# Pack Sparse Conv2D Computation to Accelerate on Systolic Array

Yuyue Wang, Yu Zuo, Xiaxuan Gao

December 13, 2021

## 1 Introduction

Neural networks have become an essential tool for solving complex problems in various domains. In order to support a broad range of applications and satisfy their requirement of accuracy, neural networks become larger and larger, suffering from over-parameterization and the problems it brings about. Therefore, various techniques are proposed to make neural networks sparse and maintain its accuracy at the same time. However, accelerators that are designed for normal NNs can barely benefit from the sparsity. This provides the opportunity of designing new architectures that can better utilize the sparsity.

The convolution layers are the most time-consuming part in a CNN. Therefore, optimizing the convolution part is the most efficient way to reduce CNN inference time. Convolution can be lowered to matrix multiplication, and systolic array is very suitable for MM computation. If there is a method to compress the sparse data and utilize its sparsity, it can improve the efficiency of CNN significantly combined with systolic array. Therefore, we decided to choose this topic.

## 2 Related Work

### 2.1 Cambricon-X: An Accelerator for Sparse Neural Networks

Due to the fact that the neural network accelerators are hard to benefit from the reduction of the synaptic weights, Cambricon-X(1) suggests a new accelerator architecture to take advantage of the NN sparsity. This new architecture introduces multiple important components, such as the synapses buffer and the buffer controller, to compress the sparse neural network into a dense format. Therefore, it can achieve 7.23x speedup and 6.43x energy saving against the state-of-the-art NN accelerators.

Figure 1 shows the overview architecture of Cambricon-X. $CP$ refers to the control processor which is responsible for routing instructions to other parts in order to orchestrate the computation and data transfer. Both $NBin$ and $NBout$ are neuron buffers which will store input neurons and output neurons respectively. $DMA$ stands for direct memory access. It is mainly used for the on-chip and off-chip data communication. On the other hand, Cambricon-X has a central buffer controller which is responsible for controlling the

PE computation. Lastly, it also has 16 PEs for the actual computation.

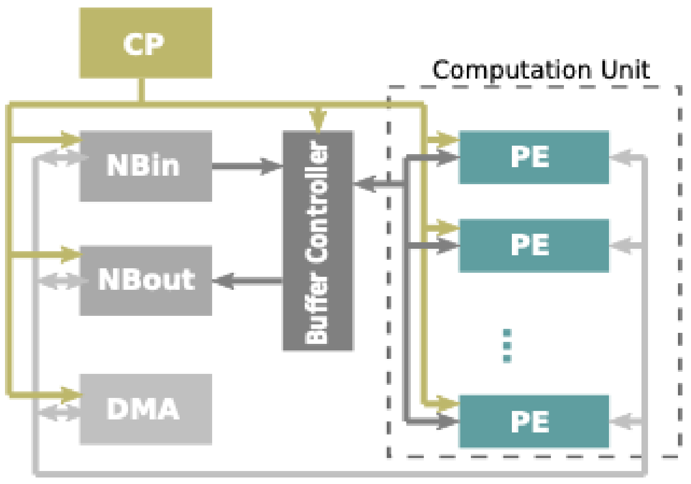


Figure 1: Cambricon-X Overview Architecture

The buffer controller is the most significant component for the computing orchestration inside Cambricon-X. Figure 2 shows the *BC* architecture. When input neurons are transfered into *BC*, they will be stored at the buffer controller functional unit (*BCFU*). As soon as PEs are ready to take in neurons, *BCFU* sends neurons to multiple indexing modules (*IM*). These *IMs* compress the incoming neurons into a condense format by the step indexing technique. The step indexing records the gaps between each necessary neurons as an array. And it puts this array inside a multiplexer such that the multiplexer is able to filter out the corresponding neurons that are needed for computation and. In this way , only necessary neurons will be used for computation and passed to PEs, which effectively accelerates the computation of PEs. After PEs finish computing, they either send the result neurons to *BCFU* for further processing or write them back to *NBout*.

