Project Abstract

1. Keywords

Convolution, CNN, Weight Stationary, Accelerator Design, Handwriting Recognition

2. Research Motivation and Intent of the Project

Convolution computation is widely applied in signal analysis, natural language processing, and image recognition [1-3]. Convolutional neural network (CNN) is also constructed with several convolutional layers to extract features from inputs [4]. Consequently, we realized a computation system which accelerates the compute process specifically for convolution based on the concept of weight stationary.

3. Teamwork

Except for the Python code for validation test was done by Hsin-Ying Chiang alone, most of the research was finished by us together, including design realization, data arrangement, coding and other work.

4. Design Principles, Research Method and Steps

Convolution aims to extract features among smaller range of data. Convolutional computation starts with an input matrix and a weight matrix. The weight matrix would slide from left to right, up to down through the whole input matrix like a frame. Every time it slides from left to right within certain rows, it produces an element in the row of the result matrix by adding all the products of an act (input element) and a corresponded weight in the weight frame. When the weight frame slides to the rightmost of the input matrix, it goes back to the leftmost and moves downwards to form a new row of the result matrix. Take Figure 1 for example, the first element of the result matrix is computed as $1 \times 1 + 1 \times 0 + 0 \times 0 + 1 \times 1 = 2$.

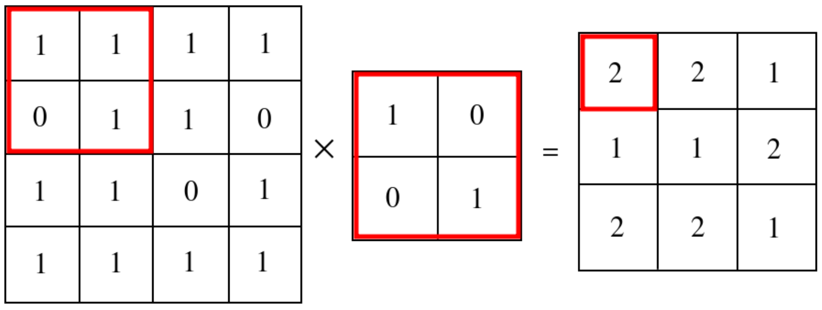


Figure 1 Convolutional Computation

By rearranging the computation order of the convolution, weight stationary is a kind of dataflow which allows the computation to spend less time and less energy consumption in a customized hardware system. Therefore, the computation system we designed is constructed based on the concept of weight stationary [5].

A. Brief Introduction of Weight Stationary

Weight stationary dataflow aims to minimize the energy consumption and the time it takes to read weights. To achieve the goal, the weights are stored in the registers near the ALU and maximize the access to the registers. By doing so, the times of access from upper part of the memory hierarchy can be reduced.

As shown in Figure 2, a PE (i.e., processing engine) is a processing unit with ALU and local registers. Weights are in the registers of PEs to make obtaining weight data more efficiently. To fit the structure, global buffer passes an act (i.e., activations, the input matrix elements) to each PE to compute psum (partial sum). To get the final result, psums need to be transported through PEs and accumulated to be the final sum.

In such a design, acts and psums need be accessed from global buffer more frequently than weight, which seems to be an inevitable issue. To deal with the problem, an improved compute system is designed, which can decrease the times of access to acts in global buffer and avoid the continuous transportation of psums through PEs.

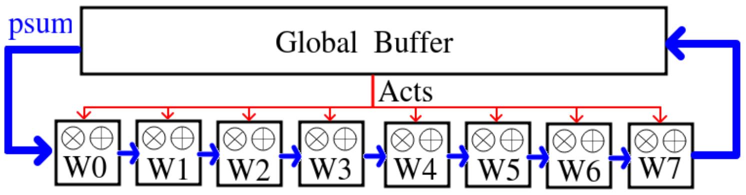


Figure 2 Structure of Weight Stationary from Fig.25(a) of [5]

B. Illustration of Our Design

To fit the usage of convolution, MNIST dataset is used as the input. By computing convolution of 28*28-pixel image input with 16*16 random weights, we can finally get a 13*13 matrix result. As shown in Figure 3, the 16*16 weight matrix is separated into 4 smaller 8*8 matrices, which take PEs 4 rounds of sliding through input matrix to finish computation. Weight data will first be passed to the global buffer and PEs. 64 weights will stay in the registers of PEs until all the related computation are finished. Therefore, we can enhance the reuse of weights.

