Energy Efficiency in Sparse Matrix Multiplication for different Hardware and Algorithms

Kevin Rohan
Dept. of Electrical and Computer Engineering
Carnegie Mellon University

Wilson Sa
Dept. of Electrical and Computer Engineering
Carnegie Mellon University

***Abstract*— Sparse Matrix-Vector Multiplication (SpMxV) is widely used in modern day applications, like feature extraction, data analytics, numerical methods etc. The throughput obtained in sparse matrix multiplication is lower compared to dense matrix multiplication due to the compression format used to represent these matrices. The paper proposes a method to improve the energy efficiency and throughput obtained by performing hardware based acceleration using FPGA for sparse matrix multiplication. A comparative analysis between the various hardware and algorithms used to compute the product of such matrices is performed. Computation is performed on FPGA and CPU. The parallel nature of the representations of the sparse matrices is exploited by using a FPGA. The look up tables and flip flops available in the FPGA make it highly programmable and parralelizable in nature. The algorithm used is an Outer Product based Sparse Matrix Multiplication. This algorithm is of high memory bandwidth, runs in parallel and reduces the number of memory accesses. This reduces the overhead.**

Keywords—SpMxV, FPGA, CPU, Energy Efficiency, Parallel Computing

I. Introduction

Sparse Matrix arises in many situations in modern day applications. The large number of data obtained in applications like machine learning, feature extraction, data analytics etc. store only the areas of interest and thus resulting in sparse matrices. These applications exploit the property of sparse matrices storing data very sparsely. With an appropriate compression algorithm it reduces the data stored very significantly.

The computation of information from these matrices involves large sparse matrix multiplications. This is due to the fact that compression algorithm has to be used to represent this matrix. The compression algorithm used results in using excessive indices and branches so as to access the data for computation from its compressed. There is a large number of irregularites in memory accessing. There are dedicated kernels which responsible for determining the product.

Compressed matrix representations offer the advantage of reduction in the data which needs to be stored by a system to depict the information it needs to convey. In the project the comparision between multiplication of compressed matrices is performed to determine which is more suitable.

The possibilities of implementation of sparse matrix multiplication is explored in CPU and FPGA. The hardware is used as a co-processor to the existing system and the operation is carried out. The efficiency is computed for two hardware implementations.

The rest of the report is organized as follows, section 2 provides a description of the related work which has already been implemented, section 3 provides details of the technical description of the project, section 4 provides the schedule in which we would like to proceed, milestones and division of work between each other.

II. RELATED WORK

[1] A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs.

This paper provides the details related to the various forms of sparse matrix representations. The basic hardware we are implementing in our project is derived from this paper where a FPGA is used as a co-processor to the existing system and a comparative analysis is provided between different hardware used.

[2] OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator.

This paper proposes an algorithm which exploits parallelism to improve the performance of sparse matrix multiplication, where an outer product is used instead of an inner product based multiplication.

III. TECHNICAL DESCRIPTION

A. *Sparse Matrix Representations:*

Sparse Matrices are represented in three forms as shown in figure 1:

Coordinate list (COO)

COO stores a list of (row, column, value) tuples.

Compressed sparse row (CSR, CRS or Yale format)

The *compressed sparse row* (CSR) or *compressed row storage* (CRS) format represents a matrix by three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices.

Compressed sparse column (CSC or CCS)

CSC is similar to CSR except that values are read first by column, a row index is stored for each value, and column pointers are stored

(a) $$A = \begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 0 & 9 & 0 & 6 \end{bmatrix}$$

(b) $$COO \begin{cases} data = [1 \quad 4 \quad 2 \quad 3 \quad 5 \quad 7 \quad 8 \quad 9 \quad 6] \\ row = [0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 3] \\ col = [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 3 \quad 4 \quad 2 \quad 4] \end{cases}$$

