Sparse Matrix-Vector Multiplication using FPGA

NAN WANG
(SID: 904-357-336)

02/11/2015
# Table of Contents

1 Introduction ................................................................................................................. 3

2 Background .................................................................................................................. 4
   2.1 SpMXV Problem .................................................................................................... 4
   2.2 Roach FPGA System ............................................................................................ 4

3 System Design ............................................................................................................... 5
   3.1 CRS Format ............................................................................................................ 5
   3.2 System diagram ...................................................................................................... 6
   3.3 Memory control ..................................................................................................... 6
   3.4 Multiplier ................................................................................................................ 8
   3.5 Accumulator .......................................................................................................... 9
   3.6 QDR and Block RAM ............................................................................................ 10

4 Simulation .................................................................................................................... 11
   4.1 Matrix and Vector Generation .............................................................................. 11
   4.2 Testbench .............................................................................................................. 11
   4.3 Simulation Result .................................................................................................. 11

5 FPGA Mapping and Performance ................................................................................. 12
   5.1 FPGA Mapping Setup ........................................................................................ 12
   5.2 Performance .......................................................................................................... 13
   5.3 Future Improvement ............................................................................................. 16

6 Conclusion ................................................................................................................... 16

References ....................................................................................................................... 17
1 Introduction

FPGAs have been used for a number of integer and fixed-point applications for many years. However, with the rapid advances in technology, current FPGAs contain much more configurable logic blocks than before. Nowadays, FPGAs are capable for a wider range of applications, like the complex floating-point operations. Some researchers have suggested that FPGAs have become highly competitive with microprocessors in both peak performance and average performance. Some research have been done such as molecular dynamics and dense matrix multiplication, using FPGAs, which achieves high performance. In other words, FPGAs have been proving its potentials in academic research field, and have begun to land on general-purpose processor and FPGA-based application accelerator market.

Floating-point Sparse Matrix-Vector Multiplication (SpMXV), $y = Ax$, is a key computational kernel that dominates the performance of many scientific and engineering applications. For example, algorithms for least squares problems and eigenvalue problems [6], as well as the image reconstruction in medical imaging all need to solve sparse linear systems using iterative methods. Such methods involve large numbers of sparse matrix-vector multiplications. Unfortunately, the performance of sparse matrix algorithms tends to be much lower than their dense matrix counterparts for two primary reasons. First, the irregularity of memory accesses causes large numbers of cache misses, while the memory speed is much slower compared to the processing speed. Second, the high ratio of load and store operations to floating-point operations stresses the load/store units, while the floating-point units are often under-utilized. [7] Although some optimizations have been proposed to improve the performance of SpMXV on cache-based memory systems, they require information on the sparsity structure of the input matrix. For matrices with very irregular sparsity structure, these optimizations provide little improvement [1].

FPGAs have become an attractive option for implementing SpMXV. Besides the high floating-point performance, the current FPGA fabrics also provide large amounts of on-chip memory as well as plenty of I/O pins. These features enable FPGA designs to provide high on-chip and off-chip memory bandwidth to I/O-bound applications. Thus, FPGA-based designs are able to avoid the long latency caused by the cache misses. [5]

This report talks about an implementation of a simple yet efficient design for SpMXV, which utilizes a reduction circuit for accumulator. The design has the following advantages. Firstly, it conforms to an existing sparse matrix storage format and makes no assumption on the sparsity structure of the input matrix. The design accepts matrices in Compressed Row Storage (CRS) format, one of the most commonly used storage formats for sparse matrices. It can be applied to any sparse matrix, regardless of its sparsity structure or size. In particular, the performance of the design only depends on the number of nonzero elements in each row, and not on the distribution of these nonzeros among the columns. For matrices with very irregular sparsity structure, the design does not suffer as much performance degradation as do the general-purpose processors.
Secondly, the design is able to achieve high performance for floating-point SpMXV. As the nonzero elements are accessed in the same order as they are stored in the memory, no complex cache management mechanism is needed, nor does the design suffer from the long access delay caused by the cache misses.

