Eyeriss Row Stationary Convolution Using Systolic Arrays

1st Sandhya Ravindran
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, USA
sravindran35@gatech.edu

2nd Tejaswini Anand Kumar
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, USA
tkumar48@gatech.edu

***Abstract*—Systolic arrays are an excellent approach to perform convolution efficiently. There exist many data flow algorithms to support systolic array based convolution- weight stationary, output stationary , input stationary etc. These strategies are suggested for reducing frequent accesses to secondary memory systems and for enabling high data re-use. One of the novel strategies that combines the ideologies of all of the above resulting in very high data re-use scenarios whilst maintaining maximum possible parallelism is the Eyeriss row stationary systolic arrays. In this project, we will try to simulate a systolic array based on row stationary data sharing and perform convolution to understand its benefits and caveats.**

I. Introduction

Convolution Neural Networks (CNN) are widely used in applications like pattern recognition, image processing, voice recognition etc. The complexities of these CNN models are exponentially increasing with increase in input data set size, their resolution etc. Hence these CNN models have many layers, millions of filter weights, and varying shapes. To achieve high performance and energy efficiency, there is a need to adopt parallelism at various levels. Compute parallelism can be achieved by having multiple processing elements parallely working on data. Even though convolution has MAC operations which can utilise high parallelism, and hence increase the throughput significantly, two issues need to be addressed which act as bottlenecks for the parallel computations:

- A huge amount of energy is utilized in simply reading all the necessary inputs from secondary memory to support the MACs owing to the high bandwidth requirement.
- Analysing where to store the partial summations that are parallelly produced by the MACs since improper storage algorithms might lead to more energy utilisation.

Hence, there is a need to maximise data reuse once it is fetched from the memory. There are various data flow techniques which perform data reuse like:

- Input stationary: minimize input feature maps movement.
- Weight stationary: minimize filter weights movements.
- Output stationary: minimize partial sums movement.
- No Local Reuse (NLR): doesn't use PE's local storage, instead uses global buffer of bigger size.

Eyeriss is a state-of-the art CNN accelerator which is re-configurable, energy efficient and has better performance than GPUs on CNN models. It uses row stationary data flow approach which maximises data reuse compared to other techniques.

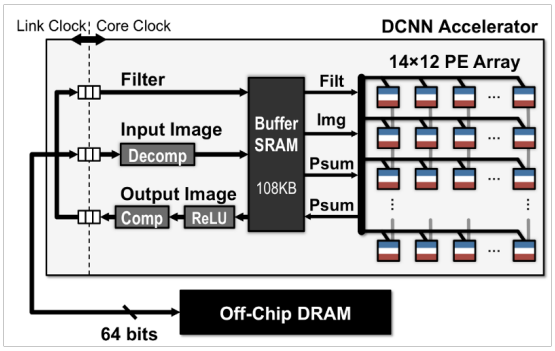

Fig. 1. Eyeriss architecture

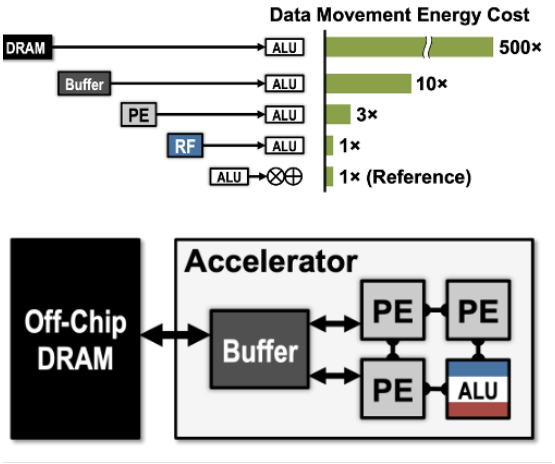

Fig. 2. Eyeriss architecture

II. Related Work

Eyeriss is a state-of-the-art CNNs: it has millions of filter weights, a lot of layers, and of different shapes. Eyeriss DUT chip's spatial array has 168 processing elements (PE) controlled by a re-configurable network on chip (NoC).