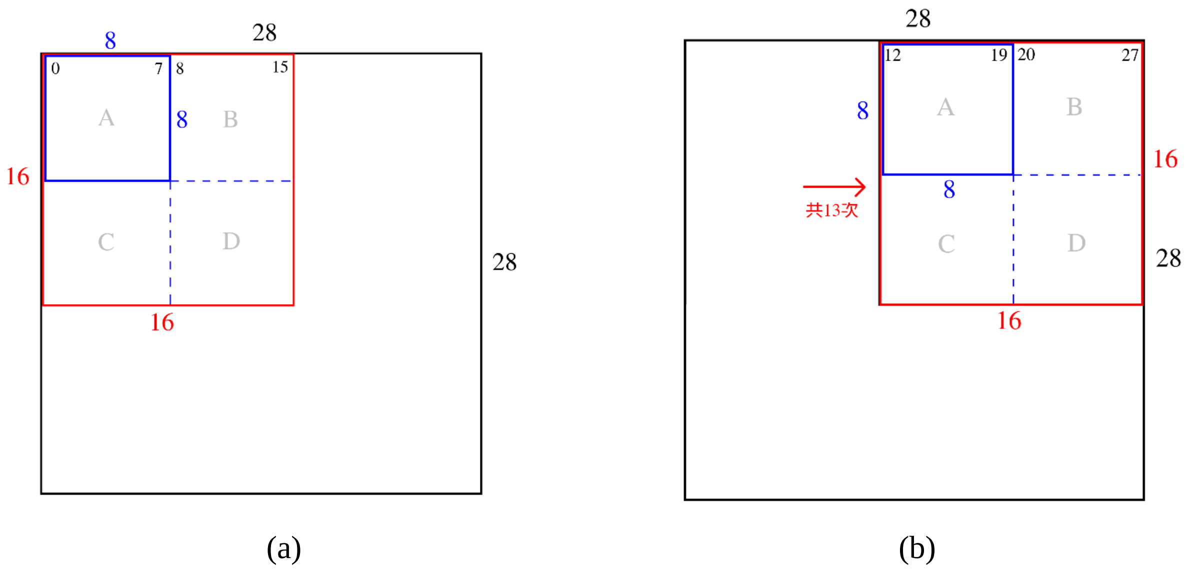


Figure 3 Sliding Weight Frame

Next, a feature of convolutional computation needs to be mentioned before illustrating the structure of PEs. During the weight frame sliding from left to right to obtain the next element of the result, only a column of acts is different compared to the acts used to compute the last adjacent element. There are columns of input data that can be reused in Figure 4. By utilizing the features, we can design a different structure for PE, but still preserve the weight stationary concept.

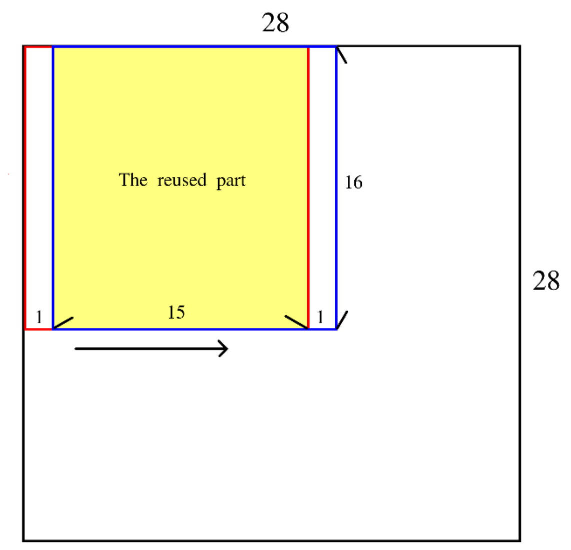


Figure 4 Sliding Way of Weight Frame

As shown in Figure 5, there are 17 registers in total in a PE, 8 for weights, 8 for acts, and 1 for the psum. 8 weight registers are used to store a row of weights in the 8*8 matrix. The other 8 act registers are used to store the corresponding act to each weight data. The acts would be multiplied with the weights and added to psum. With the design, we can decrease the times of accessing acts simply by shifting the acts in PEs left, and we only need to update 1 new act in each PE to compute the next element of the result.

