King Abdullah University for Science and Technology Academy
International Centre for Theoretical Physics
Multidisciplinary Lab at ICTP

Sparse Matrix-Vector Multiplication (SpMV) on FPGA using High-Level Synthesis

Ghala I. Buarish
Mentors: Maynor Ballina and Romina Molina

Email: ghalaibuarish@gmail.com ,maynor.ballina@gmail.com,
mromy00@gmail.com

***Abstract*—Implementation and optimization of Sparse Matrix-Vector Multiplication (SpMV) on FPGA using High-Level Synthesis (HLS). The project benchmarks different compression formats, evaluates their performance in software and hardware environments, and develops an efficient FPGA-based accelerator targeting high-performance computing applications such as scientific simulations, machine learning, and graph analytics.**

I. Research Contributions

This project was conducted individually by Ghala I. Buarish as part of the KAUST Academy Summer Research Program at the International Centre for Theoretical Physics (ICTP). The work involved the complete research cycle, from problem exploration and software prototyping to hardware deployment and documentation.

II. Objectives

The primary objective of this project was to investigate the implementation and optimization of Sparse Matrix-Vector Multiplication (SpMV) on FPGA platforms using High-Level Synthesis (HLS). The goals included:

1. Benchmarking common sparse matrix compression formats (CSR, CSC, COO, ELL, SELL) in software.
2. Analyzing performance across different environments, including local execution and the ICTP HyperFPGA cluster.
3. Translating the most suitable format(s) into FPGA hardware through HLS.
4. Designing and testing a complete FPGA system using Vivado and Vitis HLS.
5. Documenting the entire process in the form of a tutorial and flowchart to guide future FPGA developers.

III. Progress

The project progressed through distinct stages, each with measurable milestones. Initial weeks were dedicated to orientation, training on Vivado and Vitis HLS, and becoming familiar with the HyperFPGA cluster environment. Subsequent stages involved implementing and profiling five SpMV compression formats in Python, measuring execution time and memory usage. Midway through the project, parallel computing experiments were conducted using 1–4 CPU cores to understand scalability.

Based on these results, CSR was selected as the primary candidate for FPGA implementation. The project then advanced to hardware-level development: implementing SpMV in HLS, synthesizing it into IP cores, and integrating them into Vivado block designs. By the end of the internship, CSR, COO, and ELL implementations had been synthesized, tested on matrices up to 512×512, and deployed on FPGA nodes in the cluster. A full step-by-step tutorial on deploying SpMV on FPGA was also written.

a. Compressed Sparse Row (CSR)

Represents a matrix using three arrays:

1- values[ ]: Stores non-zero values of the matrix.
2- columnIndex[ ]: Stores column index of the non-zero elements.
3- rowPtr[ ]: Stores the starting indices in values[ ] for each row. Always starts from 0. Its last value is equal to the number of non-zero elements in the matrix.

b. Compressed Sparse Column (CSC)

Similar to CSR, represents a matrix by three arrays:

1- values[ ]: Non-zero elements in the matrix in column-major order.
2- rowInd[ ]: Stores the row index of the non-zero values.
3- columnOffsets[ ]: Stores the starting indices in values[ ] for each column. Always starts from 0. The last value is equal to the number of non-zero elements in the matrix.

c. Coordinate (COO) Formate

Represented by three arrays:

1- values[ ]: Non-zero elements in the matrix in row-major ordering.
2- columnInd[ ]: Holds the column indices of the non-zero elements of the matrix.
3- rowInd[ ]: Holds the row indices of the non-zero elements of the matrix.

d. ELLPACK (ELL) Format

Creates two matrices from the original sparse matrix: value matrix and column index. The storage overhead of ELLPACK is determined by the longest row, which is the maximum number of nonzero elements in one row of the matrix.

