Hardware Accelerator for Sparse Matrix Vector Multiplication

Jing-Teng Hwang
Dept. of Electrical Engineering,
National Taiwan University

Yuan-Jun Luo
Dept. of Electrical Engineering,
National Taiwan University

Chun-Ting Chen
Dept. of Electrical Engineering,
National Taiwan University

***Abstract*—As machine learning (ML) goes viral these days, demands for efficient hardware have surged to an incredible extent. Since the computation in ML encompasses a large amount of matrix vector multiplication, accelerating this process can bring a massive speedup to the overall execution time. In previous studies, sparse matrices can cause inefficiency in matrix vector multiplication, and some works has been done to overcome the bottleneck. In this paper, we design and tape out an Application Specific Integrated Circuit (ASIC) for accelerating sparse matrix vector multiplication (SMVM). In the simulation result, the chip achieves a 70x speedup compared to the software counterpart.**

***Index Terms*—ASIC, SMVM**

I. Introduction

If 10% to 90% of the entries in a matrix are zeros, we call the matrix a sparse matrix. Since the multiplication of zeros is trivial, storing a lot of zeros in the memory is actually a waste of resources. Therefore, many previous works focus on designing techniques that store sparse matrices with less memory and avoid doing zero multiplication (zero multiplied by other numbers) to boost computing efficiency.

In this work, we design and tape-out an ASIC that functions as an accelerator for SMVM. We make the following contributions:

- We revise the SMVM architecture proposed by [1] to make the chip fit I/O limitations.
- We optimize some functional blocks, such as multipliers, adders, and IPV reducer to make the chip operate in higher clock rate.
- The ASIC achieves a 70x speedup compared to its Python counterpart.

II. Approach

A. Modified Compressed Sparse Row (CSR)

Among various storage formats of sparse matrices, CSR is the one widely used. The traditional CSR turns a matrix into three arrays. The first array, called *value array*, stores the values of all non-zero entries. The second array, called *column index array*, has the same length as the first array, storing the column index of every non-zero entry. Different from the previous two arrays, the third array, called *row pointer*, stores the index of the row-changing entry in the column array. With these arrays, all information of non-zero entries is recorded. [1] revises CSR format by replacing row pointer by input pattern vector (IPV), which has the same length as the value array and the index array. IPV consists of 0 and 1, with 1 indicating that the entry is the last non-zero entry of the row. Detailed representations are shown in fig. 1.

According to previous studies, the IPV storage format will consume more memory than the normal matrix representation if more than one-third of the entries are non-zero. In our work, however, we choose IPV storage format rather than normal matrix representation due to limited I/O pins. Since IPV only needs 1 bit, consuming only 1 I/O pin, we can reserve more I/O pins for other I/O signals.

Normal matrix representation

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

IPV representation

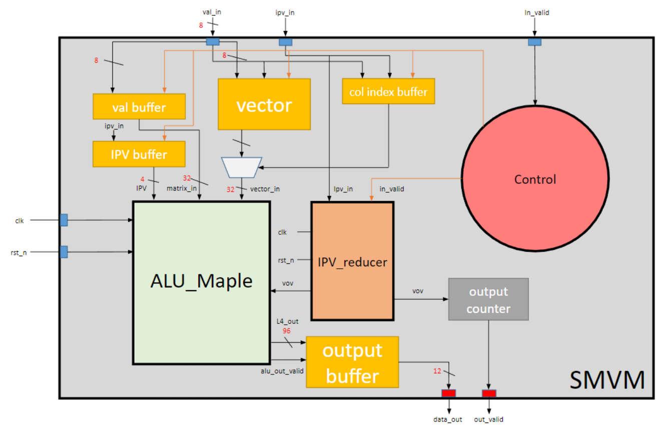
value	10	-2	3	9	3	7	8	7	3 ... 9	13	4	2	-1
col_idx	1	5	1	2	6	2	3	4	1 ... 5	6	2	5	6
IPV	0	1	0	0	1	0	0	1	0 ... 0	1	0	0	1

Fig. 1: The IPV representation proposed by [1].