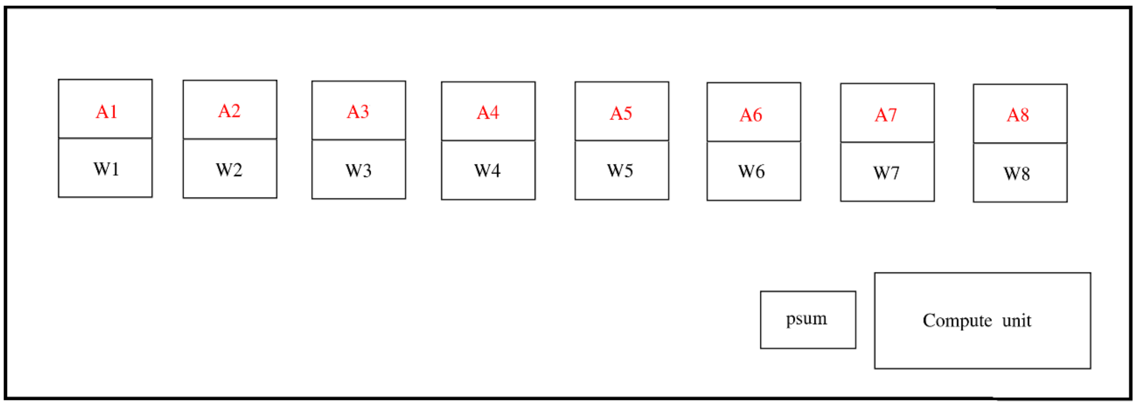


Figure 5 PE Structure

For each element in each round, psums in each PE are added together to get the completed psum of the round. In the end, the 4 psums of each round will be accumulated as the final result.

5. Realization and Experiment

The computation system consists of a memory, a global buffer, 8 PEs, a main controller, a PE controller, and some adders. The whole design is illustrated in Figure 6.

Memory keeps all the data we need and can exchange data with global buffer. Global buffer can store temporary data in need and offer a faster access to PEs. PEs are processing units that computes the partial sums for the result. Adders gather the partial sums all together to get the final result and store it back to the global buffer.

The main controller is the most complicated block in the diagram. It is charged of controlling the whole process of computation and it assigns almost all the control signals and the addresses to access data. PE controller can help the main controller to communicate with the 8 PEs. It assigns register indexes for PEs to do the computation or writing data in.

A. Block diagram

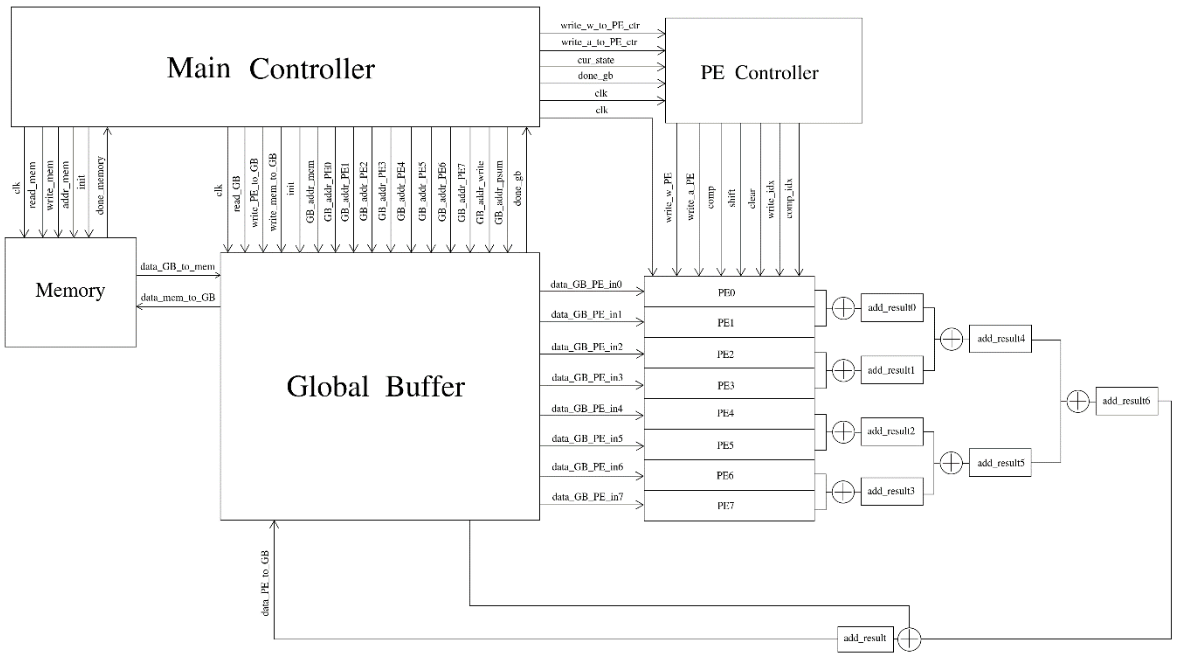
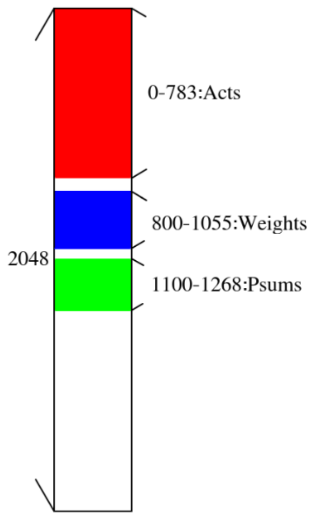


