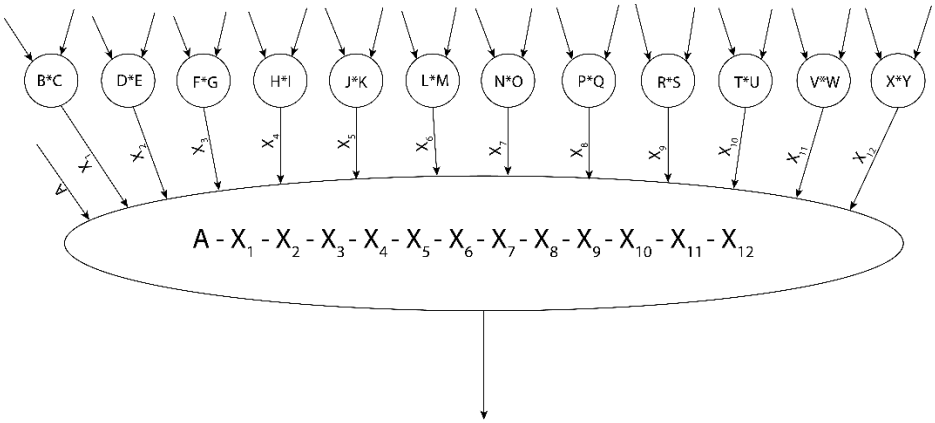



Fig 5.3(c): Modified version of MAC node with MUL and ADD arithmetic units

Now, let's assume that MUL latency is 8 cycles, ADD latency is 11 cycles and MAC latency is 19 cycles. And all the operands are available at the same time. So, time taken by the above MAC node is: -

$$Time = 12*19 = 228 \text{ cycles}$$

And the time taken by the modified version of MAC node where we have used separate MUL and ADD units is: -

$$Time = 8 + 11*4 = 52\ cycles$$

So, we can see that is we use separate MUL and DIV units instead of MAC units, for long MAC operations, time increases logarithmically instead of linearly.

Next, I will show that for matrices "fpga_dcop_50" & "fpga_dcop_01", the number of lengthy MAC operations is almost negligible as compared to the matrices "Rajat04" and "Rajat14" which has large number of lengthy MAC operations.

Let's first consider the graph showing the count of MAC operation/MAC node for "fga_dcop_01" and "fpga_dcop_50" matrices.

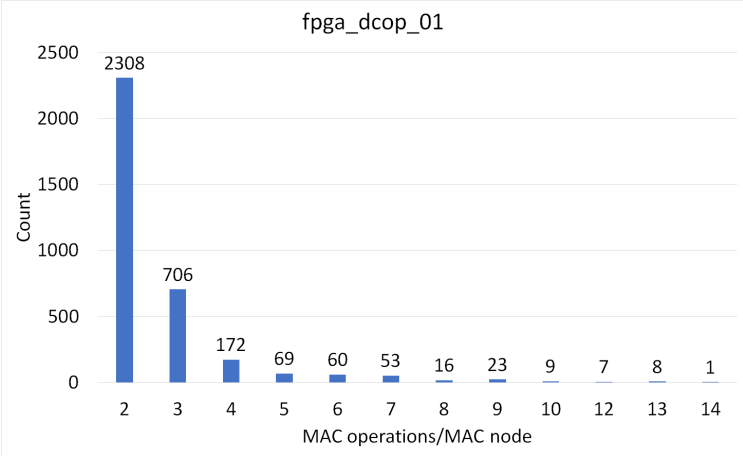



Fig 5.3(d): MAC operations/MAC node for "fpga_dcop_01"

According to the above graph, there are only 8 MAC nodes which have 13 MAC operations and 1 MAC node which has 14 MAC operations.