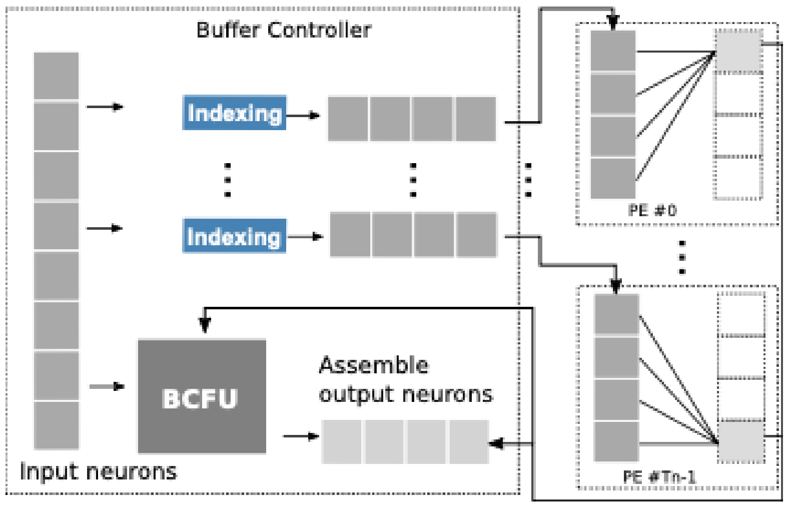


Figure 2: Buffer Controller Architecture of Cambricon-X

In addition to the buffer controller, the PE architecture (Figure 3) is another innovative design of Cambricon-X. Different from other PEs, a PE on Cambricon-X has a synapses buffer ($SB$) which compresses and stores the synapses and a PE functional unit ($PEFU$) which does the actual computation. After $DMA$ transmits weights into $SB$, $SB$ will extract the weights that are associated with each output neurons and align them in rows. More importantly, zero weights will be eliminated during this process. Accordingly, to compute output neurons, only necessary synapses will be read by $PEFU$. In order to do the computation, $PEFU$ ingests the neurons from $BC$ and synapses from $SB$. It passes them into 16 different multipliers and a 16-in adder tree such that it is able to generate one output neuron at each clock cycle. One thing to note that is $PEFU$ pipelines the two stages of multiplying and adding in order to achieve even more optimization.

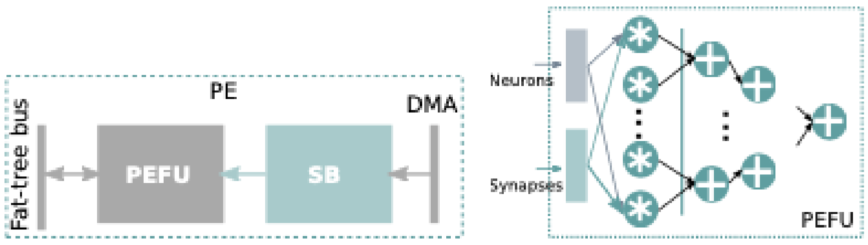


Figure 3: PE Architecture of Cambricon-X

## 2.2 SCNN: An Accelerator for Compressed-sparse Convolutional Neural Network

This paper(2) proposes a novel architecture for sparse CNN acceleration. The sparsity in all types of neural networks is very common today, and mainly comes from network pruning and nonlinear activation functions. However, architectures that are designed for dense situation can barely benefit from the sparsity, which can save a large amount of time and energy if fully utilized. The paper first introduces CNN computation shortly, and points out that dataflow is the core of accelerator design. It then proposes the PT-IS-CP-dense dataflow, and claims that PT-IS-CP-sparse dataflow is merely a natural extension.

First, the 6-layer nested loops are transformed so that the loop denoting input activation channels (loop $C$) is taken to the outermost layer. This makes sure that input activations can stay in the cache or registers until all the computation of it has been done, and it is called IS, or input stationary. However, it also brings about the problem that weights and output activations suffer from low data reuse rate. Therefore, the paper further proposes that the loop of output activation channels (loop $K$) can be tiled into loop $K$ and $Kc$. Part of weights and output activations can be saved in buffers and are not replaced until the end of every loop $K$ cycle. This can alleviate the data reuse problem.

Unlike other architectures, this design uses a cartesian product (CP) rather than dot product implementation. Every time, the cartesian product of a row of input activations and a column of weights are calculated simultaneously in the multiplexer array in PE. On the top of that, the input activations (W * H, C channels) can be tiled into data of size Wt * Ht * C. This is called planar tiled (PT), and multiple PEs can compute one tile simultaneously, which further increase parallelism. The problem of tiling is that there will be inter-PE data dependency on the edge of each tile. It then proposes two solutions: input halo and output halo. The output halo solution, which is adopted by this paper, requires inter-PE synchronization and communication, which is shown in the figure 4 below.

The sparse dataflow has little difference from the dense one. This architecture requires that sparse data should be compressed before entering PEs, and the result of compression, no matter which compressing method is used, is a dense data block and an index sequence that indicating the original coordinates. Therefore, the only difference is that the way of calculating result indices is changed. The architecture inside each PE is shown in the figure 5. The PPU (post-processing unit) is responsible for halo communication, ReLU operation and compression of output activations.

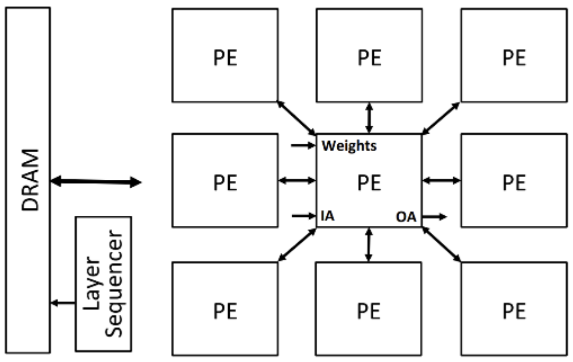


Figure 4: SCNN Architecture

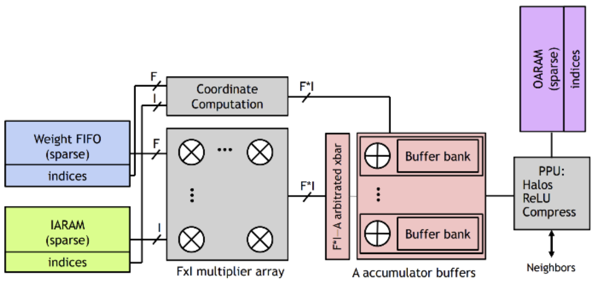


Figure 5: PE Architecture

AlexNet, GoogLeNet and VGGNet are used as benckmarks. Its performance, energy-efficiency and sensitivity to CNN sparsity are compared with other architectures, and analysis of different layers in CNN shows that SCNN works well when sparsity is not minor. The result also shows that inter-PE synchronization and intra-PE fragment is the major limitation of this architecture.