2 Background

2.1 SpMXV Problem

SpMXV is defined as follows. Consider the matrix-vector product \( y = Ax \). \( A \) is an \( n \times n \) sparse matrix with \( n_z \) nonzero elements. We use \( A_{ij} \) \((i = 0, \ldots, n-1; j = 0, \ldots, n-1)\) to denote the elements of \( A \). \( x \) and \( y \) are vectors of length \( n \), and their elements are denoted by \( x_j, y_i \) \((j = 0, \ldots, n-1; i = 0, \ldots, n-1)\) respectively. Initially \( A, x, y \) are in the external memory. Thus, to perform \( y = Ax \), \( n_z + n \) input operations are needed to bring \( A \) and \( x \) to the processor. Note that due to the irregular sparsity structure, either the row index \( i \) or the column index \( j \) or both of them need to be read together with \( A_{ij} \), if \( A_{ij} \neq 0 \). For matrix element \( A_{ij} \), the following computation is performed:

\[
y_i = y_i + A_{ij} \times x_j, \text{ if } A_{ij} \neq 0
\]  

(1)

As each nonzero element of \( A \) needs two floating-point operations, the total number of floating-point operations to be performed is \( 2n_z \). When the computation is completed, \( n \) output operations are needed to write \( y \) back to the external memory. Thus the total number of I/O operations is \( n_z + n + n \).

2.2 Roach FPGA System

The platform that is utilized in the project is ROACH which is short for Reconfigurable Open Architecture Computing Hardware. It is a standalone FPGA processing board. ROACH is a Virtex5-based upgrade to current CASPER hardware. It merges aspects from the IBOB and BEE2 platforms into a single board. The CX4/XAUI/10GbE interfaces of both are kept, while combining the Z-DOK daughter board interface of the IBOB with the high bandwidth/capacity DRAM memory and standalone usage model of the BEE2. ROACH is a single-FPGA board, dispensing with the on-board inter-FPGA links in favor of 10GbE interfaces for all cross-FPGA communications. Figure 1 shows its block diagram.
3 System Design

3.1 CRS Format

In the design, the sparse matrix is stored in CRS format, in which the nonzeros of the matrix rows are stored in contiguous memory locations. In CRS format, there are three vectors: val for floating-point nonzero elements; col for the column indices of the nonzeros; and ptr that stores the locations in the val vector that start a row.

As an example, consider the sparse matrix \( A \) as follows:

\[
A = \begin{pmatrix}
0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 4 \\
9 & 10 & 0 & 0 & 18
\end{pmatrix}
\]
The CRS format for this matrix is then specified by the vectors given below:

\[
\begin{align*}
\text{Val} & = (3 \ 4 \ 9 \ 10 \ 18) \\
\text{Col} & = (2 \ 4 \ 0 \ 1 \ 4) \\
\text{Ptr} & = (0 \ 1 \ 1 \ 1 \ 2)
\end{align*}
\]

Notably that by subtracting the consecutive elements in ptr vector, we can acquire the number of nonzero elements in each row. For the last row, the number of nonzero elements is calculated by subtracting the last entry in ptr vector from total nonzeros nz.

### 3.2 System diagram

The basic procedures of the system are memory access and data processing.

From figure 2, memory control module is connected with both two QDRs that contain value and column information respectively, and on-chip block ram ptr_bram which stores pointer information of the sparse matrix and vec_bram which stores the vector that is multiplied by matrix A. After data fetching and signal processing, two operands are sent to the arithmetic modules – multiplier and accumulator. Finally, the result vector is saved into another on-chip block ram.

![Figure 2. Block Diagram of SpMXV Multiplication System](image)

### 3.3 Memory control

Memory control is one of the most crucial parts of the whole system. The memory control module is in charge of asserting memory read enable signal and assigning memory access
address, so that correct data can be acquired by arithmetic logic units in the next stage. In the design, the methodology of pipeline is applied to row level, thus, the throughput is not affected by the latency (10 clock cycles) of QDR data reading while multiplying a row with a vector. As it is shown in figure 3, the core of the memory control module is a finite state machine.

At the beginning, the finite state machine is reset to IDLE, where all the registers are reset to predefined values. After completion of writing initial data to QDRs and BRAMs, a start signal is asserted to trigger the memory control transition from IDLE to READ_PTR. In READ_PTR state, it calculates the difference of current pointer and the next pointer to determine the row length of nonzero elements and increases the pointer location by one. With row length calculated successfully, a read_start signal is asserted to transition to READ_DATA state that reads value and column data from QDRs as its name indicates. In READ_DATA, operands and valid signals are sent to multiplication and accumulator unit. HOLD state is necessary because the design of accumulator requires a long processing time. When the all the multiplication operations are sensed by the system, state will go back to IDLE from HOLD with the prog_end signal asserting.