(c) $$CSR \begin{cases} data = [1 \quad 4 \quad 2 \quad 3 \quad 5 \quad 7 \quad 8 \quad 9 \quad 6] \\ ptr = [0 \quad 2 \quad 4 \quad 7 \quad 9] \\ col = [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 3 \quad 4 \quad 2 \quad 4] \end{cases}$$

(d) $$CSC \begin{cases} data = [1 \quad 5 \quad 4 \quad 2 \quad 3 \quad 9 \quad 7 \quad 8 \quad 6] \\ row = [0 \quad 2 \quad 0 \quad 1 \quad 1 \quad 3 \quad 2 \quad 2 \quad 3] \\ ptr = [0 \quad 2 \quad 4 \quad 6 \quad 7 \quad 9] \end{cases}$$

Figure 1. Sparse Matrix Representations

Compressed sparse matricies (CSR and CSC) are considered in this project as it occupies lesser memory and it is more suitable for modern applications.

The multiplication of a CSR based multiplication [1] with a column vector follows the flow represented in figure 4.

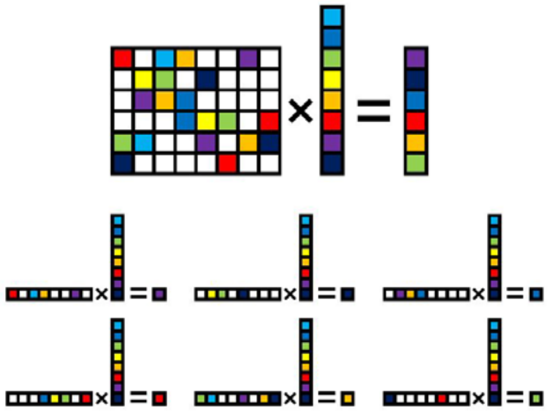


Figure 2. CSR sparse matrix multiplied with a column vector

Each iteration gives us the result of one row of the output matrix. Between iterations the value of the next row pointer must be selected.

The multiplication with of a CSC based sparse matrix [1] with a column vector follows the algorithm represented in figure 3.

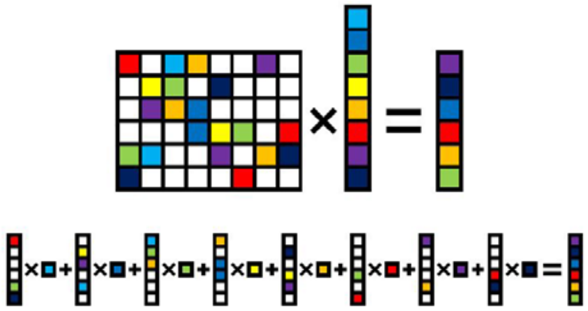


Figure 3. CSC sparse matrix multiplied with a column vector

Each iteration the a partial value of the result row is obtained. The value of the result matrix accessed keeps changing each iteration.

B. *Hardware Optimizations:*

Machine learning, data-analytics, Big Data deals with a huge amount of data which it needs to process to generate the desired results. The solution to obtaining the results of these applications is to make the code as parallelizable as possible.

The CPU is designed to deal general purpose applications for computing and the architecture of the CPU is not highly parallelizable. Moreover, generally there will be other applications which will be running on a CPU which slows down the process of such applications.

The architecture of FPGAs as shown in figure 4, consists of a large number of Configurable Logic Blocks which consists of Look up tables and Flip Flops. It also consists of programmable interconnect wires which makes the architecture highly flexible to program and parallizable.