Figure 6 Block Diagram

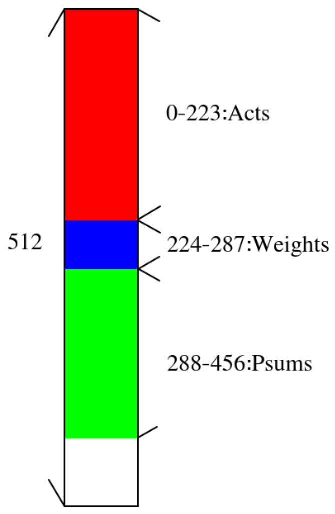
7 modules are coded in Verilog in this system, including the main controller, PE controller, memory, global buffer, PE, adders, and a module to combine them all. The main controller and PE controller are in charge of assigning control signals and addresses to other modules. Memory and global buffer are used to store the data. Adders can add all the partial sums computed by PEs together to get the final result.

B. Storage Structure of Memory and Global Buffer

Memory and global buffer can store 2048 and 512 64-bit numbers respectively. Different kinds of data are stored in fixed range of positions as shown in Figure 7.



(a) Data Storage of Memory



(b) Data Storage of Global Buffer

Figure 7 Data Storage of Memory and Global Buffer

Before the computation starts, global buffer stores the first 8 rows acts data of the 28*28 act matrix. As shown in Figure 8, the rows of these data are numbered for better explanation. The corresponding PEs on the right represents which PE each row data will be sent to. When a row of act data in Global Buffer is no longer needed, it will update the act data. Instead of reloading all 8 rows of acts again, it only needs one more new row updated since the weight frame slides down a row. As the example shown in Figure 8.a, after the computation of Row 0 to Row 7 have been finished, the next 8 rows of acts the weight frame will use is Row 1 to Row 8, so it replaces Row 0 to be Row 8 data as Figure 8.b. Following by that, again, after the computation of Row 1 to Row 8 has been finished, it replaces Row 1 to be Row 9 in Figure 8.c, and so on.