![State transition diagram of memory controller](image)

![Timing diagram of a normal flow of memory control](image)
As an example, if current row pointer points to a row with 25 nonzeros, the state machine will stay in READ_PTR state for 3 cycles to determine the row length of nonzero elements. Then it will enter READ_DATA state for 26 cycles and move to HOLD state. It will stay in HOLD state until it receives acc_end signal from accumulator, so generally it will last (10 cycles of memory access + 6 cycles of multiplication + \(\lceil \log_2 N \rceil \) * 11 cycles of accumulation) in HOLD. In brief, it is going to stall the pipeline during HOLD state.

\[
\{\text{Control Unit}\}
\]

\[
\text{READ_PTR:}
\]

\[
\text{if next row to read is last row then}
\]

\[
\text{row_len} = n_2 - \text{current Pointer}
\]

\[
\text{else}
\]

\[
\text{row_len} = \text{next Pointer} - \text{current Pointer}
\]

\[
\text{end if}
\]

\[
\text{READ\_DATA:}
\]

\[
\text{while (i < row_len) then}
\]

\[
\text{read val(i), col(i)}
\]

\[
i = i + 2
\]

\[
\text{if read_enable then}
\]

\[
\text{counter} = \text{counter} + 1
\]

\[
\text{end if}
\]

\[
\text{end}
\]

\[
\text{HOLD:}
\]

\[
\text{while (counter < predefined maximum accumulate entries) then}
\]

\[
\text{counter} = \text{counter} + 1
\]

\[
\text{end}
\]

Figure 5. Pseudo code of memory control [1]

In figure 5, it shows how to determine the addresses of QDRs and how the memory control works exactly.

3.4 Multiplier

The multiplier consists of three parts: an interface to memory, an interface to accumulator and a 32-bit floating point multiplier. Since the operand from vector BRAM has to be fetched, which takes one clock cycle, the matrix value operand input, label input and valid input are delayed by one clock cycle. As figure 6 shows, the multiplication unit reads value and vector as operands.
The 32-bit floating point multiplier is generated by Xilinx Core Generator 10.1 with non-default configurations as follows.

<table>
<thead>
<tr>
<th>Architecture Optimizations</th>
<th>High Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Family Optimizations</td>
<td>Max Usage</td>
</tr>
<tr>
<td>Latency and Rate Configuration</td>
<td>Use Maximum Latency</td>
</tr>
</tbody>
</table>

Table 1. Multiplier Configurations

3.5 Accumulator

The accumulator is the other most important module of this implementation. It is a reduction circuit. It accumulates all the intermediate results for a row and outputs the final result. Suppose a row has m nonzeros, thus m inputs are sequentially delivered to the reduction circuit. To sum up these m inputs, a straightforward method is to construct a binary adder tree with m − 1 adders and ⌈lg(m)⌉ tree levels. Since the inputs arrive at the reduction circuit sequentially, we observe that the adders at the same tree level is used in different clock cycles. Therefore, we replace the level 0 adders with a buffer feeding a single adder, replace the level 1 adders with another adder with buffer, and so forth. Thus instead of using m − 1 floating-point adders, we use ⌈lg(m)⌉ floating-point adders and ⌈lg(m)⌉ buffers, as shown in Figure 7. At level l, the adder and buffer is denoted as adder_l and buffer_l (l = 0,...,[lg(m)]−1).

![Reduction circuit with ⌈lg(m)⌉ adders](image)

Figure 7. Reduction circuit with ⌈lg(m)⌉ adders [1]
In each clock cycle, each adder checks if it has enough operands to begin execution. Normally, the adder proceeds if the buffer contains two inputs. However, when \( m \) is not a power of 2, adders at certain levels need to proceed even though their corresponding buffers only contain a single input. Take \( m = 5 \) as an example. There are 5 inputs at level 0, and the 5th input cannot be paired up with any other input. Thus if the 5th input is detected in buffer0, it is added with 0 by adder0 even if buffer0 only contains one input. To identify such cases, we label each input with a number, which indicates how many sub-rows of the same row remain to be accumulated. At level 1, if the label of an input is less than \( 2l + 1 \), it must be the last input on that level. Thus when such an input is in buffer1 and no other input is waiting to pair up with it, adder1 adds the input with 0.

