

POLITECNICO DI TORINO

Master's Degree in Mechatronics



Master's Degree Thesis

A comprehensive analysis of Sparse Matrix by Vector multiplication on FPGA with different compression formats

Supervisors

Prof. LUCIANO LAVAGNO

FILIPPO MINNELLA

Candidate

BEKZOD FAZILOV

2021-2022 academic year

Summary

Today in many applications the edge devices are used from cloud computing, internet of things (IoT) to manufacturing sectors to monitor, analyze processes through applying machine learning and other algorithms. The edge devices usually run light softwares like quantized and small machine learning algorithms due to their limited performance, energy consumption and memory. In the algorithms containing the matrix to vector multiplication, especially, in the quantized fully connected stage of Neural Networks, the weighted matrices are usually consists high percentage of zero elements. For the sake of reduction of resource usage, energy consumption and increasing the performance, the only non-zero elements can be used in this operation. Therefore, the sparse storage formats are helpful in avoiding multiplication operations involving zero elements.

In this thesis work the dataflow of sparse matrix to vector multiplication(SpMV) is analyzed where sparse matrices having different size and different sparsity are stored in different sparse storage formats. The sparse matrices are randomly generated with different sizes and different sparsity using probability algorithms and Mersenne Twister 19937 generator. The sparsity of matrices are chosen between 50% and 80%. And the sizes of the matrices were 30 by 30, 60 by 60 and 120 by 120. The C++ and Xilinx Vitis HLS are used to convert sparse matrices into ELL, CSC (compressed sparse column), CSR (compressed sparse row), COO (coordinate) sparse storage formats . Furthermore, Xilinx Vitis HLS is used also used to create the IP, estimate the resource utilization and establish the input output ports. The generated IP then utilized by Xilinx Vivado Design Suite to simulate the hardware design and obtain the bitsream of the corresponding storage format. The obtained bitstream is used to configure the PL of the PYNQ-Z2 board. The sparse storage formats obtained from C++ are used to program the PYNQ-Z2 board. The performance estimation is done on the PYNQ-Z2 board.

Acknowledgements

ACKNOWLEDGMENTS

*“HI”
Goofy, Google by Google*

Table of Contents

List of Tables	VII
List of Figures	VIII
Acronyms	XI
1 Introduction	1
1.1 Overview	1
1.2 FPGA internal structure	2
1.3 PYNQ-Z2 Development board.	3
1.3.1 Hardware overview	3
1.3.2 Programming the board	3
1.4 High level synthesis	5
1.4.1 Overview of high level synthesis	5
1.4.2 HLS tools	5
2 Background	7
3 Discussion on sparse storage formats	9
3.1 ELL sparse matrix storage format by vector multiplication	10
3.1.1 Overview	10
3.1.2 Implementation ELL format	11
3.2 COO Format	14
3.2.1 Overview	14
3.2.2 Implementation of COO(coordinate) format	14
3.3 CSC Format	18
3.3.1 Overview	18
3.3.2 Implementation CSC format	18
3.4 CSR Format	21
3.4.1 Overview	21
3.4.2 Implementation CSR format	22

4	Experimental results	26
4.1	Random sparse matrix generation	26
4.2	Evaluations	28
4.2.1	Estimation of resource utilization	28
4.2.2	Estimation of execution time	29
5	Conclusion	42
5.1	Discussion	42
5.2	Future work	42
A	Codes	43
A.1	Test bench code	43
A.2	Header file	49

List of Tables

List of Figures

1.1	Diagram of the basic FPGA structure and internal components https://digitaltagebuch.wordpress.com/2012/11/26/fpga-design-flow/	2
1.2	PYNQ board image from https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html	3
2.1	Sparse matrix	7
2.2	Sparse matrix	8
3.1	Concentration of non zeros elements in one row	10
3.2	Spreaded location of non zero elements	10
3.3	Ell format representation	11
3.4	Ell format block design	12
3.5	COO format representation	15
3.6	COO sparse matrix storage format's block design	16
3.7	CSC format representation	18
3.8	CSC sparse matrix storage format's block design	20
3.9	CSR format representation	22
3.10	CSR sparse matrix storage format's block design	24
4.1	120 by 120 matrix with 80 percent sparsity multiplied by vector	29
4.2	120 by 120 matrix with 70 percent sparsity multiplied by vector	30
4.3	120 by 120 matrix with 60 percent sparsity multiplied by vector	30
4.4	120 by 120 matrix with 50 percent sparsity multiplied by vector	31
4.5	60 by 60 matrix with 80 percent sparsity multiplied by vector	31
4.6	60 by 60 matrix with 70 percent sparsity multiplied by vector	32
4.7	60 by 60 matrix with 60 percent sparsity multiplied by vector	32
4.8	60 by 60 matrix with 50 percent sparsity multiplied by vector	33
4.9	30 by 30 matrix with 80 percent sparsity multiplied by vector	33
4.10	30 by 30 matrix with 70 percent sparsity multiplied by vector	34
4.11	30 by 30 matrix with 60 percent sparsity multiplied by vector	34
4.12	30 by 30 matrix with 50 percent sparsity multiplied by vector	35

4.13	120 by 120 matrix with 80 percent sparsity multiplied by vector	. .	35
4.14	120 by 120 matrix with 70 percent sparsity multiplied by vector	. .	36
4.15	120 by 120 matrix with 60 percent sparsity multiplied by vector	. .	36
4.16	120 by 120 matrix with 50 percent sparsity multiplied by vector	. .	37
4.17	60 by 60 matrix with 80 percent sparsity multiplied by vector	. . .	37
4.18	60 by 60 matrix with 70 percent sparsity multiplied by vector	. . .	38
4.19	60 by 60 matrix with 60 percent sparsity multiplied by vector	. . .	38
4.20	60 by 60 matrix with 50 percent sparsity multiplied by vector	. . .	39
4.21	30 by 30 matrix with 80 percent sparsity multiplied by vector	. . .	39
4.22	30 by 30 matrix with 70 percent sparsity multiplied by vector	. . .	40
4.23	30 by 30 matrix with 60 percent sparsity multiplied by vector	. . .	40
4.24	30 by 30 matrix with 50 percent sparsity multiplied by vector	. . .	41

Acronyms

AI

artificial intelligence

Chapter 1

Introduction

1.1 Overview

The matrix by vector multiplication operations are the essential part of many algorithms, such as least squares, eigenvalue problems, linear system of equation problems, and especially they are the heart of the neural network algorithms. And also a lot of researches are done to implement optimized hardware solution in order to accelerate the computation on FPGA and ASICs. Most of the algorithms running on CPU, GPU or implemented on hardware on FPGA are considered to work with non sparse matrix or with non sparse vectors. In general there are many sparse operations such as sparse matrix by vector or sparse matrix to sparse matrix multiplications and etc. And sometimes the sparsity of the object which is involved in operation can reach up to 80%. It means to obtain the results, only 20% of that object is used during the computation and significant part of the sparse matrix do not contributed anything to the result. Obviously, the hardware resources are wasted and to prevent this kind of issues the different kind of solutions are proposed.. In this thesis the sparse matrix by vector matrix multiplication analysed, where sparse matrix is represented in sparse matrix storage format.

This section also covers the FPGA and its internal structure together with short discussion on high level synthesis and HLS tools, furthermore, the target device on which the thesis experiments held. In Chapter 2 the discussion done on sparse operations, in chapter 3 the sparse storage formats are described, in chapter 4 the results of experiments are represented and chapter 5 described the conclusion.

1.2 FPGA internal structure

The FPGA is traditionally applied where the low latency is very important in computation and is now becoming more applicable in Machine Learning and Artificial Intelligence fields due to low energy consumption, low latency. It consists CLB(Configurable Logic Block), DPS(digital signal processor), Interconnect Resources, I/O-blocks, , Switch Blocks and different types of memories such as BRAM, URAM. LUT(Look-Up-Table) and flip-flop are the main parts of CLB. Basically, LUTs are consists of many multiplexers, when they are interconnected together they represent the part of the circuit's logic, the flip-flop helps to take snap shoot of that logical state of the circuit. By interconnecting CLBs together through interconnection resources and switch blocks the desired circuit can be implemented. Meanwhile the DSP blocks are consists of registers and MAC units and their purpose is to increase the performance of the computation operations. BRAM and URAM have finite size and they are used to store data on the FPGA. The Figure 1.3 represents the short description of the internal structure of FPGA.

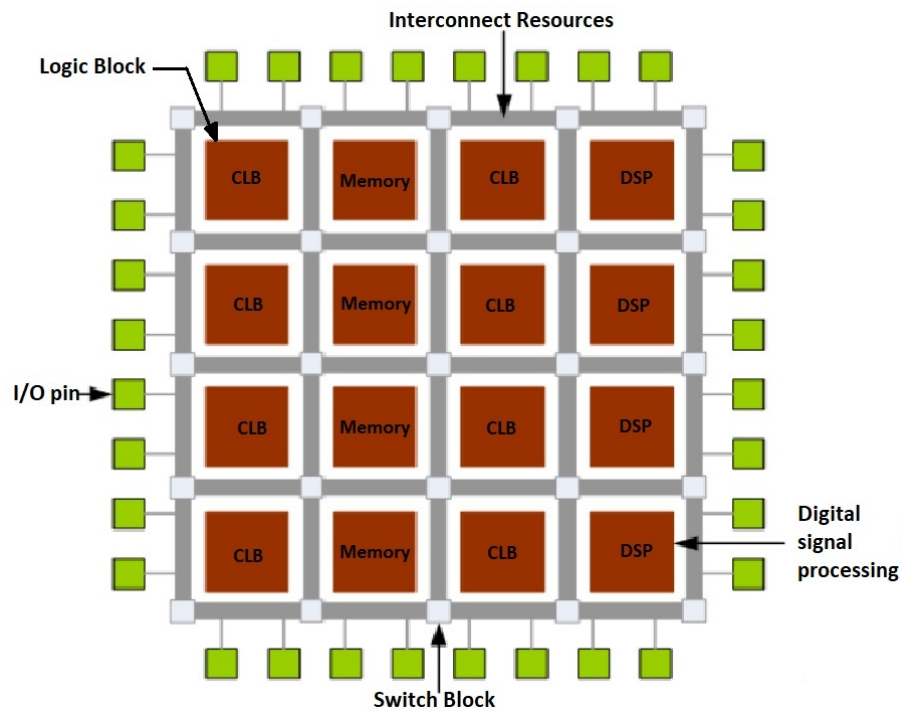


Figure 1.1: Diagram of the basic FPGA structure and internal components
<https://digitaltagebuch.wordpress.com/2012/11/26/fpga-design-flow/>

1.3 PYNQ-Z2 Development board.

1.3.1 Hardware overview

The thesis experiments were done on the PYNQ-Z2 development board. The brief description can be found in the <https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html> website. We will shortly describe the PYNQ-Z2" board. The abbreviation of PYNQ is Python Productivity for Zynq. Therefore, it is a framework for developing embedded projects. It has ZYNQ XC7Z020-1CLG400C which has PS(650MHz dual-core Cortex-A9 processor) and PL(Programmable logic equivalent to Artix-7 FPGA) on the SoC(System on chip).

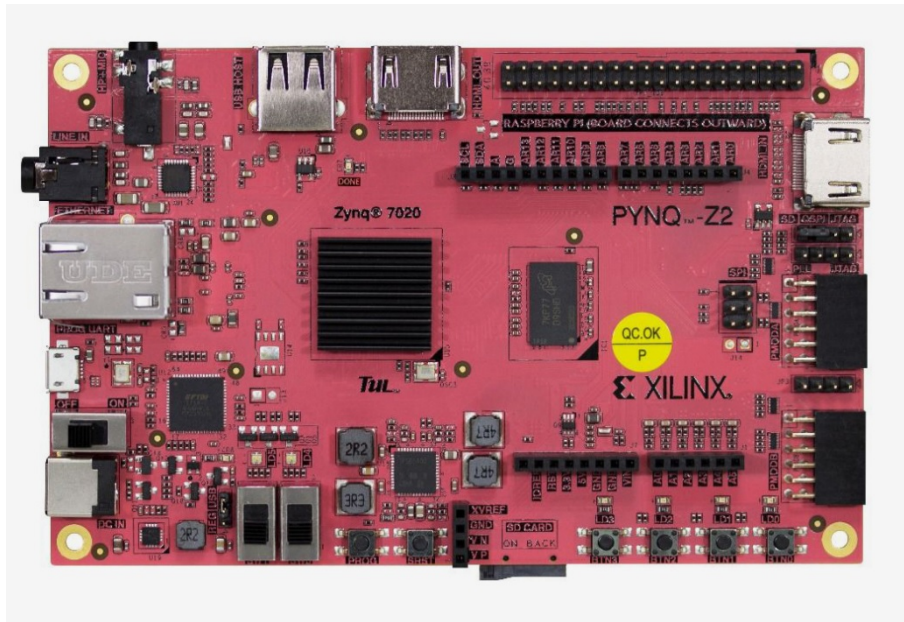


Figure 1.2: PYNQ board image from <https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html>

1.3.2 Programming the board

The board is programmed using python language in Jupyter notebook. The brief information about PL and PS connection is taken from pynq website in the section of Overlay Design Methodology. The Zynq has 9 AXI interfaces between the PS and the PL. On the PL side, there are 4x AXI Master HP (High Performance) ports, 2x AXI GP (General Purpose) ports, 2x AXI Slave GP ports and 1x AXI Master ACP port. There are also GPIO controllers in the PS that are connected to the PL.

There are four pynq classes that are used to manage data movement between the Zynq PS (including the PS DRAM) and PL interfaces.

pynq.gpio.GPIO - General Purpose Input/Output

pynq.mmio.MMIO - Memory Mapped IO

pynq.buffer.allocate() - Memory allocation

pynq.lib.dma.DMA - Direct Memory Access The class used depends on the Zynq PS interface the IP is connected to, and the interface of the IP.

Python code running on PYNQ can access IP connected to an AXI Slave connected to a GP port. MMIO can be used to do this.

IP connected to an AXI Master port is not under direct control of the PS. The AXI Master port allows the IP to access DRAM directly. Before doing this, memory should be allocated for the IP to use. The allocate function can be used to do this. For higher performance data transfer between PS DRAM and an IP, DMAs can be used. PYNQ provides a DMA class.

When designing your own overlay, you need to consider the type of IP you need, and how it will connect to the PS. You should then be able to determine which classes you need to use the IP.

PS GPIO

There are 64 GPIO (wires) from the Zynq PS to PL.

PS GPIO wires from the PS can be used as a very simple way to communicate between PS and PL. For example, GPIO can be used as control signals for resets, or interrupts.

IP does not have to be mapped into the system memory map to be connected to GPIO.