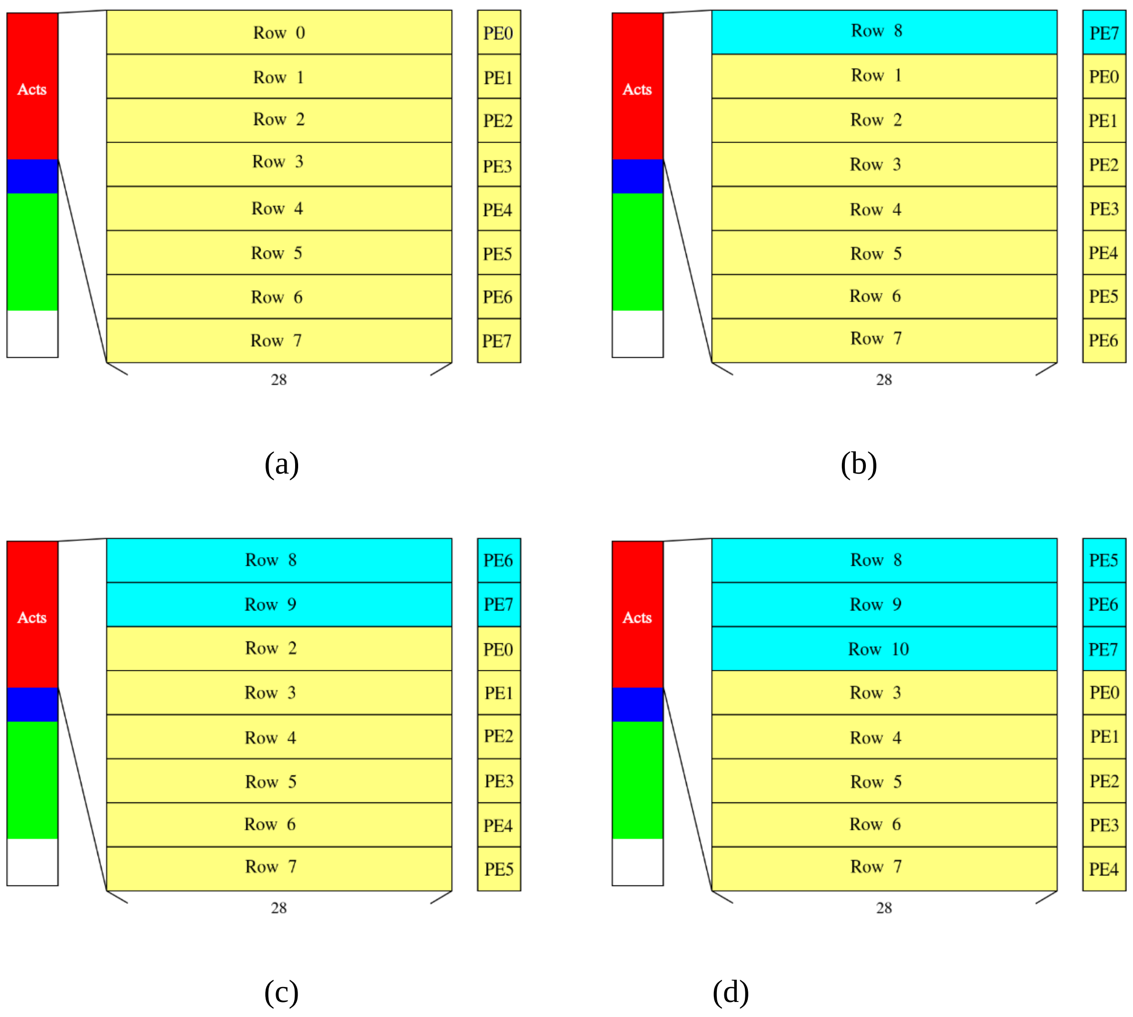


Figure 8 Arrangement of Act Data Storage

C. Working Process of the System

For the main controller, there are 11 states in total. In each state, it will give different control signals to memory, global buffer and PEs. Therefore, they can finish different work in different states as shown in Table 1.

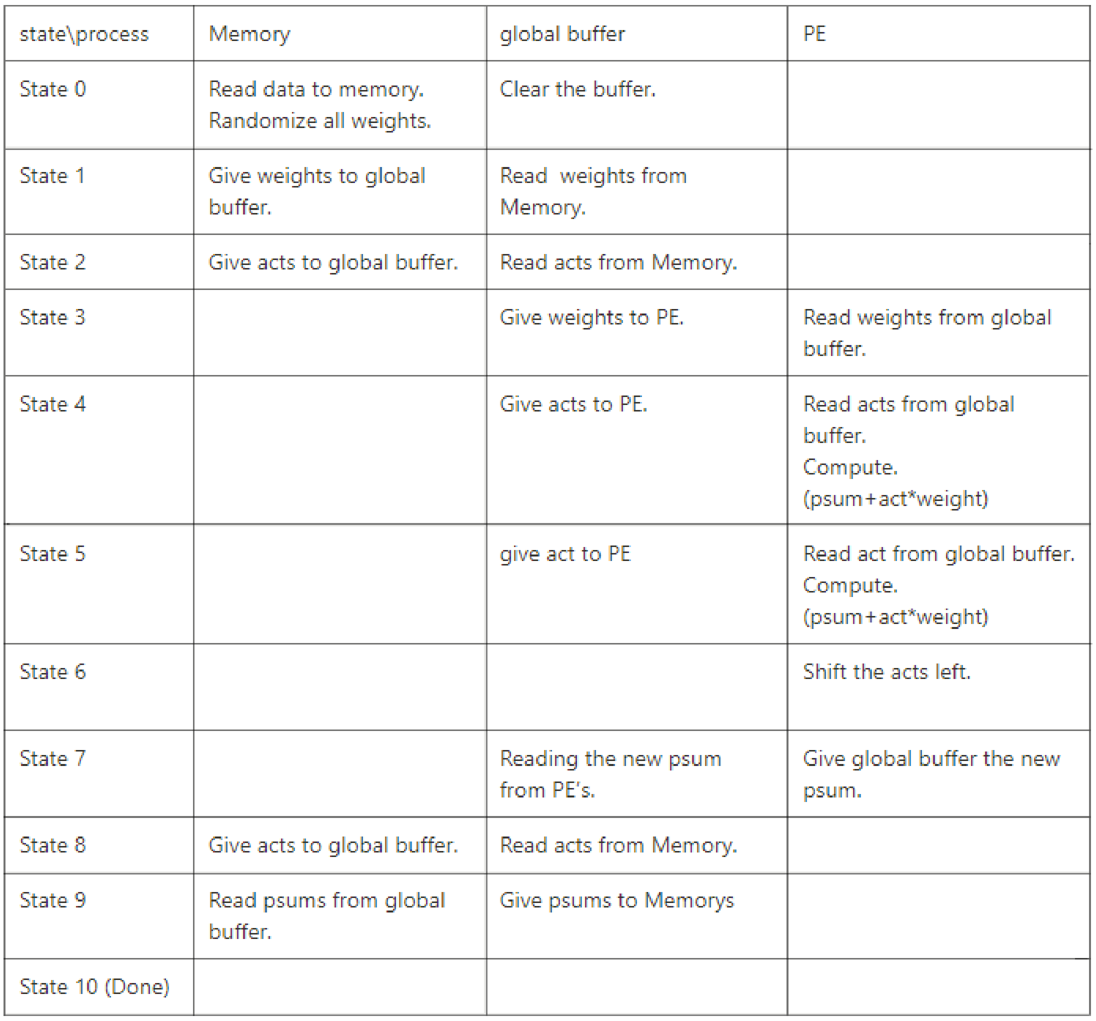
The system initializes in state 0, including preparing all the data in memory and clearing the global buffer. In state 1 and 2, it copies the weights it needs for the round and the first 8 rows of acts from memory to global buffer. To compute the convolution, each PE gets 8 weights for the round from global buffer in state 3.