Figure 8 illustrates how we accumulate a row with 5 sub-rows using 3 adders. The small boxes represent the inputs and their labels are shown below them. The inputs labeled as 0 do not really exist; they are shown in the figure to form a binary tree. If an input is added with an input labeled as 0, it is the last input on its level and cannot be paired up with any other input. We see on both level 0 and 1, the last input is labeled as 1 and is added with 0. At level 2, however, although the last input is still labeled as 1, it is paired up with another input in buffer2. This time, these two inputs are summed up, and the final result is yielded.

![Figure 8. Illustrations of addition of 5 entries using 3 adders][1]

In one word, this design consumes less resources with the latency getting longer.

Since there are at most 56181 nonzero elements a row (test matrix 14th) among test matrices, it requires 16 levels of adders to fully accumulate them. Thus, 16 pairs of buffer and adder in series are implemented in the design.

### 3.6 QDR and Block RAM

Among those blocks in the system, there are two blocks that have not been discussed yet, which are QDR and Block RAM. They all come from the ROACH CASPER platform. Block RAM modules provide flexible 36 Kbit true dual-port RAM that are cascadable to form larger memory blocks. Each block RAM can also be configured as two independent 18 Kbit true dual-port RAM blocks, providing memory granularity for designs needing
smaller RAM blocks. In design simulation, QDR_ROACH.v file which is a behavioral model of QDR and bram.v module which is generated by Xilinx core generator are used.

4 Simulation

4.1 Matrix and Vector Generation

In order to generate random sparse matrices and vectors, a stimulus generation MATLAB file is created. In this file, a randomly generated sparse matrix is translated into a compressed row storage (CRS) format. Also, the golden result data for comparison use is generated. Additionally, two dimensional information \( n_z \) (nonzero) and \( n \) (row size) are calculated and passed to the input of the design as known information.

4.2 Testbench

In the testbench, it writes initialized data to QDRs and BRAMs, and then leaves the memory control unit to read QDRs and BRAMs automatically. One special thing to mention is that some zeros are padded to some address of QDR because of the special addressing method of QDR. Figure 9 shows the waveform of zero padding.

As it is shown in figure 8, since two consecutive 36-bit data are concatenate together and stored in one address of QDR, it would have to pad a zero to every row that has odd number of nonzero elements to avoid address mismatch. Otherwise, the first element of next row will be counted as last element of current row.

4.3 Simulation Result

I ran numerous test input set and compared the simulation result with golden result in MATLAB. The Table 2 shows the results.

<table>
<thead>
<tr>
<th>Matrix Dimension n x n</th>
<th>Sparse Density</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 x 300</td>
<td>1%</td>
<td>All pass</td>
</tr>
<tr>
<td>300 x 300</td>
<td>0.1%</td>
<td>All pass</td>
</tr>
<tr>
<td>900 x 900</td>
<td>1%</td>
<td>All pass</td>
</tr>
<tr>
<td>900 x 900</td>
<td>0.1%</td>
<td>All pass</td>
</tr>
<tr>
<td>2000 x 2000</td>
<td>0.1%</td>
<td>All pass</td>
</tr>
<tr>
<td>3000 x 3000</td>
<td>0.01%</td>
<td>All pass</td>
</tr>
<tr>
<td>5000 x 5000</td>
<td>0.01%</td>
<td>All pass</td>
</tr>
<tr>
<td>10000 x 10000</td>
<td>0.01%</td>
<td>Out of MATLAB workspace memory</td>
</tr>
</tbody>
</table>

Table 2. Simulation results
5 FPGA Mapping and Performance

5.1 FPGA Mapping Setup
I used Xilinx 10.1 and Mentor Graphics Modelsim 6.6a development tools in my experiment. Our target device is Xilinx Virtex5 xc5vsx95t which has 14,720 slices and maximum 8784 Kb block RAM blocks.

Firstly, the Simulink model is built up as shown in figure 10.

