POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Implementation of a hardware accelerator for Deep Neural Networks based on Sparse Representations of Feature Maps

Supervisor

Prof. Maurizio MARTINA

Candidate

Yu HAO

March 2022

Summary

Deep learning, as one of the most currently remarkable machine learning techniques, has achieved great success in many fields such as speech recognition, image analysis, and autonomous driving. However, the neural network requires billions of multiply-and-accumulated operations, which makes the single-frame runtime enormous and energy-hungry. To optimize these imperfections, researchers from the University of Zurich and ETH Zurich developed a hardware accelerator named NullHop [1], which is a flexible and efficient hardware accelerator architecture aiming at exploiting the sparsity of neuron activations. NullHop uses a novel sparse matrix compression algorithm to encode the input data into two elements: a Sparsity Map (SM) and a Non-Zero Value List (NZVL). This scheme could enhance the overall computation time and energy consumption owing to two main features: 1) its ability to skip over zero-value pixels in the input layers without any wasted clock cycles and redundant MACs. 2) The compression scheme reduces the requirements of external memory and also the huge consumption brought with every memory access. This thesis work mainly targets implementing a hardware accelerator based on NullHop in Hardware Description Language (VHDL). The simulation results from ModelSim show that the accelerator could accomplish one input layer computation with dimension 6x6, 16 input channels, sparsity 84.375%, kernel size 3x3, 16 output channels, ReLU enabled, 2x2 max pooling enabled in 2029 clock cycles. If the input is 8x8x16, sparsity 81.05%, kernel size 3xx, ReLU and 2x2 max pooling enabled, 16 output channels the total time consumption is 4828 clock cycles. If the input layer is 64x64x16, sparsity 16.39%, kernel size 3x3, 16 output channels, the total time consumption is 738,834 cycles. The accelerator not only achieves a latency reduction thanks to the sparsity of input data but also reduces the workload of MACs since no zero-value pixel is forwarded to the computation unit(except for very few special purpose pixels). Meanwhile, the ReLU and max pooling are done on the fly during computation which could bring more enhancement. Furthermore, different output channels of the CNNs are calculated simultaneously, which gives the possibility to extend the accelerator's range of application but with negligible latency increment.

Acknowledgements

Firstly, I am very grateful to Prof. Maurizio Martina for giving me this opportunity to work on this thesis. It's my honor to have this valuable chance to study under your guidance. Thank you very much for supporting me and leading me to the right path when I'm confused, both in my study and life career.

Also many sincere thanks to Maurizio Capra, Emanuele Valpreda, Pierpaolo Mori' and Beatrice Bussolino for the technical support you offered during this thesis journey. Thank you for all the patience and all the very detailed explanations whenever I have problems.

Thanks to Annalisa Panarelli and Emanuele Rulfi. I'm truly glad to meet you and live with you these years. Thank you for everything. You are not just friends to me, but also my sister and brother.

Thanks to Bingquan Meng. All these years we have known each other from elementary school, you always stand with me and give me help without any reservation. Many thanks to Jiaqi Li, Chaojie Gao, Zhengliang Li, Manuele Damiani and Yuchen Xie. Thank you all for your support during all these years of my study in Italy. May our friendship last forever.

A huge thank you to Chen Xie. Thank you for your help and encouragement, and thank you for giving me directions when I'm stuck with problems.

Finally, I would like to thank my parents. Your patience and understanding are my strength to move on.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial Intelligence (**AI**) was first coined in the 1950s, and now it has become an essential ridgepole of many scientific systems and industries. AI works by collecting a massive number of data and analyzing with intelligent algorithms to allow the whole network to learn from data characteristics automatically.

Nowadays, AI is a broad field of study with several primary concepts, including Machine learning (**ML**), Neural Networks (**NN**), and Deep Learning (**DL**) as shown in Figure 1.1. Among these, the study of Deep Learning has far-reaching significance. The specific characteristic of Deep learning is that it trains the machine to solve the problem on its own decision. Deep Learning explores the hierarchical characterization of the model rather than the formulation and specification to learn data features. Meanwhile, DL empowers the system to learn from historical performance results to optimize its internal logic.

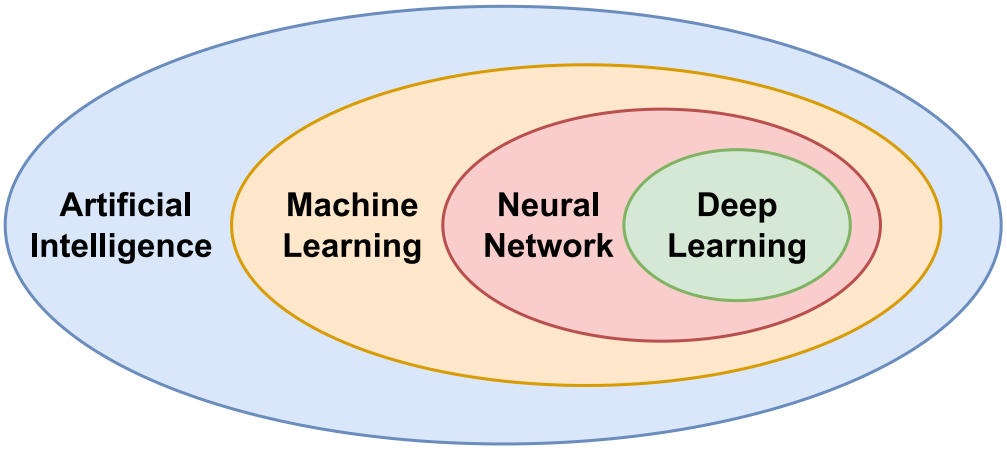

Figure 1.1: Artificial Intelligence Overview

Convolutional Neural Network (**CNN/ConvNet**) is a representative class

of Deep Learning models. CNN is a specialized type of neural network model designed for image recognition. Its essential operation is "convolution". The analysis procedure of CNN requires tons of multiply-and-accumulate (**MAC**) operations. The vast number of calculations makes the CNN model extremely energy-hungry and has enormous latency. Moreover, at least two numbers are fetched from memory for each MAC, which makes the energy and time consumption worse. Even if the CNN is performed with modern hardware platforms such as graphical processing units (GPUs), the performance is still unsatisfactory. For example, a 640x360 color input frame requires about 2 billion MAC operations. The NVIDIA Tegra X1 GPU does 60 billion operations per second (GOp/s) with a power consumption of about 10W, which means the GPU could process the input frame at a rate of 15Hz. The result shows that this GPU's power efficiency is about 6Gop/W, only about 6% of its theoretical maximum performance. [1]

Due to the tremendous calculation pressure and the efficiency limitation of commercial CPU/GPU, a hardware accelerator is one of the prime solutions to the current dilemma. The hardware accelerator implemented by this thesis is based on "NullHop" [1], which mainly focused on reducing the workload of Processing Elements (**PE**) by exploring the sparsity of neuron activations. The input feature maps are encoded with a particular algorithm to filter the zero pixels to let PEs process only non-zero pixels. Meanwhile, since the number of data forwarded to PEs is reduced, the impact due to every memory access is likewise less heavy.

The thesis is organized systematically in different chapters as follows:

- Chapter 2 reviewed the background of ML,NN and DL. The applied filed of CNNs, the architectures of CNNs, and their corresponding mathematic expressions are clarified in the following. Several techniques implemented by this accelerator to reduce hardware costs and maintain a reasonable accuracy are also explained.

- Chapter 3 interprets the architecture of this accelerator. Each hardware unit is be discussed from the architecture level. The Data Path and control units with Finite-state machines (**FSM**) are also explained.

- Chapter 4 presents the achieved results, and Chapter 5 is reserved for the conclusion.

Chapter 2

CNN Background

2.1 Machine Learning overview

Machine Learning has become a pillar industry in the development of modern society. This concept was first developed by Arthur Samuel, a computer scientist at IBM and a pioneer in AI. Samuel designed a computer program to play checkers. The program was unique in that the more it ran, the more it learned from experience, the more accurate predictions it could make.

There are five main steps of machine learning:

1. Identify relevant data set and prepare these data as input for further analysis.
2. Choose an appropriate type of Machine Learning algorithm
3. Build an analytical model based on the chosen algorithm in step 2.
4. Train the model with the data set chosen in step 1, revising it as needed.
5. Run the model to generate classification results, predictions, or other findings. See Figure 2.1.

Classical Machine Learning is often categorized by how its algorithm learns to predict more accurate or perform more reasonable. There are four basic approaches:

- Supervised Machine Learning: also known as supervised learning. It is defined since it uses labeled input data to train the analytical model to classify data or predict outputs accurately.
- Unsupervised Machine Learning: also known as unsupervised learning. It discovers hidden inference and data characteristics without any human intervention. This type of Machine Learning relies on unlabeled data to discover similarities and differences in information.

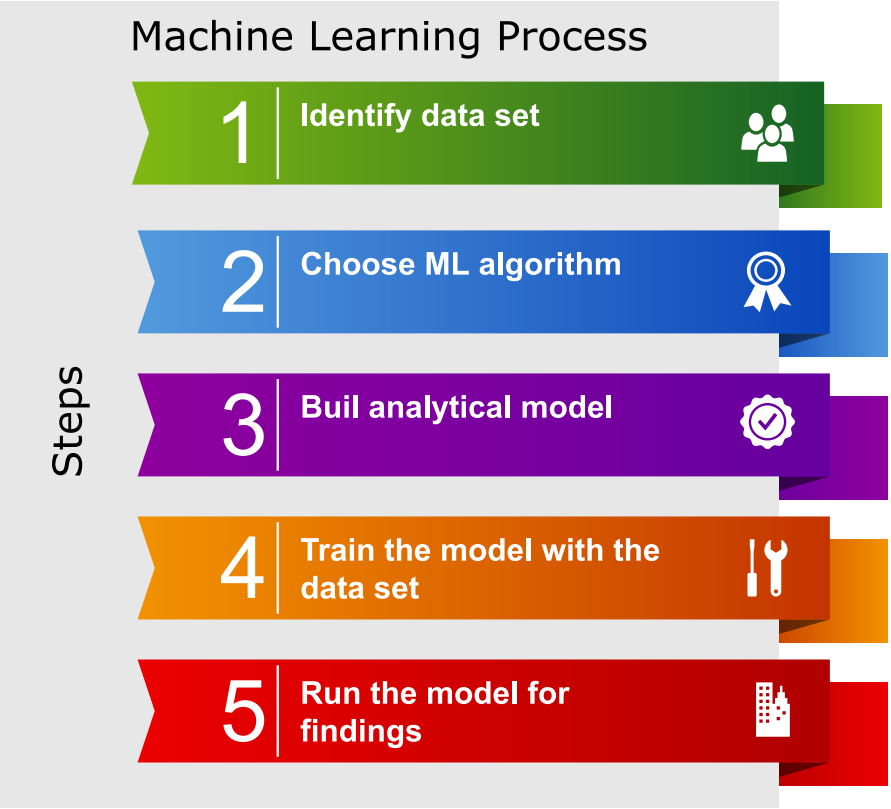

Figure 2.1: Machine Learning Steps

- Semi-supervised Machine Learning: This approach mixes the two preceding types. The model is trained with a smaller amount of labeled data and a large amount of unlabeled data for feature extraction, which makes it suitable for solving problems with insufficient labeled data or problems with heavy data pressure(because unlabeled data is less expensive and takes less effort to acquire).

- Reinforcement Machine Learning: This approach is similar to the supervised learning method but without sample data for training. It is trained through trial and error to take the best action by establishing a reward system. The reinforcement method is typically applied when training models to play games or training autonomous vehicles to drive.

Nowadays, Machine Learning is successfully employed in a broad field of human society and scientific area since it can analyze problems at a high speed and on a large scale. The applied field concludes classification, regression, clustering, or dimensionality reduction tasks of large sets of especially high-dimensional input

data. Vast parts of our daily life, for example, image and speech recognition, web searches, fraud detection, email/spam filtering, credit scores, and many more, are powered by Machine Learning algorithms [2].

2.2 Neural Network Overview

Neural networks, also known as Artificial Neural Networks(**ANN**s) or simulated neural networks(**SNN**s), are a subset of Machine Learning and also the essential part of Deep Learning. Neural networks mimic the human brain, composed of four main parameters: inputs, weights, a bias or threshold, and an output, as shown in Figure 2.3. The algebraic formula is as Equ 2.1:

$$\textbf{Output} = \sum_{i=1}^{N} w_i x_i + bias \tag{2.1}$$

The architecture of NNs are comprised of an input layer, one or more hidden layers, and an output layer. The input layer receives input data, the output layer makes a classification or prediction about the input data, and the hidden layers are considered as the networks' core engine to do the analysis. What differentiates a deep neural network and a basic neural network is the number of hidden layers. Usually, a deep neural network has more than three hidden layers, while a basic neural network has at most two or three hidden layers, as shown in Figure 2.2 [3].

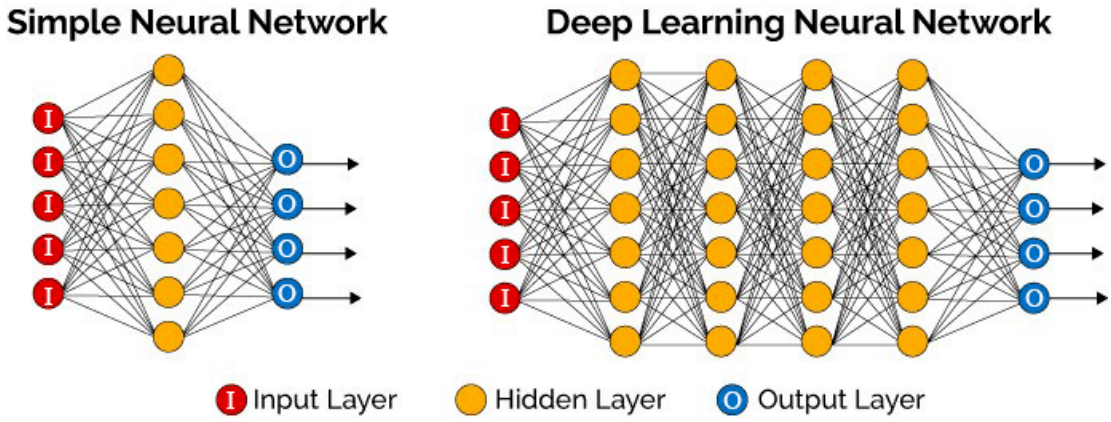

Figure 2.2: Comparison between normal neural network and Deep Neural Network [3]

There are three main categories of neural networks:

- Feedforward neural networks: also known as multi-layer perceptrons(**MLP**s). Feedforward refers to the analysis direction flows from the input layer to

the output layer. There are also various activation functions applied to determine the output of MLPs. These activation functions will be explained later in Section 2.4.1. Meanwhile, neural networks can be trained through backpropagation, which means moving in the opposite direction from output to input. This methodology is able to calculate and attribute the error of each neuron, allowing further adjustments and result improvements.

- Convolutional Neural Networks: They are similar to feedforward networks. This concept will be later explained in Section 2.4.
- Recurrent neural networks(**RNN**s): They employ sequential or time-series data and feed the output from the previous step as input to the current stage. Due to the particular input feature, RNNs are primarily leveraged in predicting future outcomes, such as stock market predictions or sales forecasting.