In state 4, global buffer gives the leftmost 8 acts of each row to each PE. During each reading and writing time from global buffer to PE, PE can compute the product of other pairs of weights and acts which are already written to PE earlier. After the computation, each PE shifts the acts left in state 6. By doing so, after writing the psums to global buffer in state 7, it enters state 5 and reads only 1 more act from global buffer to get the next element of the result matrix.

When the weight frame moves to the rightmost position of the act matrix, it needs to write a new row of acts from the memory to compute the next row of the result. Therefore, it replaces the top row of acts in the global buffer with the 28 new acts during state 8, and then goes back to state 4 to continue computing.

It will enter state 9 when all 4 rounds of computation for different part of the weight matrix are finished. In state 9, it writes the final result back to the memory. After all the writing, the system enables the Done signal and stays at state 10 until it is turned off.

Table 1 States of the Main Controller



state\process	Memory	global buffer	PE
State 0	Read data to memory. Randomize all weights.	Clear the buffer.	
State 1	Give weights to global buffer.	Read weights from Memory.	
State 2	Give acts to global buffer.	Read acts from Memory.	
State 3		Give weights to PE.	Read weights from global buffer.
State 4		Give acts to PE.	Read acts from global buffer. Compute. (psum+act*weight)
State 5		give act to PE	Read act from global buffer. Compute. (psum+act*weight)
State 6			Shift the acts left.
State 7		Reading the new psum from PE's.	Give global buffer the new psum.
State 8	Give acts to global buffer.	Read acts from Memory.	
State 9	Read psums from global buffer.	Give psums to Memorys	
State 10 (Done)			

The process shown in Table 1 can be represented as a SM chart for the main controller as shown in Figure 9.