It reduces data movement and handles multiple shapes by exploiting data reuse. Eyeriss maximizes Data Reuse within PE. It has 3 types of data reuse:

- Convolutional reuse: the input data can be shared in small amounts among multiple convolution operations because the convolution layers use weight stationary property.
- Filter reuse: Each and every filter weights can be reused in fully connected and convolution layer across a set of input feature maps.
- Ifmap reuse: Each and every input feature map data can be reused in fully connected and convolution layer across multiple filters which generate multiple output channels.

The problem statement related to partial sums can be eliminated using proper operation scheduling techniques. This way the psums generated can be immediately used up and in turn reducing memory read-write energy requirements and storage.

III. Design Milestone 1

We would like to implement from scratch - the type of data flow used in Eyeriss – Row stationary Convolution using Systolic Arrays on FPGAs. For enabling the maximum possible data reuse, the data movement and sharing happens across three directions across the PEs. Apart from this, data sharing is utilized to its best capability within a PE.

A. Data Movement Design

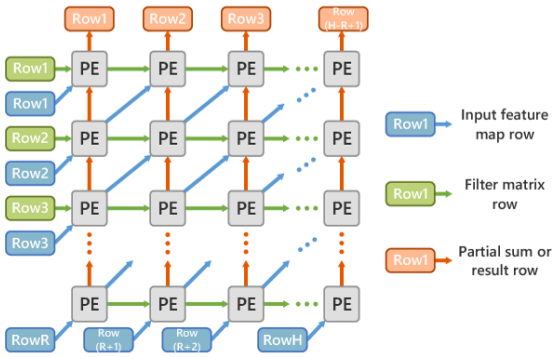

Fig. 3. Row Stationary Data Movement Summary

- **Data re-use within the PE**
  In the row stationary approach, each row (or a portion of the row in case of huge data size) of the input feature map and its corresponding row in the filter is assigned to one PE. The PE is capable of reusing that row's weights and input feature map by moving the data cyclically. It also performs a shift right operation to accommodate the next input feature map value needed for compute. It is also able to produce a partial sum with respect to the row by accumulating it at the same time. The sharing movement within a PE is shown in Fig. 4.
- **Data re-use across different PEs**
  Now this can be scaled to be inclusive of N D convolution.

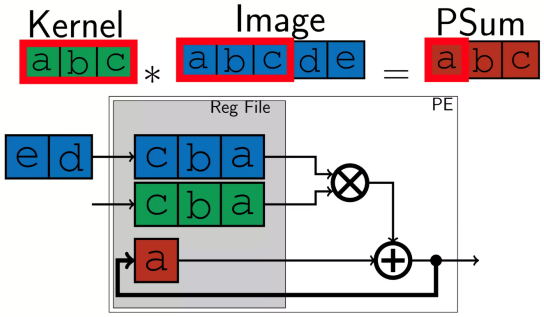

Fig. 4. Data Reuse and movement within a PE

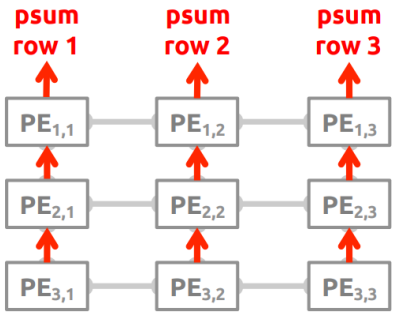

Fig. 5. Vertical Data Movement : PSUMs

- **Vertical Data Sharing:** Consider a mesh like configuration for the PEs. The logical PEs in the same column would be computing partial sums across consecutive rows of input i.e consecutive row elements of the output. Hence their partial sums could be shared in the vertical direction at a scheduled time to finally reach an external PE that produces the total sum for the output data. This also eliminates the problem of partial sum intermediate storage. (Fig. 5)
- **Horizontal data sharing:** The logical PEs in the same row would ideally be using the same filter weights. We can share the filter values from one PE to the next in the horizontal direction there by reusing weights and not fetching it from secondary memory for each PE. (Fig. 6)
- **Diagonal Data Sharing:** When the filter moves downward in the vertical direction, the 2nd row