MMIO

Any IP connected to the AXI Slave GP port will be mapped into the system memory map. MMIO can be used read/write a memory mapped location. A MMIO read or write command is a single transaction to transfer 32 bits of data to or from a memory location. As burst instructions are not supported, MMIO is most appropriate for reading and writing small amounts of data to/from IP connect to the AXI Slave GP ports.

allocate

Memory must be allocated before it can be accessed by the IP. allocate allows memory buffers to be allocated. The pynq.buffer.allocate() function allocates a contiguous memory buffer which allows efficient transfers of data between PS and

PL. Python or other code running in Linux on the PS can access the memory buffer directly.

As PYNQ is running Linux, the buffer will exist in the Linux virtual memory. The Zynq AXI Slave ports allow an AXI-master IP in an overlay to access physical memory. The numpy array returned can also provide the physical memory pointer to the buffer which can be sent to an IP in the overlay. The physical address is stored in the `device_address` property of the allocated memory buffer instance. An IP in an overlay can then access the same buffer using the physical address.

More information about using `allocate` can be found in the `Allocate` section.

1.4 High level synthesis

1.4.1 Overview of high level synthesis

As mentioned in the xilinx homepage <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Benefits-of-High-Level-Synthesis>, the High-Level Synthesis is an automated design process that takes an abstract behavioral specification of a digital system and generates a register-transfer level structure that realizes the given behavior.

The design process consists of the following steps

1. Writing the desired behaviour (what will compute the PL) algorithm in the C/C++ language
 2. Verify the functionality of that algorithm
 3. Use the HLS tool to generate the RTL for a given clock speed, input constraints
 4. Verify the functionality of the generated RTL
 5. Explore different architectures using the same input source code
- HLS can enable the path of creating high-quality RTL, rather quickly than manually writing error-free RTL.

In short words the engineer needs to write the behaviour algorithm in C/C++, insert it into HLS tool together with constraint, target device, target throughput, etc. The HLS tool generates and verifies the desired bitstream file. The pragmas are used to optimize the circuit, by changing the resource utilization, latency, throughput, etc. Consideration about state machine and pipe-lining are done automatically by HLS tool. The Vitis HLS and Vivado are HLS tools which are used in this thesis work.

1.4.2 HLS tools

Vitis HLS

Vitis HLS is the tool to generate the desired IP and the IP is generated from the C/C++ top level function. The user should do the following steps in order to generate the IP.

1. The functional behaviour of the algorithm must be written in C/C++ language in the top level function and should be validated in the test bench algorithm.
2. The top level function must be synthesized with target frequency. In the synthesis report Timing estimate, uncertainty interval, the resource utilization, Input output ports together with their addresses and etc are shown.
3. Then the Cosimulation should be done.
4. The last step is importing the IP into desired folder. The detailed information can be found in the Xilinx home page <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction>. Overall the Vitis HLS is used to configure the PL part of the PYNQ Z-2 board.

Vivado

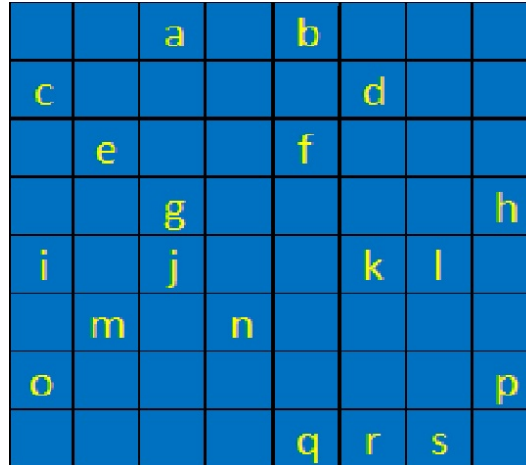
This tool is used to obtain the bitstream file from the IP. The following steps are done in vivado to generate bitstream file.

1. Importing the IP from the folder.
2. Creating Block Design, in this step the communication between PS and PL are established, by determining the type communication and which kind of pins are involved in this communication.
3. Synthesizing the Block Design
4. Implementation and generating Bitstream file.

Chapter 2

Background

The sparsity is very common feature of vectors, matrices and kernels. And the mathematical objects with sparsity feature are involved also in many computation operations in different algorithms. For instance, operations like matrix vector multiplication, matrix matrix multiplication or the convolutional stage of neural networks and so on. The mathematical objects with sparsity we define a matrix or a vector having some entries zero values. The zero elements of sparse objects (such as vector, matrix, kernel or etc.) waste hardware resources and energy execution of operation. Such kind of operations differ from each other drastically. And even to find a unique solution for each type is difficult. Because the vectors, matrices or kernels have different percentage of sparsity, and non zero elements of them are differently spread inside of them as shown in figures below.



		a		b			
c					d		
	e			f			
		g					h
i		j			k	l	
	m		n				
o							p
				q	r	s	

Figure 2.1: Sparse matrix

In this thesis , the main focus is devoted to matrix vector multiplication operation,

		a		b			
c					d		
e	f	g		h		i	
j	k	l	m	n	o	p	
	q		r				
		s					

Figure 2.2: Sparse matrix

where the matrix is considered as sparse matrix with finite size and finite percentage of sparsity. The non zero values of sparse matrices are stored in sparse storage formats. Then the computation of matrix vector multiplication is done without involving the zero elements of sparse matrix. In general, the dataflow of sparse matrix storage formats which are ELL, COO, CSC and CSR are analyzed in this thesis work. We decided not to apply this computation algorithm directly to sparse matrix, but we decided to transfer the sparse matrix into sparse matrix storage formats, then analyze the dataflow of them without pipelining in order to compare the sparse matrix storage formats with each other. The simple matrix by vector multiplication and the ELL sparse matrix storage format multiplication by matrix were not synthesised in Vitis HLS due to software issues. Therefore the results on simple matrix with vector and the matrix stored in ELL format with vector are different from other formats.

Chapter 3

Discussion on sparse storage formats

Apparently the sparse matrices can differ in their structure in terms of location of non zero elements. For instance the non zero elements of sparse matrix can be located equally from each other like in figure 3.3 or located densely in one column or in one row, even more there can be very few non zero elements inside sparse matrix like the sparsity of matrix can be higher than 95 percent. From the fact that they are different, the different sparse matrix storage formats can be created by considering this property. Certainly all of the sparse store formats can be used to different sparse matrices . But the main issue is reduction of non essential computation in sparse matrix by vector multiplication. And the sparse matrix storage formats handle it by removing the computations which involve zero elements of sparse matrix from matrix by vector multiplications. Nevertheless not all sparse matrix storage formats handle it efficiently. As already mentioned the four type of sparse matrix storage formats ELL, CSR, CSC and COO are analysed in this thesis work. And all of these formats can handle efficiently the computation only with particular type of sparse matrix. For example with the ELL format the computation of sparse matrix vector multiplication is done efficiently when sparse matrix has equally distributed non zero elements like in figure 3.2. While the CSR and CSC formats show relatively computation speed increase when the non zero elements of sparse matrix concentrated mostly in rows or in columns. And the COO format shows good performance when the sparsity percentage of matrix is very high.

		a		b			
c					d		
e	f	g		h			
i	j	k	l	m	n	o	p
	q		r				
		s					

Figure 3.1: Concentration of non zeros elements in one row

		a		b			
c					d		
	e			f			
		g					h
i		j			k	l	
	m		n				
o							p
				q	r	s	

Figure 3.2: Spreadded location of non zero elements

3.1 ELL sparse matrix storage format by vector multiplication

3.1.1 Overview

The ELL format is one of the sparse storage formats which stores the sparse matrix in compact way by creating two matrix from original sparse matrix. That created matrices most of the time are smaller in size then original sparse matrix. One matrix is called as value matrix which stores the values of the non zero elements of sparse matrix. The other one is called Column index matrix which stores the column indices of the sparse matrix. The both matrices of the ELL sparse storage format have the same row number of original sparse matrix, but different column number than original sparse matrix. The number of columns are defined by the row which contains the maximum number of non zero elements and it is called as maximum row. Therefore, the rows which has fewer values than maximum row has empty spaces, that empty spaces are filled with zeros.

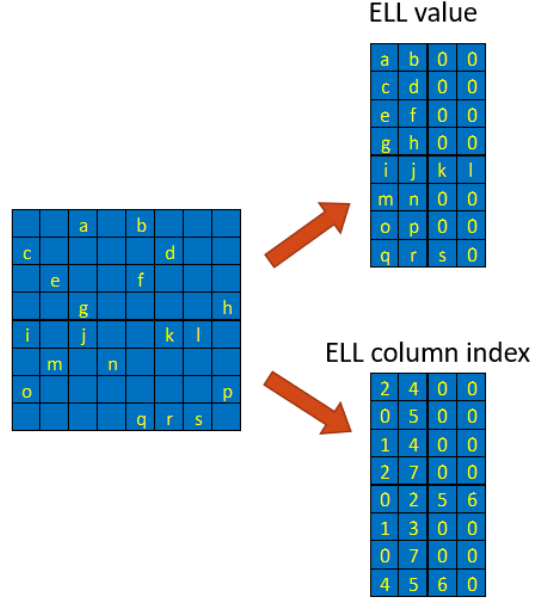


Figure 3.3: Ell format representation

3.1.2 Implementation ELL format

The multiplication between the ELL sparse matrix storage format and vector is done on PL of PYNQ-Z2 board. The logic for the PL is created using C++ language in Vitis HLS program. In the test bench file the ELL sparse matrix storage format is created from the sparse matrix. Meanwhile in the top level function file the multiplication by vector with ELL format is done by considering the fact that it creates the logic for the PL part of FPGA. The m_axi protocol is used between PS(processing system) and PL(programmable logic) communication and established by pragmas which is shown C++ code below. As we can see from the figure below the top level function has 4 arguments and this input outputs are established by master axi protocol in slave mode. The slave mode uses additional control signal in order to start the communication. The programmable logic PL is considered as slave and the processing system PS is considered as master which controls the communication between PL and PS.

In order to run this computation on the PYNQ Z2 board, the interface of that board must be programmed. The python code below shows the multiplication between the ELL sparse matrix storage format and vector where the sparse matrix is 30 by 30. First the sparse matrix is transformed into ELL format in C++ and then uploaded into PYNQ-Z2 interface. Then the buffers for the ELL format matrix with exact size are allocated and filled with the given values of sparse matrix. And the allocated memory buffers in PS should be synchronized with PL.

The buffers should be written into IP's address inside overlay in order to perform the computation. To start the computation a bit must be written into address of start address.

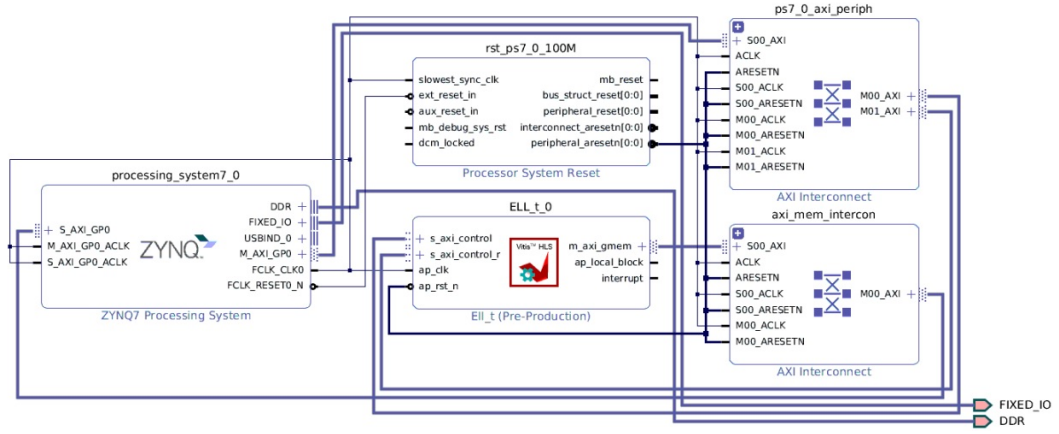


Figure 3.4: Ell format block design

```

1 #include <iostream>
2 #include "Hf4StF.h"
3 #include <ap_axi_sdata.h>
4 #include <hls_stream.h>
5
6 void ELL_t( d_in ELL_clm_idx[ROW_A][ELL_COL], d_in ELL_val[ROW_A
7             ][ELL_COL], d_in B[COL_A], d_out C_H[ROW_A] )
8 {
9     #pragma HLS PIPELINE off
10    #pragma HLS INTERFACE mode=m_axi port=ELL_clm_idx offset=slave
11    #pragma HLS INTERFACE mode=m_axi port=ELL_val      offset=slave
12    #pragma HLS INTERFACE mode=m_axi port=B            offset=slave
13    #pragma HLS INTERFACE mode=m_axi port=C_H          offset=slave
14
15    #pragma HLS INTERFACE mode=s_axilite port=return bundle=control
16
17    for(int i=0; i<ROW_A; i++)
18    {
19        #pragma HLS PIPELINE II=1
20        for(int j=0; j<ELL_COL; j++)
21        {
22            int k=ELL_clm_idx[i][j];
23            int val=ELL_val[i][j];
24            C_H[i]=C_H[i]+val*B[k];
25        }
26    }

```



```

27     }
28 }
29
30 }

```

content/ELL top level.cpp

```

1  import time
2  import numpy as np
3  import os, warnings
4  from pynq import PL
5  import pynq
6  from pynq.overlay import Overlay
7
8  ol = Overlay("./ELL_30x30.bit")
9  print("Done")
10 ol?
11
12 for i in ol.ip_dict:
13     print(i)
14
15 from pynq import allocate
16 import numpy as np
17 from numpy import loadtxt
18
19
20 ELL_col_indx = pynq.buffer.allocate(shape=(30,12), dtype=np.uint32)
21 ELL_val       = pynq.buffer.allocate(shape=(30,12), dtype=np.uint32)
22
23 B             = pynq.buffer.allocate(shape=(30,), dtype=np.uint32)
24 C             = pynq.buffer.allocate(shape=(30,), dtype=np.uint32)
25
26 ELL_v = loadtxt('./ELL_val_30x30.txt', dtype='int')
27 ELL_c = loadtxt('./ELL_col_30x30.txt', dtype='int')
28
29 for i in range(30):
30     for j in range(12):
31         ELL_col_indx[i][j]=ELL_c[i][j]
32
33 for i in range(30):
34     for j in range(12):
35         ELL_val[i][j]=ELL_v[i][j]
36
37 B[:]=1
38 C[:]=0
39
40 ELL_col_indx1=ELL_col_indx.device_address
41 ELL_val1=ELL_val.device_address

```

```

41 B1=B.device_address
42 C1=C.device_address
43
44 ELL_col_indx.sync_to_device()
45 ELL_val.sync_to_device
46 B.sync_to_device()
47 C.sync_to_device()
48
49 print(time.time_ns())
50 ol.ELL_t_0.s_axi_control.write(0x10,ELL_col_indx1)
51 ol.ELL_t_0.s_axi_control_r.write(0x18,ELL_val1)
52 ol.ELL_t_0.s_axi_control_r.write(0x20,B1)
53 ol.ELL_t_0.s_axi_control_r.write(0x28,C1)
54 print(time.time_ns())
55 ol.ELL_t_0.s_axi_control.write(0x00,1)
56 print(time.time_ns())
57 print(C)

```

content/ELL.py

3.2 COO Format

3.2.1 Overview

The second sparse matrix storage format is COO coordinate format which consists of 3 arrays. Primary sparse matrix is transformed into three arrays which are the row index array, column index array and value array. The row index array contains the row indices of the non zero elements of the sparse matrix. The column index array contains the column indices of the non zero elements of the sparse matrix and the value array contains the values of the non zero elements of the sparse matrix. This sparse matrix storage format is preferred to apply when the sparse matrix has very high sparsity in order to reduce the computation.