1- Value Matrix: Stores the values of the non-zero elements of the sparse matrix.
2- Column Index: Stores the column indices of each non-zero element from the original matrix.

e. Sliced ELLPACK (SELL)

Instead of padding all rows in the whole matrix to match the global longest row like what's done in ELL, SELL splits the original matrix into blocks of rows (number defined by the user), each group is treated separately for better memory usage and faster execution time. Elements in the slices are stored column by column. The matrix is represented by three arrays:

1- values[ ]: Non-zero values and padding, stored in column-major order.
2- columnInd[ ]: Column index of each non-zero value in the matrix, or -1 for padding.
3- sliceOffsets[ ]: Starts from 0, holds where each slice starts in the columnInd[ ] array.

All compression formats were first executed in Python on a local compiler and tested for time and memory usage, 4x4 and 10x10 matrices were used. Then they were executed on an FPGA through a server and tested for time and memory usage as well, 10x10, 100x100, and 1000x1000 were used. After that, I experimented with using a different number of CPU cores (1 CPU core is the default) and compared the results to see if SpMV is more efficient when parallel programming is used. Comparative tables and graphs were created. Accordingly, the most efficient compression format was chosen for the implementation of SpMV on the HyperFPGA cluster.

Using HLS, CSR was implemented with different directives (pipeline, unroll, array partition). The HLS implementation was optimized for time and resources usage. The final HLS implementation produced the IP which then was integrated into a VIvado block design. The block design was implemented on a Zynq UltraScale+MPSoC platform. The design combines SpMV IP , Zynq processing system, and the programmable logic. I started the process of debugging server-side output and looking for what is causing inconsistencies, referring back to the block design and the HLS implementation and taking the hardware limitations in consideration.

IV. Methodology and Approach

The methodology combined software experimentation, hardware synthesis, and iterative debugging:

Software benchmarking: Python implementations of CSR, CSC, COO, ELL, and SELL were developed. Execution time and memory usage were profiled across matrices of increasing size, from 4×4 to 1000×1000.

Parallel computing analysis: The same algorithms were executed on the ICTP cluster using 1–4 CPU cores. This revealed how SpMV behaves under different parallel workloads.

High-Level Synthesis (HLS): The best-performing formats were converted to C and synthesized with Vitis HLS. Optimization directives such as pipeline, unroll, and array partition were applied selectively to improve timing and resource usage.

FPGA integration: The synthesized SpMV IPs were incorporated into Vivado block designs with a Zynq Processing System (PS), Comblock communication modules, AXI interconnects, and reset/clock modules. Bitstreams were generated and deployed on HyperFPGA nodes, with input/output tested via Jupyter notebooks.

Documentation: A full tutorial, accompanied by flowcharts and block diagrams, was created to serve as a practical deployment guide.

Figure 1 and 2 shows the overall deployment flowchart used to guide the FPGA development process, from format selection to final implementation.

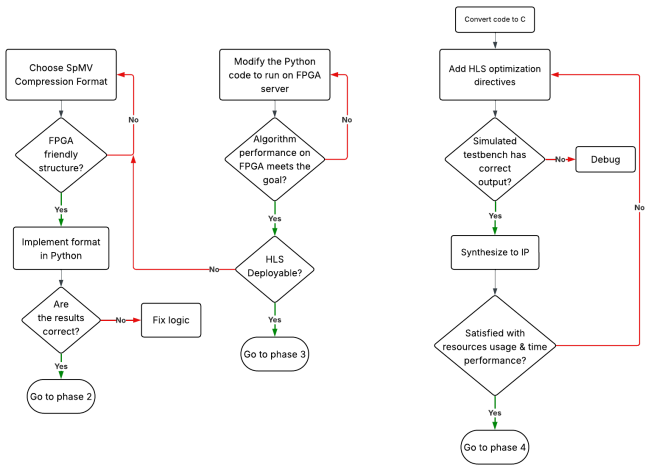


Figure 1. Deployment Flowchart for SpMV on FPGA (Phase 1-3).