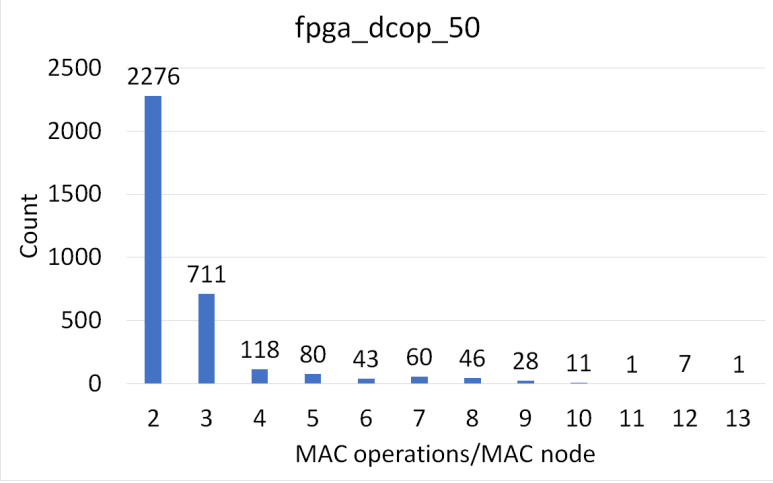



Fig 5.3(e): MAC operations/MAC node for "fpga_dcop_50"

The graph for "fpga_dcop_50" is very similar to "fpga_dcop_01", which explains why these two matrices have similar performance gain as compared to Nechma's implementation.

Now, let's consider the graph showing the count of MAC operations/MAC node for "Rajat04" and "Rajat14" matrices.

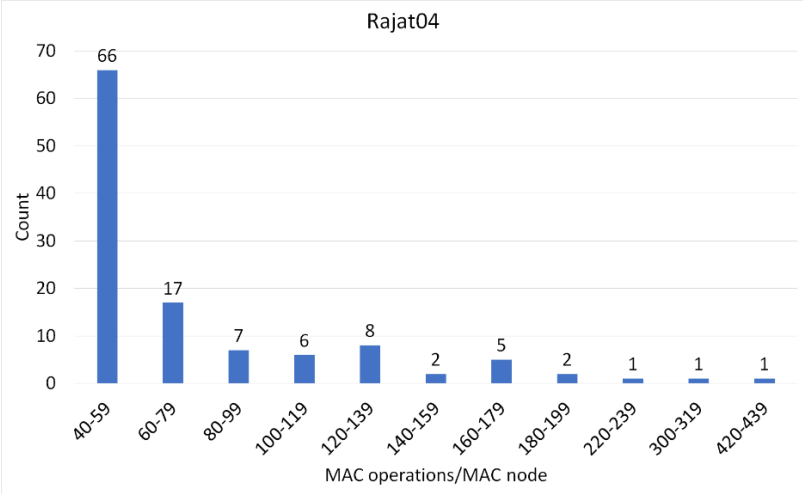



Fig 5.3(f): MAC operations/MAC node for "Rajat04"

For the above graph I have combined the x-axis values in sets of 20 in-order to make the graph length reasonable. Also, I have not displayed the range 1-19 and 20-39 as they were too large and made the rest of the bars in the graph look very small. Range 1-19 has a value 12354 and range 20-39 has a value 671.

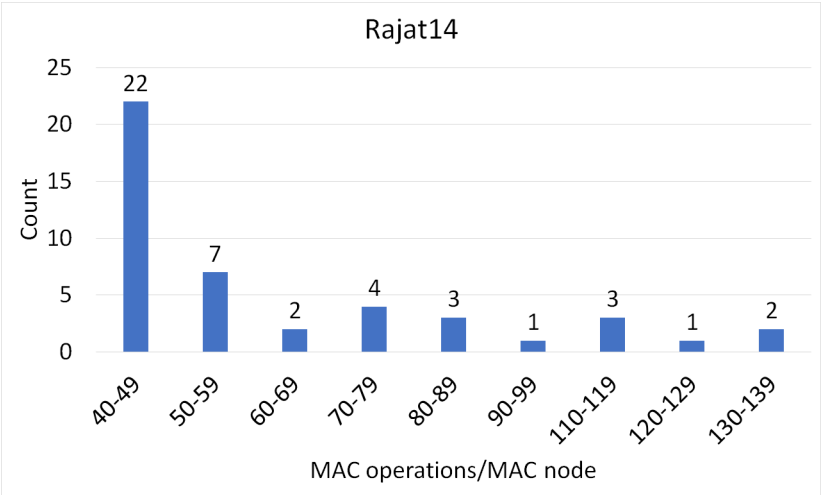



Fig 5.3(g): MAC operations/MAC node for "Rajat14"

Similar to the "Rajat04" graph, I have combined the x-axis values in sets of 10 in-order to make the graph length reasonable. Also, I have not displayed the range 1-9 and 10-19 as they were too large and made the rest of the bars in the graph look very small. Range 1-9 has a value 3537 and range 10-19 has a value 836.