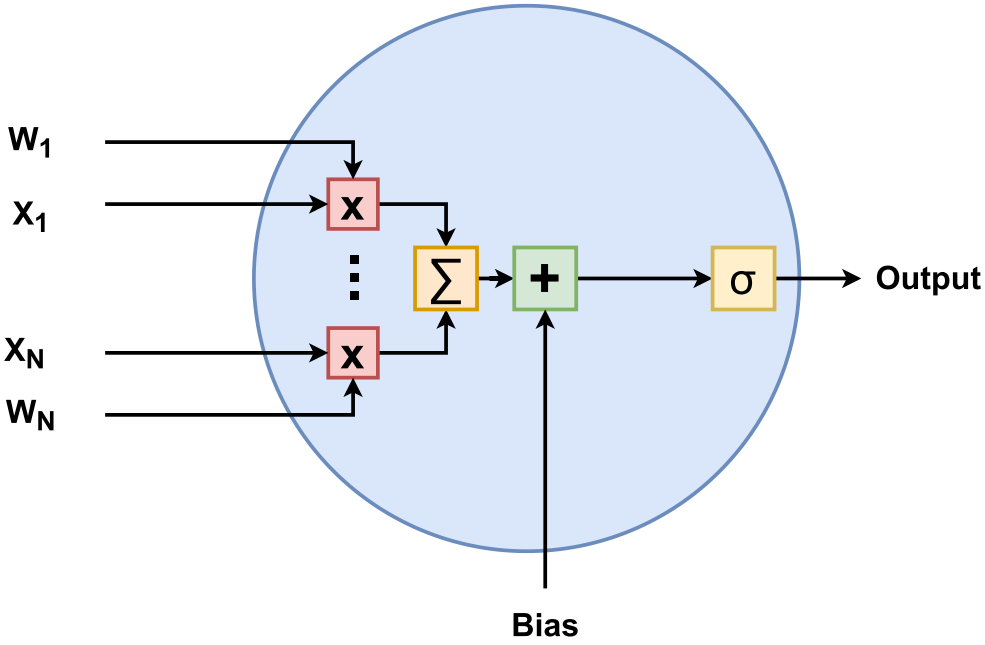

Figure 2.3: Model of an artificial neuron

2.3 Deep Learning Overview

Figure 1.1 shows that Deep Learning is indeed a sub-field of Neural Network and Machine Learning. Deep Learning is defined as a neural network that is composed of multiple hidden layers, and these layers are specialized to represent data abstractions.

Meanwhile, DL inherits the ability to imitate patterns that human brains does for making decisions. Nevertheless, Deep Learning model is highly time-consuming in analyzing and training process, it still remains superior in analysis and prediction performance compared to other Machine Learning algorithms [4][5], as shown in Figure 2.4 [6].

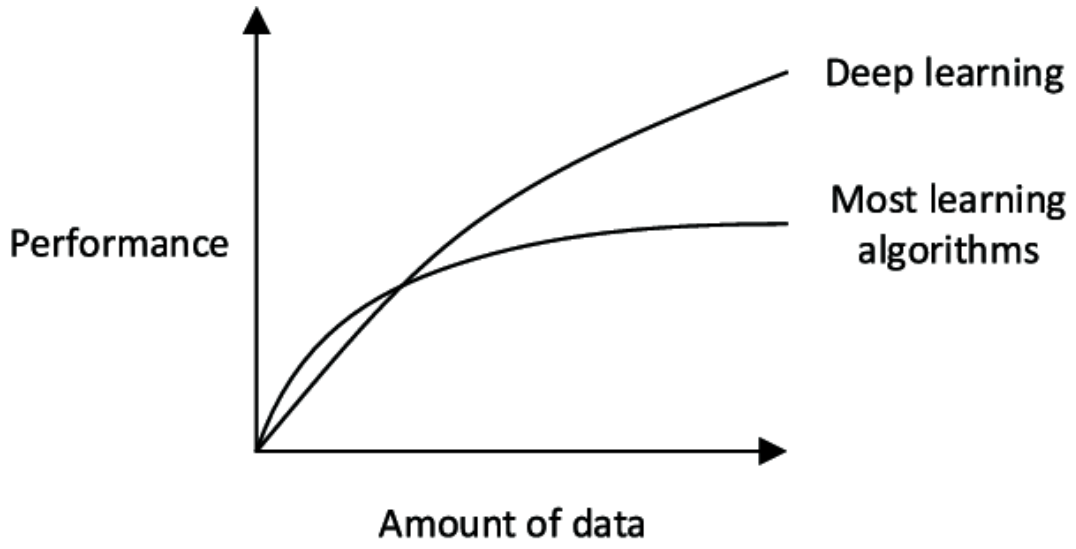

Figure 2.4: Comparison between performance of Deep Learning against other Machine Learning algorithms. Results prove that Deep Learning benefits from large amounts of data, whereas the performance increase of other machine learning models plateaus. [6]

2.4 CNN Overview

Aforementioned, Convolution Neural Network is a popular discriminative Deep Learning architecture with a deep feed-forward architecture and an astonishing ability to generalize efficiently compared to networks with fully connected layers. The superiority of CNN relies on the concept of weight sharing, which reduces the number of parameters used during training. Due to lesser parameters, CNNs can be trained smoothly and does not suffer overfitting [7]. CNNs are widely applied in various fields, such as:

- Image classification: The researchers from University of Toronto trained a large deep convolutional neural network to classify 1.2 million high-resolution images from the ImageNet LSVRC-2010 contest into 1000 different categories.

On the test data, the CNN model achieved top-1 and top-5 error rates of 37.5% and 17.0%, which is better than the previous state-of-art at that time. [8]

- Vehicle Recognition: The researchers from University of Electronic Science and Technology of China proposed a nine-layer neural network focusing on computer vision for vehicle recognition. By employing *Caffe* framework, the top-1 accuracy about the nine-layer network reached 92.25%, and the top-5 accuracy reached 97.51%. [9]

- Speech recognition: The researchers from IBM showed a tremendous improvement on the order of 10-30% in acoustic modeling for speech recognition with neural networks. This research also confirmed experimentally, with CNNs showing improvements in word error rate between 4-12% relative compared to other DNNs across different LVCSR tasks. [10]

- Medical imaging and diagnostics: The research from University of Liverpool trained a convolutional neural network using a high-end graphics processor unit on the Kaggle dataset to diagnose diabetic retinopathy from digital fundus images and accurately classify its severity. On the data set of 80,000 images used, the CNN model achieved a sensitivity of 95% and an accuracy of 75% on 5,000 validation images. [11]

- Facial expression recognition: The researchers from Universiti Brunei Darussalam used a CNN model to perform automatic facial expression recognition (AFER) on the Aff-Wild2 dataset and achieved an accuracy of 50.77% and an F1 score of 29.16% on the validation set. [12]

CNNs extract high-level characteristics from input images with three main common network layers: Convolutional Layer, Pooling Layer, and Fully Connected Layer. The Convolutional Layer and Pooling Layer perform feature extractions. The Fully Connected Layer maps the extracted features into final output. The internal connections between stages are shown in Figure 2.5 [13].

2.4.1 Convolutional Layer

The Convolutional Layer is the essential stage of a Convolution Network that does most computations. The CNNs analyze the input images as one or more matrices of pixels, depending on the input channel depth. For a gray-scale image, the channel number is one. A typical RGB image comprises three input feature map channels: red, green, and blue color channels, as shown in Figure 2.6. This input is treated as three matrices of size 3x3.

The general formula to calculate output feature map's is as Equ 2.2 [14]:

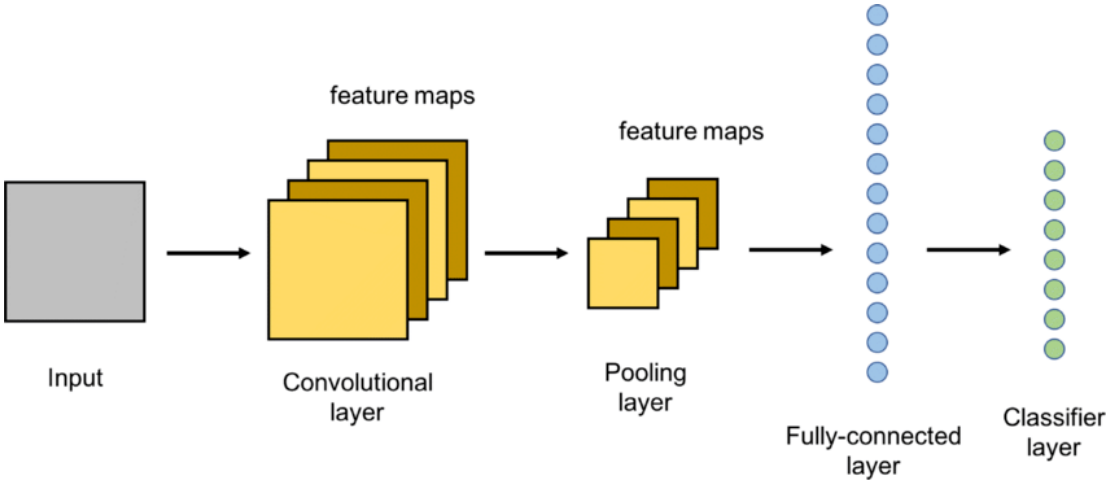

Figure 2.5: Full construction of CNN model [13]

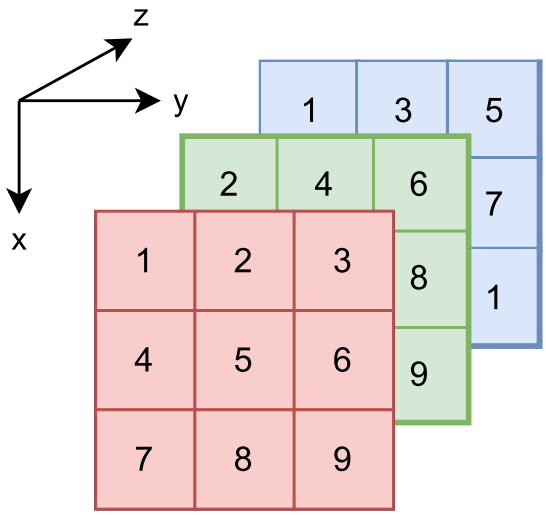

Figure 2.6: RGB image

$$\mathbf{Ofm}[c_o, h_o, w_o] = \sum_{c_i=0}^{C_i-1} \sum_{h_k=0}^{H_k-1} \sum_{w_k=0}^{W_k-1} \times (\mathbf{W}[c_i, c_o, h_k, w_k])\mathbf{Ifm}[c_i, Sh_o + h_k, Sw_o + w_k] + \mathbf{b}[c_o]) \quad (2.2)$$

$$0 \leq c_o < C_o,\ 0 \leq h_o < H_o,$$
$$0 \leq w_o < W_o,\ 0 \leq h_k < H_k,\ 0 \leq w_k < W_k$$

It describes an input feature map **Ifm** with C_i channels; each channel is a feature map of size $[H_i \times W_i]$, interacting with kernel weights **W** of size $[C_i \times C_o \times H_k \times W_k]$, and a bias term **b** of size $[C_o]$. The distance between adjacent receptive fields is defined by a stride parameter **S**. The filter's output **Ofm** is the convolution output, which is also a 3D matrix of size $[C_o \times H_o \times W_o]$. Figure 2.7 presents the pseudo code of a Convolutional Layer.

```
for (n=0; n<N; n++)
  for (co=0; co<Co; co++)
    for (ci=0; ci<Ci; ci++)
      for (ho=0; ho<Ho; ho++)
        for (wo=0; wo<Wo; wo++)
          for (hk=0; hk<Hk; hk++)
            for (wk=0; wk<Wk; wk++)
                Ofm[n][co][ho][wo] += Ifm[n][ci][ho+hk][wo+wk] * W[co][ci][hk][wk]
```

Figure 2.7: Pseudocode of Conv Layer

If the input feature map is a 6x6 RGB image for example, the input is treated as a 3D array with three matrices from the mathematical point of view, each matrix is of size 6x6. Imagine the filter is of size 3x3, since the input is three-channel type, therefore each filter set is composed of three matrices, each matrix is of size 3x3. If there exists in the filter bank four sets of filters, the output feature map will be a four-channel feature map, each channel is a 4x4 matrix as shown in Figure 2.8 [1].

The CNNs are simple cascades of linear algebra operations without a non-linear activation function. To solve the complex non-linear problems, an optional non-linear activation function could be applied to the result of the Conv layer. Some of the most popular functions are:

1. *Rectified Linear Unit*(**ReLU**) is a typical and simple activation function that forces the activations to be greater or equal to zero. See Equ 2.3.

$$y = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases} \tag{2.3}$$

2. *Sigmoid* function normalizes the output into range (0,1). It's more computationally expensive compared to ReLU. See Equ 2.4.

$$y = \frac{1}{1 + e^{-x}} \tag{2.4}$$

3. *Hyperbolic Tangent* function is the extended version of *Sigmoid* function, with the activations in the range of (-1,1), as shown in Equ 2.5.

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

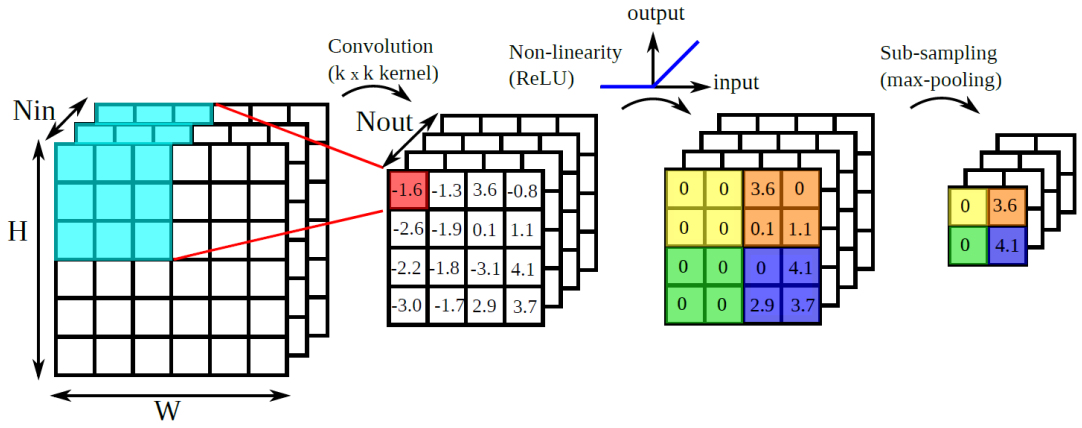

Figure 2.8: CNN main processing stages [1]

2.4.2 Pooling Layer

Pooling Layers, also known as down-sampling, conducts dimensionality reduction to reduce feature map redundancy and network computation complexity. There are two main types of pooling:

- Max pooling: An additional filter moves across output feature maps to select the pixel with the maximum value. An example of a 2x2 non-overlapping pooling stage is shown in Figure 2.9, which reduces the feature maps from 4x4 into 2x2.
- Average pooling: instead of obtaining the maximum value, the average pooling calculates the average value within a particular range inside the current feature map. See Figure 2.9.