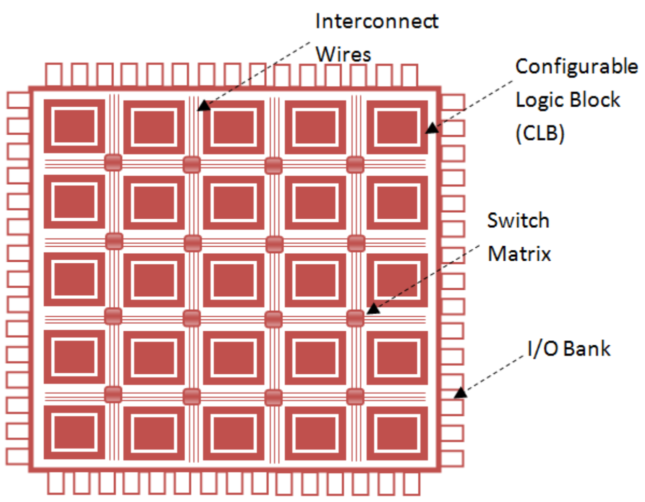


Figure 4. : Architecture of a FPGA

The above stated feeatures make the FPGA a suitable hardware accelerator to perform intensive tasks such as sparse matrix multiplication.

C. *Software Optimizations:*

To optimize Sparse Matrix Multiplication (SMM) operation, we will carry out an analysis on the common pitfalls of various algorithms on different pieces or hardware architectures. One major bottleneck we will take a critical look at is redundant memory accesses which occur in traditional Sparse Matrix Multiplication algorithms. If we are able to identity redundant memory access as the major bottleneck, we can test different commonly used algorithms to identify the

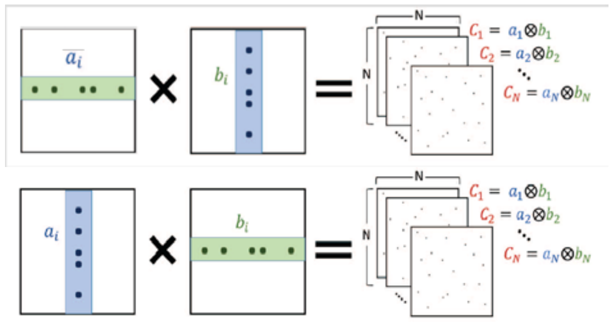


Figure 5. Top row shows an outer, bottom row illustrates an inner matrix multiplication.

operations which will most greatly reduce this memory access. We may attempt different permutations of algorithms which may exploit different combinations between multiply and merge subset of operations.

To get started, we will use the gem5 simulation tool to test various hardware architectures through a diverse set of metric multiplication operations using SuiteSparse collection. We also plan on using the Stanford Network Analysis Project as a tool measure throughput and performance of the operations.

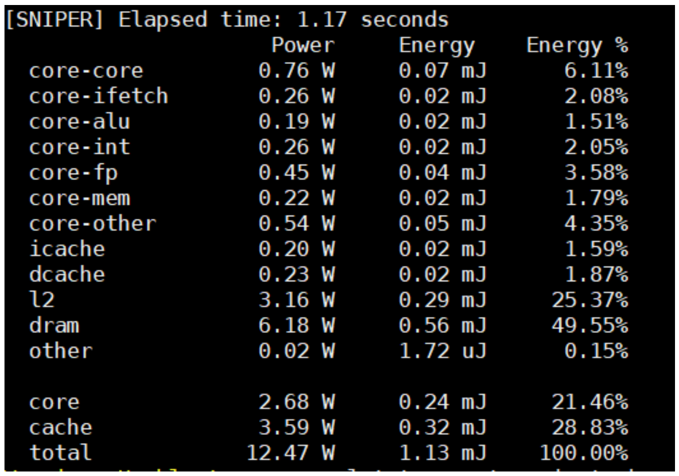
Using the SuiteSparse collection, we plan to implement the following four major algorithms for test: Intel Math Kernel Library (MKL), OuterSPACE [2],cuSPARSE [3], and CUSP [4]. The algorithms will mainly differ by methodologies of carrying out matrix multiplication. Traditionally, matrix-to-matrix multiplication is performed using an inner dot product approach. In an inner matrix multiplication, the first matrix ‘A’ is decomposed to rows, multiplied by the second matrix ‘B’ decomposed to columns. The algorithm most closely resembling this is MKL approach. Conversely, in an outer dot product approach, the first matrix ‘A’ is decomposed to columns, multiplied by the second matrix ‘B’ decomposed to rows. The OuterSPACE algorithm most closely resembles this approach. Further, once the matrices are decomposed by column and row, the matrices are *multiply* and *merge* operations to combine into a final result. These differences is where the cuSPARSE and CUSP algorithms differ from the others.