3.2.2 Implementation of COO(coordinate) format

The computation of sparse matrix and vector is done on PL of PYNQ-Z2 board. Consequently the IP for PL is created in Vitis HLS. The top level function in Vitis HLS is responsible for creation IP. The IP has four inputs and one output. The communication between PS and PL is established in slave mode using Master AXI protocol. Four inputs are for row array, column array, value array and for vector respectively, the fourth input is the vector which multiplied to sparse matrix. One output is responsible for resulted output vector from multiplication. This sparse matrix storage format is created by using the C++ in Vitis HLS tool and processed

in Vivado design suite to establish block design then further converted to the logic of the FPGA by creating the bitstream file.

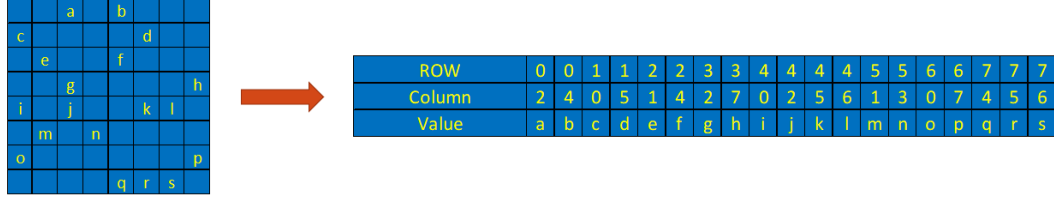


Figure 3.5: COO format representation

To run the multiplication on the PYNQ -Z2, the device's interface must be programmed. The sparse matrix transformed into COO format with three arrays using C++ language, then this COO format is uploaded on the device. The buffers for the data are created with 32 bit integer data type and filled with values of COO format arrays. And this buffers in PS are synchronized with PL. Then the address of this buffers are mapped with address of IP in PL by command write in order to give data to correct address. Since we use Master AXI in slave mode, to start computation one bit must be set in start stop address of control register inside IP.

```

1 #include <iostream>
2 #include "Hf4StF.h"
3 #include <vector>
4 #include <ap_axi_sdata.h>
5 #include <hls_stream.h>
6
7 void COO_t(
8     d_in COO_row[N],
9     d_in COO_col[N],
10    d_in COO_val[N],
11    d_in B[COL_A],
12    d_in C_H[ROW_A])
13 {
14
15
16 #pragma HLS INTERFACE mode=m_axi port=COO_row offset=slave
17 #pragma HLS INTERFACE mode=m_axi port=COO_col offset=slave
18 #pragma HLS INTERFACE mode=m_axi port=COO_val offset=slave
19 #pragma HLS INTERFACE mode=m_axi port=B offset=slave
20 #pragma HLS INTERFACE mode=m_axi port=C_H offset=slave
21
22 #pragma HLS INTERFACE mode=s_axilite port=return bundle=
    control_signals
23
24 // #pragma HLS PIPELINE off
25

```

```

26     int r=0, c=0, val=0;
27     for(int i=0; i<nz_val_A_C00; i++)
28     {
29         r=C00_row[i];
30         c=C00_col[i];
31         val=C00_val[i];
32         C_H[r]=C_H[r]+val*B[c];
33     }
34 }

```

content/COO top level.cpp

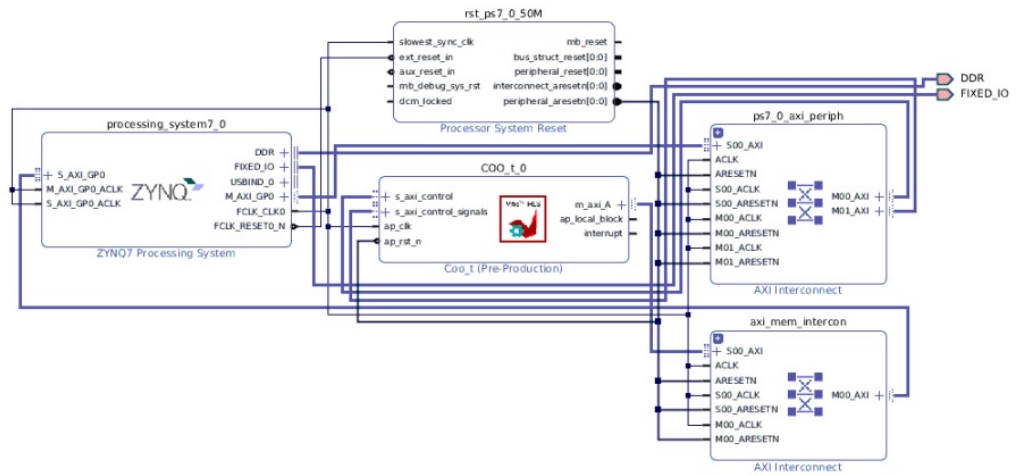


Figure 3.6: COO sparse matrix storage format's block design

```

1 import time
2 from pynq import PL
3 import pynq
4 from pynq.overlay import Overlay
5
6 ol = Overlay("./COO_30x30.bit")
7 print("Done")
8 ol.is_loaded()
9
10 for i in ol.ip_dict:
11     print(i)
12
13 from pynq import allocate
14 import numpy as np
15 from numpy import loadtxt
16
17 row_C00 = allocate(shape=(2904,), dtype=np.uint32)

```

```

18 col_C00 = allocate(shape=(2904,), dtype=np.uint32)
19 val_C00 = allocate(shape=(2904,), dtype=np.uint32)
20 B       = allocate(shape=(120, ), dtype=np.uint32)
21 C       = allocate(shape=(120, ), dtype=np.uint32)
22
23 r = loadtxt('./C00_row_indx_120x120.txt', dtype='int')
24 c = loadtxt('./C00_col_indx_120x120.txt', dtype='int')
25 v = loadtxt('./C00_val_120x120.txt',      dtype='int')
26
27 for i in range(2904):
28     row_C00[i]=r[i]
29     col_C00[i]=c[i]
30     val_C00[i]=v[i]
31
32 B[:]=1
33 C[:]=0
34 print(row_C00)
35 print(B)
36 print(col_C00)
37
38 row_C00.sync_to_device()
39 col_C00.sync_to_device()
40 val_C00.sync_to_device()
41 B.sync_to_device()
42 C.sync_to_device()
43
44 row_C001 = row_C00.device_address
45 col_C001 = col_C00.device_address
46 val_C001 = val_C00.device_address
47 B1       = B.device_address
48
49 C1       = C.device_address
50
51 print(time.time_ns())
52 ol.C00_t_0.s_axi_control.write(0x10,row_C001)
53 ol.C00_t_0.s_axi_control.write(0x18,col_C001)
54 ol.C00_t_0.s_axi_control.write(0x20,val_C001)
55 ol.C00_t_0.s_axi_control.write(0x28,B1)
56 ol.C00_t_0.s_axi_control.write(0x30,C1)
57 print(time.time())
58 ol.C00_t_0.s_axi_control_signals.write(0x00,1)
59 print(time.time_ns())
60 print(C)

```

content/COO.py

3.3 CSC Format

3.3.1 Overview

The CSC Compressed sparse column format is consists of 3 arrays which are column pointer array, row index array and value array. The row index array contains the row indices and value array contains the values of the non zero elements of sparse matrix. In order to define the values of pointer array, the concept ordering of non zero values of sparse matrix should be introduced. The sparse matrix contains certain number of non zero elements. Also every column inside sparse matrix contains different number of non zero entries. And they can be ordered in different way. In this format the ordering of the non zero elements are started from top to down through column and left to right when passing to the new column. The ordering number also starts from zero. And the order number of the first non zero entry of every column is taken as pointer. Therefore the column pointer array contains the non zero entry's order number which appeared as first element of column of spare matrix. If we compare the length of the column pointer array to the other arrays, it is length is much more shorter. Consequently this sparse matrix storage format requires less memory than the other formats which are discussed earlier.

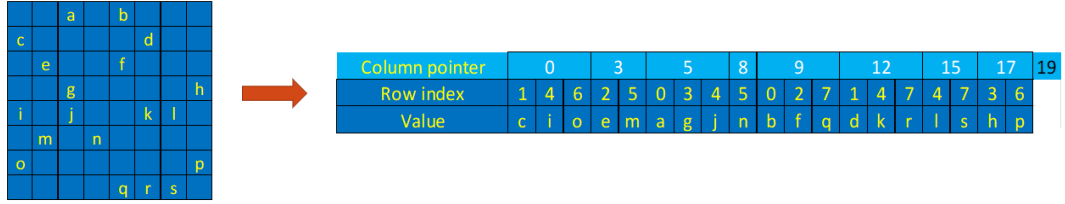


Figure 3.7: CSC format representation

3.3.2 Implementation CSC format

The multiplication between the sparse matrix and the vector is done on PL part PYNQ-Z2 board. The logic for PL part is created in Vitis HLS software in the top level function. The following code is used to create the logic for this format. The IP has 4 inputs and one output. The Master AXI protocol in slave mode is established between PS and PL of the PYNQ-Z2 board. The block design is created and bitstream file is extracted from IP in Vivado design suite. To estimate the execution time the PYNQ-Z2 board must be programmed. First the sparse matrix is transformed into CSC format with text file format using C++ language and uploaded into PYNQ-Z2 interface together with bit stream file. The required

buffers for CSC format is created in the device's interface and filled with sparse matrix values from text file version of CSC format. The buffers are synchronized with PL part of the device. To start the computation the addresses of buffers and addresses of arrays in IP are overlapped with write command, then the one bit set in the control register of IP.

```

1  #include <iostream>
2  #include "Hf4StF.h"
3
4
5  void CSC_t( d_in CSC_ar_nz[N], d_in ar_row_idx[M], d_in ar_col_ptr
6             [L], d_in B[COL_A], d_out C_H[ROW_A])
7  {
8
9  #pragma HLS INTERFACE mode=m_axi      offset=slave port=ar_nz
10 #pragma HLS INTERFACE mode=m_axi      offset=slave port=ar_row_idx
11 #pragma HLS INTERFACE mode=m_axi      offset=slave port=ar_col_ptr
12 #pragma HLS INTERFACE mode=m_axi      offset=slave port=B
13 #pragma HLS INTERFACE mode=m_axi      offset=slave port=C_H
14
15 #pragma HLS INTERFACE mode=s_axilite port=return bundle=
    control_signals
16 int clp=31;
17 int col_start=0, col_end=0;
18 for(int i = 0; i < n_clp; ++i)
19 {
20     col_start = ar_col_ptr[i] ;
21     col_end   = ar_col_ptr[i+1];
22
23     for(int nz_id = col_start; nz_id < col_end; ++nz_id)
24     {
25
26         int j=0;
27         int val=0;
28
29         j   = ar_row_idx[nz_id];
30         val = CSC_ar_nz[nz_id];
31
32         C_H[j] = C_H[j]+val*B[i];
33
34     }
35
36 }
37
38 }

```

content/CSC top level.cpp

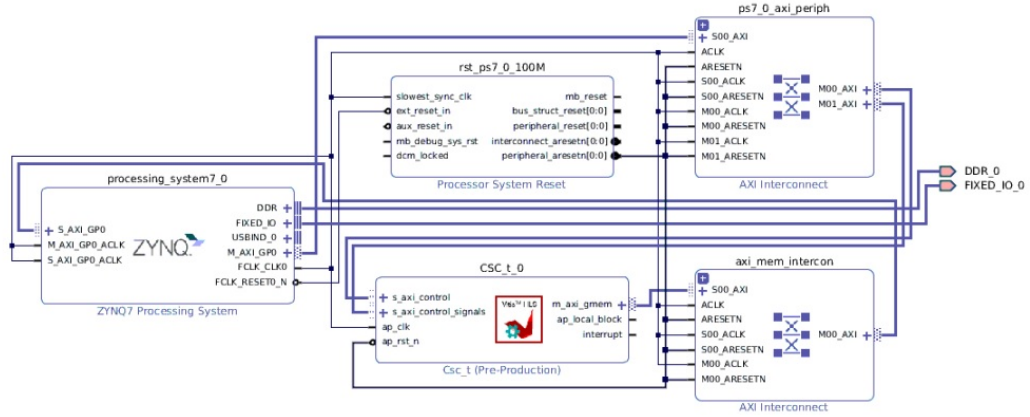


Figure 3.8: CSC sparse matrix storage format's block design

```

1 import time
2 from pynq import PL
3 import pynq
4 from pynq.overlay import Overlay
5
6 CSC_F=Overlay("./CSC_top_without_pipeline.bit")
7 print("Done")
8
9 for i in CSC_F.ip_dict:
10     print(i)
11
12 from pynq import allocate
13 import numpy as np
14 from numpy import loadtxt
15
16 col_ptr = pynq.buffer.allocate(shape=(31,), dtype=np.uint32)
17 row_id = pynq.buffer.allocate(shape=(186,), dtype=np.uint32)
18 val = pynq.buffer.allocate(shape=(186,), dtype=np.uint32)
19 B = pynq.buffer.allocate(shape=(30,), dtype=np.uint32)
20 C = pynq.buffer.allocate(shape=(30,), dtype=np.uint32)
21
22 c_ptr = loadtxt('./ar_col_ptr_CSC.txt', dtype='int')
23 r = loadtxt('./ar_row_idx_CSC.txt', dtype='int')
24 v = loadtxt('./ar_nz_CSC.txt', dtype='int')
25
26 for i in range(31):
27     col_ptr[i]= c_ptr[i]
28
29 for i in range(186):
30     row_id[i] = r[i]
31     val[i] = v[i]

```



```

32 B[:]=1
33 C[:]=0
34
35 print(B)
36 print(col_ptr)
37
38 col_ptr.sync_to_device()
39 row_id.sync_to_device()
40 val.sync_to_device()
41 B.sync_to_device()
42 C.sync_to_device()
43
44 col_ptr1 = col_ptr.device_address
45 row_id1  = row_id.device_address
46 val1     = val.device_address
47 B1       = B.device_address
48 C1       = C.device_address
49
50 print(time.time_ns())
51
52
53 CSC_F.CSC_t_0.s_axi_control.write(0x10,val1)
54 CSC_F.CSC_t_0.s_axi_control.write(0x18,row_id1)
55 CSC_F.CSC_t_0.s_axi_control.write(0x20,col_ptr1)
56 CSC_F.CSC_t_0.s_axi_control.write(0x28,B1)
57 CSC_F.CSC_t_0.s_axi_control.write(0x30,C1)
58 CSC_F.CSC_t_0.s_axi_control_signals.write(0x00,1)
59
60 print(time.time_ns())
61 print(C)
62

```

content/CSC_pynq_interface.py

3.4 CSR Format

3.4.1 Overview

The last format which we considered is CSR Compressed sparse row format. This format also consists of 3 arrays which are row pointer array, column array and value array. The column array and value array consists of column index and values of the non zero entries of sparse matrix respectively. The row pointer array consists of pointers to the rows which contains non zero entries of sparse matrix. This format is similar to CSC sparse matrix storage format, but the difference are in the ordering of non zero elements of sparse matrix and in compression with respect to rows. And the row pointer array contains the pointers to row, and their values is

equal to the order value of first non-zero element in the each row of sparse matrix. The ordering of sparse matrix's non zero elements is done from left to right within row and top to down between rows and starting the order number from zero. This format is also occupies less memory than COO format since row pointer array contains less elements than other arrays. This format shows good performance when non zero entries are concentrated in one array.