2.4.3 Fully Connected Layer

The Fully Connected Layer(**FC**) is generally placed at the end of CNNs to convert the two-dimensional feature map into a one-dimensional output feature map. This layer combines all the features learned by the previous layers across the image

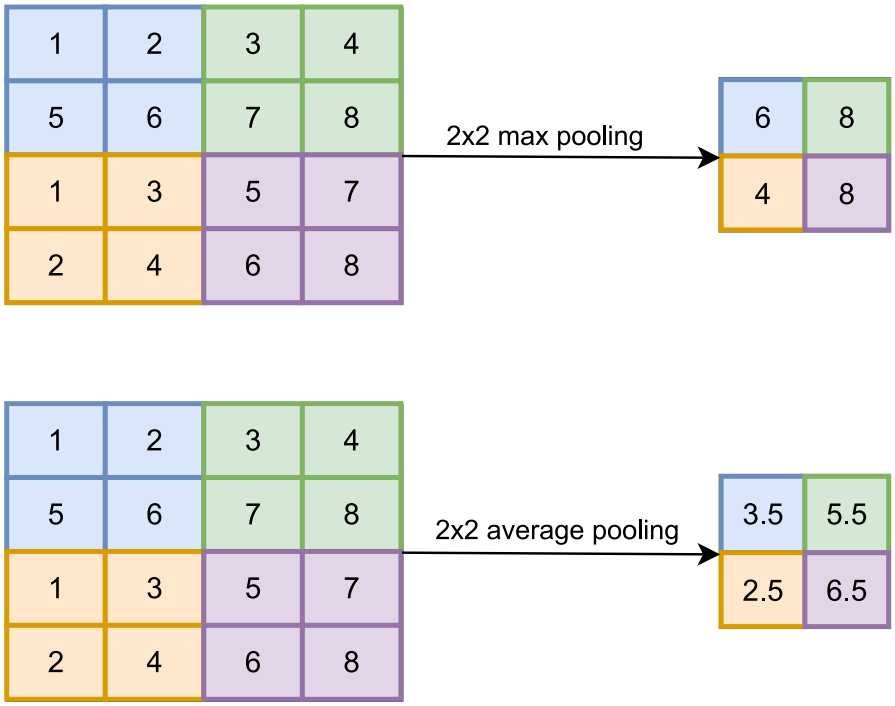

Figure 2.9: Pooling layer

to identify the larger patterns. Equ 2.6 [14] explains the calculations performed during FC layer, where C_i and C_o are the number of neurons of two consecutive layers. It also shows that an FC layer is a vector-matrix multiplication with the weights arranged in a $C_i \times C_o$ matrix(see Figure 2.10).

$$\mathbf{O}[c_o] = \sum_{c_i=0}^{C_i-1} \mathbf{W}[c_o, c_i]\mathbf{I}[c_i] + \mathbf{b}[c_o] \qquad 0 \leq c_o < C_o \quad 0 \leq c_i < C_i \tag{2.6}$$

2.5 CNN Variants

2.5.1 VGGNet

VGGNet is a Convolutional Neural Network architecture introduced by Karen Simonyan and Andrew Zisserman in 2014. It is a pioneer in exploring how the depth of the network influences the performance of a CNN. The researches demonstrate that the representation depth is beneficial for the classification accuracy. [15]

VGG16 is comprised of 13 convolutional layers and 3 Fully Connected layers. The input to VGG16 is a 224x224 RGB image; the kernel size is 3x3. Further

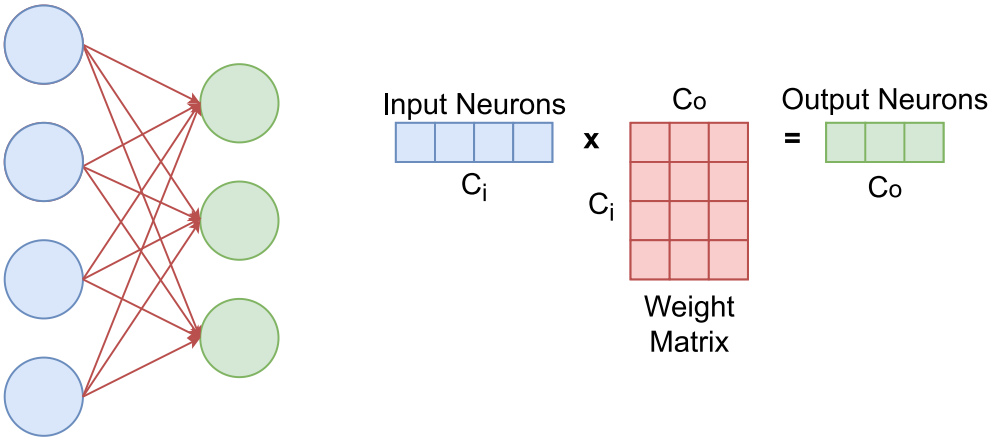

Figure 2.10: Example of an FC layer and its transformation with vector-matrix multiplication

details are shown in Figure 2.11 [16]. VGG19 extends the network to 19 layers in total, which is also the winner of ILSVRC-2014. According to the reported results from their research, by increasing the layer depth from 11 to 19, the error rate drops from 29.6% to 25.5% regarding top-1 val.error on the ILSVRC dataset in ILSVRC2014. [15]

Figure 2.11: VGG16 Architecture [16]

2.5.2 GoogLeNet

GoogLeNet is a deep Convolutional Neural Network proposed by researchers from Google in the ILSVRC2014. The network is 22 layers deep when counting only layers with parameters. The overall number of layers employed is about 100. Its architecture is shown in Figure 2.12 [17]. The network uses an average pooling layer before the classifier, instead of fully connected layers. This modification improves the top-1 accuracy by about 0.6%. GoogLeNet ranked the first among other participants with a top-5 error of 6.67% on both the validation and testing data during the ILSVRC2014. [18]

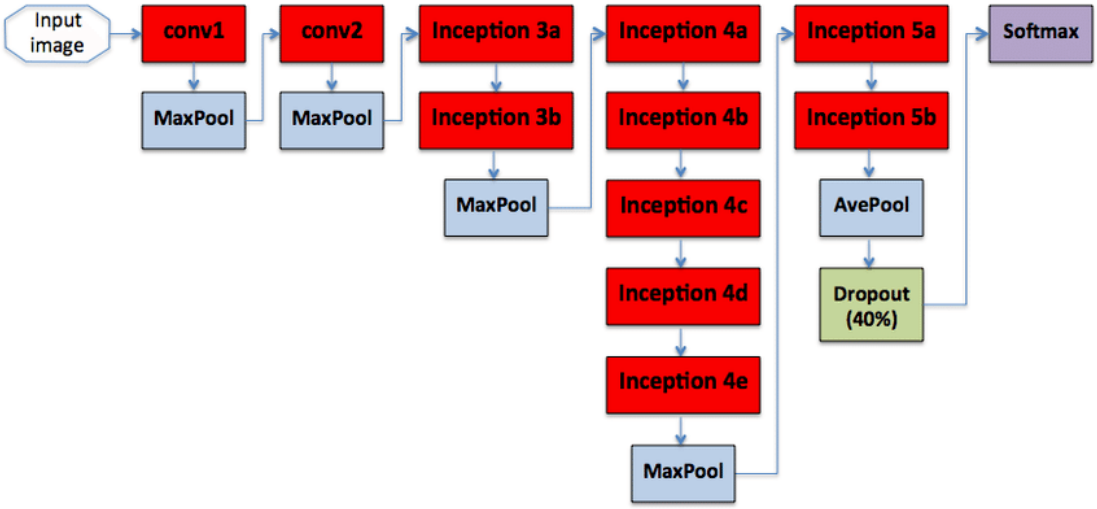

Figure 2.12: GoogLeNet Architecture [17]

2.5.3 ResNet

ResNet is proposed to overcome the degradation problem of CNNs when its depth reaches limit. The CNN's performance saturates or even starts degrading when the layer number is too high, as shown in Figure 2.13 [19]. Because the gradient is back-propagated to earlier layers, repeated analysis may cause the gradient to get smaller. The novel ResNet is, in theory, capable of being extended to an infinite depth without losing accuracy. According to the result from [19], the 152-layer ResNet reaches a single-model top-5 validation error of 4.49%. Remarkably, the 50/101/152-layer ResNets are more accurate than the 34-layer one. No degradation problem has been observed, and significant accuracy gains from the increased depth. One version of ResNet is shown in Figure 2.14 [20].

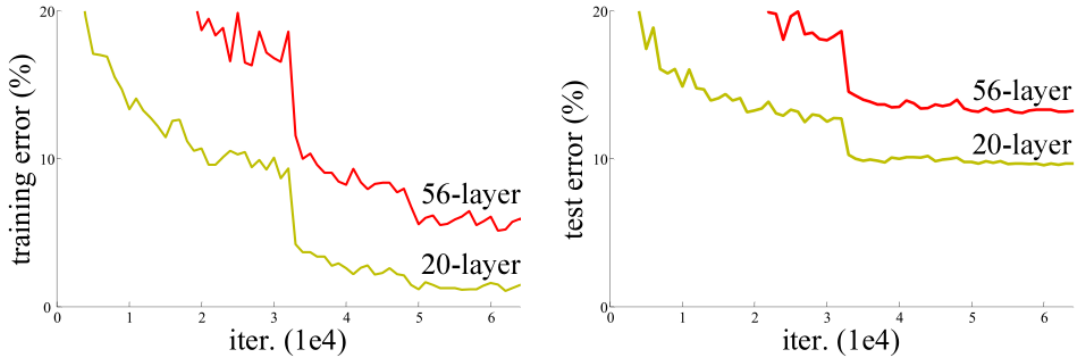

Figure 2.13: Increasing network depth leads to worse its performance [19]

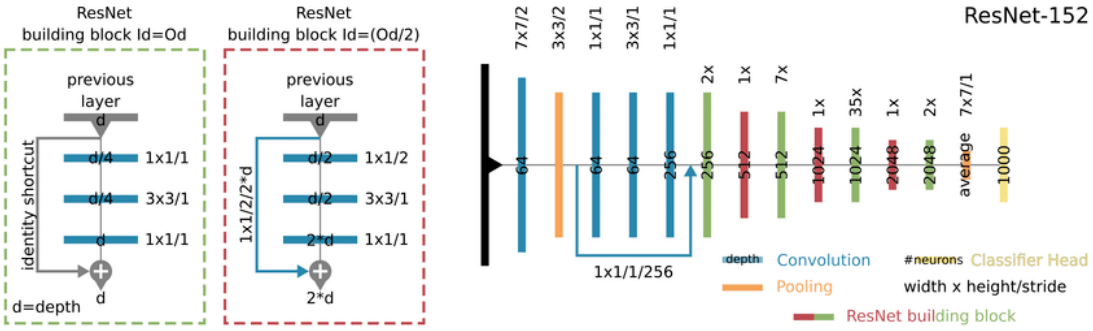

Figure 2.14: ResNet-152 Architecture [20]

2.6 Preprocessing for CNN on accelerator

2.6.1 Temporal hardware accelerator platform

Temporal architectures are usually employed on general-purpose platforms, such as CPUs and GPUs.

CPUs have multiple Arithmetic Logic Units(**ALU**s) that work synchronously and perform an instruction on a vector of data. However, they are still the least used for CNN inference or training due to the disappointing FLOPS and FLOPS/WATT performance compared to other hardware platforms. [14]

GPUs are manycore architectures that are specifically designed for parallel computation. Each core processes individual data that belongs to multiple threads running simultaneously. GPUs are more efficient in DNNs training.

The performance of commercial CPUs and GPUs have kept improving in these decades. Nevertheless, general-purpose platforms are not very prone to use sparsity as an advantage. Thus, many FPGA and ASIC architectures leverage sparse

matrices to accelerate the inference stage thanks to custom hardware. By exploiting a suitable coding algorithm, the accelerator performance could be even further advanced. This is the starting point of this thesis – to design a configurable, independent hardware accelerator architecture, working with Sparse Matrix coding algorithm, accelerating the CNNs training process using the sparsity of input data.

2.6.2 Reduced precision CNN

To solve the enormous energy consumption brought with every convolution operation, the precisions of activations and weights are limited. Meanwhile, the reduced data precision also benefits the possible sparsity. However, the reduced precision should also be sufficient to maintain a satisfying accuracy of the network. From the researches [1][21][22], a custom branch of Caffe called ADaPTION is developed to train networks from scratch and fine-tune existing 32-bit floating-point networks to any customized precision for the parameters adopted during convolution using the power2quant algorithm. This methodology achieved VGG16 67.5% Top-1 accuracy after quantizing the weights and activations from 32-bit to 16-bit, with only 0.8% compromised accuracy. Another surprising benefit is the increased sparsity brought by the reduced precision. The average sparsity among all layers is increased from 57% with 32-bit precision to an extraordinary 82% with 16-bit reduced precision. Maximum activation sparsity growth reaches nearly 50% per layer, as shown in Figure 2.15 [1]. To balance the trade-off between accuracy and hardware complexity, a 16-bit fixed-point precision is adopted for the entire accelerator design.

2.6.3 Sparse Matrix Compression Algorithm

The CNN hardware accelerator implemented in this thesis uses a novel sparse matrix compression scheme. In Machine Learning, sparsity refers to the percentage of zero values among all values inside input data. The research on sparsity is beneficial since zero value will not significantly impact a calculation. The Sparse Matrix Compression Algorithm codes the input feature maps into two elements: a Sparsity Map(**SM**) and a Non-Zero Value List(**NZVL**). The SM is a 3D matrix with the same number of elements as the input feature maps. The only difference is that the values inside SM are only 0s and 1s. The SM could be considered as replacing every non-zero value inside the input feature maps with 1 but maintaining the pixel's original spatial coordinates. Equ 2.7 shows the mathematical relationship between the original input data and SM.

$$SM(x,y,z) = \begin{cases} 1 & \text{input}(x,y,z) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

For the simplicity of explanation, in the following part of the thesis, the pixel's

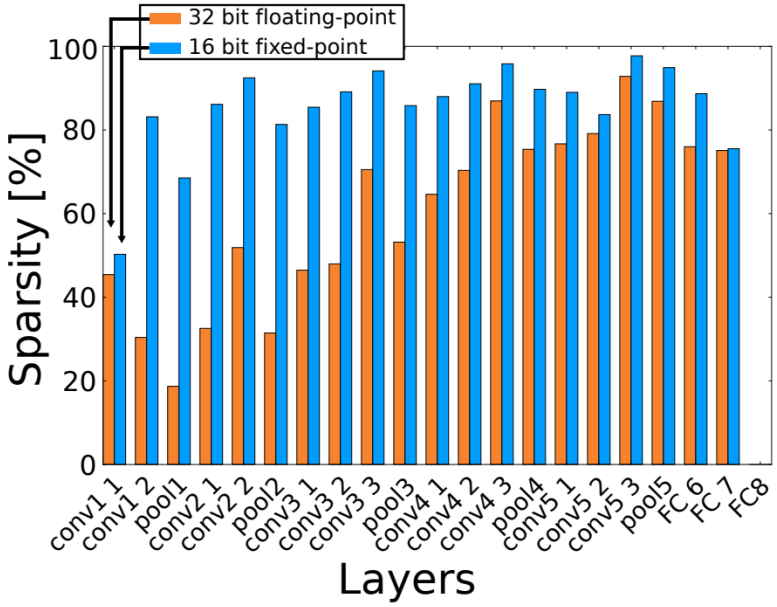

Figure 2.15: Sparsity before (orange) and after (blue) activations are quantized to 16-bit fixed-point for VGG16 layers. Average over 1000 ImageNet images. [1]

spatial coordinates are represented as in Figure 2.6: x is the row number of the pixel, y is the column number, and z is the input channel index.

The SM is applied to indicate the spatial coordinates of every non-zero pixel. And the NZVL stores the pixel's original value. The advantage of this coding scenario is the enormous data size reduction. For the example in Figure 2.16, the original input feature map is of size 4x4, if the pixel value is in 16-bit representation, this input data size is 256 bits. However, with the SM & NZVL compression scheme, the input map comprises a 16-bit SM and three pixel values. The total input data size is only 64 bits if pixel values are in 16-bit representation, which is only a quarter compared to the original data size.

The SMs and NZVLs are streamed into the accelerator as shown in Figure 2.17. To reduce the complexity during the decode phase, the SM 3D array is split into 16-bit segments, the same size as the pixel value representation. The very first 16-bit data sent by the input data bus is always a SM segment. If this SM segment has sixteen 0s, the second 16-bit data of the bus will be another SM. Otherwise, the following $N * 16$ bit data will be non-zero pixel values in binary codes, N is the number of 1s inside this SM. Figure 2.17 presents conditions that SMs have all 0s and SMs have actual pixels.

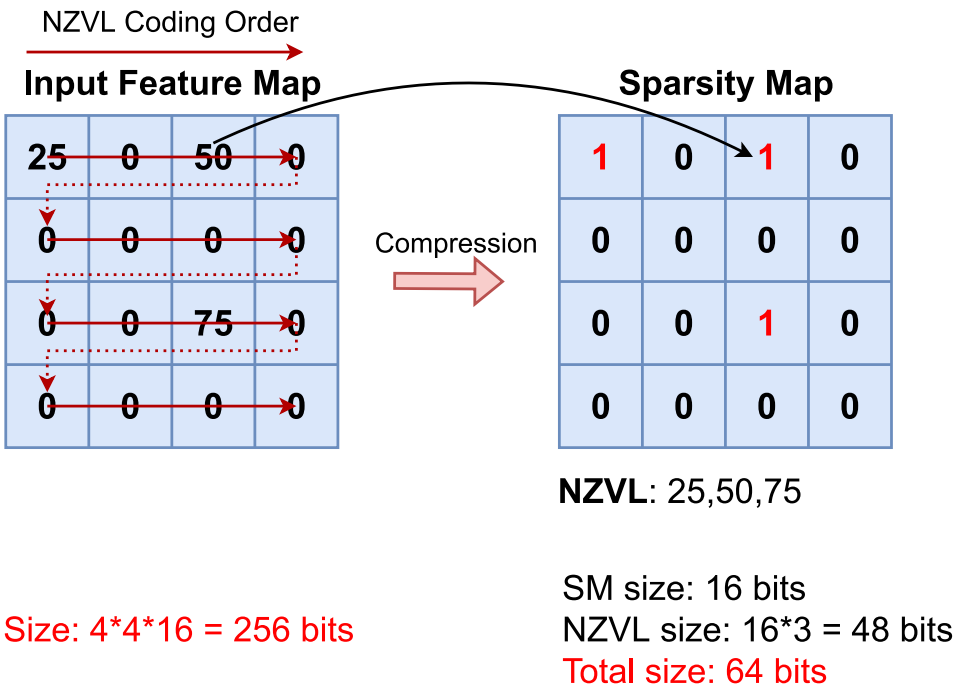

Figure 2.16: Example of Sparse Matrix Compression Algorithm

$$CIS = P_{all} \times (1 + N_{bit} \times (1 - S_p)) \tag{2.8}$$

Equ 2.8 shows how the compressed image size is influenced. CIS stands for the compressed input image size in bits. P_{all} is the total number of pixels in inputs. S_p stands for the sparsity of input images ranging from 0 to 1. N_{bit} is the adopted input precision in bits. Whether the data size could be compressed mainly depends on the sparsity of the feature maps. The data size compression could be ensured when condition 2.9 is satisfied:

$$S_p > th_p = \frac{1}{N_{bit}} \tag{2.9}$$

where th_p refers to the threshold sparsity.