The OuterSPACE paper concludes that the outer dot product approach has an improvement of throughput between 7.9x and 14.0x on comparable architectures, while keeping power budgeted and area constrained. We plan on carrying a similar comparison, as well as analysis of these software optimizations in connection with the parallelized hardware architecture optimization efforts above.

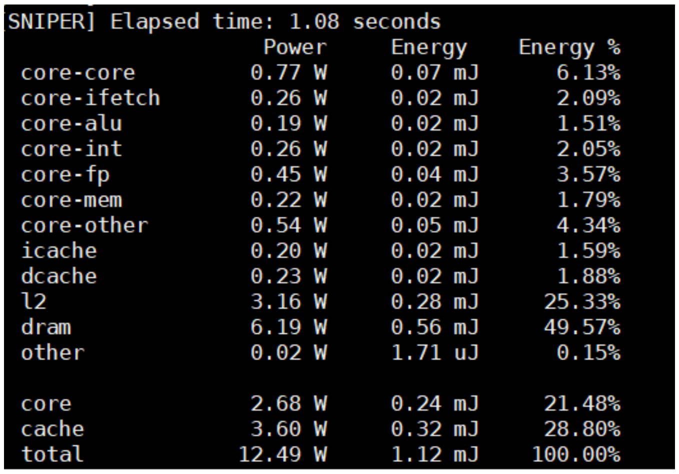
IV.RESULTS AND ANALYSIS

A. *CSR vs CSC for Small Sparse Matrices*

The column vector multiplication with CSR and CSC based respresentations were performed. The power analysis was performed on sniper-sim. The processor used was Intel Xeon X5550. The results of the CSR based multiplication obtained is in figure 6. The results of the CSC based multiplication obtained is in figure 7.


```
[SNIPER] Elapsed time: 1.17 seconds
                       Power      Energy     Energy %
  core-core           0.76 W    0.07 mJ       6.11%
  core-ifetch         0.26 W    0.02 mJ       2.08%
  core-alu            0.19 W    0.02 mJ       1.51%
  core-int            0.26 W    0.02 mJ       2.05%
  core-fp             0.45 W    0.04 mJ       3.58%
  core-mem            0.22 W    0.02 mJ       1.79%
  core-other          0.54 W    0.05 mJ       4.35%
  icache              0.20 W    0.02 mJ       1.59%
  dcache              0.23 W    0.02 mJ       1.87%
  l2                  3.16 W    0.29 mJ      25.37%
  dram                6.18 W    0.56 mJ      49.55%
  other               0.02 W    1.72 uJ       0.15%

  core                2.68 W    0.24 mJ      21.46%
  cache               3.59 W    0.32 mJ      28.83%
  total              12.47 W    1.13 mJ     100.00%
```
Figure 6: Sniper-sim simulation of CSR based multiplication.