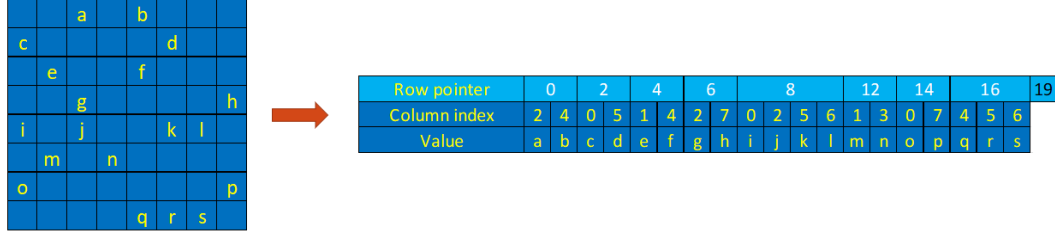


Figure 3.9: CSR format representation

3.4.2 Implementation CSR format

The computation of multiplication between sparse matrix stored in CSR format and vector is done on PL part of PYNQ-Z2 board. As usual the logic for PL is created by using Vitis HLS. The resource utilization also estimated in Vitis HLS. The following code demonstrates the top level function of CSR format in Vitis HLS. Overall, four inputs and one output is used in this computation. The communication between PL and PS is established as master AXI protocol in slave mode with the help of pragmas in Vitis HLS. In order to implement the multiplication two nested for loops are needed. The outer loop is responsible for shifting between rows which contains the non zero entries of sparse matrix and it uses the values of row pointer array. While the inner loop is needed to shift between non zero entries within rows and fetch column index and values of non zero entries of sparse matrix from column array and value array respectively and perform the computation. After the generation of IP in Vitis HLS, the block design and bitstream file is created in Vivado Design Suite. The execution time is estimated on PYNQ-Z2 board, and following code written on python shows the implementation of computation on PYNQ-Z2 board. As usual, the sparse matrix transformed into CSR format by C++ language and stored in text format. Then, the sparse matrix in CSR format together with bitstream file are uploaded into the device. The required buffers for the CSR format arrays are allocated with exact size and exact data type. These buffers are filled with CSR format values and synchronized with PL part of device. The values in the buffers are passed to correct address of IP with the help of write command. And the computation is started when one bit is set in control register

of IP.

```

1 #include <iostream>
2 #include "header_csr.h"
3 using namespace std;
4
5 void CSR_t( d_in CSR_ar_nz[M], d_in ar_col_idx[N], d_in ar_row_ptr
6            [L], d_in B[COL_A], d_out C_H[ROW_A]){
7
8 #pragma HLS INTERFACE mode=m_axi offset=slave port = CSR_ar_nz
9 #pragma HLS INTERFACE mode=m_axi offset=slave port = ar_col_idx
10 #pragma HLS INTERFACE mode=m_axi offset=slave port = ar_row_ptr
11 #pragma HLS INTERFACE mode=m_axi offset=slave port = B
12 #pragma HLS INTERFACE mode=m_axi offset=slave port = C_H
13
14 #pragma HLS INTERFACE mode=s_axilite port=return bundle=
15   control_signals
16
17 int rwp = 31;
18 for(int i = 0; i < n_rwp; i++)
19 {
20     int row_start=0, row_end=0;
21     row_start=ar_row_ptr[i];
22     row_end=ar_row_ptr[i+1];
23     for(int k=row_start; k < row_end; k++)
24     {
25         int j = 0;
26         int val = 0;
27         j = ar_col_idx[k];
28         val = CSR_ar_nz[k];
29
30         C_H[i]=C_H[i]+B[j]*val;
31     }
32 }
33 }

```

content/CSR top level.cpp

```

1 import time
2 from pynq import PL
3 import pynq
4 from pynq.overlay import Overlay
5
6 CSR_F=Overlay("./CSR_30x30_without_pipeline.bit")#without pipeline
7 print("Done")
8
9 for i in CSR_F.ip_dict:
10     print(i)
11
12 from pynq import allocate

```



```
43 C.sync_to_device()
44
45 row_ptr1= row_ptr.device_address
46 col_id1 = col_id.device_address
47 val1     = val.device_address
48 B1       = B.device_address
49 C1       = C.device_address
50
51 print(time.time_ns())
52 CSR_F.CSR_t_0.s_axi_control.write(0x10, val1)
53 CSR_F.CSR_t_0.s_axi_control.write(0x18, col_id1)
54 CSR_F.CSR_t_0.s_axi_control.write(0x20, row_ptr1)
55 CSR_F.CSR_t_0.s_axi_control.write(0x28, B1)
56 CSR_F.CSR_t_0.s_axi_control.write(0x30, C1)
57
58 print(time.time_ns())
59 CSR_F.CSR_t_0.s_axi_control_signals.write(0x00,1)
60 print(time.time_ns())
61
62 print(C)
```

content/CSR.py

Chapter 4

Experimental results

4.1 Random sparse matrix generation

The sparse matrices for our experiment are generated randomly for each required size and for each required sparsity. Overall the 12 sparse matrices are generated. The sparsity of matrices are 50, 60, 70 and 80 percents. The sizes of sparse matrices are 30 by 30, 60 by 60 and 120 by 120. The sparse matrices are generated using probability algorithm in C++. The desired sparsity of the sparse matrix is obtained using comparison condition between desired sparsity value and probability of random number in uniform distribution algorithm, where the random number is generated using Mersenne Twister 19937 generator. The uniform distribution algorithm has 2 parameters, which are $n=0.0$ and $m=0.1$. These values are chosen for simplicity in order to have the same scale with value of sparsity percentage .

The non zero values of the sparse matrix are obtained by binomial distribution algorithm. In the following code the binomial distribution algorithm has parameters $k=20$ and $p=0.5$, where the k represents the least upper bound and other values are distributed by parameter p . That's why most of the values of the non zero elements of sparse matrix is close to 10. All the non zero values of These sparse matrices are used both in Vitis HLS in bin format to generate IP and in PYNQ-Z2 in text format to estimate the execution time.

```
1
2 #include <stdio.h>
3 #include <random>
4 #include <iostream>
5 #include <fstream>
6 #define ROW_A 30
7 #define COL_A 30
8
9 #include <strstream>
10 using namespace std;
```

```

11 int main()
12 {
13
14
15     int A[ROW_A][COL_A];
16     float zero_prob_A = 0.5;
17
18     std::random_device rd;
19     std::mt19937 gen(rd());
20     std::binomial_distribution<int> binomial_distributed_values_A
        (20, 0.5);
21
22
23     std::mt19937 sparse_gen(rd());
24     std::uniform_real_distribution<float> sparse_dist(0.0, 1.0);
25
26     for (int i = 0; i < ROW_A; i++)
27     {
28         for (int j = 0; j < COL_A; j++)
29         {
30             if (sparse_dist(sparse_gen) > zero_prob_A)
31             {
32                 A[i][j] = binomial_distributed_values_A(gen);
33             }
34             else
35             {
36                 A[i][j] = 0;
37             }
38         }
39     }
40
41     for (int i = 0; i < ROW_A; i++)
42     {
43         for (int j = 0; j < COL_A; j++)
44         {
45             std::cout << A[i][j];
46             std::cout << " ";
47         }
48         std::cout << std::endl;
49     }
50
51     fstream sparse("S_30x30_sparsity_50_percent.TXT");
52
53     for (size_t i = 0; i < ROW_A; i++)
54     {
55         for (size_t j = 0; j < COL_A; j++)
56         {
57             sparse << A[i][j] << " ";
58             if (j == COL_A - 1)

```

```

59     {
60         sparse << "\n";
61     }
62
63     }
64
65 }
66 int nz_A = 0;
67 for (size_t i = 0; i < ROW_A; i++)
68 {
69     for (size_t j = 0; j < COL_A; j++)
70     {
71         if (A[i][j] != 0) {
72             nz_A++;
73         }
74     }
75 }
76
77 cout << nz_A<<endl;
78 int mval = 0;
79 for (size_t i = 0; i < ROW_A; i++)
80 {
81     int E=0;
82     for (size_t j = 0; j < COL_A; j++)
83     {
84         if ( A[i][j] != 0)
85         {
86             E++;
87         }
88     }
89     if( mval<E )
90     {
91         mval = E;
92     }
93 }
94 cout << mval;
95
96 }

```

content/Random matrix generation.cpp

4.2 Evaluations

4.2.1 Estimation of resource utilization

As we mentioned above the IP is generated using Xilinx Vitis HLS software. After the completion synthesis process the report of the synthesis appears. All the

resource utilization are reported as chart in the following figures. We can observe from following figures that the Ordinary matrix by vector multiplication and ELL sparse matrix storage format multiplication by vector consumes more resources than the other sparse storage formats due to using the pipelining technique. Meanwhile the COO, CSC and CSR sparse storage formats utilize almost the same amount of resources for every size of matrix and for every percentage of sparsity in our experiment. From the execution time bars, we can observe that the latency of COO format decreases when the sparsity of matrix increases, and the CSC and the CSR formats both have execution time which depend on structure of sparse matrix and on sparsity of matrix. When the sparsity matrix increases, the ELL format and ordinary matrix by vector multiplication with pipelining shows decrease in their execution time.

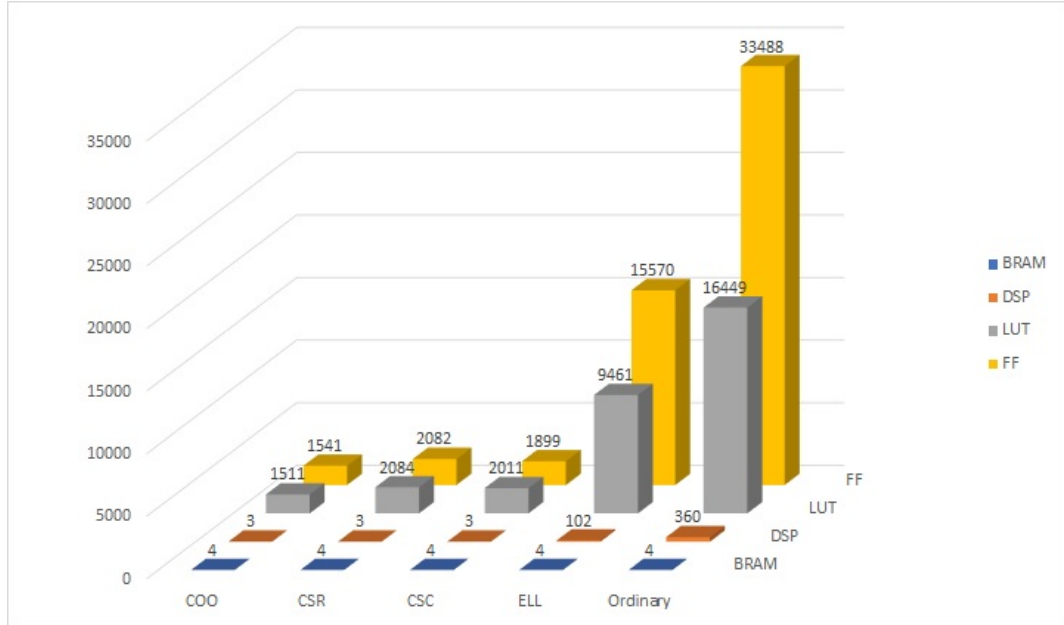


Figure 4.1: 120 by 120 matrix with 80 percent sparsity multiplied by vector

4.2.2 Estimation of execution time

The execution time is taken from the PYNQ-Z2 board by implementing the bitstream file. As mentioned in the chapter 1.