## 2.3 Adaptive Tiling: Applying Fixed-size Systolic Arrays To Sparse Convolutional Neural Networks

This paper (3) presents an approach to map sparse neural network computation onto systolic array. Generally, matrix multiplication needs to be done in tiles to fit the size of systolic array, but the matrix tiles are highly sparse and those zero values still occupy cells in the array, making it hard to exploit the sparsity. In this paper, researchers propose an idea named adaptive tiling. Adjacent sparse matrix tiles in the same row can be packed together, so that most tiles' non-zero values occupy the empty space of other tiles, and we can get a denser tile with minimum loss of non-zero values. After the packing step, the packed tile is pre-stored onto the systolic array, with each cell containing a value and the tile index of the value. Then the other matrix's tiles that can be multiplied with the packed tiles and contribute to the same result tile are pumped into the array from the left-most column in parallel. Every cell chooses the input value matching its pre-stored value's tile to multiply, and add it to the partial sum from the upper cell. It then transfers the partial sum to its bottom cell and the input values to its right cell. The result tile's values are collected at the bottom row of array.

Although this idea is novel, its primitive version has too strict constraint on how should adjacent tiles be packed together(Packed tiles should align by both column index and row index). Later the same group publicized a new research, column combining (4) to accelerate CNN on systolic array. In this research, multiple columns forming a dense column after combination are grouped together, and all matrix columns are divided into several such groups. Correspondingly, the other matrix's rows are permuted so that its row indexes match the former matrix's column indexes after column grouping. Then the column combined matrix is pre-stored onto the systolic array cells, with the row-permuted matrix's columns pumped into the array from the left-most array column in a skew manner. The result tile is still collected from the bottom row of the array.

The two papers use Lasso regularization(the summation of all weights' absolute value) in the loss function to encourage more sparse weights. In the second paper, because non-zero value loss is introduced by combining columns, the neural network needs re-training after reducing the conflicting values to regain high prediction accuracy. For the systolic array hardware design, in the second paper they convert float point computation to lower precision fixed point computation to reduce data communication bandwidth and computation, and pump data in a bit by bit manner and use bit-serial MAC unit to amortize the communication and computation to multiple cycles. These modifications to the ordinary systolic array enable it maintain high execution frequency.

## 2.4 Other Papers

Another group presents three papers that are later in time, mainly publicized in 2020.

The first paper proposes an accelerator named Tensorurus(8), and focuses more on compression methods. Based on CISR (Compressed Interleaved Sparse Row), the paper further proposes the CISS (Compressed Interleaved Sparse Slice) method. The core architecture of Tensorurus, TLU, reads data in CISR format and push them to PEs in dataflow.

The second paper, sparse-TPU(7), has much connections with the paper on adaptive tiling, and improves the column-combining process. It also packs multiple sparse columns with no conflict into one dense column. However, it analyzes the conflict probability, and claims that the length of columns is the critical factor of conflict numbers. The paper proposes a more advanced packing method: cut columns into pieces, and pack pieces at the same row base respectively. The conflict data is not discarded, but filled into vacant position of the packed column.

The third paper(6) proposes Sparse Systolic Tensor Array, which is a modified version of systolic array that can exploit a special type of sparsity – VDBB sparsity. Based on DBB sparsity, VDBB can support models with different levels of sparsity. The core architecture, Tensor-PE (TPE), contains a systolic array that supports VDBB sparsity, and each TPE can handle a small matrix multiplication.

## 3 Our Approach

### 3.1 Accelerator Selection

The three papers provide us with three different options. Cambricon-X could be considered as a special type of systolic array with other peripheral modules and data compression mechanism. SCNN is a totally different architecture compared to systolic array, because it uses multiplexer arrays that compute a cartesian product in parallel. The third paper, adaptive tiling, modifies the normal systolic array to support adaptively packed tiles. The architectures proposed by the first two papers are complex, and have little connection with AutoSA(5). We want to use this tool, so we choose the direction of adaptive tiling, and also combine ideas of (4) and (7).

The third paper in our group mainly focused on applying adaptive tiling to pointwise convolution. The reason is that in MobileNet, convolution at each convolutional layer is accomplished by applying depthwise 2D convolutions to each channel, followed by pointwise convolution over all channels. The second part consumes 95% of the time. Pointwise convolution can be expressed as matrix multiplication. More generally, convolution can be converted to matrix multiplication, so we decided to implement only MM accelerator after the failure with AutoSA (we will talk about it later in section 5).

#### 3.1.1 Tools

As a brief summary, we used PyTorch to construct, train, and prune our VGG19 neural network. And we used vitis HLS and gcc to write Cpp code with HLS pragmas to implement the hardware design. Since the tools we used are so few we decided to skip drawing the flowchart.

### 3.2 Algorithm Design and Implementation

We adopt the network training and pruning process of (4), and implement it with Python and PyTorch library. We use PyTorch's network library to consruct the network architecutre of VGG19, and its dataloader to access CIFAR10 dataset as the training and testing data. For optimizer, we use SGD optimizer with nesterov momentum 0.9, and a cosine declining learning rate with 0.1 as the initial value. At the middle point of the whole training, we firstly do unstructured pruning to prune a certain portion of non-zero values, and then do column combining to erase conflicting non-zero values, which is implemented in Python code and registered as a user-defined prune method in PyTorch. The column combining algorithm is slightly different from what's used in (4), as we use partition-wise packing, which means we firstly partition the weight matrix by dividing columns to shorter ones. According to (7), this modification can greatly reduce the probability of column collision. For the limitation of project time, we didn't implement row permutation as we don't have time to connect the software part and hardware part for a real-world application experiment.