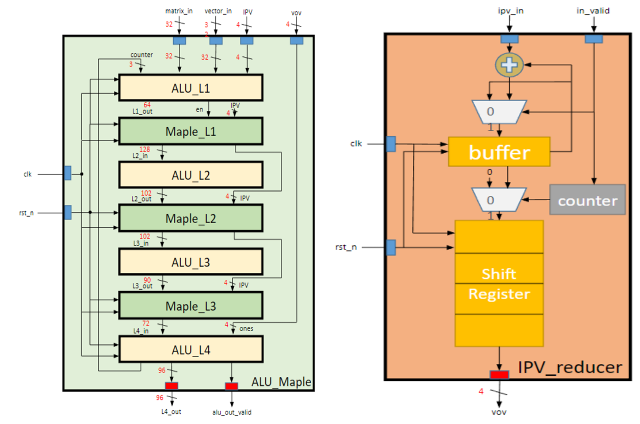
B. Hardware Architecture

The overall design of our ASIC chip is shown in fig. 2a. The column index array, value array, and the vector to be multiplied are inputted through *val_in*. After being inputted, they are stored in buffers until all required values are inputted. Pre-calculated IPV array is inputted through *ipv_in* and stored in *IPV_reducer*. *in_valid* is used to control the data flow. The output is read out from *data_out* when *out_valid* is pulled high. In the following sections, we will provide detailed hardware architecture of the main blocks in our ASIC.

1) ALU Maple: This part is the main computing unit of our ASIC. Though the high-level architecture is mainly based on the pipeline proposed by [1], we re-design the internal architecture to fit the data flow of our ASIC. ALU Maple consists of 4 ALU stages and 3 map tables, which are the same



(a) The high-level architecture of our chip design



(b) The block diagram of ALU maple and IPV reducer

Fig. 2: An overview of the proposed architecture. fig. 2a illustrates the overall architecture of SMVM design, while fig. 2b shows the detailed architecture of ALU_Maple and IPV_reducer that appear in fig. 2a.

as that in [1]. The level 1 ALU has 4 multipliers, multiplying 4 non-zero matrix entries in a clock cycle. In every map table stage, we decide which values should add together and which should directly trickle down to the next stage based on 4-bit IPV values. For instance, if IPV is 1111 in Maple L1, which means the four values are the last elements of the row and thus should not add together. Therefore, they will directly flow to buffers in ALU L2, making the 2 adder in ALU L2 idle for one cycle. On the other hand, if IPV equals to 0000, the four multiplied results will flow to the adders. In the last ALU stage (ALU L4), we decide the number of outputs based on a signal *VOV*, which will be detailed in section II-B4. If the last non-zero entry of a row doesn't show up in this cycle, the accumulated result should be stored in the adder accumulator (AAC), waiting for the rest of non-zero entries to arrive. One can refer to section II-B2 for more information about AAC. In addition, appendix A provides the detailed architecture of the ALUs and Maples.

2) AAC: Simple as it seems, we encounter difficulties when implementing AAC as a simple 24-bit adder. We find that the

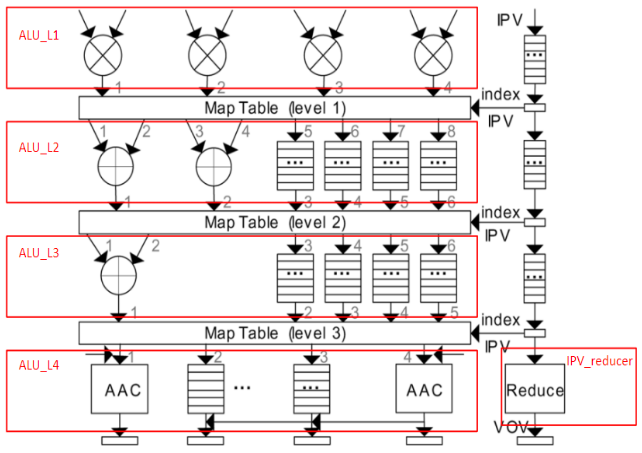


Fig. 3: The corresponding pipeline stages between our design and that proposed by [1]

critical path induced by the adder is so long that it becomes the bottleneck of the whole pipeline. As a result, we refer to the AAC design proposed by [2], which divides the 24-bit adder into 2 pipeline stages. We add the least significant 12 bits in the first pipeline stage, and the most significant 12 bits in the second pipeline stage. That way, we can cut the critical path in half, solving the issue.