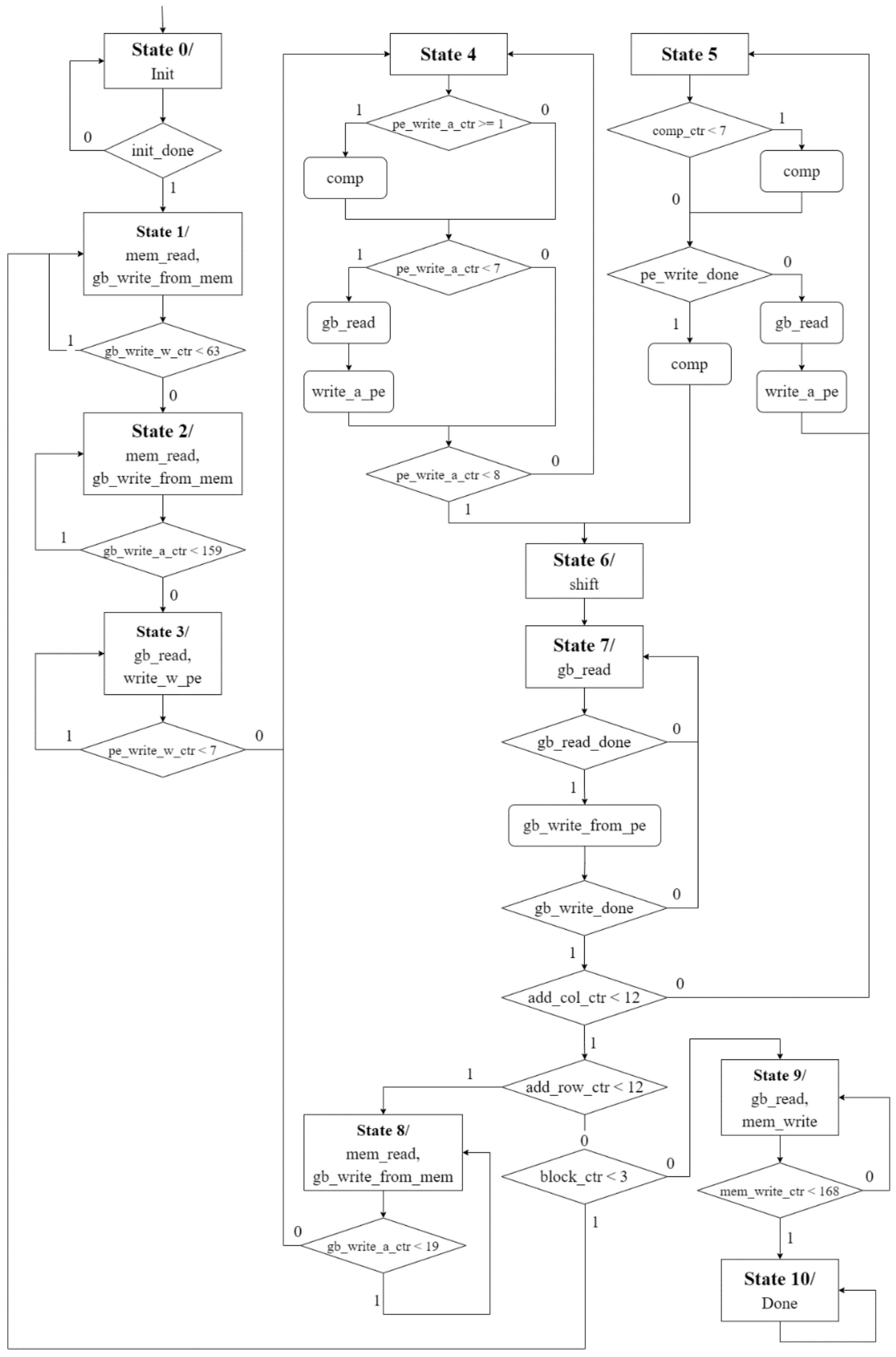


Figure 9 SM Chart of the Main Controller

6. Demonstration of the Result

The computation system is simulated under the environment of Vivado 2022.1 (64-bits).

A. Validation of Computation Result

The correctness of the result is validated by Python code in Python 3.8. The same convolutional computation is realized in Python, and it turns out that all the values correspond to the result of the hardware system has simulated.

B. Reuse of the Data

Table 2 reveals the reuse situation of weights and acts. The second column tells the total number of weight and act elements, and the following columns show how weight data and act data are accessed during the whole convolutional computation for a 28*28 input matrix in our design.

For each weight data, the total number of its usage is 169. However, for different act data, the number of times it is accessed depends on its position in the input matrix. According to the memory hierarchy, more accesses from registers and global buffer can result in less time and energy consumption. By the concept of weight stationary, 99.4% of weight usage is reused from registers which take less time to access.

For act data, the worst case occurs when it is accessed for the least time. It is accessed only once and it’s read from memory. The best case, in contrast, is read the maximum number of times. There is 80.77% of the act usage being read from registers. In average, an act data is used 55.18 times during the whole convolutional computation. It would be read 2.04 times from memory, 8.57 times from global buffer, and 44.57 times from PE registers.

Table 2 Reuse Situation of Weights and Acts

	total number	number of usage/ element	from memory	from global buffer	from registers
weights	256	169	1 (0.06%)	0 (0%)	168 (99.4%)
acts	784	Min: 1 Max: 169 Average: 55.18	Min: 1 (100%) Max: 4 (2.37%) Average: 2.04 (3.70%)	Min: 0 (0%) Max: 22 (13.02%) Average: 8.57 (15.53%)	Min: 0 (0%) Max: 143 (84.61%) Average: 44.57 (80.77%)