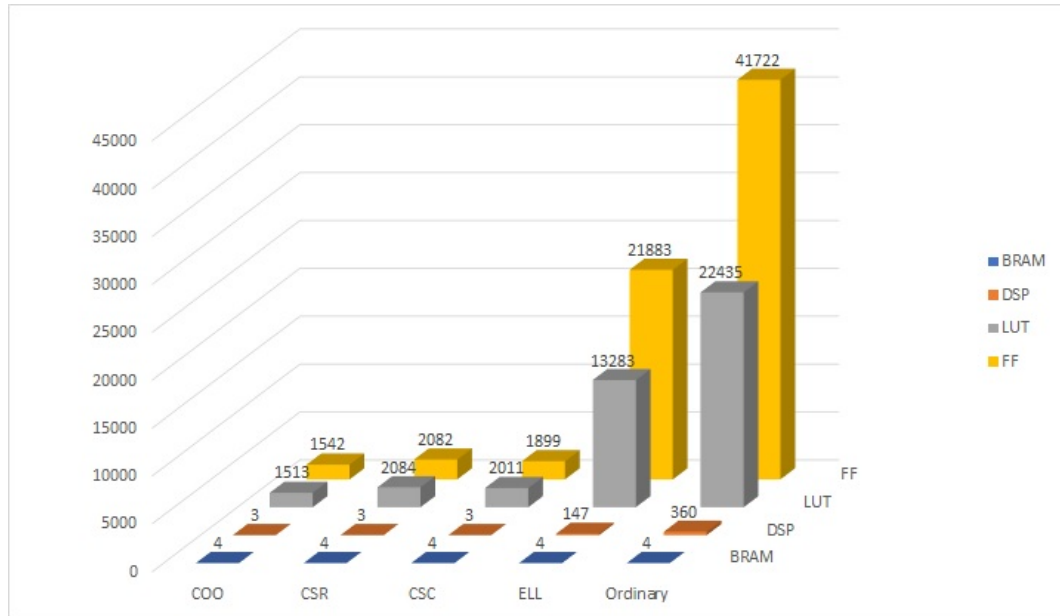


Figure 4.2: 120 by 120 matrix with 70 percent sparsity multiplied by vector

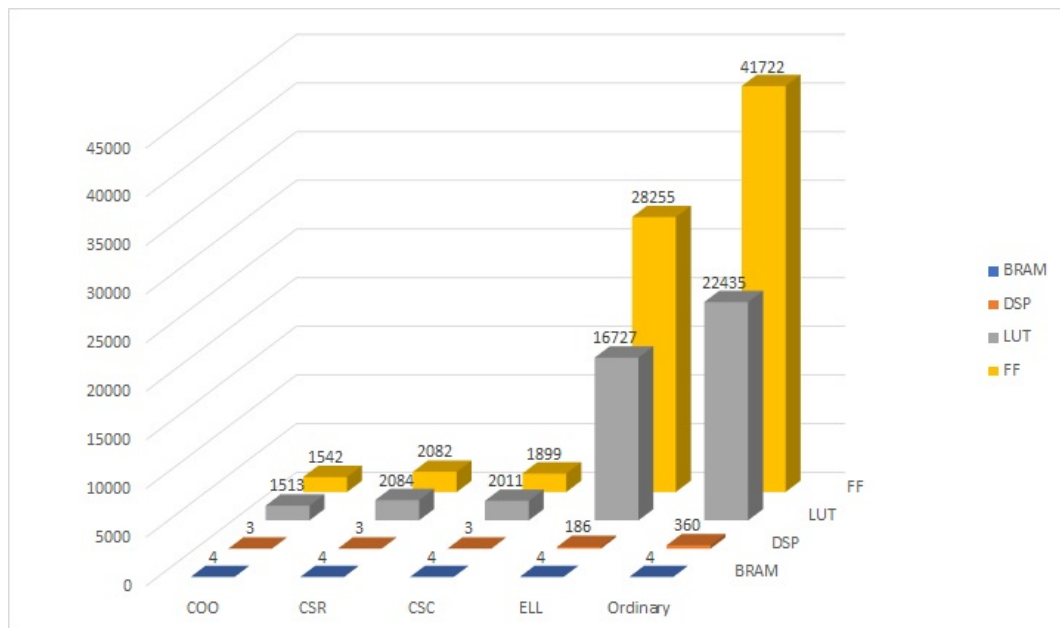


Figure 4.3: 120 by 120 matrix with 60 percent sparsity multiplied by vector

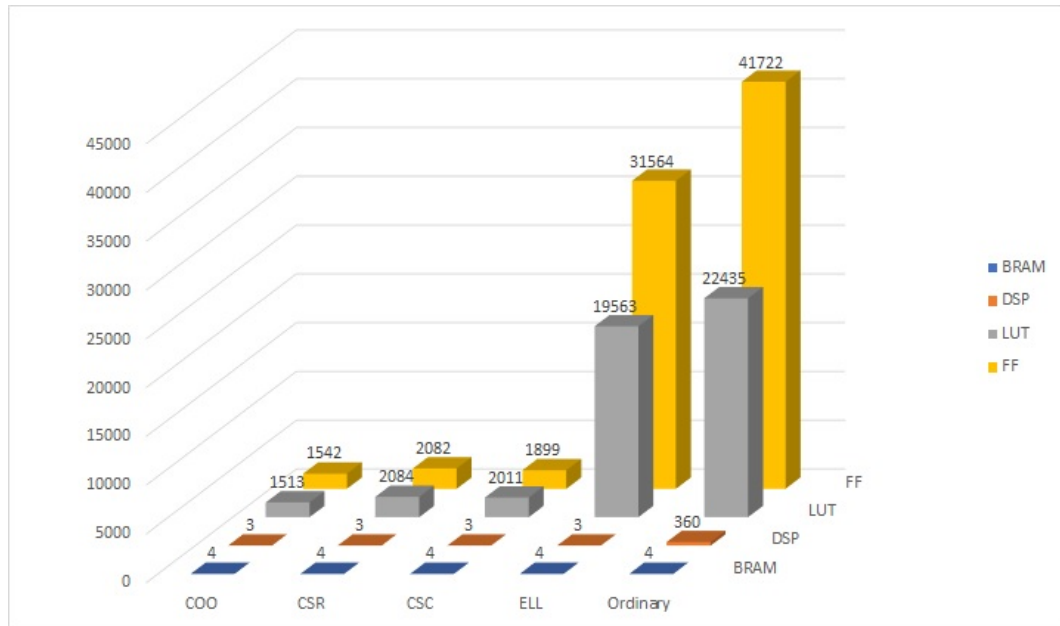


Figure 4.4: 120 by 120 matrix with 50 percent sparsity multiplied by vector

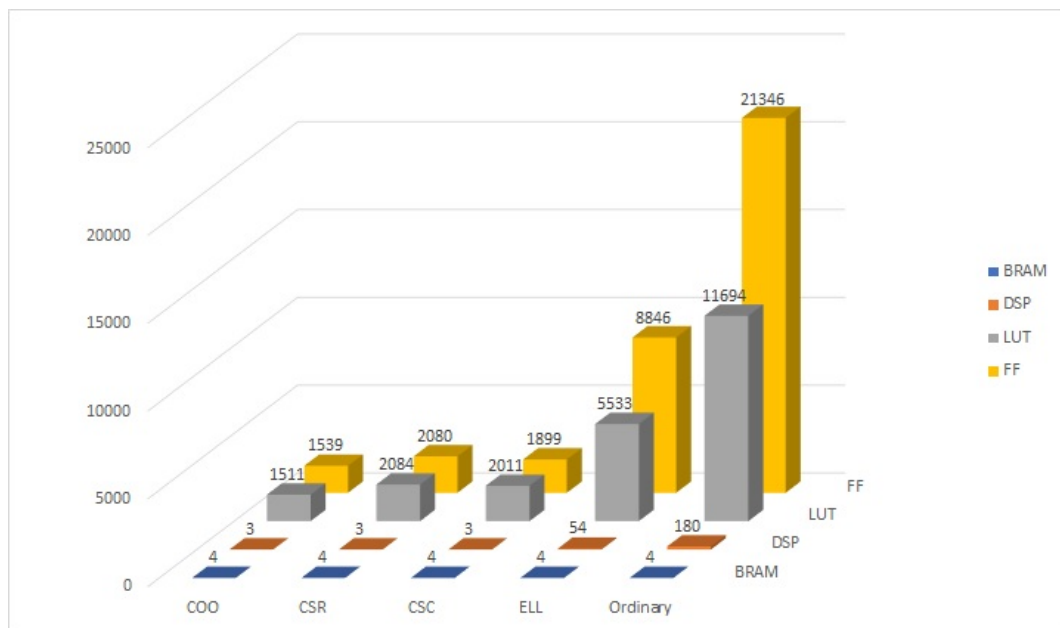


Figure 4.5: 60 by 60 matrix with 80 percent sparsity multiplied by vector

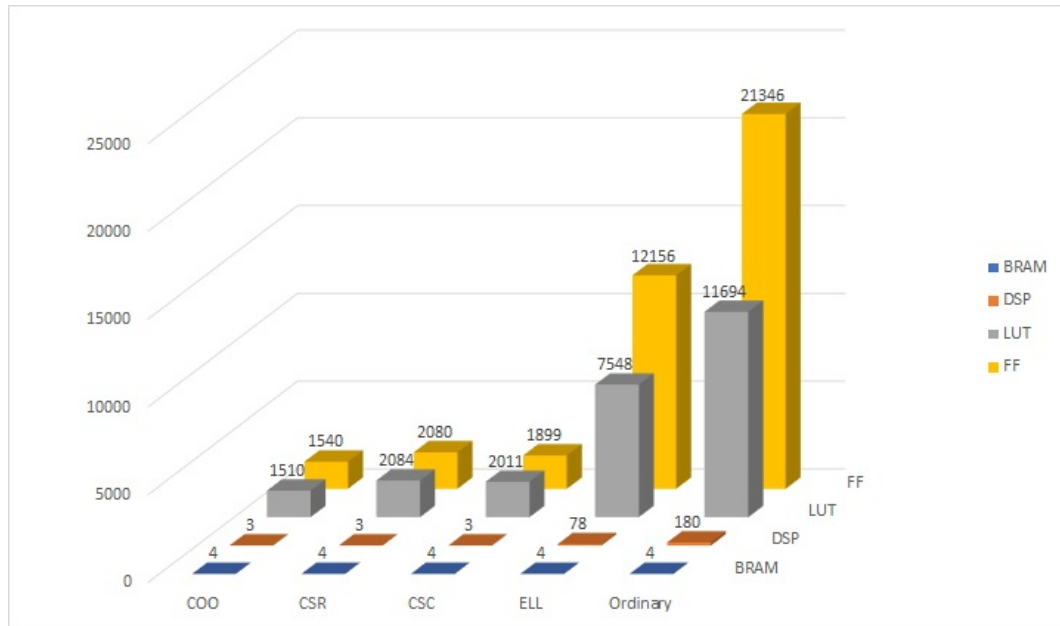


Figure 4.6: 60 by 60 matrix with 70 percent sparsity multiplied by vector

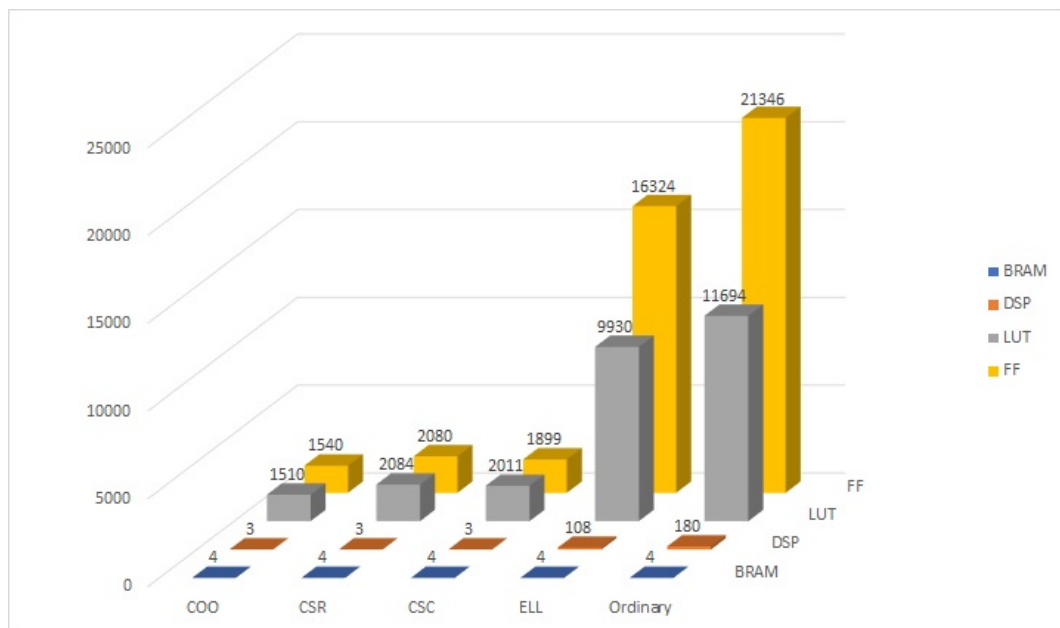


Figure 4.7: 60 by 60 matrix with 60 percent sparsity multiplied by vector

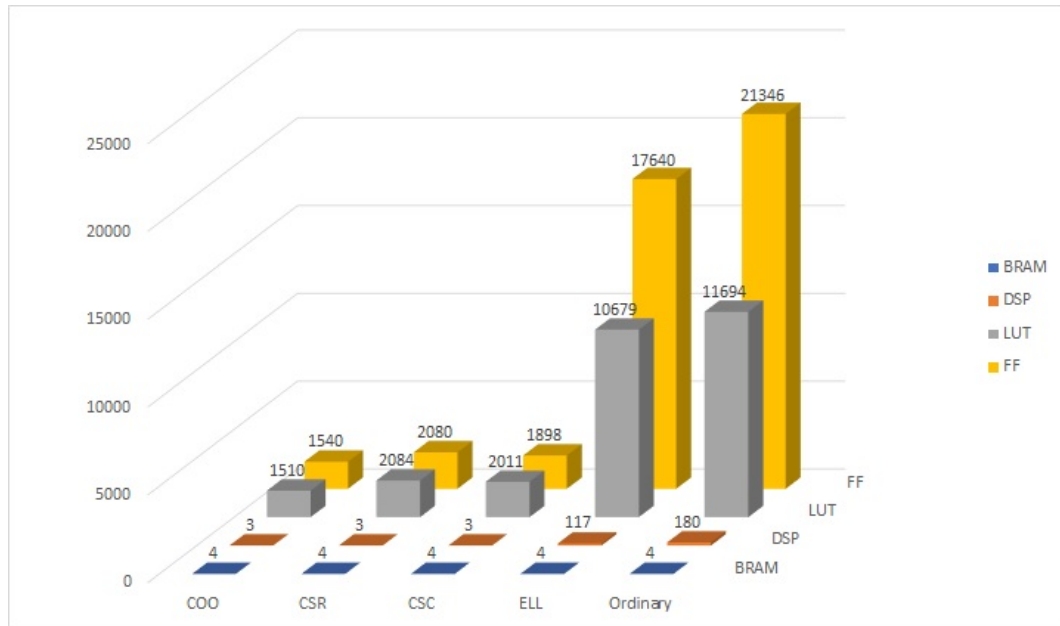


Figure 4.8: 60 by 60 matrix with 50 percent sparsity multiplied by vector

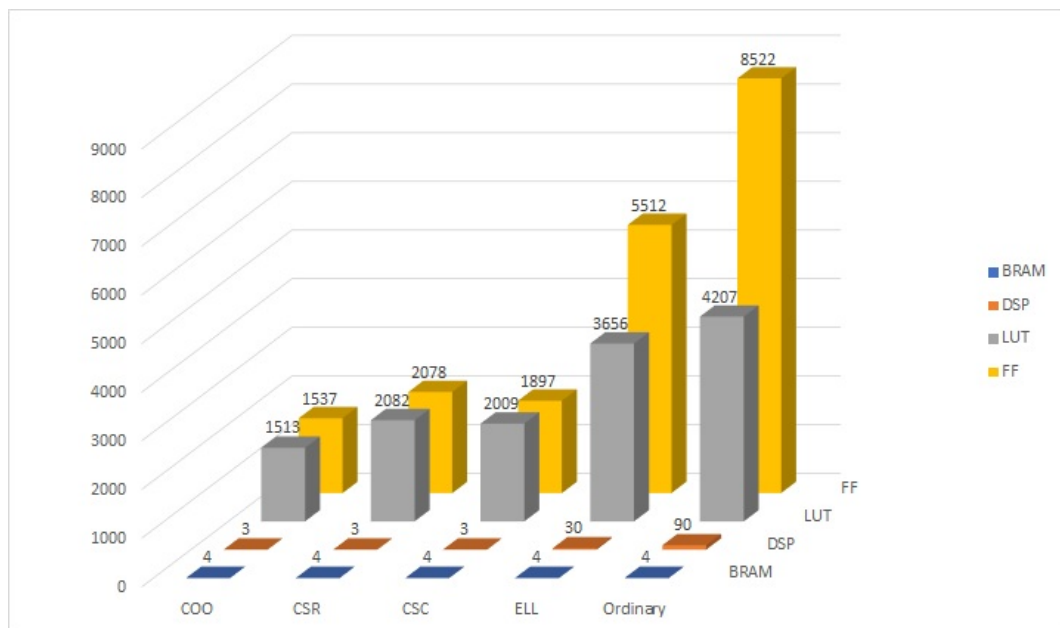


Figure 4.9: 30 by 30 matrix with 80 percent sparsity multiplied by vector

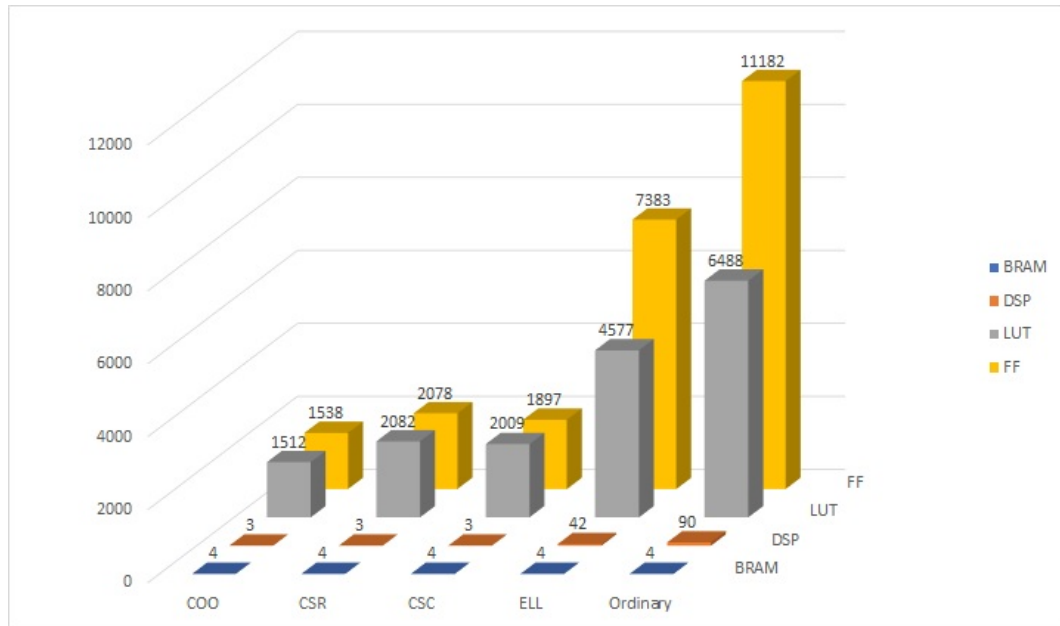


Figure 4.10: 30 by 30 matrix with 70 percent sparsity multiplied by vector

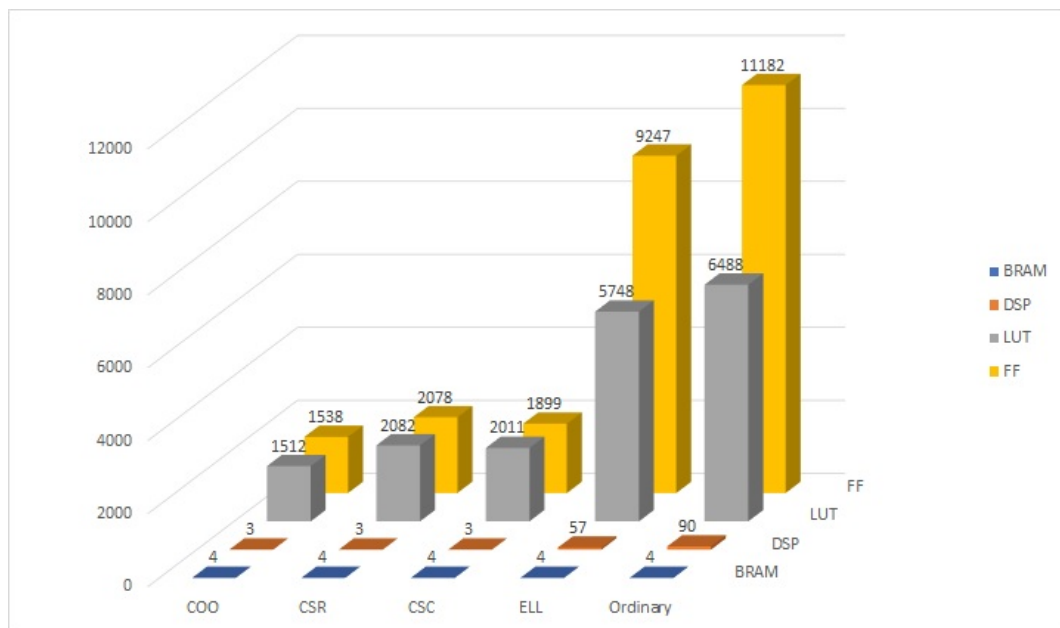


Figure 4.11: 30 by 30 matrix with 60 percent sparsity multiplied by vector

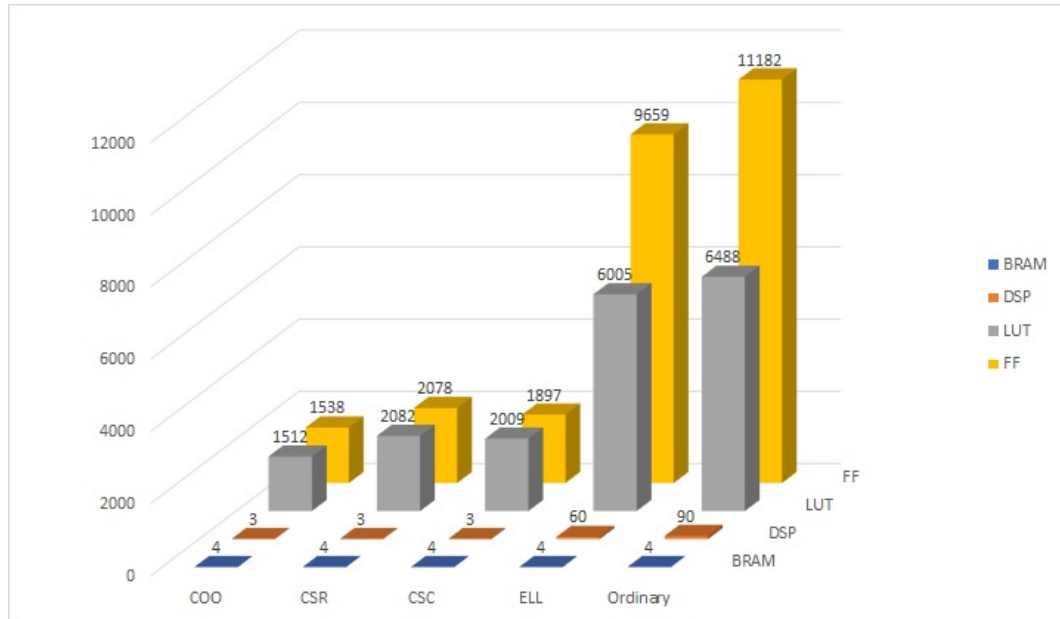


Figure 4.12: 30 by 30 matrix with 50 percent sparsity multiplied by vector

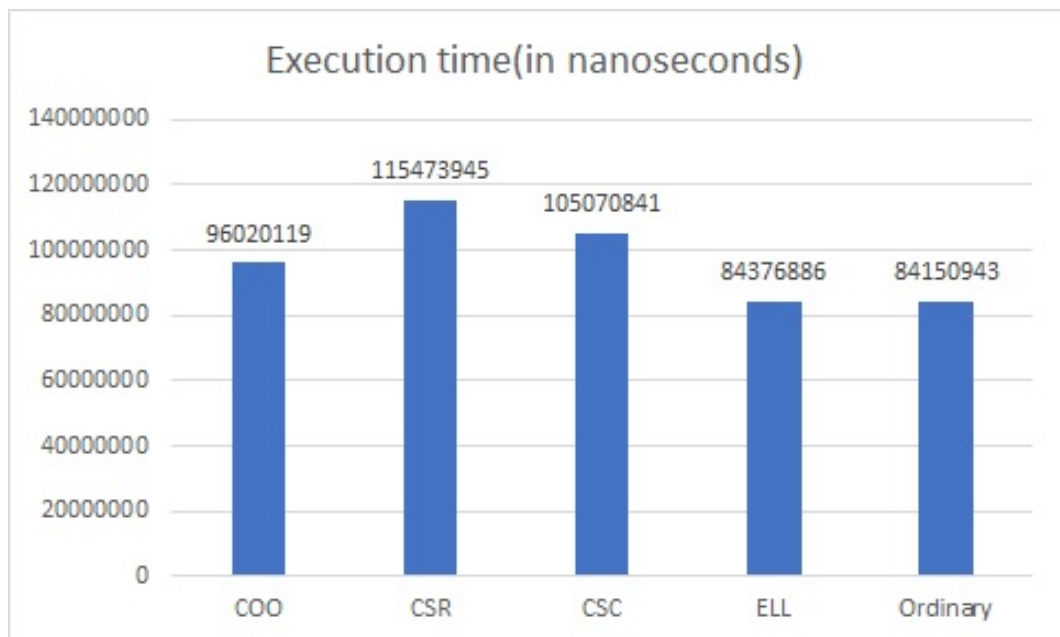


Figure 4.13: 120 by 120 matrix with 80 percent sparsity multiplied by vector

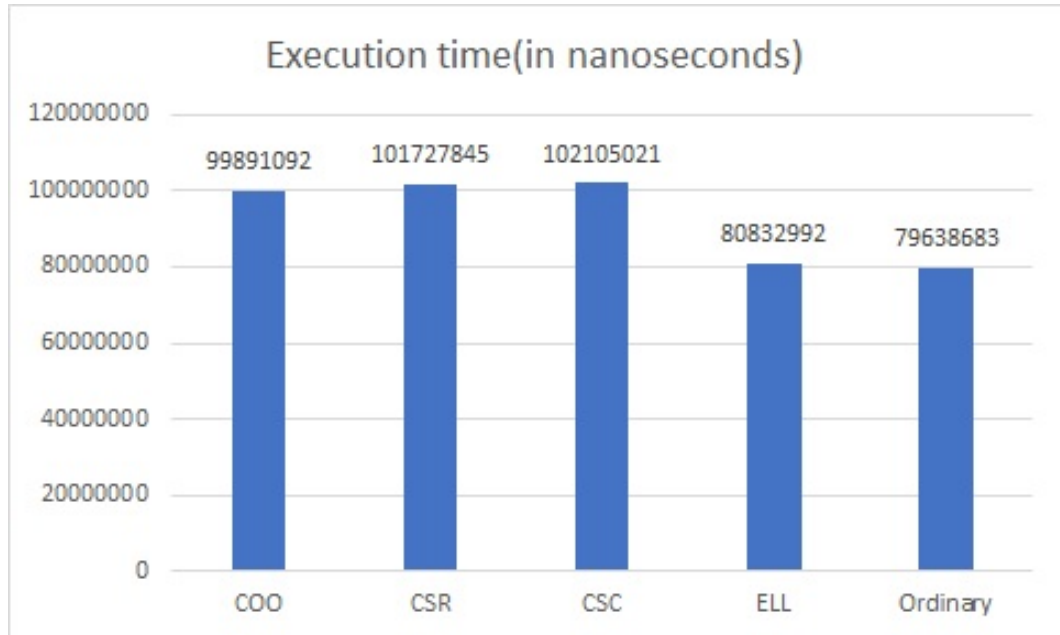


Figure 4.14: 120 by 120 matrix with 70 percent sparsity multiplied by vector

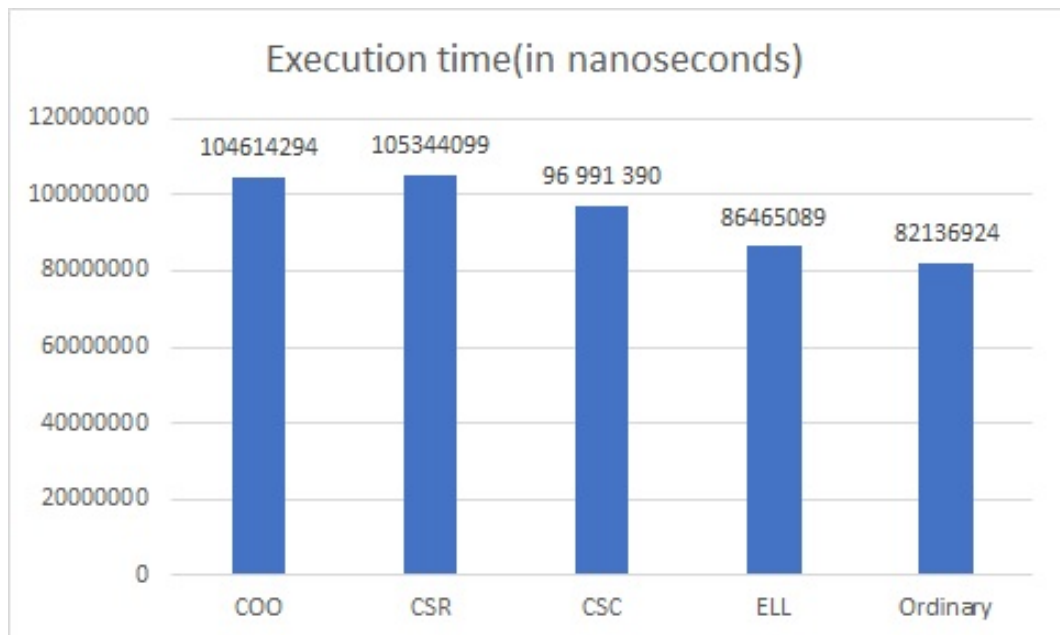


Figure 4.15: 120 by 120 matrix with 60 percent sparsity multiplied by vector

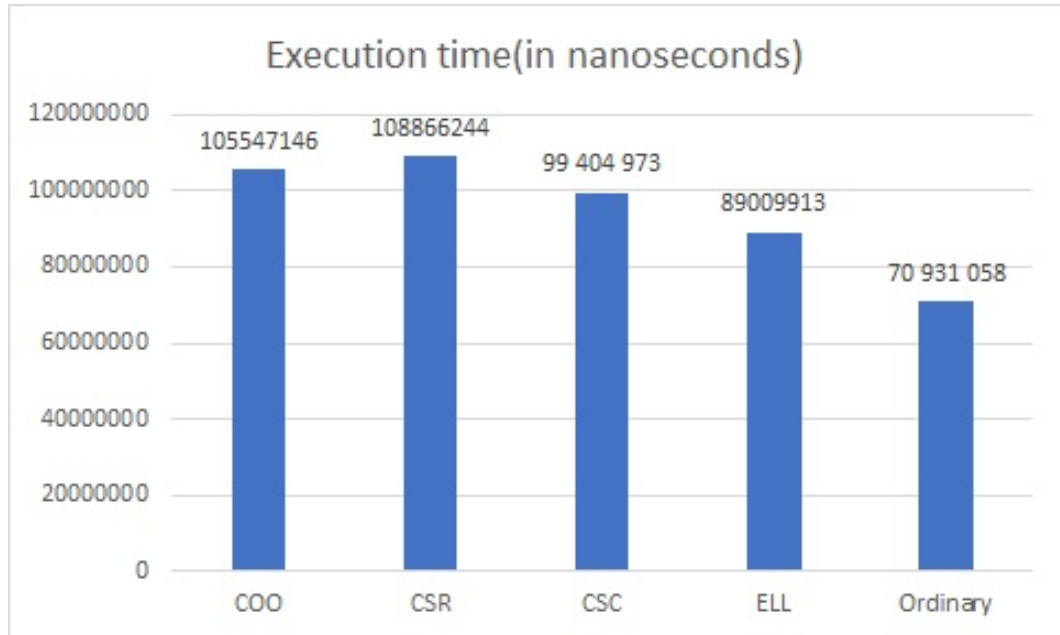


Figure 4.16: 120 by 120 matrix with 50 percent sparsity multiplied by vector

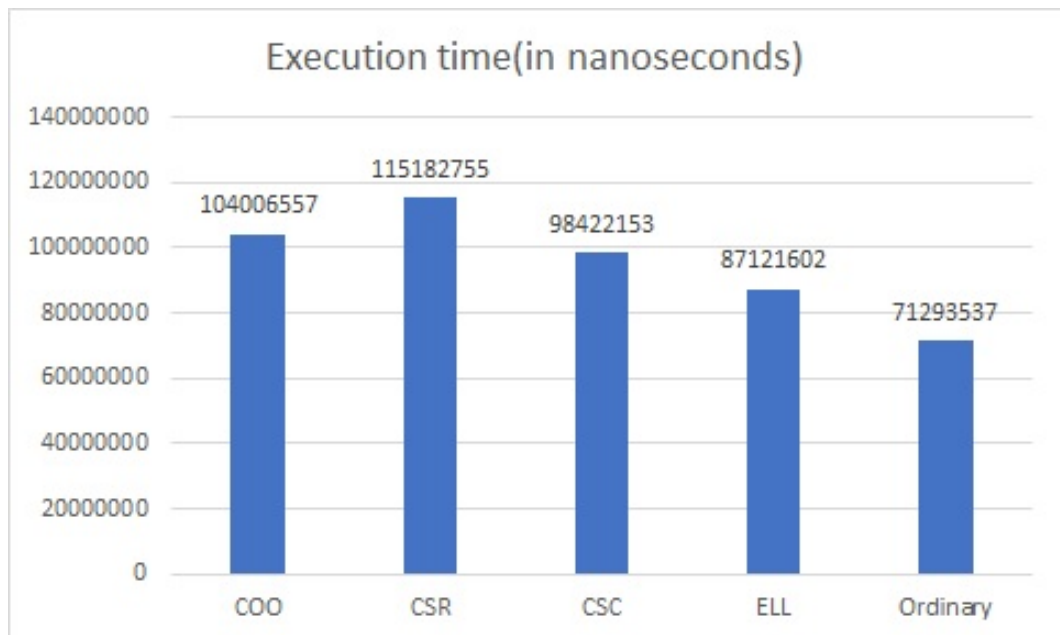


Figure 4.17: 60 by 60 matrix with 80 percent sparsity multiplied by vector

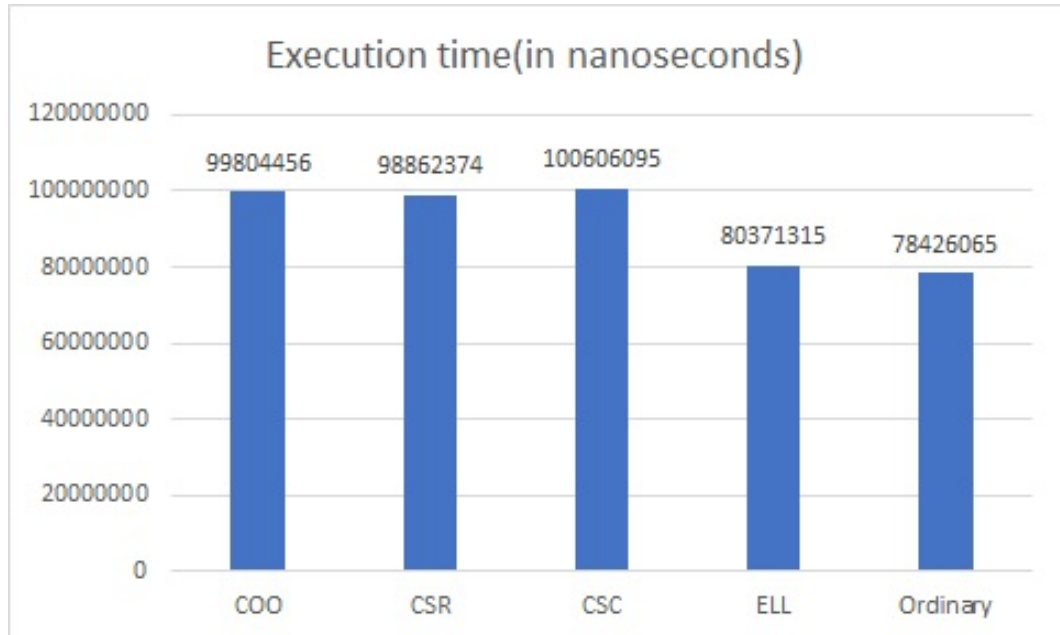


Figure 4.18: 60 by 60 matrix with 70 percent sparsity multiplied by vector

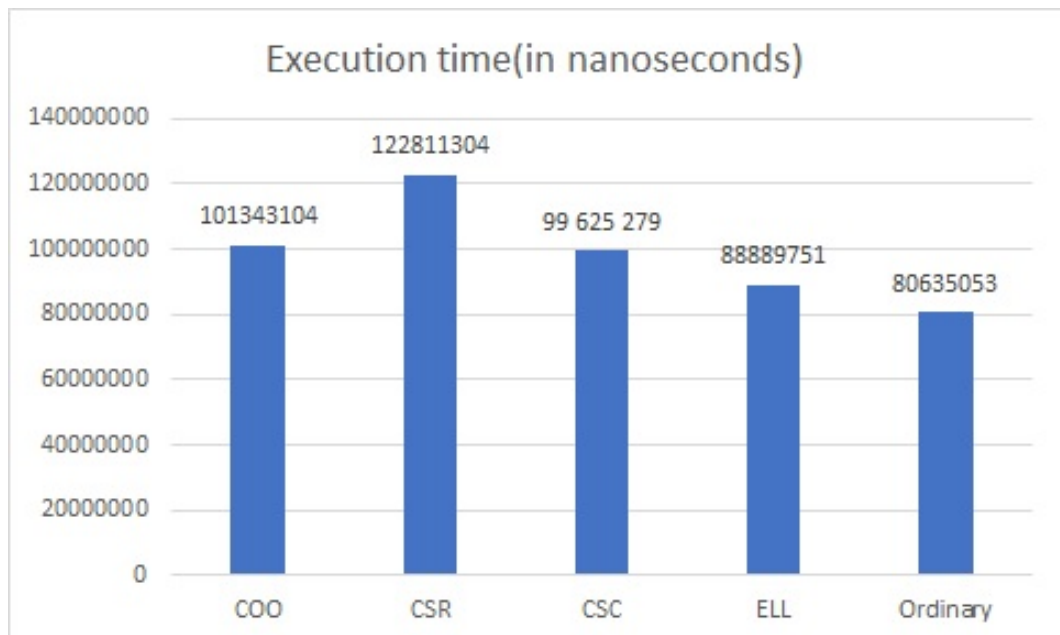


Figure 4.19: 60 by 60 matrix with 60 percent sparsity multiplied by vector

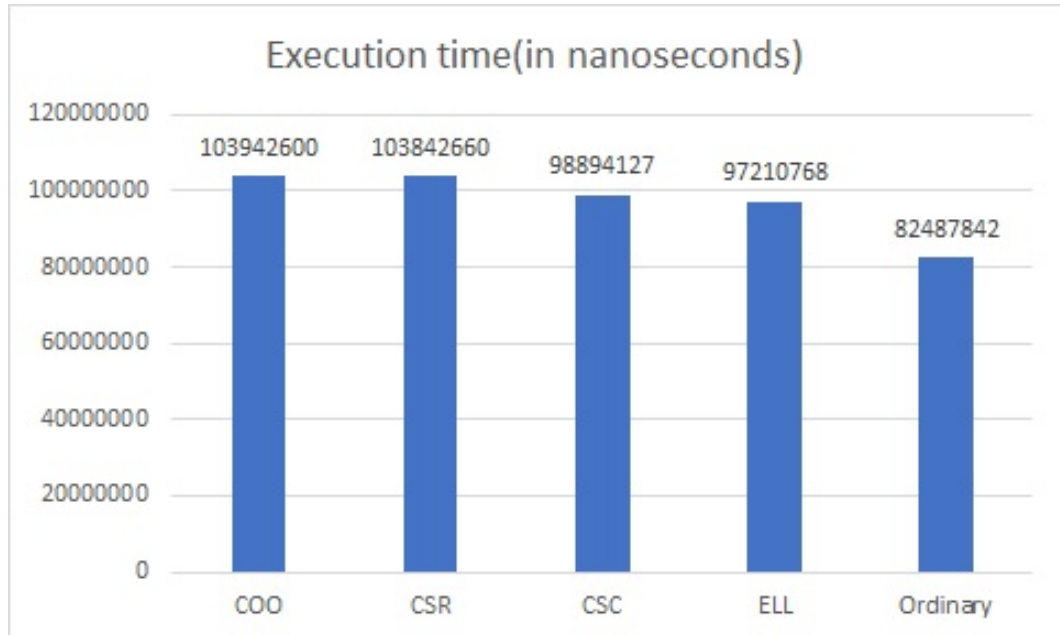


Figure 4.20: 60 by 60 matrix with 50 percent sparsity multiplied by vector

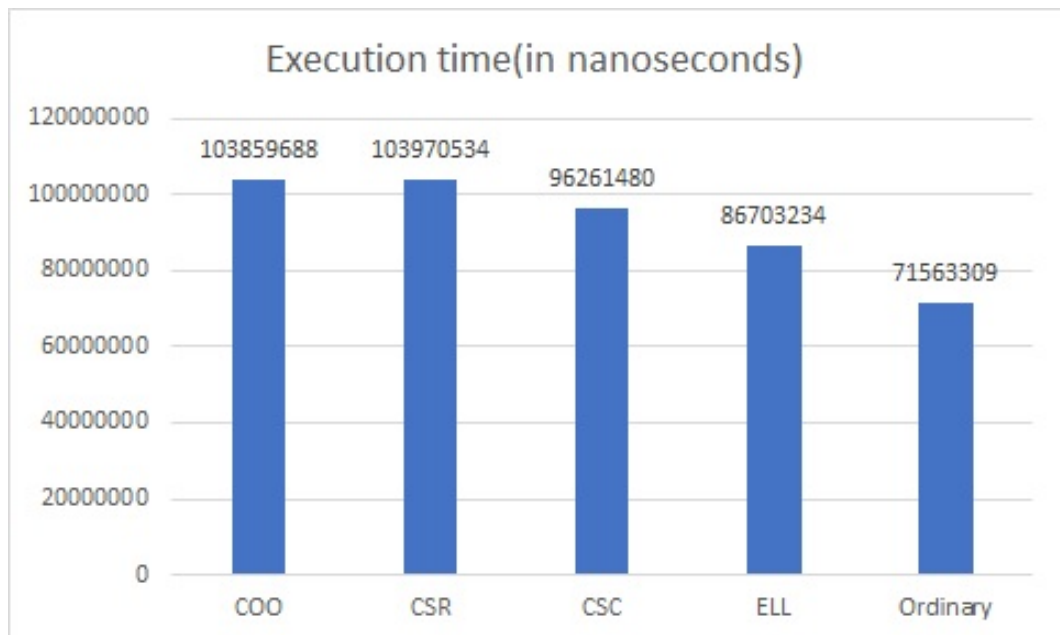


Figure 4.21: 30 by 30 matrix with 80 percent sparsity multiplied by vector

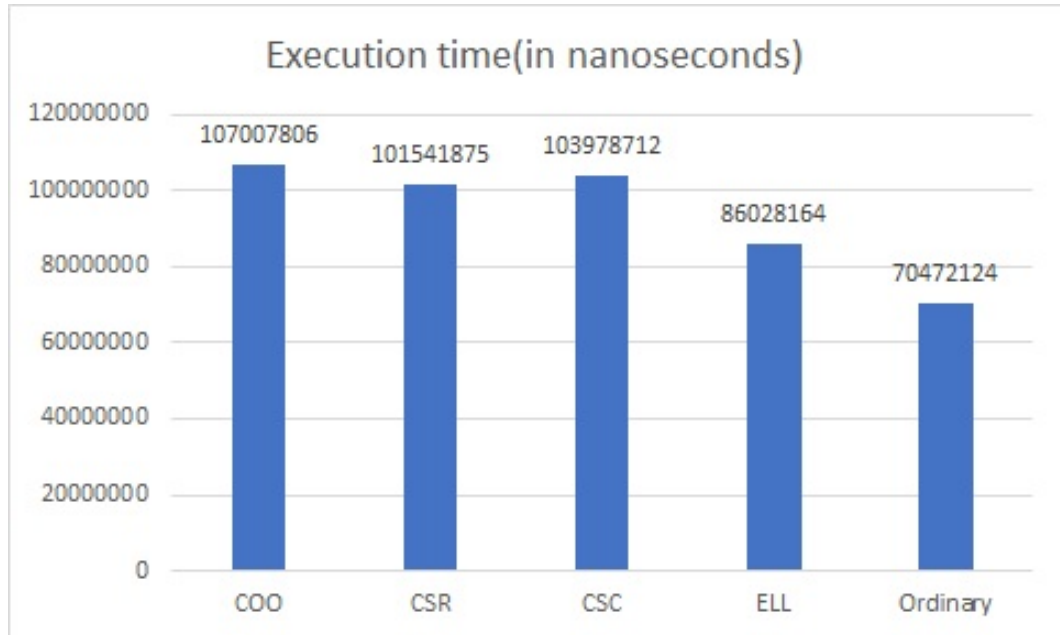


Figure 4.22: 30 by 30 matrix with 70 percent sparsity multiplied by vector