In the current implementation, the 16-bit data precision is used to balance the accuracy and hardware costs. The threshold sparsity to guarantee compression under 16-bit is 0.0625. This threshold is low enough for most CNNs to reach. The other threshold sparsities with different bit precisions are presented in Table 2.1. This SM & NZVL compression algorithm achieves better results than the run-length(**RL**) compression algorithm proposed by [23], as demonstrated for the VGG19 example in Figure 2.18 [1] and Figure 2.19 [1].

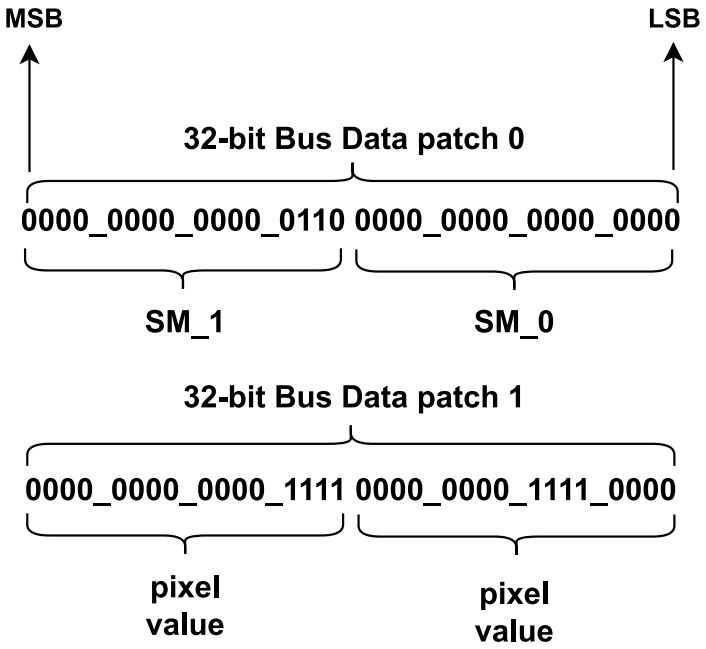

Figure 2.17: SM and pixel value distribution inside data bus

Bit Precision	Threshold Sparsity
8	0.1250
12	0.0833
16	0.0625
24	0.0416
32	0.0312

Table 2.1: Minimum sparsity for compression

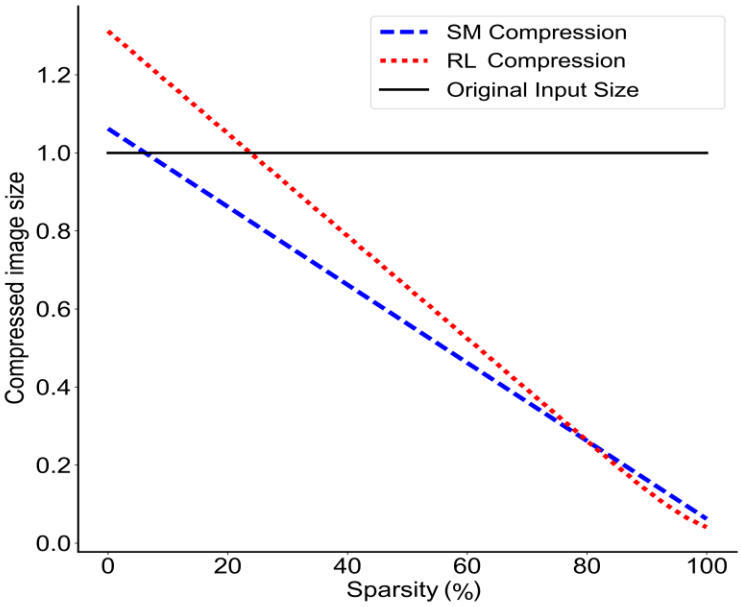

Figure 2.18: Comparison of compression methods over different sparsity amounts. Results from 10,000 images. [1]

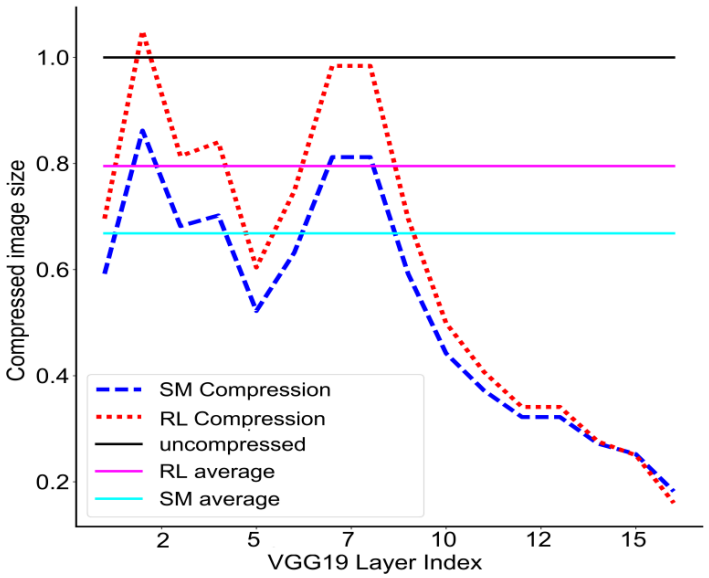

Figure 2.19: Comparison on 1000 runs of VGG19 [1]

Chapter 3

Accelerator Architecture

3.1 Architecture Overview

The high-level schematic of the accelerator is presented in Figure 3.1. The input feature maps are transmitted into the accelerator with a 32-bit data bus. Moreover, an input configuration interface is implemented, which can be used by a host microcontroller for configuring the system. The enable, reset, clock, and bus handshake signals are also working as part of the control signals of the whole accelerator. The accelerator processes the convolutional stages one at a time in a sequential mode. The ReLU stage, max pooling stage, and a customized encoding stage could be concatenated after the convolutional stage. The input feature maps and the kernel values for the current convolutional layer are stored in two independent SRAM blocks. The output feature maps of the current convolutional layer are streamed off-chip to the external memory. Every feature map is always stored in the SM & NZVL compressed mode and never decompressed during the whole computation phase to reduce the workload of memory and PEs. Valuable data will be decoded by Input Data Processor(**IDP**) and be forwarded to Compute Core Module(**CCM**). The Pooling, ReLU and Encoding Unit(**PRE**) does further processing and compression of the convolution results.

The following subsections explain the functions of every hardware unit within the accelerator with the compression scheme employed. The high-level overview of the workflow is as follows:

1. The Input Decoding Processor receives the input feature maps sent from the input bus. The input data is be decoded since the data is compressed into SMs and NZVLs. Only a small portion of pixels are decoded at one time. Along with the non-zero pixel value, the pixels' positions are also forwarded to the Compute Core Module.

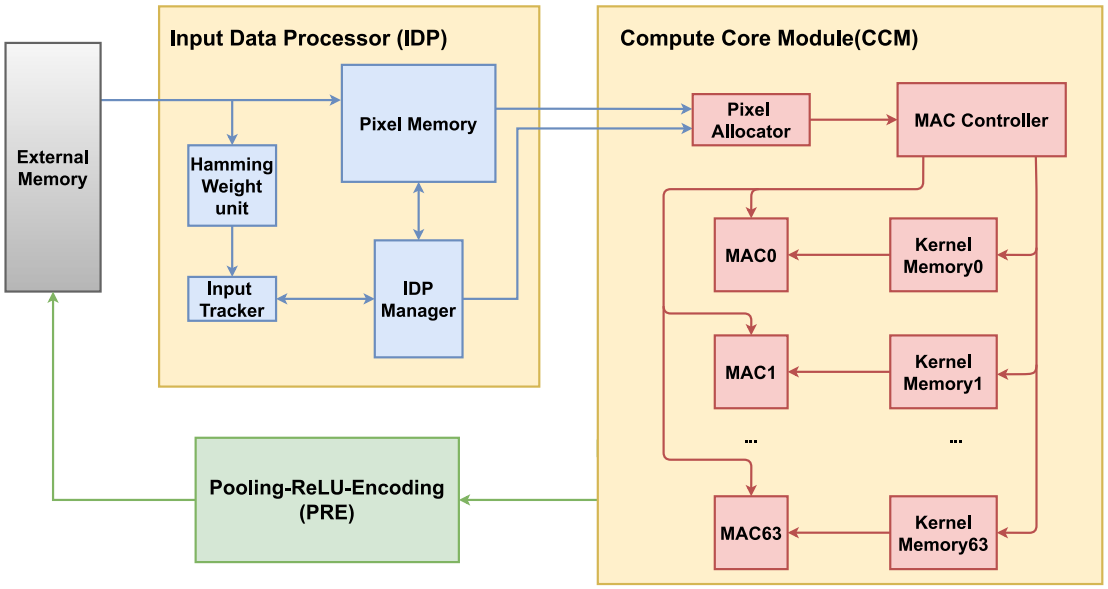

Figure 3.1: High-level schematic of the accelerator

2. The Pixel Allocator(**PA**) inside CCM distributes the received pixels to the MAC controller. The controller generates necessary kernel weight addresses inside the Kernel Memory. The MACs use the pixel values streamed from the MAC controller and the kernel values sent from kernel memory to do the convolutions. Each MAC is in charge of computing one output channel. Different MACs receive the same pixel value but potentially different kernel values fetched from their individual Kernel Memories.

3. The results from CCM are forwarded into PRE unit which composes a ReLU unit, a max pooling unit, and an encoding unit. ReLU, max pooling and encoding are done sequentially. The encoded output feature maps are then sent off-chip.

3.2 Input Data Processor

3.2.1 Input Data Format

The input data streamed into the CNN accelerator is first received by the IDP unit. These 16-bit length SM segments and non-zero pixel values are interleaved in the Bus Data. The low 16-bit data of the first 32-bit Bus Data sent by the input bus will always be a SM segment. As shown before in Figure 2.17, the number of 1s indicates the number of pixel values behind the SM segment. Starting with this information, all the following data could be decoded. In the IDP Manager, the

position of 1s will also be used to calculate the non-zero pixel's spatial coordinates. The compressed input feature maps are streamed row by row into the accelerator starting from the top row. If the kernel size is 3x3, the pixels are streamed into the accelerator as follows : $p(0,0,0)$, $p(0,0,1)$, $p(0,0,2)$,..., $p(0,0,N_i)$, $p(1,0,0)$,..., $p(1,0,N_i)$, $p(2,0,0)$,..., $p(2,0,N_i)$, $p(3,0,0)$,..., $p(3,0,N_i)$. The coordinate representation is as shown in Figure 2.6, and N_i stands for the input channel number. For the simplicity of explanation, the pixels with the same x and y will be referred to as a "**bar**" in the following chapters. Therefore, one bar is composed of 16 pixels if the input channel depth is 16. The input data bus first streams $bar(0,0)$, then $bar(1,0)$, $bar(2,0)$, $bar(3,0)$ as shown in Figure 3.2. Then the input data bus switches to send $bar(0,1)$, $bar(1,1)$, $bar(2,1)$ and $bar(3,1)$ as shown in Figure 3.3.

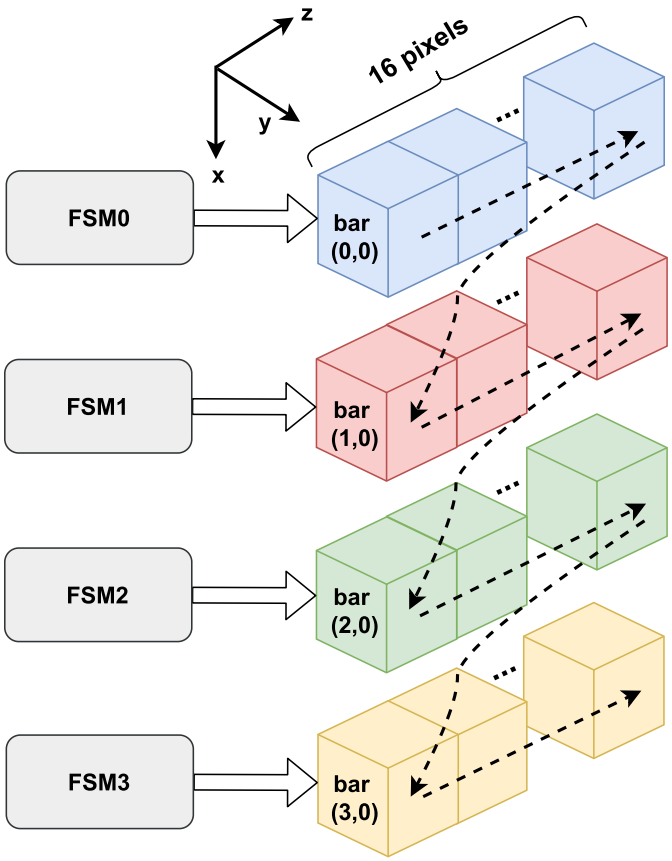

Figure 3.2: Input data stream scheme

Under the condition of a 3x3 kernel, four pixels with the same input channel depth are decoded at one clock cycle by four FSMs as shown in Figure 3.2. The pixels within these four rows, referred to as a "**Pixel Patch**", are the necessary pixels to calculate two rows of output feature maps with a vertical convolution

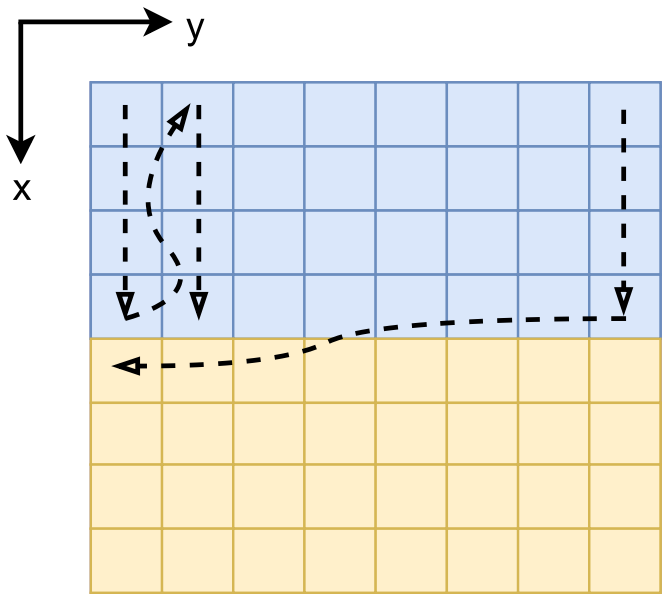

Figure 3.3: Input data stripe switch scheme

stride of 1 as shown in Figure 3.4.