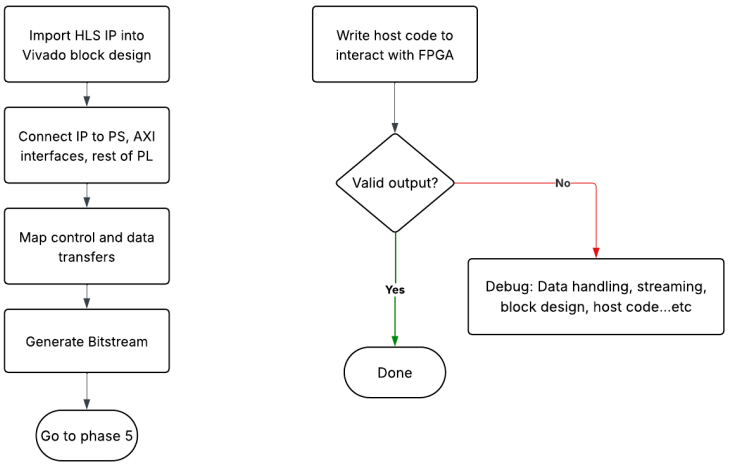


Figure 2. Deployment Flowchart for SpMV on FPGA (Phase 4-5).

V. Results and Findings

The results are divided into three categories: software benchmarks, parallel CPU analysis, and FPGA hardware implementation.

a. Software Benchmarks

All compression formats were tested in Python and on the cluster. Results showed that COO was fastest for very small matrices, while CSR scaled most efficiently as matrix size increased. ELL and SELL consumed more memory due to padding, making them less suitable for FPGAs. CSC behaved similarly to CSR but did not provide advantages.

Figure 3 illustrates the cluster performance results, showing execution time and memory usage for CSR, COO, ELL, and SELL.

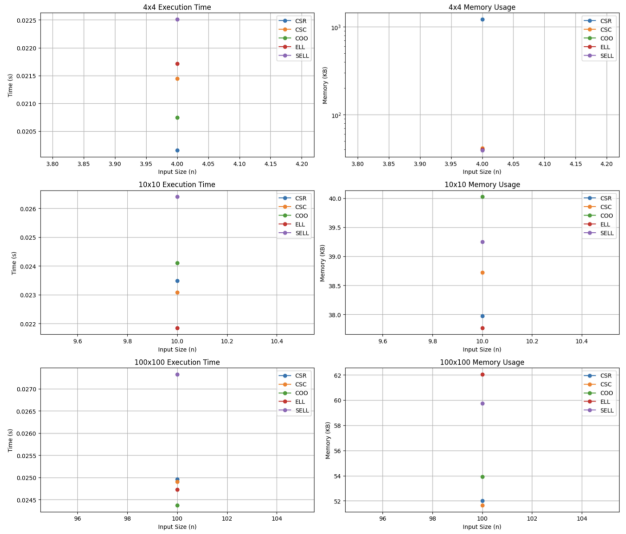
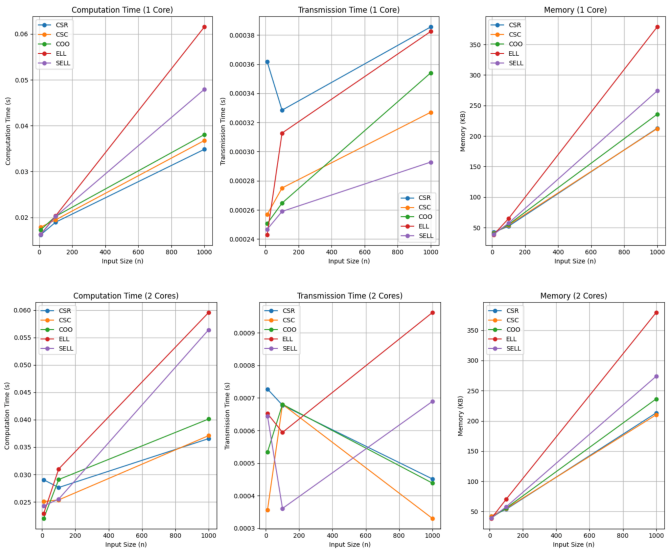


Figure 3. Cluster execution time and memory usage for different SpMV formats (CSR, COO, ELL, SELL).

b. Parallel CPU Analysis

SpMV was executed with 1–4 CPU cores. Contrary to expectations, adding cores often increased overhead due to synchronization and data transmission. For large matrices (1000×1000), CSR and CSC performed best.