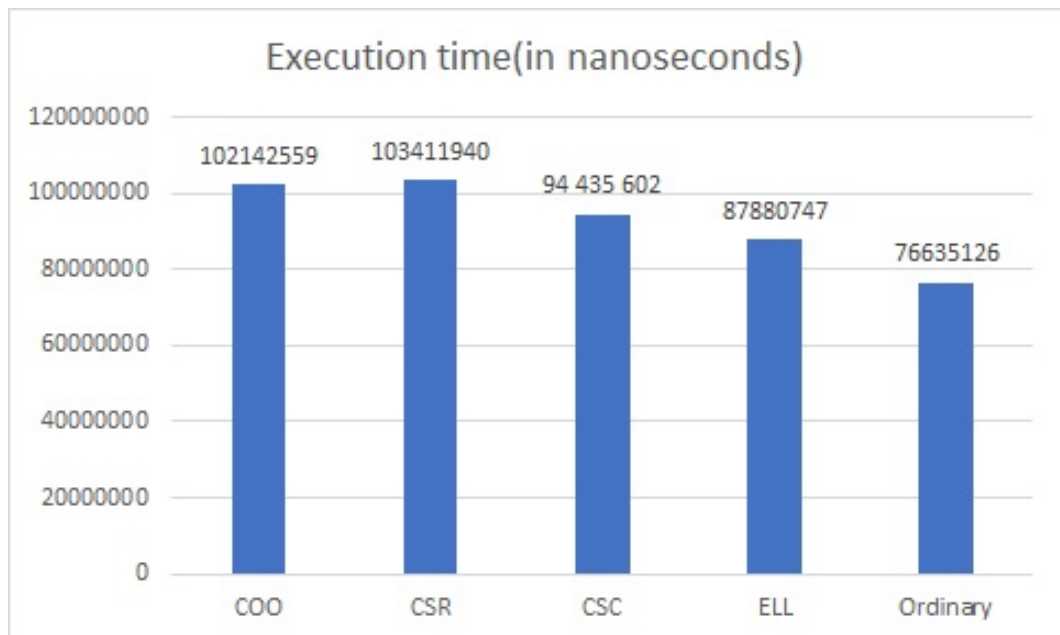


Figure 4.23: 30 by 30 matrix with 60 percent sparsity multiplied by vector

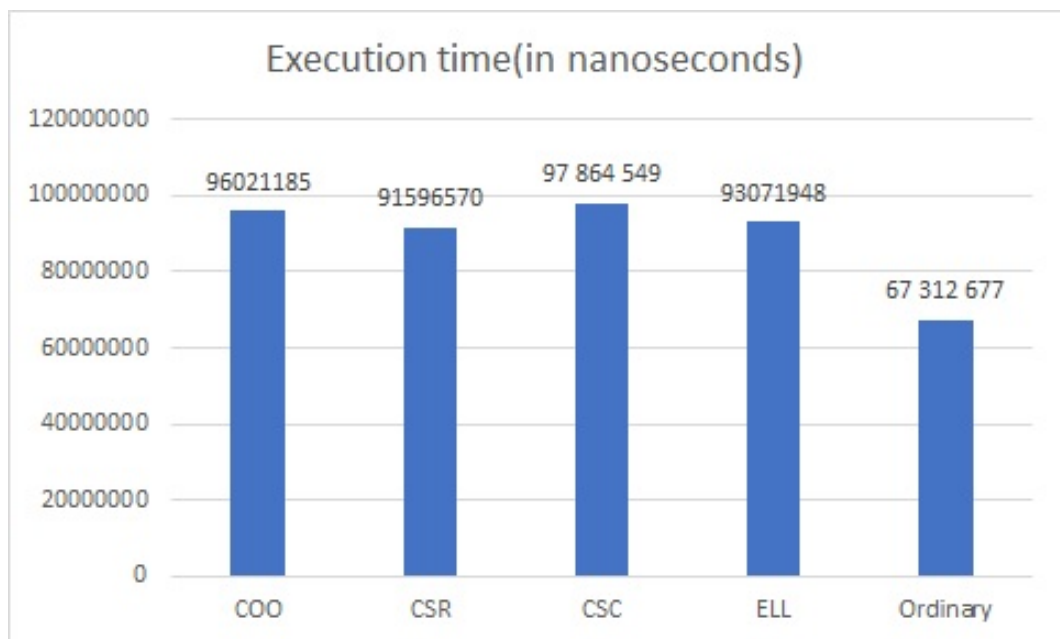


Figure 4.24: 30 by 30 matrix with 50 percent sparsity multiplied by vector

Chapter 5

Conclusion

5.1 Discussion

5.2 Future work

Appendix A

Codes

A.1 Test bench code

```
1 int main()
2 { //defining the sparse matrix, vector and output vector
3   d_in A[ROW_A][COL_A], B[COL_A],C[ROW_A],C_H[ROW_A];
4   d_in error_counter=0;// to check the results from test bench
   and hardware
5
6   for(int i = 0; i < ROW_A; i++)
7   {
8     for(int j = 0; j < COL_A; j++)
9     {
10      A[i][j] = 0;
11    }
12  }
13
14  fstream sparse_file("S_30x30_50%.bin");
15
16  for(int i = 0; i < ROW_A; i++)
17  {
18    for(int j = 0; j < COL_A; j++)
19    {
20      sparse_file>>A[i][j];
21    }
22  }
23
24  for(int i = 0; i < COL_A; i++)
25  {
26    B[i] = 1;
27  }
28
29
```

```

30     for(int i = 0; i < ROW_A; i++)
31     {
32         C[i] = 0;
33         C_H[i] = 0;
34     }
35
36     // COO format
37
38     int nz_val_A=0;
39     std::vector<int> row_indx;;
40     std::vector<int> col_indx;
41     std::vector<int> val_vec;
42
43     d_in COO_row[N];
44     d_in COO_col[N];
45     d_in COO_val[N];
46
47     for(int i = 0; i < N; i++)
48     {
49         COO_row[i]=0;
50         COO_col[i]=0;
51         COO_val[i]=0;
52     }
53
54     for(int i = 0; i < ROW_A; ++i)
55     {
56         for(int j = 0; j < COL_A; ++j)
57         {
58             if( A[i][j]!=0 )
59             {
60                 row_indx.push_back(i);
61                 col_indx.push_back(j);
62                 val_vec.push_back(A[i][j]);
63                 nz_val_A++;
64             }
65         }
66     }
67
68 }
69
70 std::copy(row_indx.begin(), row_indx.end(), COO_row);
71 std::copy(col_indx.begin(), col_indx.end(), COO_col);
72 std::copy(val_vec.begin(), val_vec.end(), COO_val);
73
74 // CSC_format
75
76 std::vector<int> nz_vector;
77 std::vector<int> row_idx_vector;
78 std::vector<int> col_ptr_vector;

```



```
79  int ar_nz[N], ar_row_idx[M], ar_col_ptr[L], k=0;
80  int n_nz_A=0;
81  int clp=0;
82
83
84  for(int i = 0; i < N; i++)
85  {
86      ar_nz[i]=0;
87      ar_row_idx[i]=0;
88  }
89
90  for(int i = 0; i < L; i++)
91  {
92      ar_col_ptr[i]=0;
93  }
94
95  //  computation of vector containing column start pointers.
96
97  col_ptr_vector.push_back(0);
98  for(int j = 0; j < COL_A; j++)
99  {
100     if( j==0 )
101     {
102         k++;
103     }
104