### 3.3 Hardware Design

We implement our systolic array architecture directly in HLS. For the ease of implementation, we only implement two versions of systolic array both using 32bit float point data type for input and output value(Thus we don't need to implement bit-serial MAC and balance the input ratio of feature map values and partial sums). The first version's systolic array is written implicitly by setting column number of input feature map as outermost loop and pipelining it. In this way, values of multiple feature map rows are pumped in in parallel and executed for every cycle. The second version's systolic array is written explicitly by setting the whole execution sequence(timestamp) as outermost loop and pipelining it. And input values are pumped in one by one instead of in parallel, and adjacent array rows skew input for one cycle, and every cell computes every max($\text{factor}_{\text{combine}}$) cycles, after the arrival of all inputs and the partial sum, like the architecture proposed in (7).

## 4 Experiment

### 4.1 Model Accuracy and Sparsity of Different Epochs

As discussed in Section 3.2, we construct, train and prune VGG19 neuron network with PyTorch. We wrote experiment code to show the network accuracy and sparsity after different epochs. Some unsaved accuracy and sparsity results show that our implementation works as expected, as both the accuracy and sparsity remain high enough after the pruning and retraining. Unfortunately we didn't save checkpoints and due to the time limit there is no time left to re-perform the whole procedure to collect and analyze the data.

### 4.2 Algorithm Verification

For the column combining algorithm, we combine the techniques we learned from both (4) and (7). In order to verify the correctness of this algorithm, we manually construct a small benchmark and compare the software result against the golden result.

### 4.3 Hardware Execution Verification

Similarly to how we verify the column combining algorithm in the last section, we also construct randomly generated MM benchmarks and compare the hardware simulation result against the golden result by software execution.

### 4.4 Hardware Synthesis Analysis

We implemented multi-column systolic array in two versions as described in (3.3). They both use $16 \times 16$ systolic array, and the workload is matrix multiplication between a $8 \times 64$ matrix and a $64 \times 8$ matrix(packed to $8 \times 8$ matrix with combination factor=8). The target technology is xcvu9p-flga2104-2-i with clock rate 5.

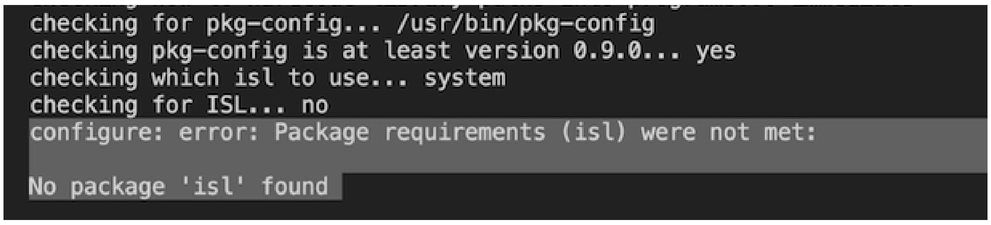
The first version (parallel input) achieves 57 cycles total latency, with iteration latency=51 and initial interval=1. As the array size is small, all resources' utilization rate is less than 5%. The second version (sequential input) achieves 112 cycles total latency, with iteration latency=36 (To be honest I think it should be 1, but it's so high and HLS seems not to completely unroll the 1 step computations) and initial interval=1. I cannot explain why the two versions have so big iteration latency as I'm not a HLS expert and don't know the magic process under the pipeline pragma in the two versions of code. And I'll be really glad to know the reason.

## 5 Challenge

While doing this project, we encounter a set of obstacles. Those obstacles are primarily related to the development of AutoSA(5). Based on our understanding, to generate systolic array with column combining logic requires modifying the AutoSA repository. But then we figure out that it would be a little bit easier if we just write the whole column combining logic inside our C program. Therefore, we start our experiment by installing AutoSA and modifying the provisioning examples inside AutoSA. However, this process is much more difficult than what we expect.

### 5.1 Installing AutoSA

Installing AutoSA is already very hard. The given tutorial of AutoSA does not specify the Amazon machine image(AMI) version of AWS. Therefore, we have attempted multiple AWS AMI, such as FPGA Developer AMI which we used in our lab2, Xilinx Vitis 2020.2 Developer AMI, and Xilinx Vitis 2021.1 Developer AMI. With hours of experiments, we figure out Xilinx Vitis 2021.1 is the only AMI that is compatible with the requirement of AutoSA. And all other images have some environment issues. For the FPGA Developer AMI, the package installation command given by the AutoSA tutorial would always fail. Usually, there are three packages that AutoSA can not correctly install because of environment incompatibility: ISL, xgboost, and barvinok. The figure below shows an example of ISL pakcage error we had during setup. What makes it worse is that the barvinok package does not have a explicit installation guide since it is a 20 years old project, which means we are unable to manually install it. In this way, we decided to use other AMI. Furthermore, for the Vitis 2020.2, even if all the packages are correctly installed, this AMI fails to compile the code of AutoSA due to file management issues shown in figure 2.