From the above 2 graphs of "Rajat04" & "Rajat11" it's quite evident that there exist MAC nodes with have above 100 MAC operations also!!! This proves that my hypothesis was correct. The hypothesis said that the cases where my scheduler performed worse than Nechma's implementation was because those cases had MAC nodes with large number of MAC operations.

**Conclusion: -**

Exploiting fine-grained parallelism shows promising performance improvements over medium-grained parallelism (as done by Nechma) which can be seen from the performance gain in "fpga_dcop_01" and "fpga_dcop_50" matrices. But to exploit the fine-grained parallelism properly, we must separate the MAC arithmetic unit into MUL and ADD arithmetic units.

## 5.4 Total error comparison while using 32-bit float as compared to 64-bit double

Once I compute the L and U values in HDL simulation, I dump them in a text file and compare the results with C simulation for correctness. The issue here is that, as the L and U values become small enough to reach the highest precision which the respective data types can offer (i.e. $2^{-23}$ for float and $2^{-52}$ for double) the C simulation results start to diverge from the HDL simulation result. I think this is mostly due to the way MAC IP has been implemented by Xilinx. The error count for two different hardware configurations is shown in the figures below: -

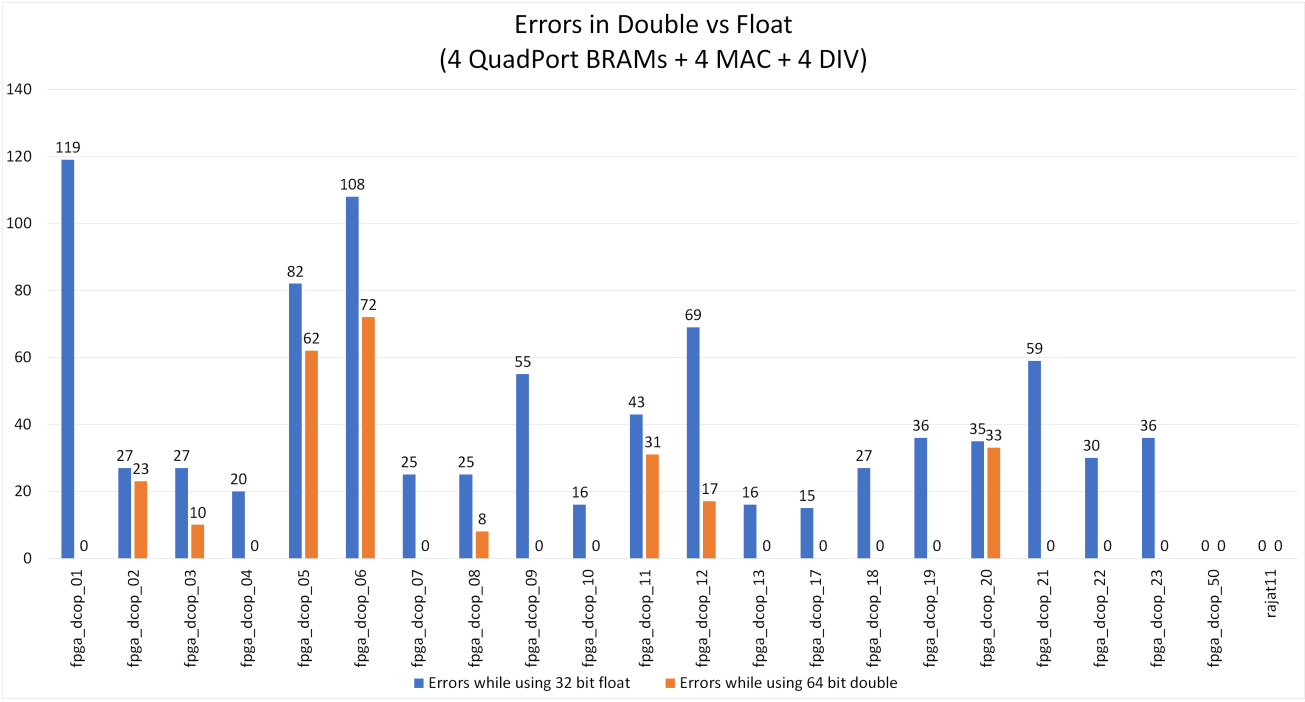



Fig 5.4(a): Error count for 1st hardware

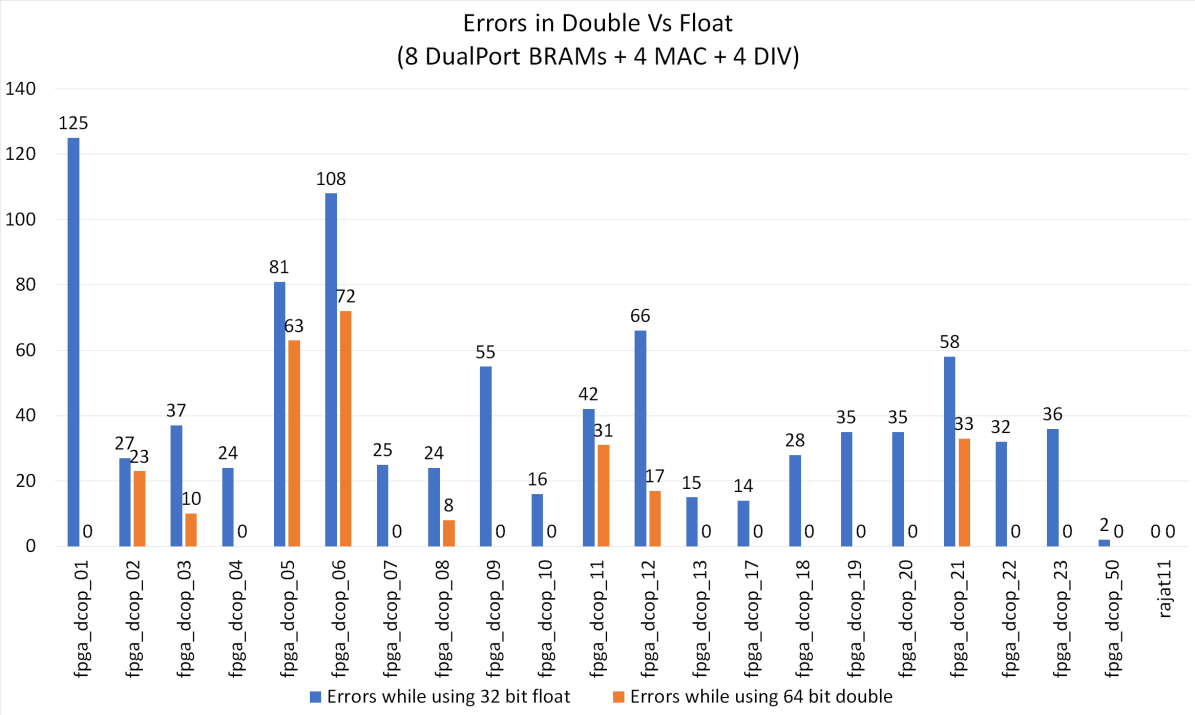



Fig 5.4(b): Error count for 2nd hardware

Note how the error count varies between the two hardware for the same matrix. When we use 64-bit double precision, the errors are fewer as compared to using 32-bit single precision.

**Conclusion: -**

Such errors can be ignored and we must ensure that the errors in result are not due to the errors in implementing the data flow graph or the scheduler.

# Chapter 6

# Future Work

## 6.1 Synthesizable Asynchronous Quad Port BRAM wrapper

As discussed in section 4.4.1.2, the present asynchronous BRAM wrapper was not synthesizable without generating timing violations because the false paths have not been specified in the constraints file. Hence providing proper constraints so that the asynchronous Quad Port BRAM wrapper to get synthesizable can be one on the future work.

## 6.2 Separating the MAC Arithmetic Unit into MUL and ADD Arithmetic Unit

I have discussed in section 5.3 that the main reason our approach is dramatically lagging in performance for certain matrices as compared to Nechma's implementation is because we have used MAC AU instead of using MUL and ADD AU separately. Hence for next future work, I would suggest to make appropriate changes to data flow graph, scheduler and hardware to incorporate the above change.