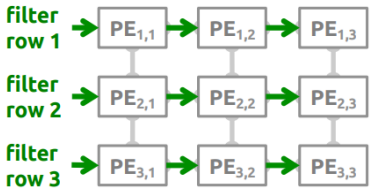

Fig. 6. Horizontal Data Movement : Weights

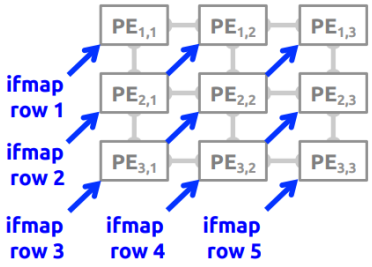

Fig. 7. Diagonal Data Movement : Input feature map

of input feature map now becomes needs to be multiplied with the first row of the filter. So the 2nd row of input can be shared to the diagonal PE which would contain the weights of the first row and would be idle to perform this MAC. Likewise, this would happen parallelly across all PEs and the edge PEs would receive new data from the secondary memory. This way the input feature map is also re-used across the PEs. Thus at the end of all operations, each logical column of the PE grid would produce one (N rows/row+col-1) row outputs.(Fig. 7)

B. Data Movement Implementation

Implementation of the data movements/sharing was done in a step wise manner as described below.

- Initially implemented 1 D convolution CPP functional code that would not require diagonal data movement.
- Completed the CPP implementation of 2D convolution using the row stationary implementation without limitations on the available internal buffers to each PE. Implemented all three types of data movement - horizontal, vertical and diagonal with appropriate scheduling for movement of PSUMs for a 5x5 input feature map and 3x3 filter. The systolic array in our design consists of 3x3 PEs. Initially two different approaches were considered to get to the final result - one based on synchronised approach where each PE moves data externally (vertically moves the PSUMs, horizontally moves the weights and diagonally moves the input feature map), performs a shift internally and computes MACs based on simulated cycles and the other is a function based approach where each operation inside the PE is called as iterations.
- We have chosen the cycle-based code as we feel it closely resembles systolic array movement in real time simulating similar delays while the other approach might produce high/unrealistic latency as the design scales.
- Structures are used for PEs which form the systolic array -3x3 grid. There are other summing PEs external to the systolic array which sum the PSUMs collected to produce a final result which can be written to secondary memory. The main convolution function simply calls the the nine PE functions (arranged like a 2D grid) N number of times based on the amount of data that needs to be processed (Fig. 8 - NOTE: this is the revised code snippet after the second design milestone). All the three different types of external data sharing and internal data re-use is written as a common code to all PEs and handled within the PE function. There are three external functions apart from this,one to read data from the DRAM to the internal buffers, one to handle collecting the PSUMs in vertical movement and perform a summation,another to handle writing the final data to the correct position in the DRAM.

IV. Next Steps after Design Milestone 1

- The above implementations of CPP codes are still not synthesizable due to structure of code and usages of some data structures and STL libraries. The internal buffers are not bound by size and use the deque data structure. This needs to be changed to arrays (with a logic of acting like queues) of a fixed size. The fixed size needs to smaller than the row size of the input image so that it can be tested keeping in mind that the current model can support any length of the row. The code needs to be altered to make it synthesizable and the necessary test bench needs to be written.
- Following this, necessary HLS based optimisations have to be added to enable all PEs to act in unison parallely with the necessary data-flow systolic array like movement.
- The optimisations need to be analysed to make conclusive results. Some amount of time would also be required for debugging the said approaches.

V. Design Milestone 2 : Design for Synthesizable Code

A. Choice of design

- The initial implementation strategy was to use BRAM as secondary storage by using the bind storage pragma to show a higher latency to the BRAM that has the entire array as compared to data movement in the local PE buffers. However, the pragma was unable to bind to the required latency.This would mean that row stationary convolution performs poorly compared to no data reuse design because the latency for all these memory hierarchies are same. Hence the benefit of data reuse is cannot be shown by simulation.
- After discussions, the implementation strategy was changed to directly considering DRAM as secondary storage and using completely paritioned BRAM as the local buffers assigned to each PE. This behavior more closely resembled the systolic array setup.

B. Design Changes for Synthesis

The implemented design from milestone1 although functionally correct and C simulation passes, synthesis cannot be achieved. Below listed are the previous design problems and the changes to overcome the same.