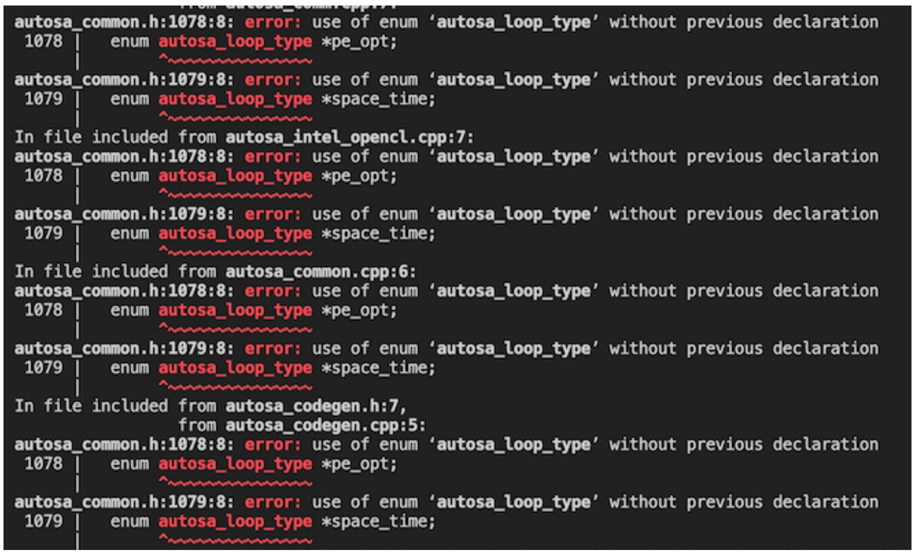
```
checking for pkg-config... /usr/bin/pkg-config
checking pkg-config is at least version 0.9.0... yes
checking which isl to use... system
checking for ISL... no
configure: error: Package requirements (isl) were not met:

No package 'isl' found
```

Figure 6: Installing AutoSA on incorrect AMI would invoke this package ISL unavailable error.

### 5.2 Running AutoSA Examples

After correctly installing AutoSA on Xilinx Vitis 2021.2, we then begin our experiment with the provisioning examples of AutoSA, for instance, the small and large convolutional neural network. However, there are a couple of issues with the existing examples. First, the hardware simulation of the large CNN doesn't terminate in 3 hours. As a result,



```
autosa_common.h:1078:8: error: use of enum ‘autosa_loop_type’ without previous declaration
 1078 |   enum autosa_loop_type *pe_opt;
      |        ^~~~~~~~~~~~~~~~
autosa_common.h:1079:8: error: use of enum ‘autosa_loop_type’ without previous declaration
 1079 |   enum autosa_loop_type *space_time;
      |        ^~~~~~~~~~~~~~~~
In file included from autosa_intel_opencl.cpp:7:
autosa_common.h:1078:8: error: use of enum ‘autosa_loop_type’ without previous declaration
 1078 |   enum autosa_loop_type *pe_opt;
      |        ^~~~~~~~~~~~~~~~
autosa_common.h:1079:8: error: use of enum ‘autosa_loop_type’ without previous declaration
 1079 |   enum autosa_loop_type *space_time;
      |        ^~~~~~~~~~~~~~~~
In file included from autosa_common.cpp:6:
autosa_common.h:1078:8: error: use of enum ‘autosa_loop_type’ without previous declaration
 1078 |   enum autosa_loop_type *pe_opt;
      |        ^~~~~~~~~~~~~~~~
autosa_common.h:1079:8: error: use of enum ‘autosa_loop_type’ without previous declaration
 1079 |   enum autosa_loop_type *space_time;
      |        ^~~~~~~~~~~~~~~~
In file included from autosa_codegen.h:7,
                 from autosa_codegen.cpp:5:
autosa_common.h:1078:8: error: use of enum ‘autosa_loop_type’ without previous declaration
 1078 |   enum autosa_loop_type *pe_opt;
      |        ^~~~~~~~~~~~~~~~
autosa_common.h:1079:8: error: use of enum ‘autosa_loop_type’ without previous declaration
 1079 |   enum autosa_loop_type *space_time;
      |        ^~~~~~~~~~~~~~~~
```

Figure 7: Installing AutoSA on incorrect AMI would cause this compiling error.