3) Multiplier: The 4 multipliers in ALU_L1 stage also induces a long critical path. To address this issue, we implement them by the *Modified Baugh-Wooley algorithm* and split the computing cycle to 2. That is, performing multiplication in 2 clock cycles.

4) IPV Reducer: In our design, IPV is an important signal that control the data flow in ALU Maple. In fact, it is also used to generate a control signal, called *VOV*, which decides the number of outputs in the last pipeline stage (ALU_L4). The logic that takes IPV as input and outputs VOV is named *IPV reducer*. It counts the the number of ones in 4-bit IPV, which equals to the number of the last entry in a row and should output a result for that row.

5) I/O: Due to limited size of the ASIC, we don't store the matrix data on the chip; instead, we input those values during runtime. First of all, the chip will receive the shape of the matrix, then the values of the vector, and finally the non-zero entries of the matrix. As mentioned in section II-A, the non-zero entries will be encoded into the value array, the column index array, and IPV. Theoretically, these arrays should be inputted in parallel to attain high efficiency. However, due to limited number of I/O pins, we have no choice but to input them in 2 cycles. Therefore, though we have 4 multipliers that can operate in parallel in one cycle, achieving 4x speedup, the data for the multipliers will not be ready for 8 clock cycles, which drastically counteracts the speed gained from parallelism.

As for the output, it is a 24-bit value. Facing the same bandwidth limitation as the input side, we output each value in 2 clock cycles. Hence, in the worst case, where $IPV = 1111$, there will be 4 values to be outputted simultaneously, and the output process should take 8 cycles to complete. One can refer to table I for detailed I/O specifications.

TABLE I: The I/O pins of our ASIC chip

Signal Name	I/O	Width
clk	I	1
rst_n	I	1
val_in	I	8
ipv_in	I	1
in_valid	I	1
out_valid	O	1
out_data	O	12

III. Evaluation

In evaluation, We set the matrix size to 256×256, with 20% non-zero percentage. The test data is fed into our RTL design through a testbench. After the correctness is verified, we synthesize our design with an EDA tool, *Design Compiler*, to get the gate-level design. After that, we verify it through another testbench, with the clock cycle set to 10ns. At last, we turn the gate-level design into layout with another EDA tool, *Innovus*, with the *UMC* $0.18\mu m$ *Mixed-Mode and RFCMOS 1.8V/3.3V* process. The power supply of the ASIC is 1.8V. After passing the design rule check (DRC) and the layout versus schematic (LVS), the layout design is sent to UMC for final tape-out.

IV. Result

table II presents the specifications of the ASIC and the measured values in pre-layout and post-layout stage. fig. 4 shows the layout and the connections between the chip area and the package. In fig. 5, the ASIC has been taped out by UMC, and here we show 4 of them.

After detailed analysis, our ASIC is found to perform SMVM 70x faster than its Python counterpart under 10ns clock cycle.

TABLE II: The specifications of our ASIC

Specification	**Spec.**	**Pre-layout(tt)**	**Post-layout(tt)**
Frequency	100 MHz	100 MHz	
Chip size (μm^2)	< 1500 x 1500	1499.9 x 1499.9	
Power	-	14.7 mW	21.1 mW
PADs	40	40	

V. Conclusion

In this work, we implement an ASIC for SMVM acceleration. Due to I/O bandwidth limitation, We adjust the pipeline architecture and the data flow proposed by [1] to make the chip fit our requirements. To address issues from long critical paths, we optimize the multipliers and AACs, enabling our chip to operate at higher clock frequency. Afterwards, we synthesize the RTL code into a layout design and measure the power consumption as well as the chip area. The post-layout simulation shows a 70x speedup compared to the software implementation. At last, our design is approved by TSRI and taped out under the UMC $0.18\mu m$ process.

