SparseCNN Hardware Design

R09921132 Liu, Yen-Fu, R09921129 Huang, Yi-Yao

***Abstract*—Sparsity, as an intrinsic property of convolutional neural networks (CNNs), has been widely employed for hardware acceleration. In this project, we follow the prior work [1] to implement the compressed sparse row (CSR) based Sparse CNN architecture and propose two different architectures to compare the results with the traditional CNN baseline. We get a conclusion that CSR-based Sparse CNN should save computing effort from algorithm perspective but need more memory access than traditional convolution which results in a large area and power/time-consuming memory blocks. Also, we follow what we learn in this class and the fan-out 4 delay model in HW3 to minimize the power*area term of SparseCNN with a data-stream 2x parallel CSR converter, a 2x pipeline PE ,and a 16x parallel adder memory module.**

***Index Terms*—Accelerator, architecture, convolutional neural networks (CNNs), sparsity.**

I. Introduction

In recent years, deep convolutional neural networks (CNNs) have made dramatic success in various applications, including image recognition, object detection, and semantic segmentation. However, CNNs provide impressive accuracy at the cost of huge computing complexity. To resolve the inefficiency of CNNs, AI accelerator is a hot research topic in recent years and one of the solutions is the SparseCNN hardware design.

Sparsity is an inherent property of CNNs, which refers to a fraction of weights/features that are exactly zero and can be used to reduce ineffective computation and storage consumption. Recent advances have demonstrated that pruning is an effective method to further increase the sparsity of CNNs, and negligible accuracy loss is introduced even more than half of the parameters are pruned. Together with the runtime sparsity in features, more potential benefits are available.

Although taking the benefits of sparsity may reduce the algorithm complexity, in this class, DSP in VLSI, we learn the optimization of an algorithm may cause more cost in hardware implementation. For this reason, we choose the SparseCNN hardware design as our final project topic. We can be more familiar with AI accelerators and study CNN hardware architectures. Also, we can experience whether SparseCNN is more energy-efficient or not.

In this article, we introduce the algorithm of SparseCNN first. Then, we demonstrate how we implement it and show more details of submodules including the CSR converter, PE, and adder memory module. Experiments of different sparsity input images demonstrate how SparseCNN can significantly reduce energy and power. In the last section, Architecture Exploration, we try to explore better architectures by power and area analysis with the fan-out 4 delay model.

II. SparseCNN

CNNs are widely used for solving machine intelligence problems today. In full convolution mode, the typical CNN scans the input feature maps (ifmaps) with kernel weights to generate the output feature maps (ofmaps), as shown in Fig. 1. So there are total $W_i * H_i * K_w * K_h$ MACs to complete a layer in a dense CNN. Sparsity is an opportunity, which can be used to cut the number of computations by skipping the calculations with zero values.

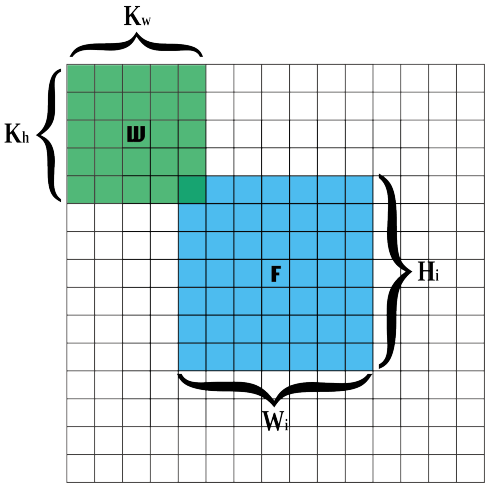

Fig. 1. Computation of a convolutional kernel.