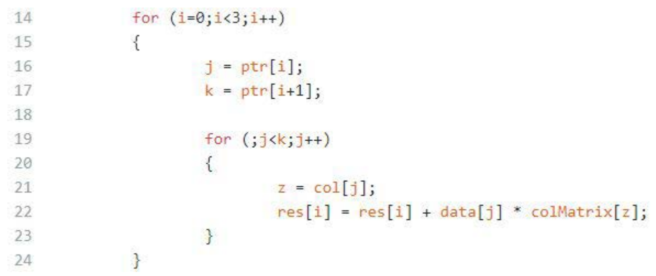


```
SNIPER] Elapsed time: 1.08 seconds
                    Power      Energy     Energy %
  core-core         0.77 W     0.07 mJ      6.13%
  core-ifetch       0.26 W     0.02 mJ      2.09%
  core-alu          0.19 W     0.02 mJ      1.51%
  core-int          0.26 W     0.02 mJ      2.05%
  core-fp           0.45 W     0.04 mJ      3.57%
  core-mem          0.22 W     0.02 mJ      1.79%
  core-other        0.54 W     0.05 mJ      4.34%
  icache            0.20 W     0.02 mJ      1.59%
  dcache            0.23 W     0.02 mJ      1.88%
  l2                3.16 W     0.28 mJ     25.33%
  dram              6.19 W     0.56 mJ     49.57%
  other             0.02 W     1.71 uJ      0.15%

  core              2.68 W     0.24 mJ     21.48%
  cache             3.60 W     0.32 mJ     28.80%
  total            12.49 W     1.12 mJ    100.00%
```

Figure 7. Sniper-sim simulation of CSC based multiplication.

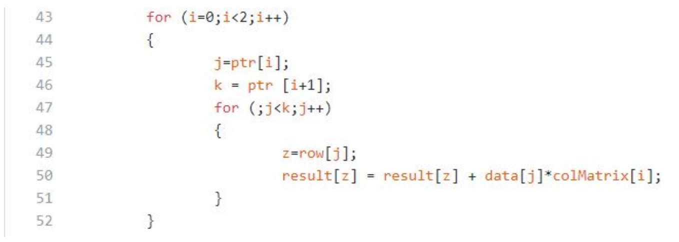
There is only a reduction of 0.02W in CSR based implementation whereas the time taken is larger for CSR which leads to higher amount of energy consumption of 0.01mJ. The values obtained are very less significant due to the fact the test matrices chosen were very small. Moreover both have a very similar based implementation as seen in the code snippets in figure 8 and figure 9.



```
    for (i=0;i<3;i++)
    {
        j = ptr[i];
        k = ptr[i+1];

        for (;j<k;j++)
        {
            z = col[j];
            res[i] = res[i] + data[j] * colMatrix[z];
        }
    }
```

Figure 8. CSR Sparse Matrix Multiplication Code snippet



```
    for (i=0;i<2;i++)
    {
        j=ptr[i];
        k = ptr [i+1];
        for (;j<k;j++)
        {
            z=row[j];
            result[z] = result[z] + data[j]*colMatrix[i];
        }
    }
```

Figure 9. CSC Sparse Matrix Multiplication Code Snipper

The code for both of these implementations look very identical to each other. There is a subtle difference in CSR based implementation there is a double indexing of *“colMatrix[z]”* which is later followed by multiplication. Both are intensive tasks thus taking more time. In CSC based sparse matrix multiplication there is double indexing of *“result[z]”* which gets updated on every iteration thus making to slightly less cache friendly but not significantly. The time taken is lower in this case due to the fact that the multiplication being more intensive.

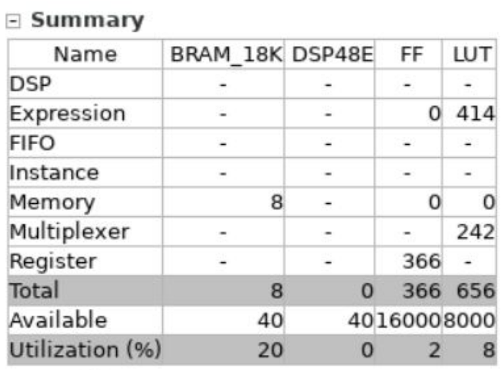
Thus, the values obtained are nearly the same due to the fact that their representations follow similar approach only thing is CSR is row-wise approach vs CSC is column-wise. The huge similarity leads to the fact the algorithm to perform multiplication is also similar. The subtle difference in power consumed is due to the fact that of the reasons stated above.