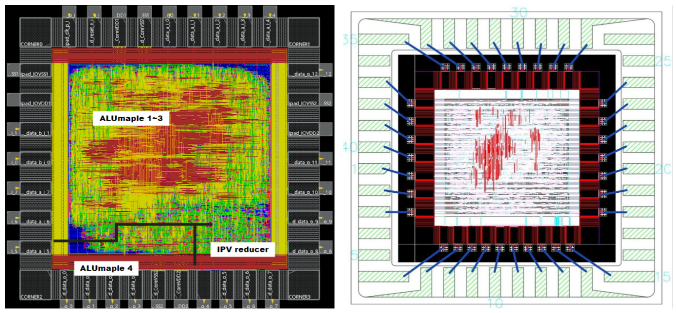


Fig. 4: On the left, we synthesize the layout and mark the corresponding functional blocks on it. On the right, we connect the I/O pins of our ASIC to the I/O pads on the package.

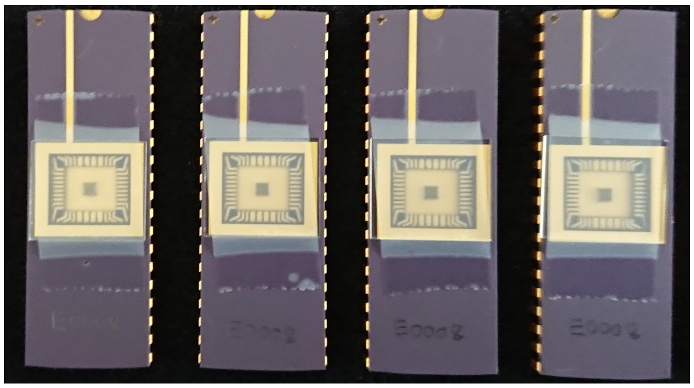


Fig. 5: The ASIC that taped out by UMC

VI. Future Work

Due to limitations from chip area and the number of I/O pins, we have to sacrifice some speed gain to make our chip work. For instance, the IPV encoding process is done by PC in our work. If the hardware constraints are loosened, we hope to implement IPV encoding in hardware. Furthermore, we fix the number of multipliers to 4, which may be inefficient when dealing with matrices with different sparsity. Therefore, we hope to find the relation between the parallelism and the sparsity.

VII. Acknowledgement

We want to thank Prof. Tzi-Dar Chiueh for advising us in the course Integrated Circuits Lab, and we also want to thank Taiwan Semiconductor Research Institute (TSRI) and UMC for taping out our design.

References

[1] S. Sun, M. Monga, P. H. Jones and J. Zambreno, ”An I/O Bandwidth-Sensitive Sparse Matrix-Vector Multiplication Engine on FPGAs,” in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 59, no. 1, pp. 113-123, Jan. 2012, doi: 10.1109/TCSI.2011.2161389.
[2] P. Zicari, S. Perri, P. Corsonello and G. Cocorullo, ”An optimized adder accumulator for high speed MACs,” 2005 6th International Conference on ASIC, 2005, pp. 757-760, doi: 10.1109/ICASIC.2005.1611425.