## 6.3 Splitting the DFG and assigning each graph to a separate LUD Hardware block

This approach will be helpful when we know that adding additional AU and BRAM to the cross-bar network would significantly reduce the clock cycles required for LU decomposition of the matrix, but adding any more blocks to the cross-bar network is making the design un-synthesizable due to routing congestions. As an example, consider the graph given below: -

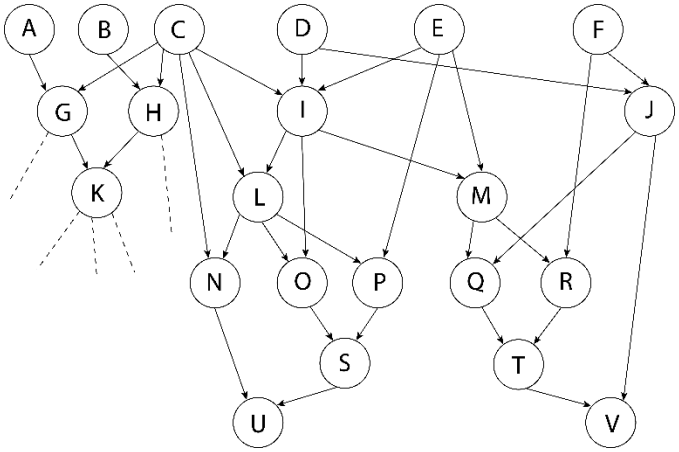



Fig 6.3(a): A DFG example

Suppose we want to split the graph into 3 separate graphs. Before splitting the graph, I have color coded each node. The color of each node represents which graph the node belongs to. The color-coded graph is shown below: -

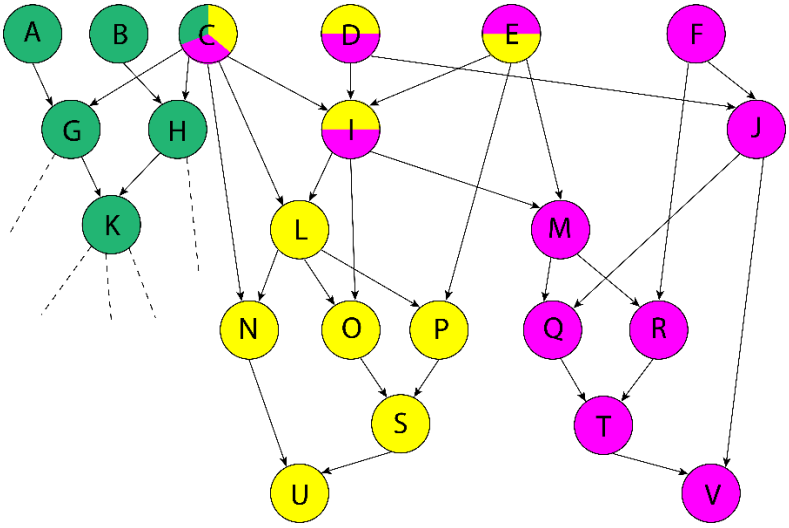



Fig 6.3(b): Color coded DFG graph

Note that a node can belong to more than 1 graph also. For example, node C belongs to all the three graphs. Finally, the three graphs are shown below: -