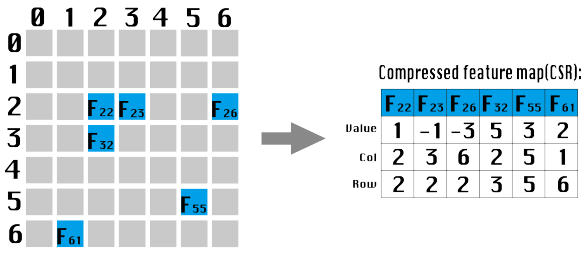

Fig. 2. Compressing of both features and weights using CSR encoding for a parse convolutional layer.

In this project, our SparseCNN adopts the CSR format to compress both sparse features and weights, and the encoded indices refer to the original row and col index, as shown in Fig. 2. Then, the features CSR and the weights CSR will be passed into the processing element (PE). PE performs a vector multiplication each cycle and during the processing, SparseCNN delivers the corresponding indices of weights and features to the Coordinate Computation module to obtain the output coordinates in the dense output feature space.

Then it sends the products and the output coordinates to product adder. Product adder includes several adders and stores the products to the correct index in the dense output feature map.

Fig. 3 shows how we implement our processing element.

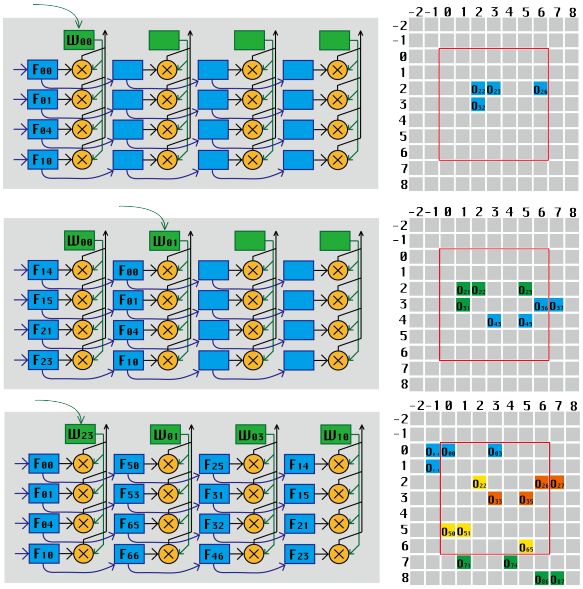

Fig. 3. Processing of the 2-D sparse convolution with the dataflow introduced in [1]. Assume the size of multiplier array is $4 * 4$. Destinations of the products over ofmap for each cycle are shown in the right column.

As the configuration of the multiplier array is $4 * 4$, the array reads four compressed features per cycle. At cycle 0, features are fed into the multiplier array while only one weight is transferred to the left-most weight register. At the same time, the products are routed to the product adder for accumulation according to their destinations. At cycle 1, the new features are fed into the leftmost column of the array, and prior features $F_{00}, F_{01}, F_{04}, F_{10}$ move to the second column. At cycle 3, the features of this ifmap meet the end, and the weight w_{00} is replaced by w_{23}. At the same time, the feature pointer rotates back to the beginning, and the first four features are fed into the array once again, starting the subsequent processing. Note that, in this example, as features are insufficient to fill up the multiplier array, more than one weight needs to be updated for higher resource utilization.

The output coordinates can be expressed as

$$o_r = F_r + \left\lfloor \frac{K_h}{2} \right\rfloor - w_r, o_c = F_c + \left\lfloor \frac{K_w}{2} \right\rfloor - w_c$$

where (F_r, F_c), (w_r, w_c), and (o_r, o_c) are the coordinates of features, weight and output product, respectively, and (K_w, K_h) denotes the kernel size of the convolutional layer. PE follows the equation to calculate the corresponding indices and sends them to product adder.

The main job of product adder is to add the output products to the correct position in the dense output feature map. Unlike CNN, SparseCNN produces irregular products whose destinations spread over the ofmaps randomly due to the irregular sparse patterns, causing access contentions. To overcome these problems, we provide several different designs and readers can see the detail in the Implementation Section.