B. *FPGA based Results*

The C code was expanded to accommodate for larger matrices. The test matrix was obtained from Suite Sparse [5]. The operations were performed on 292x292 size matrix. The C code was syntesised using Vivado HLS using Artix 7 as the FPGA board as the reference.

B.1: Utilization based results:

The utiliation of the FPGA resources is shown in figure 10:



Summary

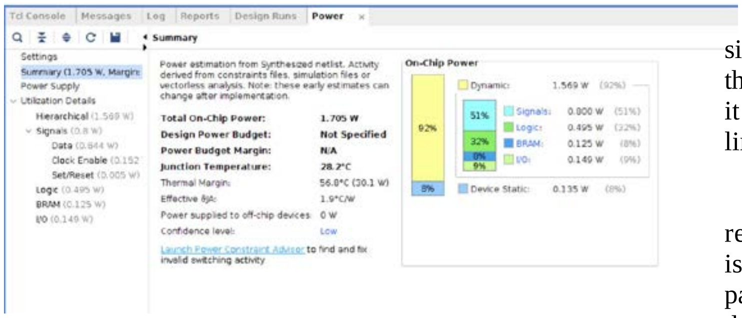
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	414
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	8	-	0	0
Multiplexer	-	-	-	242
Register	-	-	366	-
Total	8	0	366	656
Available	40	40	16000	8000
Utilization (%)	20	0	2	8

Figure 10. Utilization of the FPGA resources

The multiplexer and expression blocks are utilized for decision making and computing. The main reason there exists so many look up tables is for expression is due to the large number of addition operations which are performed on the FPGA. The multiplexers exist due to the indexing and branch statements which are part of the code. The variables which are needed for operation is stored in registers. There are a lot of intermediate results too which need to be stored for which the amount of registers is high in number.

B.2: Power Analysis

The amount of power consumed on the FPGA is shown in figure 11.



Tcl Console | Messages | Log | Reports | Design Runs | Power

Summary

Settings
Summary (1.705 W, Margin:
Power Supply
Utilization Details
Hierarchical (1.569 W)
Signals (0.8 W)
Data (0.644 W)
Clock Enable (0.152
Set/Reset (0.005 W)
Logic (0.495 W)
BRAM (0.125 W)
I/O (0.149 W)

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**1.705 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**28.2°C**
Thermal Margin:	56.8°C (30.1 W)
Effective θJA:	1.9°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	1.569 W	(92%)
Signals:	0.800 W	(51%)
Logic:	0.495 W	(32%)
BRAM:	0.125 W	(8%)
I/O:	0.149 W	(9%)
Device Static:	0.135 W	(8%)

Figure 11. Power analysis on FPGA board

The total power consumed on an FPGA is 1.7W. The majority of the contribution towards power consumption is due to the signals. That is the variables involved in the operation of the signal. They account for 51% of the power consumed. This is due to the fact there are a large number of declarations as it is a 292x292 matrix. The second most contributor to power consumption to power consumption is Logic flow which is 32% this is due to the fact that there is a large number of indexing which is taking place, along with branches. BRAM and I/O power consumed is the least this is due to the fact the amount of memory accessed is less as most of the variables are pre declared. The I/O contribution is also very less as most of the variables are pre-declared in the code. The variables were pre-declared as even in the CPU the variables should be pre-declared and the processing takes place based on the pre-declared variables and it does not receive inputs externally.