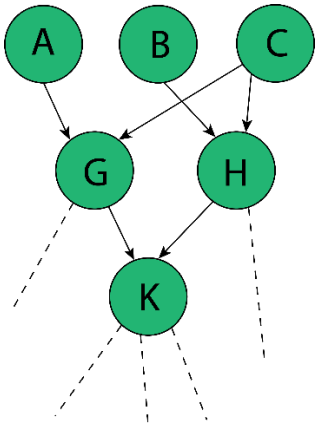



Fig 6.3(c): DFG graph 1

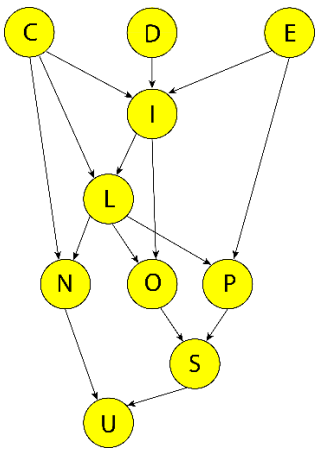



Fig 6.3(d): DFG graph 2

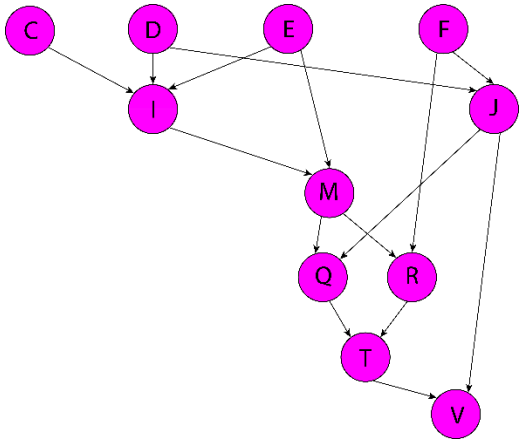



Fig 6.3(e): DFG graph 3

Note that this approach assumes that the three LUD hardware to which these 3 graphs will be distributed to, will not communicate with each other. This causes some redundancy because there will exist nodes which have to be computed in more than 1 graph. Like in this example, node "I" has to be computed in graph no. 2 and 3 both.

## 6.4 Implementing the hardware on an NOC (Network on Chip)

This is by far the most ambitious and most difficult approach. Implementing the hardware on NOC will not achieve the same performance as a cross-bar network but it will be scalable which is an equally important design consideration. An example of an NOC architecture is shown below: -

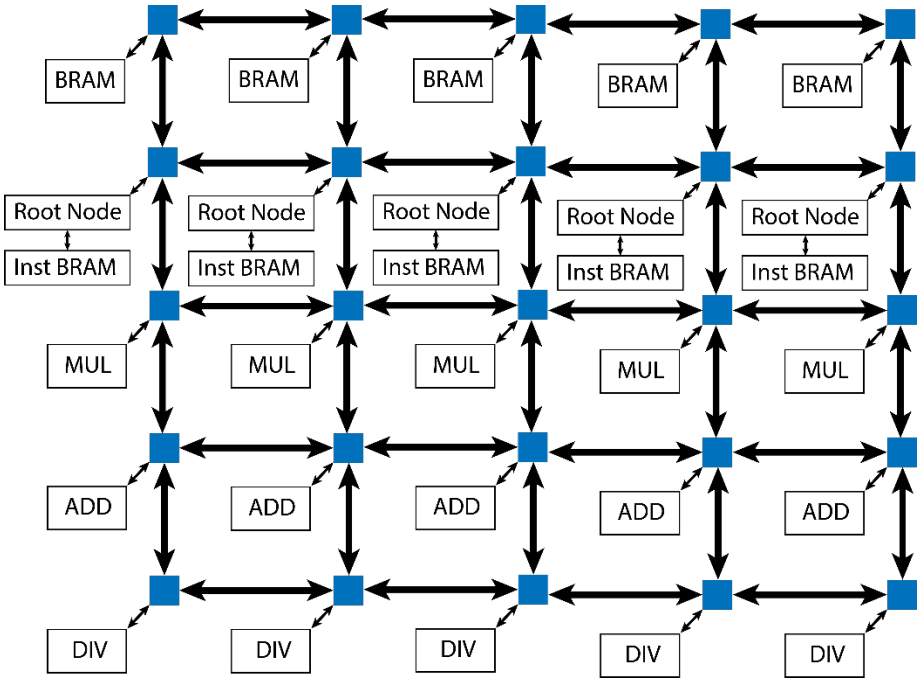



Fig 6.4(a): Proposed NOC architecture

Note that unlike traditional NOC architectures, this proposed NOC architecture has multiple Root Nodes. The purpose of Root Node is to give instructions to BRAMs and other arithmetic units. As

the AUs we are using are already pipelined, it's necessary to have multiple Root Nodes if we want to use more than 1 AU at a time. This will also solve the problem of increased latency as the size of NOC increases. Because if the size of NOC is very large and the Root Node is present in one corner of the NOC, the time it takes for an instruction to reach an arithmetic unit will be much higher than the latency of that arithmetic unit.

# References

[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] 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.