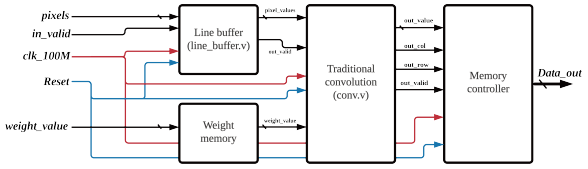

Fig. 4. The implementation of traditional CNN.

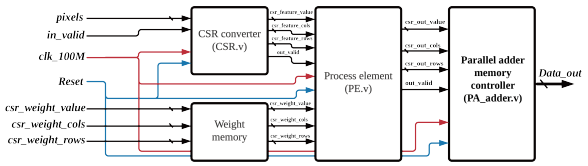

Fig. 5. The implementation of SparseCNN with Parallel adder memory module.

III. Implementation

A. Traditional CNN

In Figure 4, we show how we implement the traditional CNN baseline. Read feature map pixel by pixel into a 5x5 line buffer and wait for the line buffer to be ready. Send 25 pixels in line buffer and 25 weights into traditional convolution module, it will output one result which is the weighted sum of 25 pixels and its address. The result will be written into memory by the address given to adder memory module. Compare to sparse convolution, the disadvantage of traditional convolution is that if a pixel is zero, it will still perform multiplication.

B. SparseCNN with Parallel Adder (SparseCNNPA)

In Figure 5, we show how we implement the Sparse CNN. The input feature is sent into our CSR converter pixel by pixel and the converter produces three register arrays including values, columns, and rows. Because of the low sparsity of weights, we calculated the weight CSR first and store it in memory. When PE receives features CSR and weight CSR, it multiplies them concurrently and produces 16 products in each cycle.

Then, we implement a parallel adder memory module to store the 16 values into the dense output feature map with their indices in one cycle. However, due to the memory contention problem, indices may be different and we have to create at least $16 \times W_o \times H_o$ lines to make sure each product have the access to every memory position. This design needs pretty large area and high power consuming. Because of its disadvantage, we use another faster clock frequency to control adder memory module in the next design.

C. SparseCNN

In Figure 6, we show how we implement the Sparse CNN with only one adder in adder memory module. In previous section, we know it takes lots of areas to store the patterns in one cycle. Therefore, we apply 16 times clock frequency here and only store one pattern in a clock cycle to make the same function with less area. We also apply clock gating to make sure the adder only works at valid input.

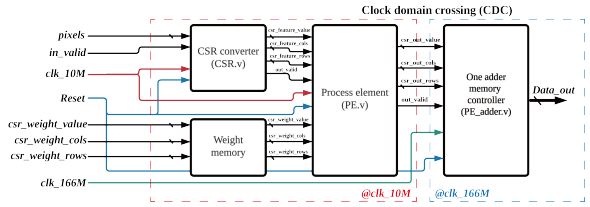

Fig. 6. The implementation of SparseCNN. CDC means that we already consider clock domain crossing problems here.

IV. Experimental Methodology

We use 28×28 as our input feature size and we implement the full mode convolution kernel. We use Synopsys Design Compiler version R-2020 to synthesize our circuit and Synopsys PrimeTime version S-2021 to do our time-based power analysis.

To evaluate how sparsity influences power and energy, we randomly generate 10 features with different sparsity which range from 0% to 90%. All features are fetched into different design and get the corresponding analysis results.

V. Evaluation

A. Area

Design	CSR Converter	PE	Adder memory	Total(um^2)
Traditional CNN	NA	NA	NA	1,569,046
SparseCNNPA	1,314,150	346,730	19,598,723	21,267,287
SparseCNN	1,210,688	342,196	1,435,061	2,991,302

TABLE I
The area of each design. The unit is um^2. Sparse CNN PA means the design: SparseCNN with parallel adder memory module.

In Table I, we can find the area of either SparseCNN or SparseCNNPA is larger than Traditional CNN. This is a trade-off between energy and area in CNN architecture. For SparseCNN and SparseCNNPA, the area of CSR Converter and PE are almost the same and the area of SparseCNN adder memory module is 13 times smaller than SparseCNNPA.