C. Efficiency

The efficiency of our design can be shown by comparing the time our system spends on computing a 13*13 result matrix from a 28*28 input matrix, with the time spent on a weight stationary system without act registers (i.e., without shifting acts). Take the time of computing one addition and one multiplication as a time unit, in which a shift or an access to PE registers can also be included. Assume that the time spent on accessing to memory and global buffer are M and G respectively. The time formulas of the 2 systems are as follow:

$$\begin{aligned} Total\ time = {} & Weight\ Access\ Time + Act\ Access\ from\ Memory\ Time \\ & + Act\ Access\ from\ GB\ Time + Compute\ Time \\ & + Psum\ Writing\ to\ GB\ Time + Result\ Writing\ Time \end{aligned}$$

$$Weight\ Access\ Time = 256(M+G)$$

$$Act\ Access\ from\ Memory\ Time = (20M \times 20) \times 4$$

$$Psum\ Writing\ to\ GB\ Time = 169(2G+1) \times 4$$

$$Result\ Writing\ Time = 169M$$

For the following description, we define our system as system (a), and the system without act registers as system (b). Except for the variables above, others represent different quantity of time units in system (a) and system (b).

For system (a):

$$Act\ Access\ from\ GB\ Time = (20G \times 13) \times 4$$

$$Compute\ Time = 13[1 + 12(8 - min(7, G))] \times 4$$

$$Shift\ Time = (12 \times 13) \times 4$$

For system (b):

$$Act\ Access\ from\ GB\ Time = (8G \times 169) \times 4$$

$$Compute\ Time = 169 \times 4$$

$$Shift\ Time = 0$$

For Act Access from GB Time in system (a), each time the weight frame slides from left to right, 20 acts are written from global buffer to PEs, and it contains 13 such actions in each round. Therefore, it will take $(20G \times 13) \times 4$ time units to access acts from global buffer. In system (b), since the acts in system (b) are never reused, it takes 8 time units each time to compute a psum for 169 result elements during the 4 rounds. So, system (b) will spend $(8G \times 169) \times 4$ time units for the act access from GB.

For Compute Time in system (a), in state 4, the compute time can be completely covered by act access time because a computation can always be completed while the next act is being written. Consequently, the only compute time that is counted is 1 time units for the last compute action. On the other hand, after shifting, an act access and 7 compute actions can be processed simultaneously, meaning that an act access action can cover several compute actions, depending on the magnitude of G. As the result, the time that compute actions will take is $(7 - min(7, G) + 1)$ It repeats 12 times as weight frame sliding from left to right, so the total time units spent in this part is $13[1 + 12(8 - min(7, G))] \times 4$. In system (b), the same as the state 4 in (a), only 1 compute action will be counted each time computing a psum, so the total time units spent in this part is 169×4.

For Shift Time in system (a), the act data shifts 12 times each time the weight frame slides from left to right, so the total time units shifting will take is $(12 \times 13) \times 4$. In system (b), there is no shifting action, so the total time units spent here is 0.

$$Total\ Time\ for\ system\ (a) = 2025M + 2648G + 6344 - 624min(7, G)$$

$$Total\ Time\ for\ system\ (b) = 2025M + 7016G + 1352$$

For example, if M = 100, and G = 10, then time spent on system (a) is $4368G - 624$ time units less than that spent on system (b), which is obviously a huge reduction on time. By adding act registers, our system may save 43,056 time units (i.e., about 15.71% of time taken by another system) in total compared to the system without the act registers.

7. Conclusion

Machine learning application requires many kinds of algorithm, and if not designed properly, it may cause the difficulty of practicing it. As a result, developing a suitable accelerator in hardware level is necessary, since it can expand the computation anility of computer, so that the software level be designed more freely. While studying for the project, we realize that before designing an accelerator for an algorithm, it is important to fully understand the whole computation process first. Ignorance to a detail may cause an absolute different result, which was why we redesigned the system for so many times. Although it was not easy for us, we still have learned a lot in this project, and we are both grateful to this opportunity of doing research on our own.

8. Reference

[1] S. G. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 11, no. 7, pp. 674-693, July 1989, doi: 10.1109/34.192463.

[2] E. R. Fonseca and J. L. G. Rosa, "A two-step convolutional neural network approach for semantic role labeling," The 2013 International Joint Conference on Neural Networks (IJCNN), Dallas, TX, USA, 2013, pp. 1-7, doi: 10.1109/IJCNN.2013.6707118

[3] R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, USA, 2014, pp. 580-587, doi: 10.1109/CVPR.2014.81.

[4] A. Shrestha and A. Mahmood, "Review of Deep Learning Algorithms and Architectures," in *IEEE Access*, vol. 7, pp. 53040-53065, 2019, doi: 10.1109/ACCESS.2019.2912200.

[5] V. Sze, Y. -H. Chen, T. -J. Yang and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE, vol. 105, no. 12, pp. 2295-2329, Dec. 2017, doi: 10.1109/JPROC.2017.2761740.