- Using the deque structure for the C simulation was one of the major reasons why the code is not synthesizable.

```
ITERATION_LOOP:
    for (int cycles = 0; cycles < active_cycles; cycles++)
    {
#pragma HLS pipeline

        // reads neccessary data from DRAM
        read_data_DRAM(DRAM_ip_data, DRAM_Wt_data, horizontal_pipes[0][0], horizontal_pipes[1][0],
                       horizontal_pipes[2][0], diagonal_pipes[0][0], diagonal_pipes[1][0], diagonal_pipes[2][0],
                       diagonal_pipes[2][1], diagonal_pipes[2][2]);

        // call the PEs unrolled PE number of times
    PE_LOOP:

        for (int j = 0; j < 3; j++)
        {
#pragma HLS unroll

            for (int i = 0; i < 3; i++)
            {

#pragma HLS unroll

                int k = i * 3 + j;
                // the below function handles data movement to neighbouring PE's , to external PSUM PE, and also the MAC operation
                processingelement(psum_accumulation[0], psum_accumulation[1], psum_accumulation[2], horizontal_pipes[i][j],
                                  horizontal_pipes[i][j + 1], vertical_pipes[i][j], vertical_pipes[i - 1][j], diagonal_pipes[i][j], diagonal_pipes[i - 1][j + 1], i, j, k);
            }
        }

        // PSUM outside systolic array
        sum_of_psums(psum_DRAM[0], psum_DRAM[1], psum_DRAM[2], psum_accumulation[0], psum_accumulation[1], psum_accumulation[2]);

        // write to DRAM
        write_data_DRAM(DRAM_op_data, psum_DRAM[0], psum_DRAM[1], psum_DRAM[2]);
    }
}
```

Fig. 8. Convolution Code Flow

The second reason is that the local buffers (or in this case queues) do not have a definite depth, and hence might continue to take in data and read out data appropriately during C simulation but not during synthesis. To avoid this,the number of available buffers to each PE is made to a limited fixed size. Hence each PE, irrespective of the size of the input is given a fixed size of 3 input buffers and 3 weight buffers in this design. This is done with the help of static arrays.

- Now that the data structure side of the problem is resolved, movement of data happens needs to be scheduled properly . The vertical movement needs to happen every after the PSUM accumulation i.e. 3 MAC operations for the given filter size (3x3). This needs to happen across all PEs at the same time. This has been accomplished in this design using simulated cycles (counter) for synchronizing movement. A similar setup is done for horizontal movement.
- In the case of diagonal movement, the input row size (number of elements per row) can be very large. To handle this, new data should come into a PE only when an old data is no longer required or has been completely re-used. In our design, since the filter size is 3, this happens every three cycles. Hence every 3 cycles, new data should come in, but prior to that the oldest data must be popped out and the remainder data present in the buffers must go through shift operation like a shift register.A CPP logic is used here for the static array to function like a shift register every three cycles.
- Finally, the last component to make the code synthesizable is to create another internal shift register for data re-use within a PE. Initial attempts were based on the shift register IP which was available on the official portal for the same. However, this IP library can only be used for C simulation and does not work for synthesis. Hence, this part of the code was also transformed using static array and appropriate indexing logic.

VI. Optimisation and Results

A. Baseline Results

Our initial code which is synthesizable with no optimization pragmas provides a latency of 8641 cycles. The number of DSPs used simultaneously by the PEs is just 3 since it is sequential. Around 10 BRAMs are used for the storage of the elements. Refer Fig. 9

B. Optimisation Techniques

- **Unrolling**: The first optimisation is to unroll the inner most loops completely so that all PEs are unrolled at the same time. This ensures parallel operations of PE and the system all together functions like a systolic array.
- **Array Partition**: To support the above, array partition of the local buffers is required. This maps the arrays to registers - FFs/LUTs. Hence the BRAM column in the final report is blank. Although array partitioning completely is advisable because BRAM percentages might get significantly high, it is not the case in this systolic array code. This is because we are only partitioning the internal buffers which are limited to a very small number of 6 per