B. Power and Energy

In Table II, we can find that even though the power of SparseCNNPA is much smaller than SparseCNN at 10M operating frequency, Traditional CNN is still the most energy-efficient design. **The reason is the access contention problem in our adder memory module takes too much energy and power** as shown in Figure 8 and Figure 7. Although the adder memory module of SparseCNNPA has a much lower power and energy percentage than the adder memory module of SparseCNN, the adder memory module still dominates power and energy terms as shown in Figure 9, Figure 10, Figure 11, and Figure 12. However, sparsity help SparseCNN save much more energy than CNN. In Figure 13, the energy of SparseCNN drops dramatically when sparsity rises.

Design	Frequency	Sparsity	Power[mW]	Latency[ns]	Energy[μ J]
Traditional CNN	100M	0%	157.2	12,996.6	2.043
Traditional CNN	100M	10%	155.9	12,996.6	2.026
Traditional CNN	100M	20%	155.9	12,996.6	2.026
Traditional CNN	100M	30%	152.3	12,996.6	1.979
Traditional CNN	100M	40%	148.1	12,996.6	1.924
Traditional CNN	100M	50%	144.3	12,996.6	1.875
Traditional CNN	100M	60%	140.9	12,996.6	1.931
Traditional CNN	100M	70%	133.4	12,996.6	1.733
Traditional CNN	100M	80%	130.3	12,996.6	1.693
Traditional CNN	100M	90%	109.6	12,996.6	1.424
SparseCNNPA	100M	0%	482	21,265.3	10.250
SparseCNNPA	100M	10%	485	19,515.3	9.464
SparseCNNPA	100M	20%	477	17,695.3	8.440
SparseCNNPA	100M	30%	462	16,155.3	7.463
SparseCNNPA	100M	40%	449	14,825.3	6.656
SparseCNNPA	100M	50%	408	13,425.3	5.477
SparseCNNPA	100M	60%	393	12,235.3	4.808
SparseCNNPA	100M	70%	365	11,185.3	4.082
SparseCNNPA	100M	80%	329	10,415.3	3.426
SparseCNNPA	100M	90%	256	9,015.3	2.307
SparseCNNPA	10M	0%	51	204,144.3	10.417
SparseCNNPA	10M	10%	51.4	187,344.3	9.629
SparseCNNPA	10M	20%	50.5	169,872.3	8.578
SparseCNNPA	10M	30%	48.9	155,088.3	7.583
SparseCNNPA	10M	40%	47.6	142,320.3	6.774
SparseCNNPA	10M	50%	43.3	128,880.3	5.581
SparseCNNPA	10M	60%	41.7	117,456.3	4.897
SparseCNNPA	10M	70%	38.8	107,376.3	4.166
SparseCNNPA	10M	80%	35.1	99,984.3	3.509
SparseCNNPA	10M	90%	27.4	86,544.3	2.371
SparseCNN	10M/166M	0%	154	203,955.3	31.592
SparseCNN	10M/166M	10%	149	187,155.3	27.886
SparseCNN	10M/166M	20%	140	169,683.3	23.755
SparseCNN	10M/166M	30%	130	154,899.3	20.136
SparseCNN	10M/166M	40%	121	142,131.3	17.197
SparseCNN	10M/166M	50%	106	128,691.3	13.641
SparseCNN	10M/166M	60%	94.2	117,267.3	11.046
SparseCNN	10M/166M	70%	77.0	107,187.3	8.253
SparseCNN	10M/166M	80%	67.5	99,795.3	6.734
SparseCNN	10M/166M	90%	36.9	86,355.3	3.186

TABLE II
The power and energy of our designs with different input image sparsity. Note that because the adder memory module of SparseCNN have to operate at 16 times of top module frequency, the whole design can only work at 10MHz. (MHz denotes as M)

In summary, sparse convolution should save computing effort which is also true from an algorithm perspective. From a hardware perspective, sparse convolution need more memory access than traditional convolution which results in a large area and power/time-consuming memory blocks.