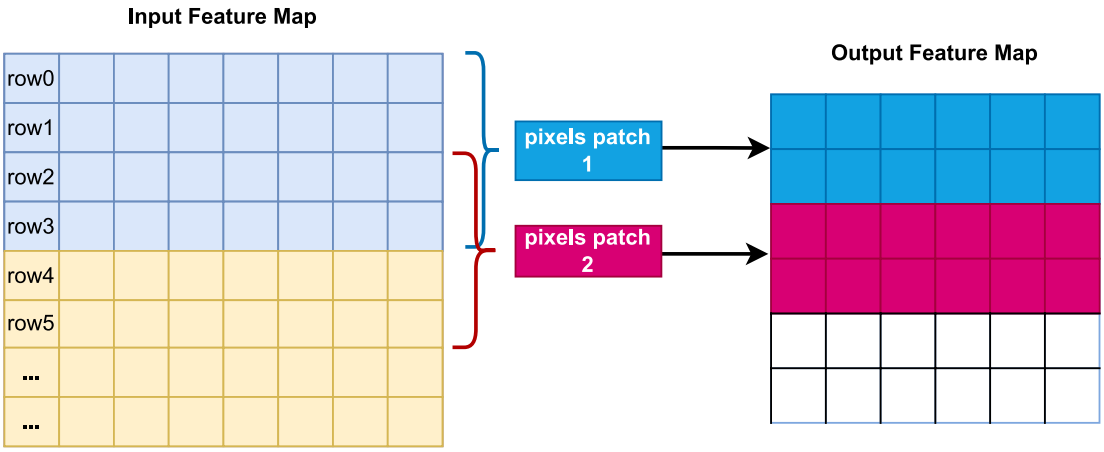

Figure 3.4: Relation from Pixel Patch to final output

3.2.2 Input Data Bus and Pixel Memory

The input data bus is in charge of reading data from External Memory and sending it into the Pixel Memory. The data storage scenario follows Little-Endian. The low 16-bit is the data to be analyzed first. The Pixel Memory stores the input feature

maps to be processed and sends data to the IDP Manager for decoding and the CCM for convolution. During simulations, the input feature maps are generated with a C file and stored into a TXT file which imitates as an external memory. And the Pixel Memory is designed with 16-bit per cell.

3.2.3 Hamming Weight Unit

The Hamming Weight Unit keeps monitoring the input data to calculate the number of 1s inside each 16-bit word. Its output port *hamming1* refers to the number of 1s of low 16-bit of Bus Data, and output port *hamming2* refers to the number of 1s of high 16-bit of Bus Data. *Hamming1* and *hamming2* are forwarded to the Input Tracker for the further bar's SM segment address calculation.

3.2.4 Input Tracker

To simplify the decode phase, it's necessary to know the address of each bar's SM segment inside the Pixel Memory. The Input Tracker accomplishes this task with an FSM and a memory. The Input Tracker memory stores the pixel memory address of every bar's SM segment and sends this information to the IDP Manager, depending on the read request and address also transmitted from the IDP Manager. This data interaction will be later explained in Section 3.2.5.

Due to the equality of input channel number(16) and the SM segment length(16), the current SM segment and the next SM segment always represent two adjacent bars. The Input Tracker FSM is built based on this regularity. The Input Tacker FSM(**IT-FSM**) comprises six states, and its internal state transfer logic is shown in Figure 3.5. State *row1* is in charge of calculating the SM segment address of *bar*(1,0), *bar*(1,1),... or *bar*(5,0).... as shown in Figure 3.3. State *row2* is in charge of the SM segment address of *bar*(2,0),*bar*(2,1),...or *bar*(6,0).... IT-FSM returns to state *reset_state* if the reset signal is set. For cleanness, all the returns to *reset_state* are not shown in Figure 3.5. The arrows mainly show the two conditions during state transfer: if the next SM segment is an all-zero SM segment, the state transfer follows the green arrow; if the next SM segment contains 1, the state transfer follows the red arrow.

To be specific, IT-FSM first enters state *row1* since *bar*(0,0) SM segment is always the low 16-bit of *Bus Data0*. State *row1* calculates the *bar*(1,0) SM segment address using the hamming weight of the low 16-bit of *Bus Data0*. If this hamming weight is 0, IT-FSM will enter state *row3* since the second SM segment is definitely the high 16-bit data of *Bus Data0*. If this hamming weight is $x(x \neq 0)$, IT-FSM counts x words starting from the high 16-bit data of the current 32-bit Bus Data. Figure 3.6 shows a concrete example.

It's worth noting that state *row1new* is set separately aside state *row1* due to

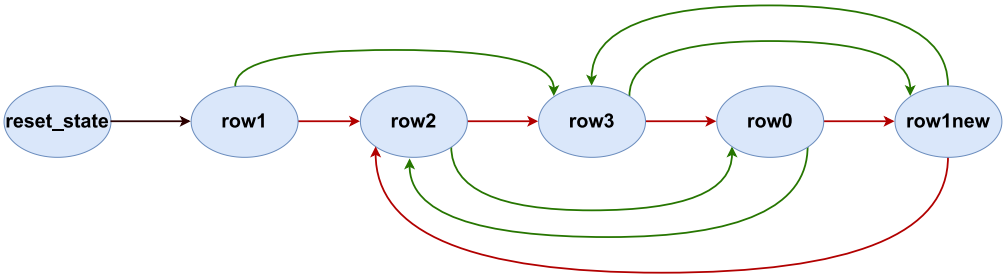

Figure 3.5: Input Tracker FSM

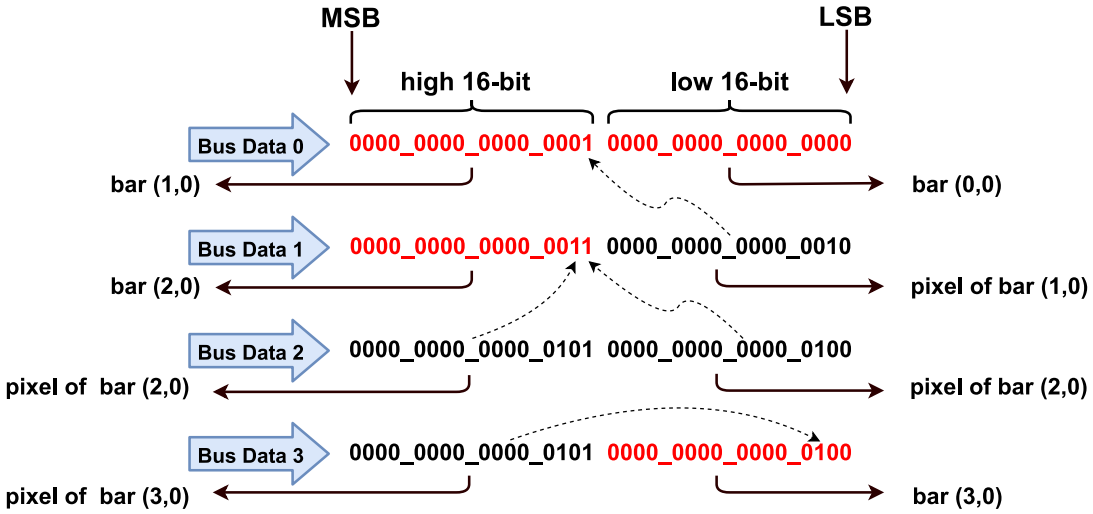

Figure 3.6: Input Tracker decodes Bus Data

the irregularity when the Input Tracker just begins to analyze the input data bus. As stated before, the very first 16-bit word sent by the input data bus is always a SM segment. This mechanism makes state *row1* different from the other states since it doesn't require any inspections to enter. Once the data begins to transfer, the FSM will first enter into state *row1*. And this state won't be accessed anymore unless the accelerator is reset. However, all the other states, including the state *row1new* must use the outputs generated from the previous states to verify if it's the correct time to have the state transfer.

3.2.5 IDP Manager

The IDP Manager is the headquarter of the whole IDP unit. It's composed of a set of FSMs(**M-FSM**). The number of M-FSMs is equal to the current kernel size plus 1. This additional M-FSM allows the accelerator to calculate two rows of

output feature maps simultaneously, as shown in Figure 3.4. During processing, each M-FSM is in charge of one bar's decoding process, as shown in Figure 3.2. Figure 3.7 explains the interactions between the IDP Manager, Input Tracker, and Pixel Memory. The decode workflow of one M-FSM is as follows:

1. The M-FSM sends a read address to Input Tracker memory. This memory sends back the location of the bar's segment inside the Pixel Memory. Then M-FSM stores this address into its SM segment address register(**SMAddReg**).

2. The M-FSM uses the address stored in SMAddReg as a read address to the Pixel memory. The Pixel Memory returns the read content, which is the SM segment of the bar being decoded. Afterwards, this SM segment is stored into the SM register(**SMReg**) and used as a map for the pixel value coordinates calculation. During the state *new_start*, M-FSM stores the address sent from Input Tracker Memory. During the state *prepare_sm*, M-FSM stores the SM segment fetched from Pixel Memory into SMReg.

3. When the SM segment is ready, depending on the content of that SM segment, the M-FSM enters either state *value_true* if that bit is 1 or state *value_false* if that bit is 0. Every time one bit is analyzed, coordinate z is increased by 1. Meanwhile, during state *value_true*, M-FSM increases the address stored inside SMAddReg by 1 and sends this increased address as a read address to the Pixel Memory. The Pixel memory sends the read content to the CCM unit.

   Moreover, the M-FSM also sends out a signal "*sm_or_not*" to indicate if the content sent out of the read port of Pixel Memory is a non-zero pixel or a zero pixel or a SM segment.

4. After all 16 bits are decoded, the M-FSM enters state *update*. The state *update* and state *new_start* are responsible for fetching new SM segments and recounting the coordinates when one pixel patch is finished as shown in Figure 3.4.

5. When the reset signal is set, the M-FSM returns to the state *reset*. The convolution process inside the CCM unit consists of several multiplications and accumulations. Therefore the M-FSM could be stopped by signal *stop_fsm* from CCM when the CCM unit is busy. When the *stop_fsm* signal is set, the M-FSM enters the state *halt* until the signal is unset. For cleanness, the state transfers to state *reset* and state *halt* are not shown, while the other state transfers are presented in Figure 3.8.

As shown in Figure 3.4, the pixel patch used to generate row0 and row1 of output feature maps and the pixel patch used to generate row2 and row3 are

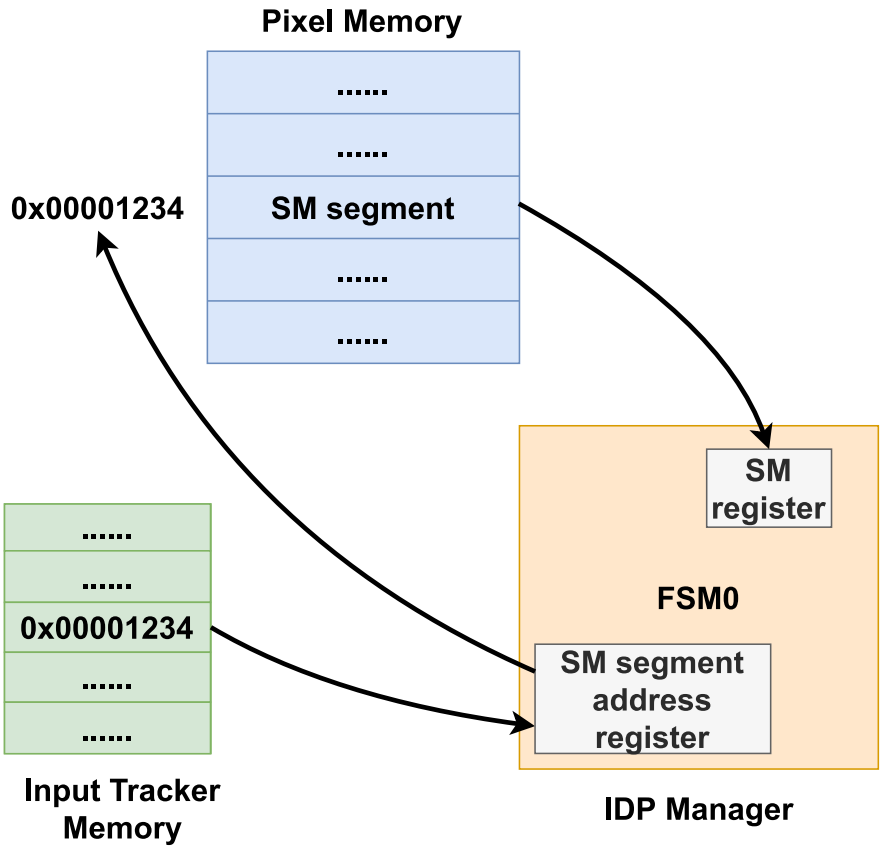

Figure 3.7: Interaction between IDP Manger, Input Tracker and Pixel Memory

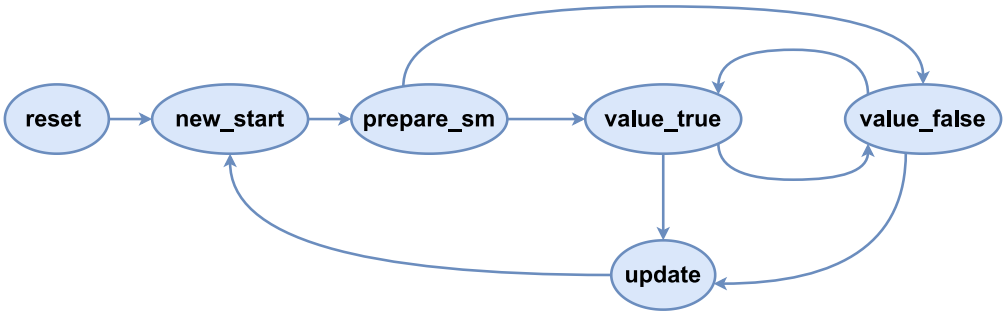

Figure 3.8: IDP Manager FSM

overlapped. The M-FSMs reuse the bar's segment addresses stored inside Input Tracker Memory to adapt this pixel repetition. This re-usage scheme is triggered only when one pixel patch is finished. For example, as shown in Figure 3.9. *Pixel Patch 0* is first handled by four M-FSMs, each M-FSM takes one row inside the pixel patch. Initially, M-FSMs take the content of *cell0*,*cell1*,*cell2*, and *cell3* inside

Input Tracker Memory. The contents are the bar's segment addresses inside Pixel Memory, as shown in Figure 3.7. After all these four bars are fully decoded, the M-FSMs switch to take the contents of *cell4*,*cell5*,*cell6*, and *cell7* inside Input Tracker Memory. This process repeats until the last column of this pixel patch is done. Afterwards, *Pixel Patch 1* is fetched to calculate another two rows of output. The M-FSMs take *cell2*,*cell3*,*cell32*, and *cell33* inside Input Tracker Memory and repeat the previous scheme. Due to the reuse of several intermediate rows, the pixel coordinates after decode are out-of-range. For example, for an input with size 8x8, as in Figure 3.9, row2, row3, row4 and row5 are reused, so the final coordinates after decode are from $x = 0$ to $x = 11$, while the original pixel coordinates are from $x = 0$ to $x = 7$.

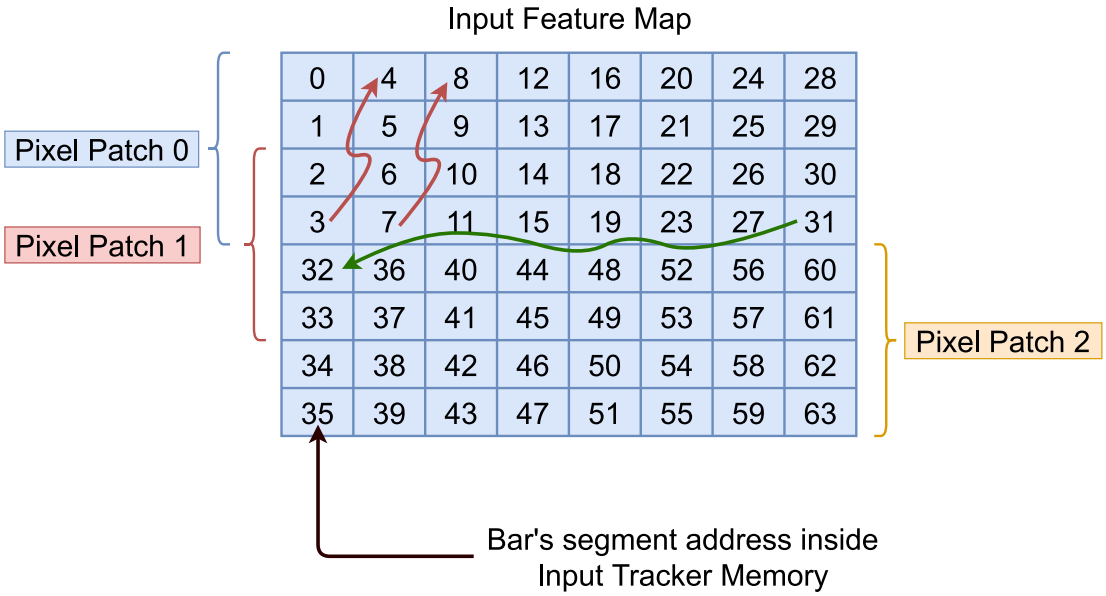

Figure 3.9: The Reuse Scheme of Bar's segment address

Another aspect to note is that:

1. the pixels in row0, depth0 of each pixel patch, referred to as **Column-detecting Pixels**.
2. the **Triggering Pixels**.

are marked as non-zero pixels, even if the actual pixel values are 0s. These pixels are referred to as "**Special Purpose Pixels**", as shown in Figure 3.10. The "column-detecting pixels" in the first case is to handle the condition that input feature maps have consecutive all zero pixel stripes. Since the output feature maps are calculated step by step, the temporary outputs are stored into a buffer which

will be later explained in Section 3.3.3. Once the buffer is full, the leftmost column of the buffer will be shifted out. These "special purpose pixels" will force the shift operation to be one column at a time, even though several stripes are all zeros. The definition of "triggering pixels", and its function will be explained in Section 3.3.3.