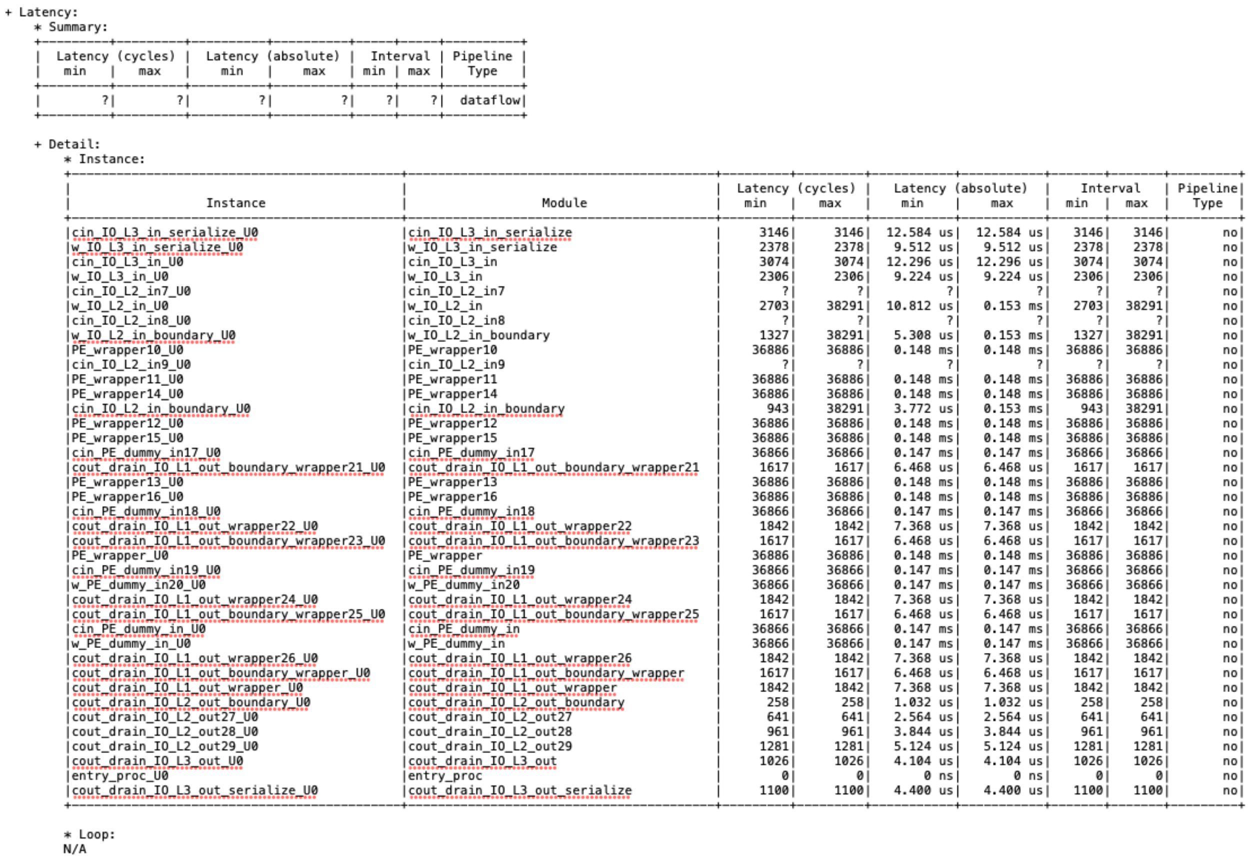
we turn to the small CNN example and hope it would work. Indeed, it works, but the hardware utilization report, showed in the figure 3, has a lot of question marks, which indicates that the generated HLS code of AutoSA has undetermined loop counts. As the HLS code is automatically generated by AutoSA, this issue becomes extremely difficult to address. We have tried to modify the result HLS code, but it didn't work out. And with these undetermined loop count, we are incapable of observing our code performance on the hardware.

## 5.3 Modifying HLS Code Generated By AutoSA

After these setbacks, we finally made the decision to implement the architecture in pure HLS code. However, implementing systolic array in HLS is not easy, because the dataflow and coordinate-calculation logic is very complex. On the other hand, the systolic array we need is slightly different from the original one: it must support indexing, receiving plural floating numbers in one column and feeding different data into corresponding multiplexers. After considering these issues, we tried to generate HLS code with AutoSA, write our own HLS code by imitation, and modify the part of systolic array.

By reading the original code, we found out that the adding column combining support did not only change the code of PE itself. In the figure below is the basic architecture of systolic array generated by AutoSA. Apart from PEs, multilevel memory hierarchy is also built, including L3 module and L2 module. L3 module is responsible for receiving data from main memory and streaming it to L2 modules. Each L2 module should take the data required by its corresponding PEs and stream the rest to the next module.