VI. Architecture Exploration

In this section, we follow what we learn in the DSP in VLSI class and the HW3 FO4 inverter delay model in UMC 0.18μm

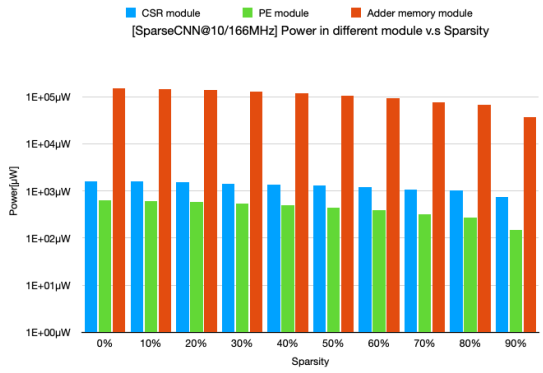

Fig. 7. The total power of each module in SparseCNN.

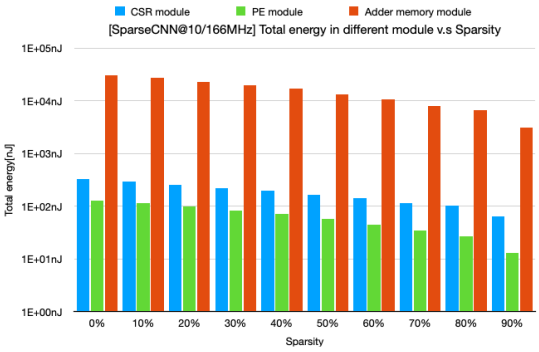

Fig. 8. The total energy of each module in SparseCNN.

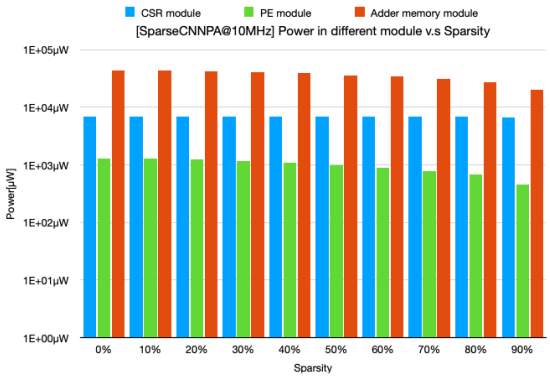

Fig. 9. The total power of each module in SparseCNNPA@10MHz.

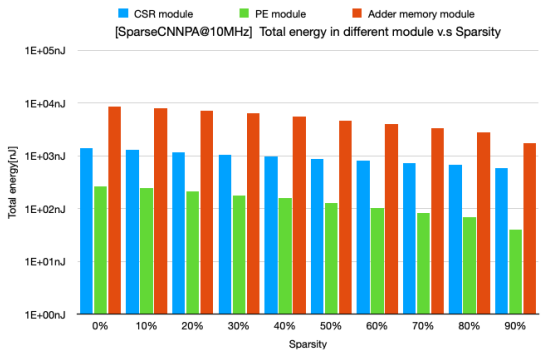

Fig. 10. The total energy of each module in SparseCNNPA@10MHz.

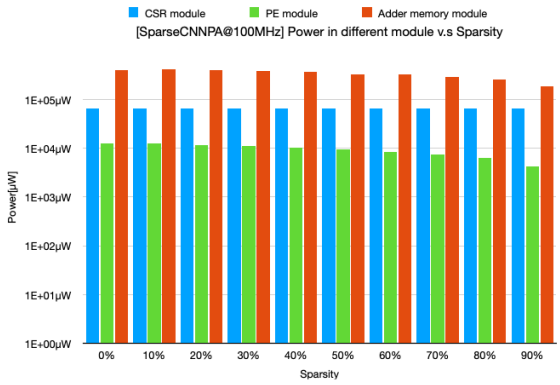

Fig. 11. The total power of each module in SparseCNNPA@100MHz.