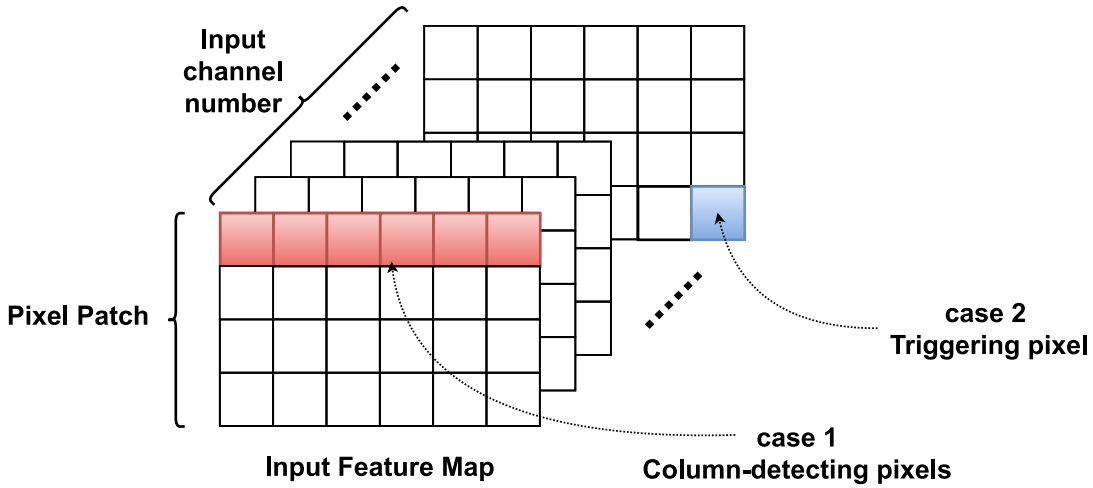

Figure 3.10: Special Purpose Pixels during decoding phase

3.3 Compute Core Module

The Compute Core Module (CCM) is the arithmetic unit of the accelerator. It is composed of a Pixel Allocator, a MAC block containing M MACs, a Kernel Memory with M banks, and a controller. M refers to the output channel numbers. The CCM unit uses the pixel values and coordinates sent from the IDP unit to calculate the convolution result. The results are temporarily stored in the buffers connected to every MAC. Along with the calculation, the results are shifted step by step to the PRE unit. The overall structure of CCM is shown in Figure 3.11.

3.3.1 Pixel Allocator

The Pixel Allocator is composed of an FSM and several registers. Its main functionalities are:

1. Stop and restart the FSMs depending on the CCM workload.

2. Store the valid pixel values and their spatial coordinates.

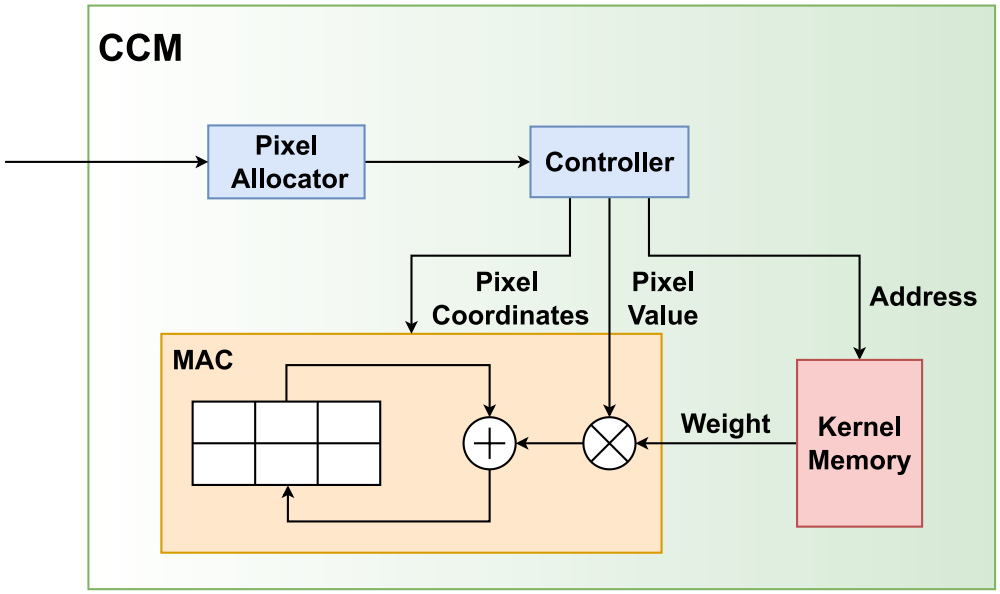

Figure 3.11: Overall structure of CCM

3. Stream out new pixel values and coordinates when MACs are free.

With kernel size 3x3, 4 FSMs are working inside the IDP Manager. These FSMs can produce up to 4 non-zero pixels per cycle. The Pixel Allocator is designed to contain 16 pixel values and their spatial coordinates. Once the Pixel Allocator is full, all the FSMs inside IDP Manager are stopped. Then Pixel Allocator streams out a valid pixel and maintains its value until MAC finishes all the calculations with this pixel. The IDP-FSMs are back to work when all the pixels inside Pixel Allocator registers have finished their calculations.

A Pixel Allocator FSM(**PA-FSM**) supervises the whole working mechanism of this hardware block. Its internal structure is as shown in Figure 3.12.

The state *stripe1*, *stripe2*, *stripe3* and *stripe4* store the pixels and coordinates streamed out of IDP. The state name starting with "*two_state*" means two valid pixels exist in that stripe. The number after indicates which FSM sends out a valid pixel. Therefore, as an example, state *two_state_01* means that FSM0 and FSM1 are sending out valid pixels. This rule applies to all the other states. The signal *sm_or_not* sent from IDP-FSM indicates the pixel's validity. When the PA-FSM is in state *prepare*, it uses this signal to decide the next state. The state *locator_stripe_switch* is used to switch to another pixel stripe for the next round of streaming. When all the data inside the registers of PA has finished calculation, the PA-FSM enters state *start* to store another three new pixel stripes.

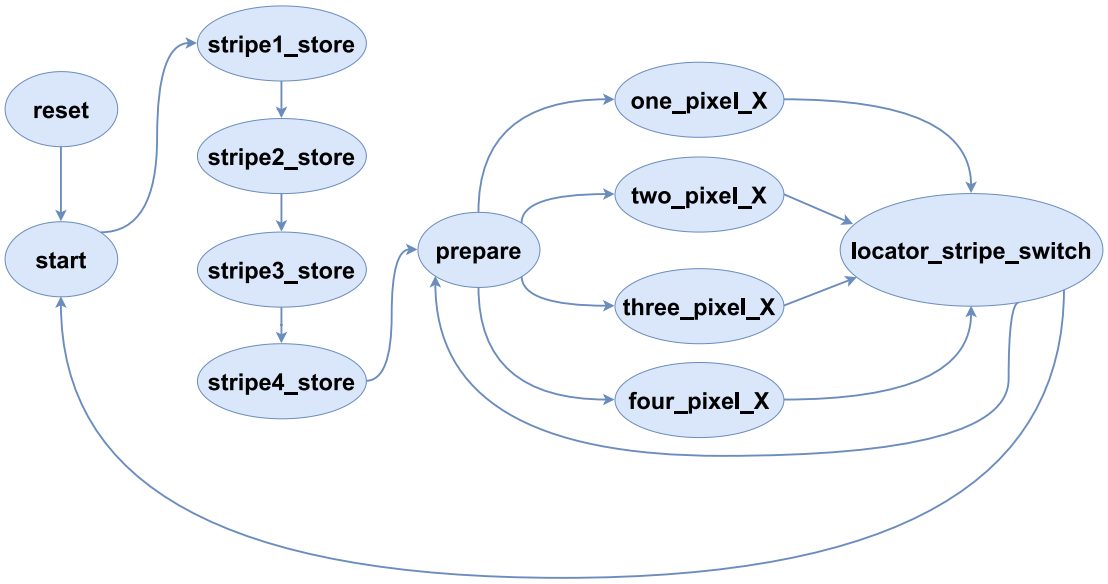

Figure 3.12: Pixel Allocator FSM

3.3.2 MAC Controller and Kernel Memory

The accelerator configuration is to realize same number of MAC units and output channel depth. At the start of a layer computation, the weights are loaded into the Kernel Memory. Each kernel is stored in a different Kernel Memory Bank and assigned to a MAC. And each MAC is in charge of computing one output channel. The MAC Controller's duty is to generate the corresponding weights' locations depending on the pixel coordinates.

The necessary kernel weights for each pixel are shown in Figure 3.13. For example, pixel (1,1) inside the pixel patch must multiply with kernel weights inside the kernel matrix with coordinates (0,0),(0,1),(1,0) and (1,1) when the complete convolution is done. If the input feature map size is 6x6, the pixel stripes with y=2 and y=3 have the same kernel weights correspondence as shown in the same figure. This pattern works for each pixel patch.

The FSM inside MAC Controller(**MAC-Control-FSM**) is built based on this regularity. This FSM detects the incoming pixel coordinates and generates necessary kernel weights' addresses inside the kernel memory. These addresses are sent to the kernel memory as reading addresses. The kernel memory returns the corresponding kernel weights to MACs. Meanwhile, the MAC Controller passes the current pixel value and its coordinates to MACs. The workflow is presented in Figure 3.14.

State *one_state_XX* is a cluster of states that generates one kernel memory address depending on the pixel's location. There are four pixels that need only one kernel weight inside a pixel patch. The XX in the state name distinguishes the four

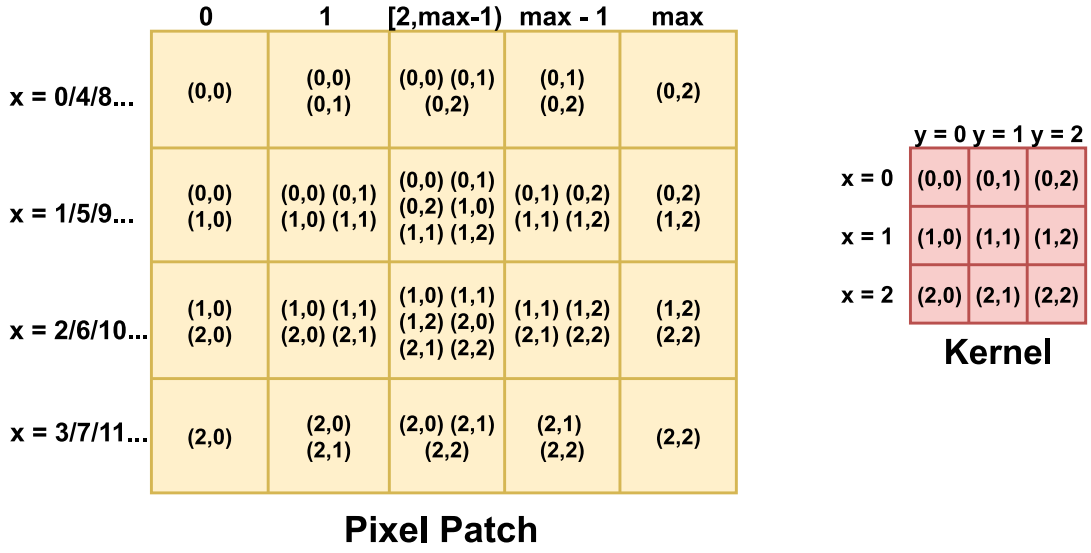

Figure 3.13: Correspondence between Input and Kernel

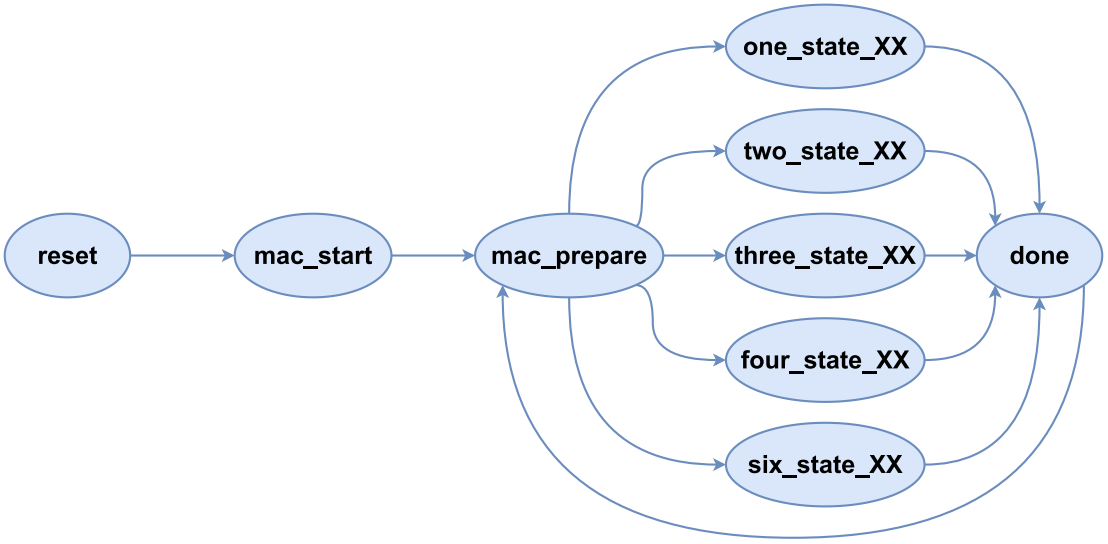

Figure 3.14: MAC-Control-FSM state representation

different cases as shown in Figure 3.13. Similarly, state *two_state_XX* is a cluster of states that generates two kernel memory addresses for one pixel.

3.3.3 Multiply-Accumulate Unit

Under the preparations done by the previous stages, the MACs now have correct pixel values and kernel weights as inputs. Figure 3.15, Figure 3.16, Figure 3.17 and Figure 3.18 illustrate four different cases that how pixels with different coordinates contribute to the final output feature map.