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
+ convolution	Timing	-2.29	8641	8.641e+04	-	8642	-	no	10 (3%)	12 (5%)	15719 (14%)	14989 (28%)	-
o ITERATION_LOOP	-	7.30	8640	8.640e+04	576	-	15	no	-	-	-	-	-
+ read_data_DRAM	-	0.00	152	1.520e+03	-	152	-	no	-	9 (4%)	4774 (4%)	5182 (9%)	-
+ convolution_Pipeline_PE_LOOP	Timing	-2.29	364	3.640e+03	-	364	-	no	4 (1%)	3 (1%)	8179 (7%)	6197 (11%)	-
o PE_LOOP	II	7.30	362	3.620e+03	42	40	9	yes	-	-	-	-	-
+ processingelement	II	0.10	39	390.000	-	7	-	yes	4 (1%)	3 (1%)	7752 (7%)	5410 (10%)	-
+ convolution_Pipeline_VITIS_LOOP_116_1	-	3.16	6	60.000	-	6	-	no	-	-	72 (~0%)	115 (~0%)	-
o VITIS_LOOP_116_1	-	7.30	4	40.000	3	1	3	yes	-	-	-	-	-
+ convolution_Pipeline_VITIS_LOOP_125_2	-	3.16	6	60.000	-	6	-	no	-	-	72 (~0%)	115 (~0%)	-
o VITIS_LOOP_125_2	-	7.30	4	40.000	3	1	3	yes	-	-	-	-	-
+ convolution_Pipeline_VITIS_LOOP_134_3	-	3.16	6	60.000	-	6	-	no	-	-	72 (~0%)	115 (~0%)	-
o VITIS_LOOP_134_3	-	7.30	4	40.000	3	1	3	yes	-	-	-	-	-
+ write_data_DRAM	-	0.00	21	210.000	-	21	-	no	-	-	345 (~0%)	573 (1%)	-

Fig. 9. Baseline Results

PS: '+' for module; 'o' for loop; '*' for dataflow

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
+ convolution	-	0.00	2001	2.001e+04	-	2002	-	no	-	36 (16%)	58780 (55%)	44040 (82%)	-
o ITERATION_LOOP	II	7.30	1999	1.999e+04	180	130	15	yes	-	-	-	-	-
+ read_data_DRAM	II	0.00	128	1.280e+03	-	11	-	yes	-	9 (4%)	14777 (13%)	9264 (17%)	-
+ write_data_DRAM	II	0.00	9	90.000	-	3	-	yes	-	-	618 (~0%)	640 (1%)	-

Fig. 10. Optimised Results

== Bind Storage Report

Name	BRAM	URAM	Pragma	Variable	Storage	Impl	Latency
+ convolution	0	0					
psum_accumulation_0_fifo_U	-	-		psum_accumulation_0	fifo	srl	0
psum_accumulation_1_fifo_U	-	-		psum_accumulation_1	fifo	srl	0
psum_accumulation_2_fifo_U	-	-		psum_accumulation_2	fifo	srl	0
psum_DRAM_0_fifo_U	-	-		psum_DRAM_0	fifo	srl	0
psum_DRAM_1_fifo_U	-	-		psum_DRAM_1	fifo	srl	0
psum_DRAM_2_fifo_U	-	-		psum_DRAM_2	fifo	srl	0

Fig. 11. FIFOs as SRLs

PE and this is never going to go up. So the amount of resources assigned to this algorithm would stay fixed.

- **Pipelining**: The outer loop which iterates through the number of cycles required to produce results is pipelined. This ensures data reading from DRAM does not have to completely wait for one iteration of PE functional execution.
- **Use of FIFO storage**: Data structure elements like the PSUM summation storage, and DRAM transfer intermediate storage buffers which require only a single write and single read can be implemented using FIFO streaming buffers because the latency for this is lesser than BRAM (as shown in the figure Fig. 12).
- **Dataflow pragma**: PE buffers could not be converted into streaming buffers since they operate based on internal data re-use, and hence multiple read operations would be required. Read and write might happen within the same function. Due to this kind of multiple read/writes and also the feedback between functions in terms of cycles synchronisation - dataflow pragma for this design is not possible.

C. Optimised Design results

The optimised design has a total latency of 2001 cycles as compared to 8641 in the baseline code. As shown in fig.11, the DSP count has also increased to 36 from 12. This indicates the parallel operation of PEs in the systolic array. Since all the storage elements used in the design are fully partitioned, the synthesis has mapped these to LUT/registers hence the BRAM column in the synthesis report is blank.

This design produces significantly more improvement in comparison to the baseline as you begin scaling upwards in terms of the size of the input array with respect to both row and column.