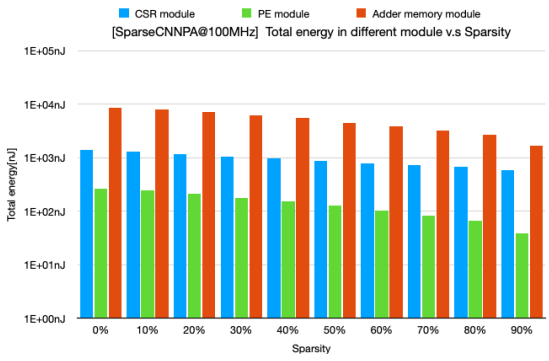

Fig. 12. The total energy of each module in SparseCNNPA@100MHz.

CMOS technology to explore the best architecture.

The fitting curve of FO4 delay and V_{DD} is

$$T_p = \frac{0.0753 \times V_{DD}}{(V_{DD} - 0.4)^2} \quad [ns] \tag{1}$$

This curve is from our HW3 and we use this equation to calculate the operating supply voltage when we change the operating frequency.

The reference designs are the modules of SparseCNN including CSR Converter, PE, and Adder memory. We randomly choose an image from MNIST which sparsity is 85.84% to measure the power of reference design. The goal of our exploration is **to minimize the power*area**.

Our methodology is using the fan-out 4 delay curve to evaluate a target operating point which is the supply voltage and clock period. Dynamic power is the sum of internal power and net switching power. Dynamic power is $\propto {V_{DD}}^2 \times f$ and leakage power is $\propto A \times V_{DD}$ where V_{DD} is the supply voltage, f is the clock frequency, and A is the area.

In two times parallel design, simply double the area of the reference design and calculate the supply voltage at the double of clock period of the reference design which is 1.213 volts. Dynamic power can be calculated by equation 2 and leakage power can be calculated by equation 3.

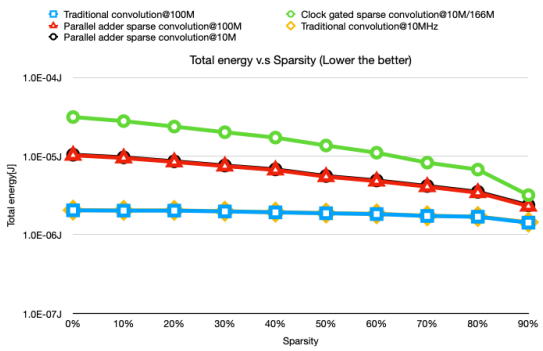

Fig. 13. Line chart of energy versus sparsity between traditional convolution and sparse convolution.

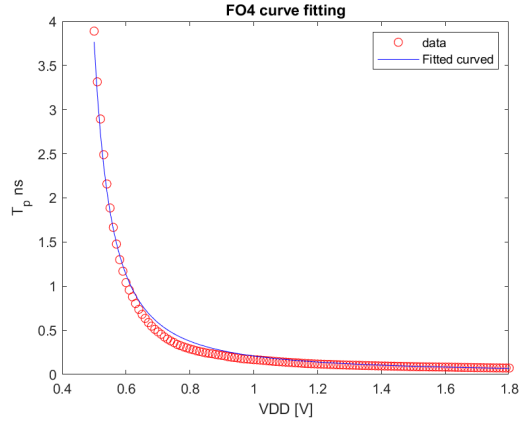

Fig. 14. The fan-out 4 inverter delay curve model in UMC 0.18 μm CMOS technology.