105     for(int i = 0; i < ROW_A; i++)
106     {
107         if( A[i][j]!=0 )
108         {
109
110             if(k!=j && j!=0)
111             {
112                 col_ptr_vector.push_back(n_nz_A);
113                 clp++;
114                 k--;
115                 if((k-j)>1)
116                 {
117                     k=j;
118                 }
119             }
120             n_nz_A++;
121         }
122     }
123
124     k++;
125     if( k-j<2 )
126     {
127
```

```

128         k = j+2;
129     }
130 }
131
132 col_ptr_vector.push_back(n_nz_A);
133
134 for(int j = 0; j < COL_A; j++)
135 {
136     for(int i = 0; i < ROW_A; i++)
137     {
138         if( A[i][j]!=0 )
139         {
140             nz_vector.push_back(A[i][j]);
141             row_idx_vector.push_back(i);
142         }
143     }
144 }
145
146 }
147
148
149 std::copy(nz_vector.begin(), nz_vector.end(), ar_nz);
150 std::copy(row_idx_vector.begin(), row_idx_vector.end(),
151 ar_row_idx);
152 std::copy(col_ptr_vector.begin(), col_ptr_vector.end(),
153 ar_col_ptr);
154
155 // CSR_format
156
157 std::vector<int> nz_vector;
158 std::vector<int> col_idx_vector;
159 std::vector<int> row_ptr_vector;
160 int ar_nz[N], ar_col_idx[M], ar_row_ptr[L], k=0;
161 int n_nz_A=0;
162 int rwp=0;
163
164 for(int i = 0; i < M; i++)
165 {
166     ar_nz[i]=0;
167     ar_col_idx[i]=0;
168 }
169 for(int i = 0; i < L; i++)
170 {
171     ar_row_ptr[i]=0;
172 }
173
174 // computation of vector containing the row start pointers.

```

```
175 row_ptr_vector.push_back(0);
176 for(int i = 0; i < ROW_A; i++)
177 {
178     if( i==0 )
179     {
180         k++;
181     }
182
183     for(int j = 0; j < COL_A; j++)
184     {
185         if( A[i][j]!=0 )
186         {
187
188             if(k!=i && i!=0)
189             {
190                 row_ptr_vector.push_back(n_nz_A);
191                 rwp++;
192                 k = i;
193
194
195             }
196             n_nz_A++;
197         }
198     }
199
200     k++;
201
202     if(k-i<2)
203     {
204         k=i+2;
205     }
206 }
207
208 rwp++;
209 n_nz_A++;
210 row_ptr_vector.push_back(n_nz_A);
211
212 for(int i = 0; i < COL_A; i++)
213 {
214     for(int j = 0; j < ROW_A; j++)
215     {
216         if( A[i][j]!=0 )
217         {
218             nz_vector.push_back(A[i][j]);
219             col_idx_vector.push_back(j);
220         }
221     }
222 }
223 }
```

```

224
225     std::copy(nz_vector.begin(),      nz_vector.end(),      ar_nz)
226 ;
227     std::copy(col_idx_vector.begin(), col_idx_vector.end(),