VII. Scalability

- Convolution applications in real life contains input feature map size and output size of thousands of primitives. It is impractical to have a systolic array whose size is

Table: Supported Combinations of Memory Type, Implementation, and Latency

Type	Implementation	Min Latency	Max Latency
FIFO	BRAM	0	0
FIFO	LUTRAM	0	0
FIFO	MEMORY	0	0
FIFO	SRL	0	0
FIFO	URAM	0	0
RAM_1P	AUTO	1	3
RAM_1P	BRAM	1	3
RAM_1P	LUTRAM	1	3
RAM_1P	URAM	1	3
RAM_1WNR	AUTO	1	3
RAM_1WNR	BRAM	1	3
RAM_1WNR	LUTRAM	1	3
RAM_1WNR	URAM	1	3
RAM_2P	AUTO	1	3
RAM_2P	BRAM	1	3
RAM_2P	LUTRAM	1	3
RAM_2P	URAM	1	3
RAM_S2P	BRAM	1	3
RAM_S2P	BRAM_ECC	1	3
RAM_S2P	LUTRAM	1	3
RAM_S2P	URAM	1	3
RAM_S2P	URAM_ECC	1	3
RAM_T2P	BRAM	1	3
RAM_T2P	URAM	1	3
ROM_1P	AUTO	1	3
ROM_1P	BRAM	1	3
ROM_1P	LUTRAM	1	3
ROM_2P	AUTO	1	3

Fig. 12. FIFO vs BRAM latency

proportionally big to perform convolution over such a large dataset. This kind of one-to-one mapping of the dataset size to the systolic array also consumes a large amount energy and area.

- The better approach would be to have 2 primitive mapping stages:
- Logical Mapping - in logical mapping, it is assumed that the systolic array is as big as the one required to perform full convolution over the entire dataset.
- Physical mapping - in physical mapping, the logical PE array is folded so that it fits into the physical PE array.
- The design has a fixed systolic array size of 3x3 = 9 PEs.
- This can support input array of size 5x5, weight 3x3 and output array of size 3x3.
- The input array's width can be increased by increasing the total operating cycle count.But, to perform convolution on an input array with number of rows greater than 3, we need to instantiate the fixed size systolic array multiple times using FOLDING logic.
- Folding implies serializing the computation and is determined by the amount of on-chip storage.

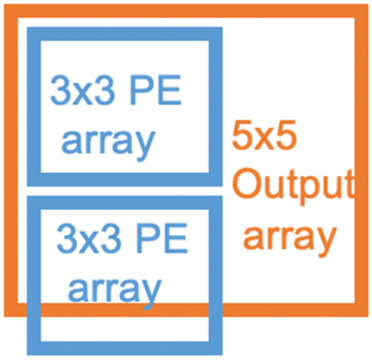

Fig. 13. Scaling by Folding

VIII. Future Work

Attempted a few optimisations that are beyond the scope of the timeline of this semester. These would be the ideal next steps to make this project more realistic.

- To attempt to use pipes for the internal buffers and perform non blocking reads and writes so that PE function is not sequentially blocked internally. This means that the MAC operations would not be blocked by data movement or vice versa within a PE. We are able to run c simulation on the same, however the RTL synthesis requires more time for analysis.
- To try to perform folding in a weight stationary manner; while performing folding to scale the design, the weights are passed in again along with the inputs. However, the weights assigned to each row would be the same even when the new inputs are passed in. So when reset() function is called for folding, the design can be changed such that it selectively clears the input data but not the weight data.
- To try to implement a design where the systolic array PE grid size is scalable i.e. filter size scalable.

References

[1] Yu-Hsin Chen, Joel Emer, Vivienne Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture.

[2] Yu-Hsin Chen, Joel Emer, Vivienne Sze, "Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators," IEEE Micro's Top Picks from the Computer Architecture Conferences, May/June 2017.

[3] Yu-Hsin Chen, Joel Emer, Vivienne Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," International Symposium on Computer Architecture (ISCA), pp. 367-379, June 2016.

[4] Yu-Hsin Chen, Tushar Krishna, Joel Emer, Vivienne Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," IEEE International Conference on Solid-State Circuits (ISSCC), pp. 262-264, February 2016.