In two times pipeline design, it is more complicated than two times parallel design. First, insert an extra stage of flip-flops in the input of the reference design, set the same clock period of the reference design but supply voltage set at 1.213 volts, and calculate the target clock period at 1.8 volts. Perform the design synthesis at the target clock period with register retiming function and get the area result. Dynamic power can be calculated by equation 2 and leakage power can be calculated by equation 3. The area on the target design is about 1.15 times of area on the reference design which is almost the same as we discussed during the lecture.

$$P_{tar_{dynamic}} = P_{ref_{dynamic}} * (\frac{Vdd_{tar}}{Vdd_{ref}})^2 * \frac{fclk_{tar}}{fclk_{ref}} \quad [W] \quad (2)$$

$$P_{tar_{leakage}} = P_{ref_{leakage}} * \frac{Vdd_{tar}}{Vdd_{ref}} * \frac{area_{tar}}{area_{ref}} \quad [W] \quad (3)$$

A. CSR Converter

Data batch is reading one channel of the frame in one clock period at the input stage and provides more flexibility of conversion process. Data streaming is reading one pixel of the frame in one clock period at the input stage and provide more area-efficient of the design. In table III, it is clear that data batch design is more power and area consuming than

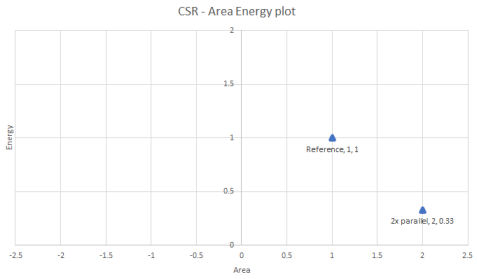

Fig. 15. CSR design space exploration.

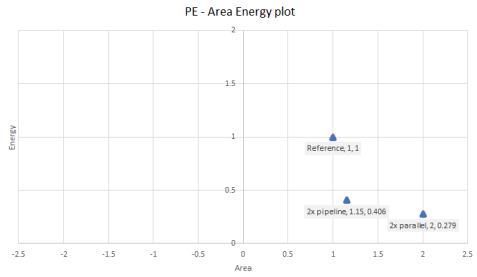

Fig. 16. PE design space exploration.

data streaming. In power times area term and total energy, data streaming is 8x smaller than data batch design than data batch and we decided to use the data streaming method in our design.

In the table IV, We estimate the two times parallel design of the CSR converter module and discovered that it achieves 70% power times the area of the reference design. Figure 15 shows the design space exploration.

B. PE

In table V, the reference design is a 4x4 multiplier array that simply outputs 16 data at each clock cycle. The 2x parallel design has twice the clock period of the reference design and it operates at 1.213 volts, it achieves 55% on power times area of reference design. Even more, two times pipeline design with one extra clock period in multiplication which is done by register retiming on design compiler and it operates at 1.213 volts, it achieves 46% on power times area of reference design. In the lecture, we discussed about pipeline design will cause roughly 1.1 times the area of the reference design, after PE synthesis, we confirm that pipeline design is roughly 1.15 times the area of reference design in UMC 0.18 μm CMOS technology. Figure 16 shows the design space exploration.

C. Adder memory module

In table VI, the reference design uses a single adder and operates at 16 times the clock frequency of the previous stage PE module. In 16 times parallel design, it uses 16 individual adders at the same time and operates at the same clock frequency of previous stage PE module, it solves the memory contention problem easily by selecting each adder at a different time and it operates at 0.6 volts, it achieves 22% on power times area of reference design in theory. But due to the low

Design	Supply voltage[V]	Frequency [MHz]	Area [μm^2]	Power[W]	Internal power[W]	Net switching power[W]	Cell leakage power[W]	Power[W]* Area[μm^2]	Latency [ns]
Data-stream	1.8	100	1,435,564	$6.59 * 10^{-2}$	$6.40 * 10^{-2}$	$9.47 * 10^{-5}$	$8.97 * 10^{-5}$	98,910	7,870
Data-batch	1.8	100	6,573,244	$2.06 * 10^{-1}$	$1.07 * 10^{-1}$	$9.83 * 10^{-2}$	$3.32 * 10^{-4}$	775,642	7,880

TABLE III
Different CSR converter data flow design.