![Simulink model of system](image)

In the design, the topology of accumulator is independent of the latency of adders. Since the latency of adder ranges from 3 to 11, there is a variable delay unit passing control signals along with processing unit like adders and multiplier. The latency of processing elements used in the design is listed in the table below:

<table>
<thead>
<tr>
<th>Processing Elements</th>
<th>Latency (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>6</td>
</tr>
<tr>
<td>Adder</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3. Latency of processing elements

Actually, the number of stage of adder in the accumulator is configurable according to the maximum number of non-zero elements in one row of a testing matrix. For the synthesis, 16 adders are applied to the accumulator.
After running bee_xps, the system has been successfully mapped to ROACH FPGA platform with the following design summary.

<table>
<thead>
<tr>
<th>Slice Logic Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Registers</td>
</tr>
<tr>
<td>Number of Slice LUTS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slice Logic Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of occupied Slices</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IO Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bonded IOBs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specific Feature Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of BlockRAM/FIFO</td>
</tr>
</tbody>
</table>

Table 4. Design resource utilization summary

The timing closure information is provided below in Table 5.

<table>
<thead>
<tr>
<th>Timing Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum period</td>
</tr>
<tr>
<td>Maximum frequency</td>
</tr>
<tr>
<td>Minimum input arrival time before clock</td>
</tr>
<tr>
<td>Maximum output required time after clock</td>
</tr>
<tr>
<td>Maximum combinational path delay</td>
</tr>
</tbody>
</table>

Table 5. Design Timing Summary

5.2 Performance

After mapping it to ROACH FPGA, I tested the runtime of calculating the enormous sparse matrices from various scientific and engineering fields. Table 6 details the size and overall sparsity structure of each matrix [3]. All of the matrices are publically available online from the University of Florida Sparse Matrix Collection [8].

<table>
<thead>
<tr>
<th>Tag</th>
<th>Test Matrice</th>
<th>Dimension</th>
<th>Nonzeros</th>
<th>Sparse Density</th>
<th>Nonzeros/Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dense</td>
<td>2000 x 2000</td>
<td>4,000,000</td>
<td>100.00000%</td>
<td>2000.00</td>
</tr>
<tr>
<td>2</td>
<td>Protein</td>
<td>36417 x 36417</td>
<td>4,344,765</td>
<td>0.32761%</td>
<td>119.31</td>
</tr>
<tr>
<td>3</td>
<td>FEM/Spheres</td>
<td>83334 x 83334</td>
<td>6,010,480</td>
<td>0.08655%</td>
<td>72.13</td>
</tr>
<tr>
<td>4</td>
<td>FEM/Cantilever</td>
<td>62451 x 62451</td>
<td>4,007,383</td>
<td>0.10275%</td>
<td>64.17</td>
</tr>
<tr>
<td>5</td>
<td>Wind Tunnel</td>
<td>217918 x 217918</td>
<td>11,524,432</td>
<td>0.02427%</td>
<td>52.88</td>
</tr>
<tr>
<td>6</td>
<td>FEM/ Harbor</td>
<td>46835 x 46835</td>
<td>2,374,001</td>
<td>0.10823%</td>
<td>50.69</td>
</tr>
<tr>
<td>7</td>
<td>QCD</td>
<td>49152 x 49152</td>
<td>1,916,928</td>
<td>0.07935%</td>
<td>39.00</td>
</tr>
<tr>
<td>8</td>
<td>FEM/Ship</td>
<td>140874 x 140874</td>
<td>3,568,176</td>
<td>0.01798%</td>
<td>25.33</td>
</tr>
<tr>
<td>9</td>
<td>Economics</td>
<td>206500 x 206500</td>
<td>1,273,389</td>
<td>0.00299%</td>
<td>6.17</td>
</tr>
<tr>
<td>10</td>
<td>Epidemiology</td>
<td>525825 x 525825</td>
<td>2,100,225</td>
<td>0.00076%</td>
<td>3.99</td>
</tr>
<tr>
<td>11</td>
<td>FEM/Accelerator</td>
<td>121192 x 121192</td>
<td>2,624,331</td>
<td>0.01787%</td>
<td>21.65</td>
</tr>
</tbody>
</table>
Table 6. Sizes of testing matrices

We use MFLOPS to measure the sustained performance of the design, as it is the most widely used performance metric for SpMXV. The MFLOPS performance is calculated as

\[
MFLOPS = \frac{\text{total number of floating point operations}}{\text{total time}}
\]