C. *Comparision of Power with that of CPU:*

The comparision of power on CPU is as shown in figure 12:

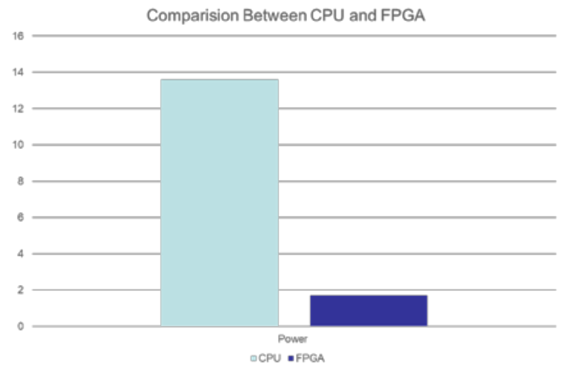


Figure 12. Comparision of power consumption between CPU (intel core i7) and FPGA (Artix 7)

The power was analysed on the laptop which uses intel i7 processor. The power was calculated using mathematical estimation:

$$\text{Power} = (V_{dd})^2 * f$$

$$\text{Power} = (V_{dd})^2 / \text{time}$$

The time taken to execute the C program was displayed after the running of the program on Dev C++ editor.

The simulation was not possible to perform on sniper sim which is present on the enyac machine due to the fact that the system was throwing a segmentation fault for the C code as it utilized a large amount of resources more than what is the limit available for use.

The implementation of the algorithm on the FPGA results in 87.5% reduction in power of the implementation. This is due to the fact that the FPGA architecture is more parallelizable as well as it can be programmed to process a dedicated set of instructions. A CPU is meant for more general purpose applications.

V. CONCLUSION

Sparse matrix multiplication was implemented on a 292x292 Matrix which has 292 non zero elements. The multiplication was initially performed using both CSR and CSC compressed sparse matrix multiplication both of them gave lower energy consumption due to better locality. The values obtained were nearly identical this is due to the fact both have very similar compression method but CSR is row-major and CSC is column-major. However, a slightly smaller value of energy consumption is obtained for CSC multiplication due to lesser number of indexing along with multiplication which reduces the time taken for multiplication.
The C code for CSC based sparse matrix multiplication was synthesised using Vivado HLS which generated the VHDL files. This was used to simulate using Vivado keeping Artix 7 as the target FPGA. There was a 87.5% reduction in power compared to that of a CPU due to the fact that the nature of implementation in FPGA is more parallel compared to that of the conventional CPU.

VI. SCHEDULE FOLLOWED

October 16: Get the conventional Sparse Matrix Matrix Multiplication Implementation running. (Rohan)

October 18: Compare analysis for CSR and CSC representations (Sa/Rohan).

October 23: Presentation – II (Sa/Rohan)

October 26: Complete gem5 analysis of different algorithms (Sa)

This was not been able to implement as the setup for gem5 along with GPU was difficult due to the time constraints.

November 2: Complete hardware comparison tests on FPGA (Rohan).

Original plan was to integrate it using virtex 5 FPGA. There was liscensing issues with virtex 5 FPGA with Vivado software in the ECE machines.

November 15: Complete integration between algorithm and hardware to begin tests (Sa)

November 30: Complete comparisons with algorithms across different architectures (Rohan)

December 4: Paper Presentations (Sa/Rohan)

December 6: Project Posters and Demo (Sa/Rohan)

December 14: Complete report (Sa/Rohan)

VII. PORJECT URL

https://github.com/kevinRohan8/18743_sparse_matrix

VIII. REFERENCES

[1] Dorrance, R., Ren, F. and Marković, D., 2014, February. A Scalable Sparse Matrix-vector Multiplication Kernel for Energy-efficient Sparse-blas on FPGAs. In Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays (pp. 161-170).

[2] Pal, S., Beaumont, J., Park, D.H., Amarnath, A., Feng, S., Chakrabarti, C., Kim, H.S., Blaauw, D., Mudge, T. and Dreslinski, R., 2018, February. OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 724-736).

[3] (2014) cuSPARSE Library. https://developer.nvidia.com/cuSPARSE.

[4] N. Bell, S. Dalton, and L. N. Olson, “Exposing fine-grained parallelism in algebraic multigrid methods,” SIAM J. Scientific Comput., 2011.

[5] ash292 sparse matrix from: https://sparse.tamu.edu/