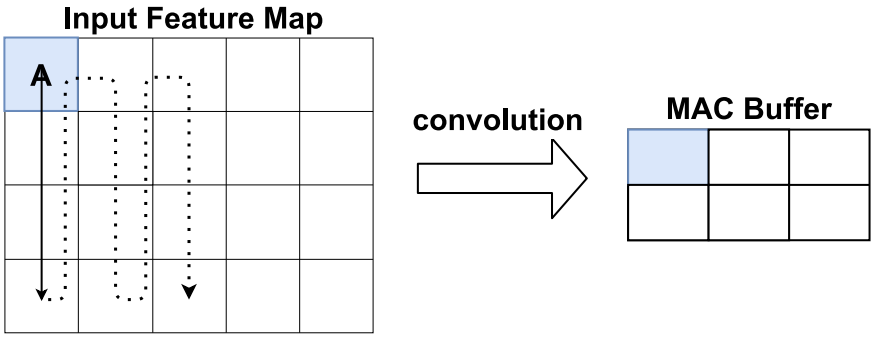

Figure 3.15: Example 1 Convolution process from one pixel to final output

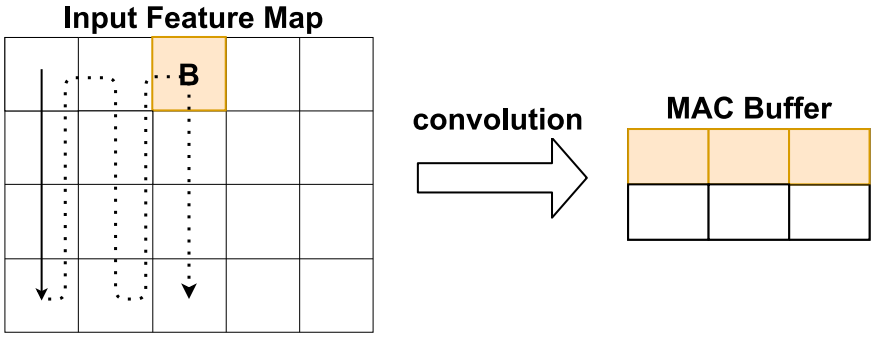

Figure 3.16: Example 2 Convolution process from one pixel to final output

For one pixel inside a pixel patch, it's not necessary to multiply it with all weights of the kernel matrix. As shown in Figure 3.17, pixel C is at (1,1) of that pixel pitch, it only needs to multiply weights in position (0,0),(0,1),(1,0),(1,1). And the four results contribute to the four red blocks inside MAC Buffer. More detailed contribution connections are shown in Figure 3.19. The detailed correspondence between pixels and kernel weights are summarized in Figure 3.13.

The same rule applies to pixel D in Figure 3.18. It does multiplications with kernel weights in position (0,0),(0,1),(0,2),(1,0),(1,1) and (1,2). The contributions to the final output are presented in Figure 3.20.

The contribution connections to output feature maps and kernel weights correspondence are regular. These regularities are implemented and controlled by an

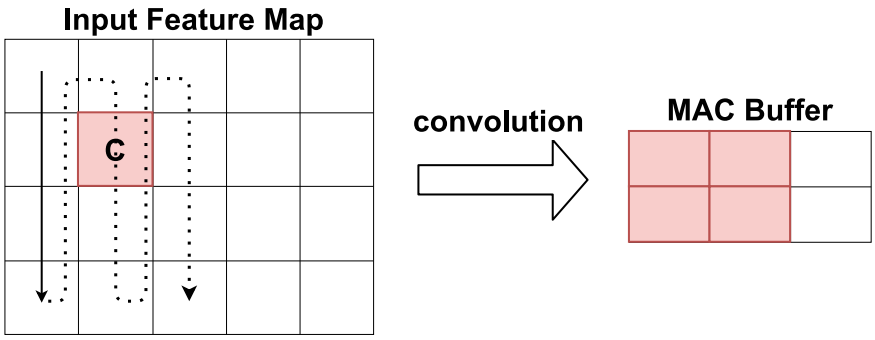

Figure 3.17: Example 3 Convolution process from one pixel to final output

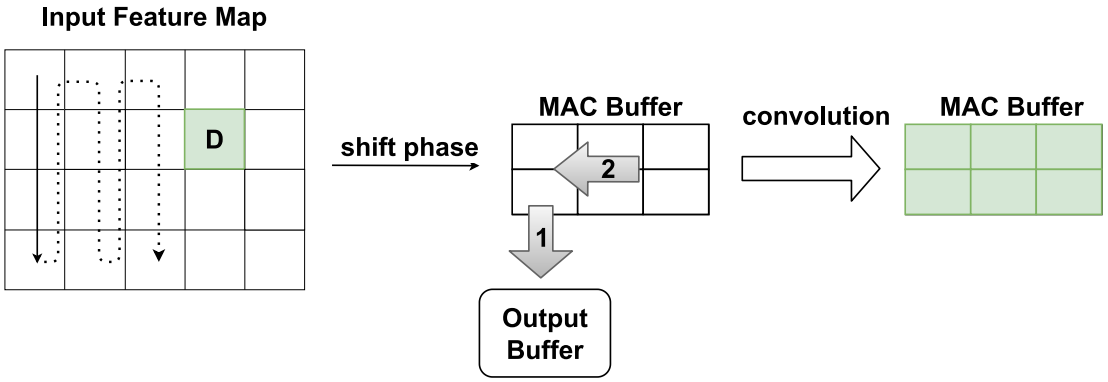

Figure 3.18: Example 4 Convolution process from one pixel to final output

FSM(**MAC-FSM**). The state representation of MAC-FSM is shown in Figure 3.21.

As before, the state transfers to state *reset* are omitted for simplicity. The *Calculation cluster* is composed of several states to do convolution depending on the incoming pixel's coordinates. Each state is in charge of one convolution case as shown in Figure 3.13.

Figure 3.18 presents the case when a pixel lies in a different column than the previous one. In this example, the MAC-FSM detects the increment of column index. Then MAC-FSM first enters state *buffer_shift*. The current stripe won't contribute to the two results stored in the leftmost entries of the MAC Buffer. So these two results are shifted out to the PRE unit during state *buffer_shift*. Meanwhile, the middle and rightmost entries inside the MAC Buffer are shifted left by one column to reserve two blank entries for the following accumulations.

This process will repeat until the *triggering pixels* with coordinates (3,P,Q), (7,P,Q),... of the input feature map finish calculations, where P stands for the input image horizontal size minus 1, Q stands for the input image channel depth.

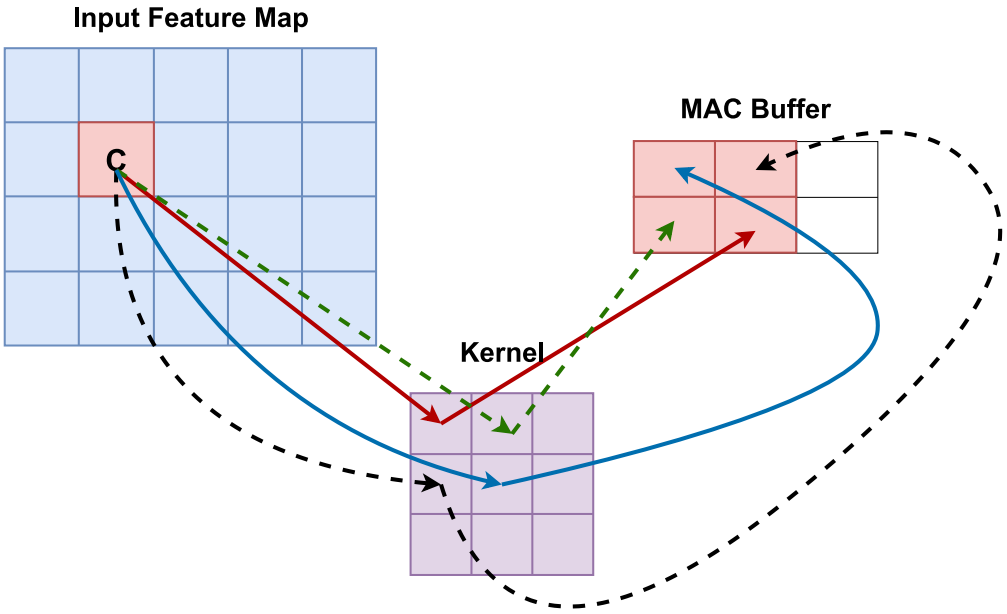

Figure 3.19: Example 1 Contribution of pixel to output feature maps

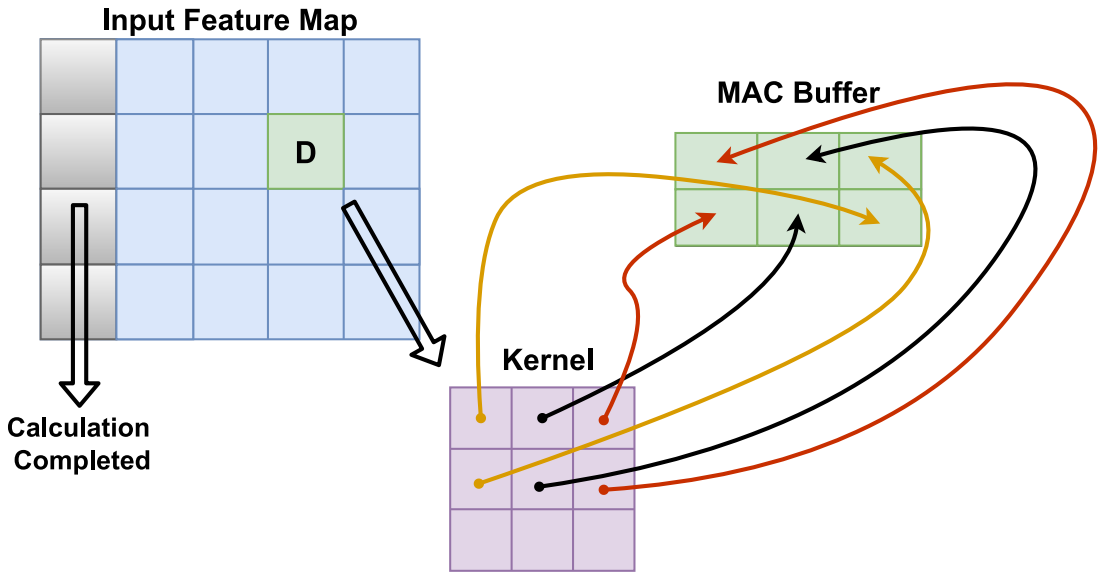

Figure 3.20: Example 2 Contribution of pixel to output feature maps

For example, if the input image is of size 6x6, with 16 input channels. The first triggering pixel coordinate is (3,5,16). And due to the reuse mechanism done in the IDP Manager, the second triggering pixel coordinate is (7,5,16). This out-of-range condition is explained in Section 3.2.5. In this case, the MAC-FSM enters state

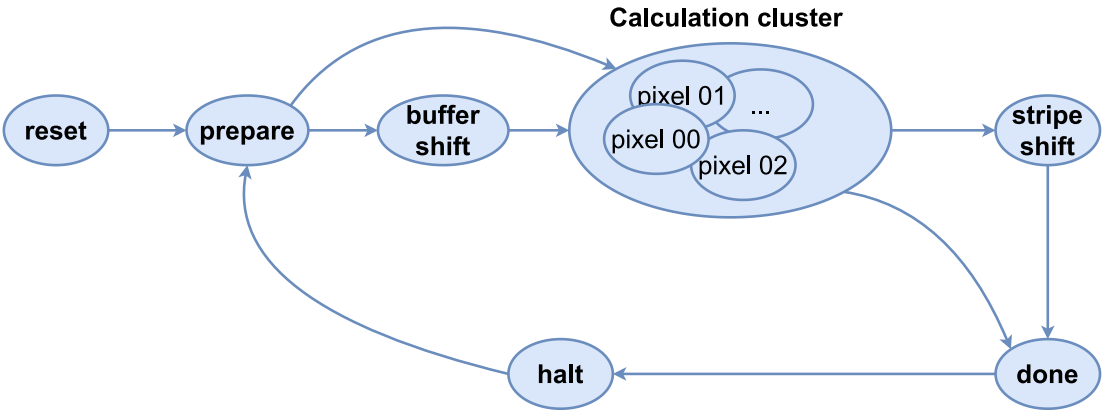

Figure 3.21: MAC-FSM state representation

stripe shift to stream out all the results inside the MAC Buffers.

Afterward, MAC-FSM enters state *halt* to wait for another valid pixel number and its corresponding kernel weights.

3.4 Pooling, ReLU and Encoding Unit

The Pooling, ReLU and Encoding Unit (PRE) is responsible for the last processing stage of the accelerator. It stores the convolution results streamed out of the MACs into the PRE Buffer. The PRE Buffer's size is 2*M, where M stands for the number of MACs, also the number of output channel channels, and 2 is the pooling dimension. The significant benefit of this PRE unit is its capability to perform ReLU, max pooling and encoding on the fly. The overall structure of PRE is shown in Figure 3.22.

3.4.1 ReLU and Max pooling

The working mechanism of PRE is controlled by an FSM(**PRE-FSM**) composed of 9 states, as shown in Figure 3.23. The ReLU stage is designed with a higher priority than the max pooling stage to simplify the internal logic. The detailed explanations of these two stages are as follows:

1. If both ReLU and max pooling are disabled, the PRE-FSM enters state *none* when the MACs raise the signal *shift_out*. In this case, all the values shifted from MACs inside CCM are stored into the PRE Buffer. Then PRE-FSM enters state *done* for encoding and state *print* for printing into a TXT file. The workflows of encoding performed during state *done* and state *done_pooling* are explained in Section 3.4.2.

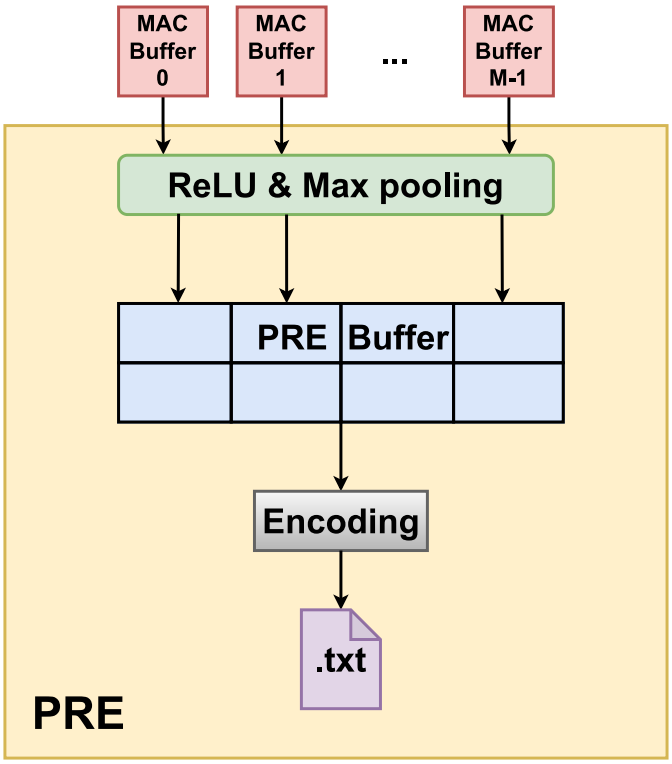

Figure 3.22: Overall structure of PRE

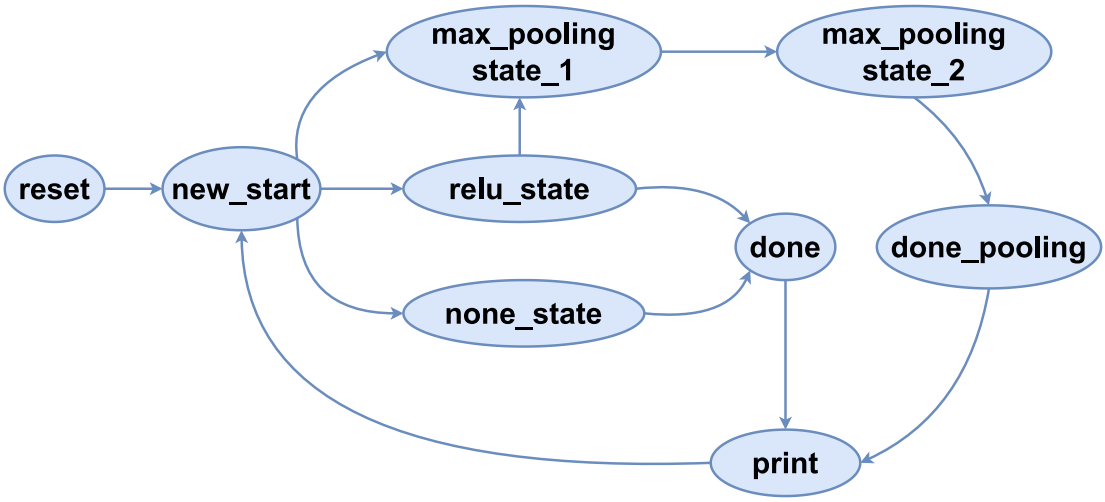

Figure 3.23: PRE-FSM state representation

2. If ReLU is enabled and max pooling is disabled, the PRE-FSM enters state *relu_state* when the MACs raise the signal *shift_out*. The negative values from

MACs are forced to be 0s while entering the PRE Buffer. Then PRE-FSM enters state *done* and state *print* for encoding and printing into a TXT file.