228 ar_col_idx);
229     std::copy(row_ptr_vector.begin(), row_ptr_vector.end(),
230 ar_row_ptr);
231
232 //  ELL_format
233
234 int ELL_clm_idx[ROW_A][ELL_COL], ELL_val[ROW_A][ELL_COL];
235 for(int i = 0; i < ROW_A; i++)
236 {
237     for(int j = 0; j < ELL_COL; j++)
238     {
239         ELL_clm_idx[i][j] = 0;
240         ELL_val[i][j]=0;
241     }
242 }
243
244 for(int i=0; i < ROW_A; i++)
245 {
246     int E=0;
247     for(int j=0; j < COL_A; j++)
248     {
249         if( A[i][j]!=0 )
250         {
251             ELL_val[i][E]=A[i][j];
252             ELL_clm_idx[i][E]=j;
253             E++;
254         }
255     }
256 }
257
258 // TEST BENCH COMPUTATION
259
260 for(int i = 0; i < ROW_A; i++)
261 {
262     for(int j = 0; j < COL_A; j++)
263     {
264         C[i] += A[i][j]*B[j];
265     }
266 }
267
268 //  calling the top level function
269
270 COO_t( COO_row,COO_col,COO_val,B,C_H);
271 CSC_t( clp,ar_nz,ar_row_idx,ar_col_ptr,B,C_H);

```

```

270 CSR_t( CSR_nz_val, CSR_col_idx, CSR_row_ptr,B,C_H);
271 ELL_t( ELL_clm_idx,ELL_val,B,C_H);
272
273 // checking the results
274
275 for(int i = 0; i < ROW_A; i++)
276 {
277     if(C[i] != C_H[i])
278     {
279         error_counter++;
280     }
281 }
282
283 std::cout<< " " << std::endl;
284 std::cout<<"The error counter: " <<error_counter<<std::endl;
285 std::cout<< " " << std::endl;
286
287 if( error_counter == 0 )
288 {
289     std::cout << "The test is passed correctly " <<std::endl;
290
291 }else{
292
293     std::cout << "There is an error in the code " <<std::endl;
294 }

```

content/Test_bench_new1.cpp

A.2 Header file

```

1 #ifndef __Hf4StF__
2 #define __Hf4StF__
3 #include <iostream>
4 #include <time.h>
5 #include <random>
6 #include <vector>
7
8
9 #define ROW_A 30
10 #define COL_A 30
11 #define L 31
12 #define M 442
13 #define N 442
14
15 typedef int d_in, d_out;
16
17 #define ELL_COL 20//the number of columns in ELL format

```

```
18
19
20 void ELL_t ( d_in ELL_clm_idx[ROW_A][ELL_COL], d_in ELL_val[
    ROW_A][ELL_COL], d_in B[COL_A], d_out C_H[ROW_A] );
21
22 void COO_t (d_in COO_row[N], d_in COO_col[N], d_in COO_val[N],
    d_in B[COL_A], d_in C_H[ROW_A]);
23
24 void CSR_t (d_in CSR_ar_nz[M], d_in ar_col_idx[N], d_in
    ar_row_ptr[L], d_in B[COL_A], d_in C_H[ROW_A]);
25
26 void CSC_t (d_in CSC_ar_nz[N],d_in ar_row_idx[M], d_in ar_col_ptr
    [L], d_in B[COL_A], d_in C_H[ROW_A]);
27
28 void full_MVM_t(d_in A[ROW_A][COL_A], d_in B[COL_A] ,d_in C_H[
    ROW_A]);
29
30 #endif
```

content/Header file.cpp