These memory modules created significant difficulty for us when modifying the code. The workload of adding index system to these memory modules was prohibiting. It was extremely hard to maintain the correctness of the whole accelerator, because its functionality could not be verified until every module had been modified. Although testing a single module was rather easy, the problem mainly came from the relation of indices between different modules. For example, AutoSA used various length of datatype. Adding extra index to the data required us to change the union length (or structure) and data



```
+ Latency:
    * Summary:
    +---------+---------+----------+----------+-----+-----+----------+
    |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline |
    |   min   |   max   |    min   |    max   | min | max |   Type   |
    +---------+---------+----------+----------+-----+-----+----------+
    |        ?|        ?|         ?|         ?|    ?|    ?|  dataflow|
    +---------+---------+----------+----------+-----+-----+----------+

    + Detail:
        * Instance:
        +--------------------------------------------+------------------------------------------+---------+---------+-----------+-----------+-------+-------+---------+
        |                                            |                                          |  Latency (cycles) |   Latency (absolute)  |    Interval   | Pipeline|
        |                  Instance                  |                  Module                  |   min   |   max   |    min    |    max    |  min  |  max  |  Type   |
        +--------------------------------------------+------------------------------------------+---------+---------+-----------+-----------+-------+-------+---------+
        |cin_IO_L3_in_serialize_U0                   |cin_IO_L3_in_serialize                    |     3146|     3146|  12.584 us|  12.584 us|   3146|   3146|       no|
        |w_IO_L3_in_serialize_U0                     |w_IO_L3_in_serialize                      |     2378|     2378|   9.512 us|   9.512 us|   2378|   2378|       no|
        |cin_IO_L3_in_U0                             |cin_IO_L3_in                              |     3074|     3074|  12.296 us|  12.296 us|   3074|   3074|       no|
        |w_IO_L3_in_U0                               |w_IO_L3_in                                |     2306|     2306|   9.224 us|   9.224 us|   2306|   2306|       no|
        |cin_IO_L2_in7_U0                            |cin_IO_L2_in7                             |        ?|        ?|          ?|          ?|      ?|      ?|       no|
        |w_IO_L2_in_U0                               |w_IO_L2_in                                |     2703|    38291|  10.812 us|   0.153 ms|   2703|  38291|       no|
        |cin_IO_L2_in8_U0                            |cin_IO_L2_in8                             |        ?|        ?|          ?|          ?|      ?|      ?|       no|
        |w_IO_L2_in_boundary_U0                      |w_IO_L2_in_boundary                       |     1327|    38291|   5.308 us|   0.153 ms|   1327|  38291|       no|
        |PE_wrapper10_U0                             |PE_wrapper10                              |    36886|    36886|   0.148 ms|   0.148 ms|  36886|  36886|       no|
        |cin_IO_L2_in9_U0                            |cin_IO_L2_in9                             |        ?|        ?|          ?|          ?|      ?|      ?|       no|
        |PE_wrapper11_U0                             |PE_wrapper11                              |    36886|    36886|   0.148 ms|   0.148 ms|  36886|  36886|       no|
        |PE_wrapper14_U0                             |PE_wrapper14                              |    36886|    36886|   0.148 ms|   0.148 ms|  36886|  36886|       no|
        |cin_IO_L2_in_boundary_U0                    |cin_IO_L2_in_boundary                     |      943|    38291|   3.772 us|   0.153 ms|    943|  38291|       no|
        |PE_wrapper12_U0                             |PE_wrapper12                              |    36886|    36886|   0.148 ms|   0.148 ms|  36886|  36886|       no|
        |PE_wrapper15_U0                             |PE_wrapper15                              |    36886|    36886|   0.148 ms|   0.148 ms|  36886|  36886|       no|
        |cin_PE_dummy_in17_U0                        |cin_PE_dummy_in17                         |    36866|    36866|   0.147 ms|   0.147 ms|  36866|  36866|       no|
        |cout_drain_IO_L1_out_boundary_wrapper21_U0  |cout_drain_IO_L1_out_boundary_wrapper21   |     1617|     1617|   6.468 us|   6.468 us|   1617|   1617|       no|
        |PE_wrapper13_U0                             |PE_wrapper13                              |    36886|    36886|   0.148 ms|   0.148 ms|  36886|  36886|       no|
        |PE_wrapper16_U0                             |PE_wrapper16                              |    36886|    36886|   0.148 ms|   0.148 ms|  36886|  36886|       no|
        |cin_PE_dummy_in18_U0                        |cin_PE_dummy_in18                         |    36866|    36866|   0.147 ms|   0.147 ms|  36866|  36866|       no|
        |cout_drain_IO_L1_out_wrapper22_U0           |cout_drain_IO_L1_out_wrapper22            |     1842|     1842|   7.368 us|   7.368 us|   1842|   1842|       no|
        |cout_drain_IO_L1_out_boundary_wrapper23_U0  |cout_drain_IO_L1_out_boundary_wrapper23   |     1617|     1617|   6.468 us|   6.468 us|   1617|   1617|       no|
        |PE_wrapper_U0                               |PE_wrapper                                |    36886|    36886|   0.148 ms|   0.148 ms|  36886|  36886|       no|
        |cin_PE_dummy_in19_U0                        |cin_PE_dummy_in19                         |    36866|    36866|   0.147 ms|   0.147 ms|  36866|  36866|       no|
        |w_PE_dummy_in20_U0                          |w_PE_dummy_in20                           |    36866|    36866|   0.147 ms|   0.147 ms|  36866|  36866|       no|
        |cout_drain_IO_L1_out_wrapper24_U0           |cout_drain_IO_L1_out_wrapper24            |     1842|     1842|   7.368 us|   7.368 us|   1842|   1842|       no|
        |cout_drain_IO_L1_out_boundary_wrapper25_U0  |cout_drain_IO_L1_out_boundary_wrapper25   |     1617|     1617|   6.468 us|   6.468 us|   1617|   1617|       no|
        |cin_PE_dummy_in_U0                          |cin_PE_dummy_in                           |    36866|    36866|   0.147 ms|   0.147 ms|  36866|  36866|       no|
        |w_PE_dummy_in_U0                            |w_PE_dummy_in                             |    36866|    36866|   0.147 ms|   0.147 ms|  36866|  36866|       no|
        |cout_drain_IO_L1_out_wrapper26_U0           |cout_drain_IO_L1_out_wrapper26            |     1842|     1842|   7.368 us|   7.368 us|   1842|   1842|       no|
        |cout_drain_IO_L1_out_boundary_wrapper_U0    |cout_drain_IO_L1_out_boundary_wrapper     |     1617|     1617|   6.468 us|   6.468 us|   1617|   1617|       no|
        |cout_drain_IO_L1_out_wrapper_U0             |cout_drain_IO_L1_out_wrapper              |     1842|     1842|   7.368 us|   7.368 us|   1842|   1842|       no|
        |cout_drain_IO_L2_out_boundary_U0            |cout_drain_IO_L2_out_boundary             |      258|      258|   1.032 us|   1.032 us|    258|    258|       no|
        |cout_drain_IO_L2_out27_U0                   |cout_drain_IO_L2_out27                    |      641|      641|   2.564 us|   2.564 us|    641|    641|       no|
        |cout_drain_IO_L2_out28_U0                   |cout_drain_IO_L2_out28                    |      961|      961|   3.844 us|   3.844 us|    961|    961|       no|
        |cout_drain_IO_L2_out29_U0                   |cout_drain_IO_L2_out29                    |     1281|     1281|   5.124 us|   5.124 us|   1281|   1281|       no|
        |cout_drain_IO_L3_out_U0                     |cout_drain_IO_L3_out                      |     1026|     1026|   4.104 us|   4.104 us|   1026|   1026|       no|
        |entry_proc_U0                               |entry_proc                                |        0|        0|       0 ns|       0 ns|      0|      0|       no|
        |cout_drain_IO_L3_out_serialize_U0           |cout_drain_IO_L3_out_serialize            |     1100|     1100|   4.400 us|   4.400 us|   1100|   1100|       no|
        +--------------------------------------------+------------------------------------------+---------+---------+-----------+-----------+-------+-------+---------+

        * Loop:
        N/A
```