The whole system is running at 100 MHz in one clock domain. Table 7 show the runtime cycles, latency and MFLOPS of all the test cases respectively.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Runtime Cycles</th>
<th>Hardware Latency</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4,400,000</td>
<td>0.0440 s</td>
<td>181.82</td>
</tr>
<tr>
<td>2</td>
<td>10,572,728</td>
<td>0.1057 s</td>
<td>82.19</td>
</tr>
<tr>
<td>3</td>
<td>17,759,498</td>
<td>0.1776 s</td>
<td>67.69</td>
</tr>
<tr>
<td>4</td>
<td>12,537,462</td>
<td>0.1254 s</td>
<td>63.93</td>
</tr>
<tr>
<td>5</td>
<td>33,928,414</td>
<td>0.3393 s</td>
<td>67.93</td>
</tr>
<tr>
<td>6</td>
<td>7,361,576</td>
<td>0.0736 s</td>
<td>64.50</td>
</tr>
<tr>
<td>7</td>
<td>7,667,712</td>
<td>0.0767 s</td>
<td>50.00</td>
</tr>
<tr>
<td>8</td>
<td>16,282,340</td>
<td>0.1628 s</td>
<td>43.83</td>
</tr>
<tr>
<td>9</td>
<td>12,412,922</td>
<td>0.1241 s</td>
<td>20.52</td>
</tr>
<tr>
<td>10</td>
<td>25,239,558</td>
<td>0.2524 s</td>
<td>16.64</td>
</tr>
<tr>
<td>11</td>
<td>12,406,821</td>
<td>0.1241 s</td>
<td>42.30</td>
</tr>
<tr>
<td>12</td>
<td>9,997,896</td>
<td>0.1000 s</td>
<td>19.18</td>
</tr>
<tr>
<td>13</td>
<td>38,702,730</td>
<td>0.3870 s</td>
<td>16.05</td>
</tr>
<tr>
<td>14</td>
<td>18,751,842</td>
<td>0.1875 s</td>
<td>120.30</td>
</tr>
</tbody>
</table>
The histogram of MFLOPS of testing matrix is plotted in figure 11.

![Histogram of MFLOPS](image1.png)

**Figure 11. Throughput of testing matrices**

From figure 11, the MFLOPS of different test matrices vary greatly. However, they are dependent of the nonzeros/row of test matrices which is proportional of sparsity of test matrices. Theoretically, the relationship between efficiency and nonzeros/row is shown in figure 12.

![Efficiency vs. nonzeros/row](image2.png)

**Figure 12. Relationship between efficiency and nonzeros/row**
From Table 7, we can see that the throughputs of test matrix A9, A10, A12 and A13 are rather low compared to other test matrices, because their nonzeros/row are so low that the overheads dominate the actual effective processing time.

From figure 12, it is obvious that the efficiency is linear to the nonzeros/row in every separate interval. For example, in the interval of [2049, 4096], the higher the number of nonzeros/row, the better efficiency and throughput it can attain. However, 2049 nonzeros in a row not necessarily result in higher throughput than 2047, because in the design 2049 belongs [2049, 4096] which takes more execution time, while 2047 belongs to [1025, 2048] which takes less processing time.

5.3 Future Improvement

There are several limitations on the design. One of the shortcomings of the design is that the MFLOPS is not high enough comparing to the effort of an application specific design. The reason is that I intended to keep the control logic simple and clean, so some executions in the design are not fully pipelined and would have to stall to wait. Therefore the area/delay tradeoff is not optimized. That’s the major limitation on the design. Future work can be focused on increasing the throughput by adding more states in the controller finite state machine so that it doesn't have to wait to fetch elements of next row until accumulator finishing execution. If applied, the throughput can be greatly improved for this set of testing matrices.

Another limitation on this design is that it only has one multiplier as processing element which result in longer latency. One solution to this limitation is applying parallel structural technique which uses more multipliers in parallel. In this way, the latency can be shorter but the resource utilization is greater.

6 Conclusion

I have implemented an FPGA-based design for floating-point sparse matrix-vector multiplication. The design utilizes the commonly used CRS format for sparse matrices. The design requires number of total nonzero elements in the sparse matrices and the dimension of the operating matrices. The design is parameterized which is adaptive to the latency of 32-bit floating point adder as required. The design contains memory control unit which fetches data from memory and passes on valid signals along with data, multiplier which does the multiplication of corresponding matrix elements and vector elements and accumulator which utilizes less area but increases the latency. The functionality of this system is verified by numerous sets of test cases under a certain size.
References