Figure 4 presents the performance of the algorithms under different core counts. It highlights how scaling up cores does not always reduce execution time.



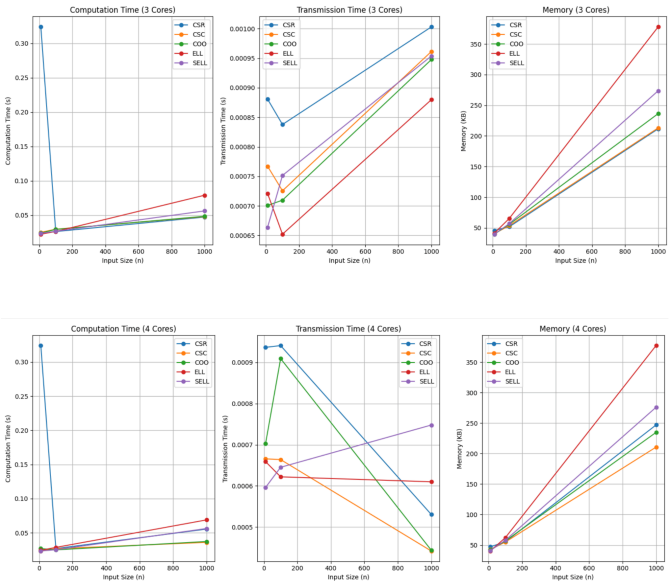


Figure 4. Multi-core scaling results (1–4 CPU cores) for SpMV execution.

c. FPGA Hardware Implementation

The final stage was HLS synthesis and FPGA deployment. CSR consistently outperformed other formats in terms of resource usage and computation time. COO was simple but inefficient at scale, while ELL consumed excessive resources.

Figure 5 shows the FPGA block design, illustrating how the SpMV IP connects with the Zynq PS, Comblock, and AXI interconnects. This design enabled real-time communication between the processing system and programmable logic.

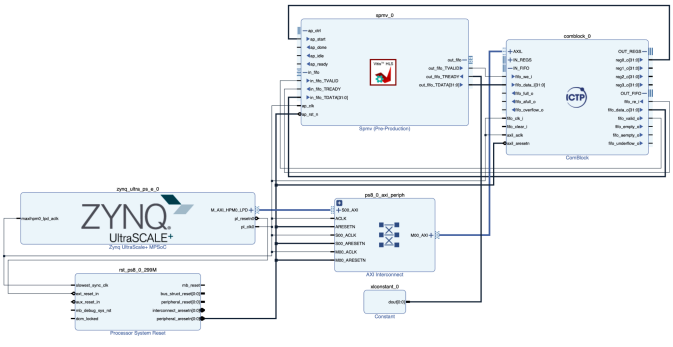


Figure 5. Vivado block design of the FPGA SpMV system (Zynq PS, SpMV IP, Comblock, AXI interconnects).

Figure 6 summarizes shows the handshake mechanism used.

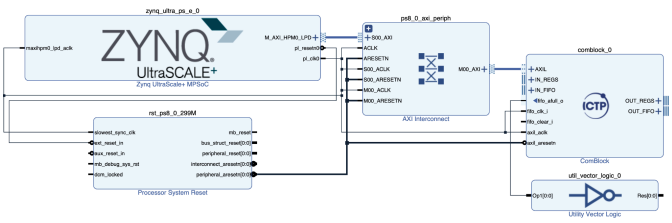
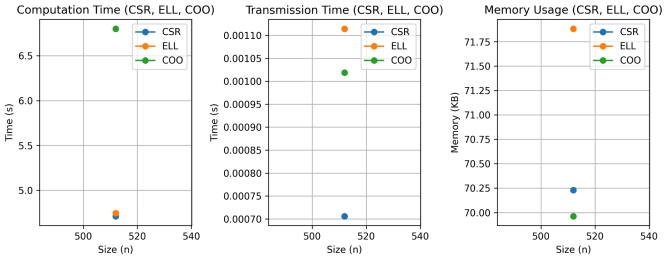


Figure 6. Handshake mechanism

d. Distributed FPGA Computing

Finally, SpMV was executed in a distributed fashion across multiple FPGA nodes in the HyperFPGA cluster. This demonstrated how CSR, COO, and ELL can be executed simultaneously on different nodes to reduce runtime.