Figure 8: The hardware utilization report of the small CNN example.

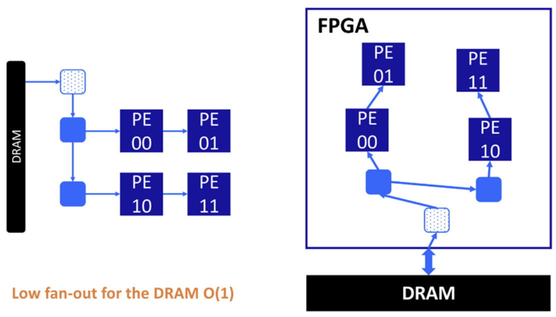


Figure 9: Systolic array structure (generated by AutoSA)

length, which may lead to bugs that are fatal and hard to detect. After spending several days on it, we had to admit that modifying generated HLS code was a dead end.

## 6 Extension

### 6.1 Column Combining Optimization

The paper(6) presented by the other group proposes more efficient column packing strategies. The first one is called partition-wise packing. The idea comes from the fact that the probability of collision increases with column length. If the length is over 1000, even two sparse columns can collide at a high probability. Therefore, our original greedy strategy may result in no compression at all. In partition-wise packing, the sparse matrix is first partitioned vertically by the length of the systolic array, then column packing is performed on each partition. (Already adopted by our latest program)

Another strategy is called collision-aware packing. The original method may tolerate 2 or 3 collisions, and only store the number with max absolute value. Collision-aware packing can be divided into two steps. The original collision-tolerate packing is performed in an iterative way. The columns that collide in the first iteration are used as additional packing candidates for the second iteration. Then the collision-free strategy is used, and all the left columns are packed.

### 6.2 Microarchitecture Optimization

Our implementation didn't fully realize the hardware design in (4), which can be a potential improvement. Also, we didn't optimize the hardware IO, which by using techniques like double buffering can further reduce the latency.

## 7 Conclusion

By combining several ideas from (3), (4) and (7), we successfully realize the key steps of the pipeline of sparse CNN acceleration on systolic array. Our implementation has the most basic functionality and can be further improved with more engineering work and HLS optimization.

## 8 Acknowledge

This work was done by Yuyue Wang, Yu Zuo, and Xiaxuan Gao. Thanks Yuyue for writing the pruned VGG-19 neural network with column combining logic and the plain HLS code for the column combining in matrix multiplication. Xiaxuan and Yu mainly focused on exploring AutoSA. Their works consist of trying to modify AutoSA and its existing examples to support column combining. Last but not the least, thanks Linghao Song, Jason Cong, and Jie Wang for the instructions during our development.

# References

[1] S. Zhang et al., "Cambricon-X: An accelerator for sparse neural networks," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1-12, doi: 10.1109/MICRO.2016.7783723.

[2] Parashar, Angshuman et al. “SCNN: An accelerator for compressed-sparse convolutional neural networks.” *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017): 27-40.

[3] H. T. Kung, B. McDanel and S. Q. Zhang, "Adaptive Tiling: Applying Fixed-size Systolic Arrays To Sparse Convolutional Neural Networks," *2018 24th International Conference on Pattern Recognition (ICPR)*, 2018, pp. 1006-1011, doi: 10.1109/ICPR.2018.8545462.

[4] Kung, H. T. et al. “Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization.” *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*(2019): n. pag.

[5] Wang, Jie et al. “AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA.”*The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2021): n. pag.

[6] Liu, Zhi-Gang, Paul N. Whatmough, and Matthew Mattina. "Sparse systolic tensor array for efficient CNN hardware acceleration." *arXiv preprint arXiv:2009.02381* (2020).

[7] He, Xin, et al. "Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices." *International Conference on Supercomputing (ICS’20).*2020.

[8] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi and Z. Zhang, "Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations," *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020,*pp. 689-702, doi: 10.1109/HPCA47549.2020.00062.

[9] Spcl. "Spcl/gemm-hls: Scalable Systolic Array-Based Matrix-Matrix Multiplication Implemented in Vivado HLS for Xilinx Fpgas." Github, https://github.com/spcl/gemm-hls.