Design	Supply voltage[V]	Frequency[MHz]	Area[μm^2]	Power[W]	Internal power[W]	Net switching power[W]	Cell leakage power[W]	Power[W]* Area[μm^2]
Ref	1.8	10	1,210,688	$7.20 * 10^{-4}$	$5.50 * 10^{-4}$	$9.68 * 10^{-5}$	$7.33 * 10^{-5}$	871.69
2x Parallel	1.213	5	2,421,376	$2.46 * 10^{-4}$	$1.25 * 10^{-4}$	$2.19 * 10^{-5}$	$9.88 * 10^{-5}$	595.66

TABLE IV
Architecture exploration of CSR converter.

Design	Supply voltage[V]	Frequency[MHz]	Area[μm^2]	Power[W]	Internal power[W]	Net switching power[W]	Cell leakage power[W]	Power[W]* Area[μm^2]
Ref	1.8	10	342,196	$2.16 * 10^{-4}$	$9.78 * 10^{-5}$	$1.08 * 10^{-4}$	$1.01 * 10^{-5}$	73.91
2x parallel	1.213	5	684,392	$6.03 * 10^{-5}$	$2.22 * 10^{-5}$	$2.45 * 10^{-5}$	$1.36 * 10^{-5}$	41.27
2x pipeline	1.213	10	393,684	$8.77 * 10^{-5}$	$3.80 * 10^{-5}$	$4.19 * 10^{-5}$	$7.83 * 10^{-6}$	34.54

TABLE V
Architecture exploration of PE.

Design	Supply voltage[V]	Frequency[MHz]	Area[μm^2]	Power[W]	Internal power[W]	Net switching power[W]	Cell leakage power[W]	Power[W]* Area[μm^2]
Ref	1.8	166	1,435,061	$5.81 * 10^{-2}$	$4.61 * 10^{-2}$	$1.10 * 10^{-2}$	$7.5 * 10^{-5}$	83377.04
16x parallel	0.6025	10	22,960,976	$8.02 * 10^{-4}$	$3.23 * 10^{-4}$	$7.70 * 10^{-5}$	$4.02 * 10^{-4}$	18414.70

TABLE VI
Architecture exploration of adder memory module.

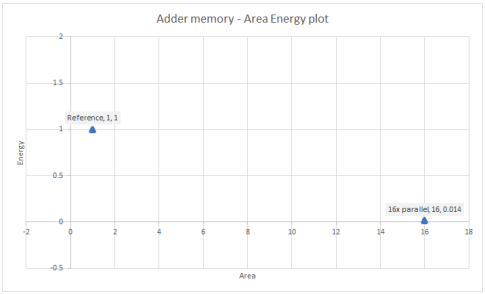

Fig. 17. Adder memory design space exploration.

voltage supply, there maybe has some IR drop in a memory module that the teaching assistant and I had discussed during the lecture. Figure 17 shows the design space exploration.

D. SparseCNN

Based on the above discussion, we get the conclusion that the SparseCNN architecture can minimize the power*area by using data-stream 2x Parallel CSR converter, 2x pipeline PE, and 16x parallel adder memory module.

VII. Conclusion

In this project, we implement a simple AI accelerator, SparseCNN, and get a conclusion that SparseCNN should save computing complexity from an algorithm perspective but the access contention problem causes more memory access than traditional CNN which results in large area and power/time-consuming memory blocks. Also, we follow what we learn from this class and find a better architecture, SparseCNN with a data-stream 2x Parallel CSR converter, a 2x pipeline PE, and a 16x parallel adder memory module, to minimize the power*area term.

References

[1] F. Li, G. Li, Z. Mo, X. He, and J. Cheng, "Fsa: A fine-grained systolic accelerator for sparse cnns," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3589–3600, 2020.