Figure 7 show performance comparisons of distributed SpMV. These figures highlight the scalability challenges and opportunities when deploying across multiple FPGAs.



	Algorithm	Input	Computation Time (s)	Memory (KB)	Transmission Time (s)
0	CSR	512	4.710890	70.229492	0.000706
1	ELL	512	4.744235	71.881836	0.001115
2	COO	512	6.800936	69.962891	0.001019

Figure 7. Distributed SpMV execution across multiple FPGA nodes).

VI. Challenges

The project encountered several technical obstacles. The most significant was a race condition in streaming designs: the FPGA computation sometimes began before all input data had been loaded, causing alternating incorrect outputs. This was resolved by removing the directive #pragma HLS PIPELINE off, allowing FIFOs to use the TREADY/TVALID handshake protocol properly.

Other challenges included dependency violations when applying unroll and pipeline directives on loops with variable trip counts, BRAM limitations

when scaling ELL to larger matrices, and discrepancies in cluster vs. local execution times due to transmission overhead. Each obstacle required iterative debugging and careful adjustment of HLS directives and block design connections.

VII. Next Steps and Future Work

Future work should focus on scaling the design to even larger matrices by introducing flow-control mechanisms such as FIFO "almost full" signals. Distributed FPGA computing should also be explored to accelerate SpMV across multiple nodes. Additional compression formats, such as hybrid CSR/ELL approaches, could be investigated. Finally, the accelerator could be tested in real-world applications such as graph neural networks or scientific simulations.

VIII. Resources and Tools

a. Software:

1- Vitis HLS 2022
2- Vivado
3- Jupyter Notebooks
4- Git and PyCharm
5- Python, C/C++

b. Hardware Platforms:

1- Zynq UltraScale+ MPSoC: A heterogeneous SoC with ARM processing system and programmable logic fabric.
2- HyperFPGA Cluster: A distributed platform provided by the lab I am working with; includes 16 nodes with MPSoC-FPGA that can be used to test and validate the block design and SpMV computation at scale.

c. Python Libraries:

NumPy, Pandas, Seaborn, Matplotlib for numerical operations, performance visualization, and data profiling.

d. Datasets and Examples

1- Custom-generated sparse matrices for compression testing.
2- Sample matrices (e.g., 4x4, 10x10, 100x100, 1000x1000) for benchmarking.

e. References and Research

1- Parallel Programming for FPGAs* (Kastner et al.)
2- Efficient sparse matrix-vector multiplication on FPGA (Cretì, 2023)
3- NVIDIA cuSPARSE documentation (SELL format)
4- Internal documentation and tutorials from ICTP & HyperFPGA project

IX. References

Kastner, R., Matai, J., & Neuendorffer, S. (2018). *Parallel Programming for FPGAs.* Morgan & Claypool.

Cretì, A. (2023). *Efficient Sparse Matrix-Vector Multiplication on FPGA: A Comparative Study of Memory-Aware Formats.* Master's Thesis, Politecnico di Torino.

NVIDIA. (2023). *cuSPARSE Library: Storage Formats — Sliced Ellpack (SELL).* CUDA Toolkit Documentation v12.2.0.

AMD/Xilinx. (2022). *Vitis HLS User Guide (UG1399).*

AMD/Xilinx. (2022). *Vivado Design Suite User Guide.*

ICTP MLab. (2025). *HyperFPGA Cluster Documentation.* Internal documentation, International Centre for Theoretical Physics, Trieste.