APPENDIX

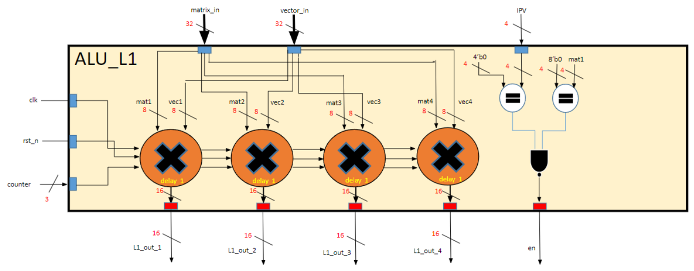


Fig. 6: The high-level architecture of our chip design

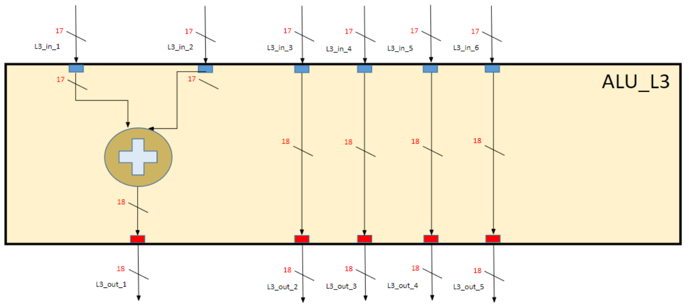


Fig. 10: The high-level architecture of our chip design

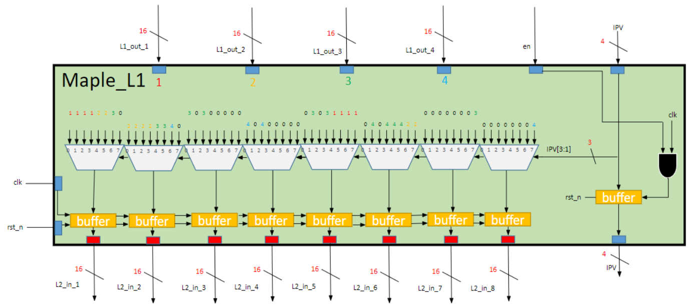


Fig. 7: The high-level architecture of our chip design

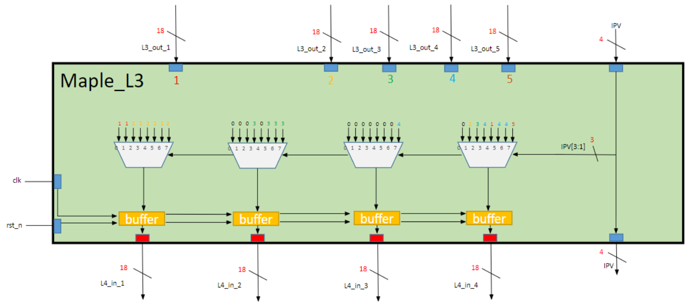


Fig. 11: The high-level architecture of our chip design

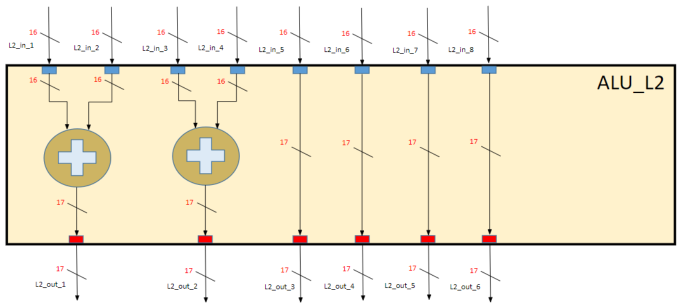


Fig. 8: The high-level architecture of our chip design

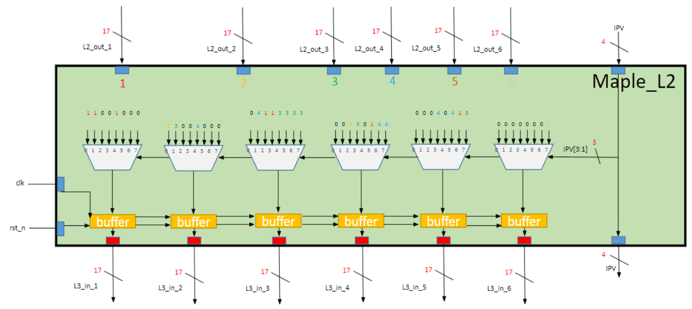


Fig. 9: The high-level architecture of our chip design

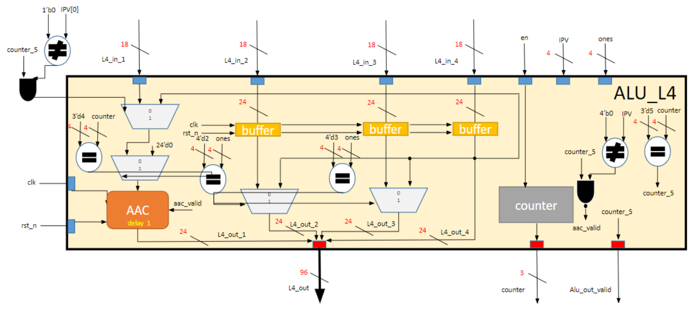


Fig. 12: The high-level architecture of our chip design