3. If max pooling is enabled and ReLU is disabled, the PRE-FSM enters state *max_pooling_state_1*. During this state, the pixel pair shifted from one MAC does intercomparison. The bigger one is stored in row0 of the PRE Buffer. The PRE-FSM enters state *max_pooling_state_2* when the second pixel pair comes. Afterward, these three values do intercomparison again and store the biggest value into row0 of the PRE Buffer.

   For example in Figure 3.24, Within *Pixel Pair 0*, A is bigger than B. So A is temporarily stored in row0 of PRE Buffer. The *Pixel Pair 1* is composed of C and D. After comparison, the biggest value among A, C and D is stored into the PRE Buffer as one result of max pooling. After that, the PRE-FSM enters state *done_pooling* to encode the results of the max pooling stage.

4. If both the max pooling and ReLU are enabled, the PRE-FSM first enters state *relu_state* to convert the negative values. This could omit a ReLU stage of the second pixel pair since the minimum value stored into the PRE Buffer during the first pixel pair is 0. Even if the second pixel pair is all-negative, the final result is 0 after the comparison.

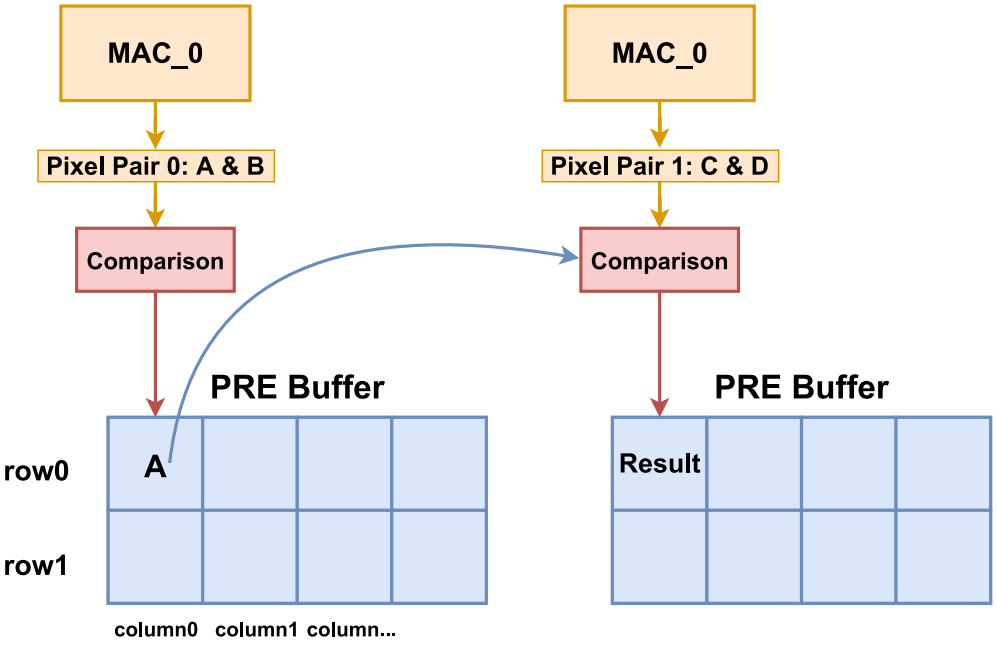

Figure 3.24: Max pooling stage inside PRE unit.

3.4.2 Encoding

The values stored inside the PRE Buffer are encoded according to the scheme described in Section 2.6.3 during state *done* and state *done_pooling*.

If the max pooling is off, all the values inside PRE Buffer are encoded and shifted out. The row0 of PRE Buffer is indeed one bar of the output feature map and row1 is the bar below it, as shown in Figure 3.25. If the max pooling is enabled, only the values of row0 are part of the output feature map.

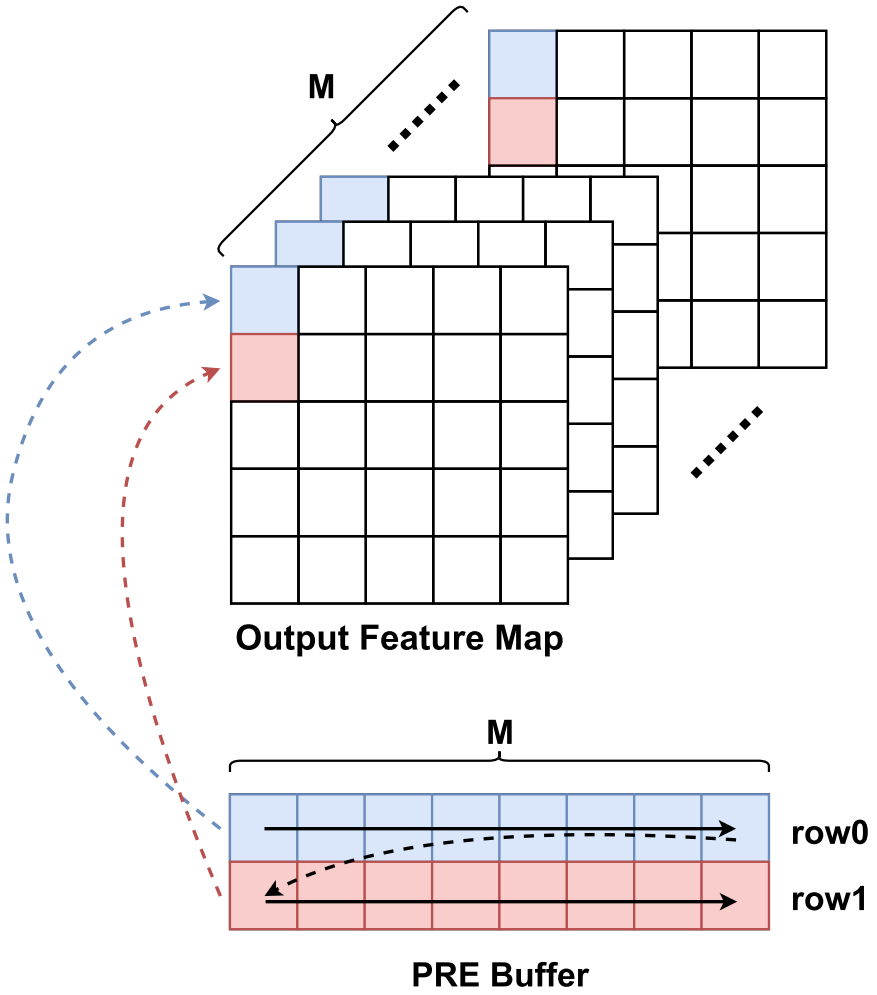

Figure 3.25: Encoding stage inside PRE unit with no max pooling

Chapter 4

Results

With the clock period of 20 ns, the simulation results from ModelSim show that:

1. If the input feature map is of size 6x6x16, composed of 126 data segments. Among these data, 36 segments are SM segments, the rest 90 segments are pixel values. The sparsity is 84.375%. The kernel is of 3x3x16x16. The output feature map is of size 4x4x16 with stride 1, no zero padding, ReLU on, and 2x2 max pooling off. If the 2x2 max pooling is on, the output feature maps are of size 2x2x16.

   The IDP begins to work at 40 ns and ends at 40230 ns, which are 2010 clock cycles. The CCM starts to work at 70 ns and ends at 40520 ns, which are 2023 clock cycles. The PRE unit begins to work at 70 ns and ends at 40610 ns, which are 2027 clock cycles. The total time consuming is 2029 clock cycles. See Figure 4.1 to Figure 4.7. The latency increment brought with max pooling stage is negligible in total time consuming.

2. The input feature map is of size 8x8x16, composed of 258 data segments. Among these data, 64 segments are SM segments, the rest 194 segments are pixel values. The sparsity is 81.05%. If the 2x2 max pooling is on, the output feature maps are of size 3x3x16.

   The IDP begins to work at 40 ns and ends at 96210 ns, which are 4809 clock cycles. The CCM starts to work at 70 ns and ends at 96490 ns, which are 4821 clock cycles. The PRE unit begins to work at 8010 ns and ends at 96590 ns, which are 4429 clock cycles. The total time consuming is 4828 clock cycles. See Figure 4.8 to Figure 4.14.

3. If the input feature map is of size 64x64x16 composed of 58886 data segments. Among these data, 4096 segments are SM segments, the rest 54790 segments are pixel values. The sparsity is 16.39%. The kernel is of 3x3x16x16. The

output feature map is of size 62x62x16 with stride 1, no zero padding, ReLU on, and 2x2 max pooling off. The output feature maps are of size 31x31x16 if the 2x2 max pooling is on.

The IDP begins to work at 40 ns and ends at 14,776,370 ns, which are 738,817 clock cycles. The CCM starts to work at 70 ns and ends at 14,776,610 ns, which are 738,827 clock cycles. The PRE unit begins to work at 70 ns and ends at 14,776,710 ns, which are 738,832 clock cycles. The total time consuming is 738,834 clock cycles. See Figure 4.15 to Figure 4.21.

The summary of the simulation results is as shown in Table 4.1.

Input Size	6x6x16	8x8x16	64x64x16
SM Segment Number	36	64	4096
Non-zero Pixel Number	90	194	54790
Sparsity	84.375%	81.05%	16.39%
Coded Data Size(bytes)	252	516	117,772
Data Size Ratio	21.875%	25.195%	89.85%
Workload Ratio	15.625%	18.94%	83.60%
Total Time-consuming(clock cycles)	2029	4828	738,834

Table 4.1: Simulation Results Summary. Data Size Ratio is the ratio between after SM & NZVL compression and the original data size. Workload Ratio is the ratio between non-zero pixel numbers and total pixel numbers.

From the statistical point of view, during the simulation of 8x8x16 input feature maps, the simulation result shows that the MACs in CCM processed only 194 pixels during the whole procedure. However, there are 1024 pixels that need to be processed without applying the hardware accelerator. The workload is approximately 18.94% compared to the original CNN implementation after applying the accelerator. In the case of 6x6x16 as input feature maps, only 90 pixels are really calculated by the MACs instead of 576 pixels. The workload is reduced to 15.625%. In the case of 64x64x16, 54790 pixels are really calculated by the MACs instead of 65536 pixels. The workload is reduced to 83.6% comparing to the non-accelerated version. These results prove that the implemented accelerator does bring an enormous enhancement in the CNN processing.

Meanwhile, thanks to the Sparse Matrix Compression Algorithm, the size of input feature map is reduced to 21.875%, 25.195% and 89.85% in the tested three cases compared to their original data sizes.

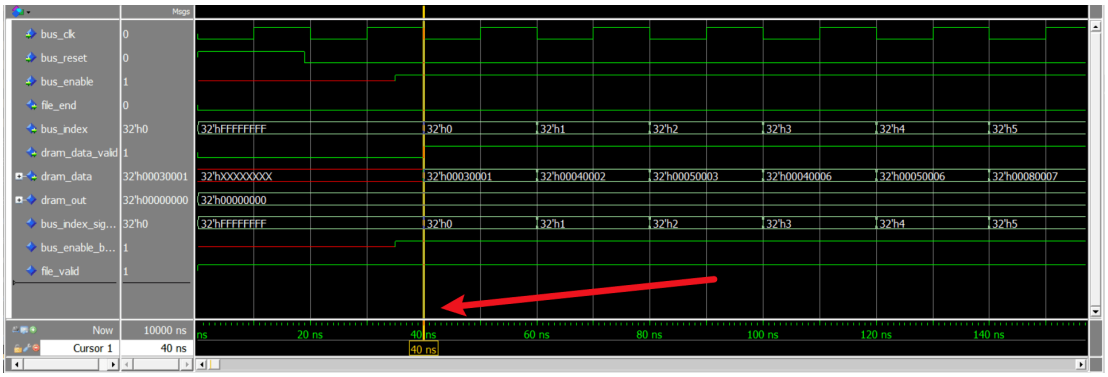

Figure 4.1: Input size 6x6: Input Bus begins to stream input data

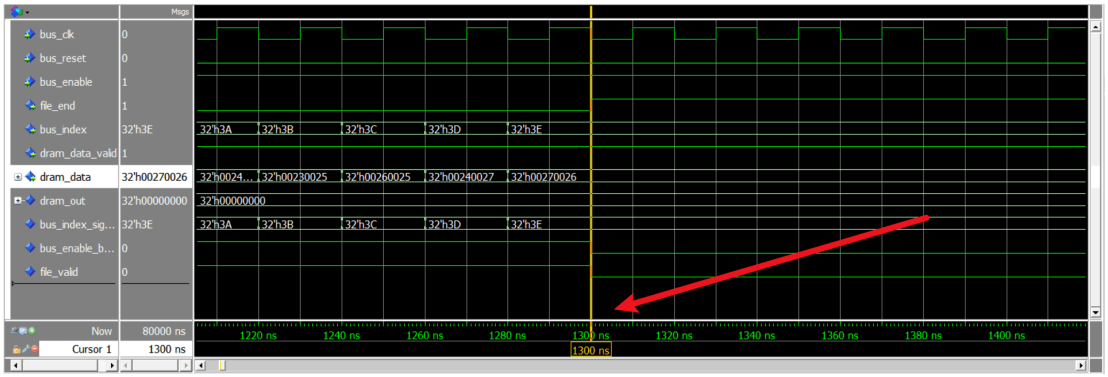

Figure 4.2: Input size 6x6: Input Bus finishes streaming

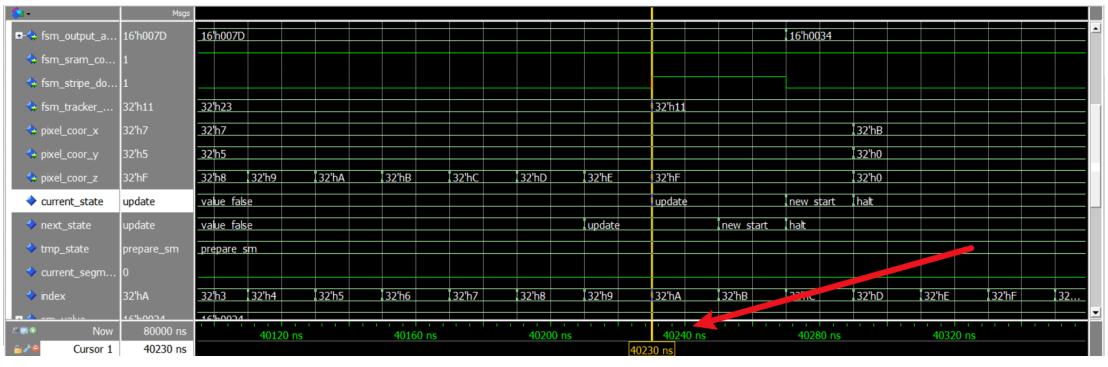

Figure 4.3: Input size 6x6: Input Data Processor streams out the last pixel in the input feature map

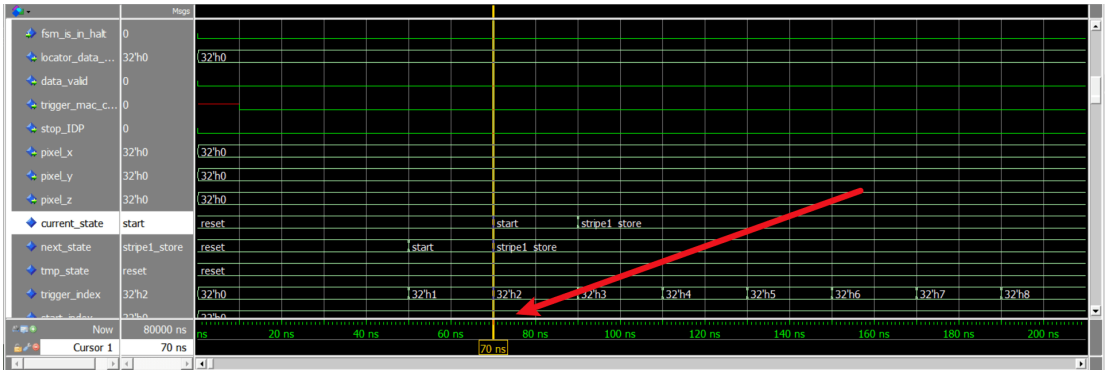

Figure 4.4: Input size 6x6: Compute Core Module begins to process decoded input feature maps

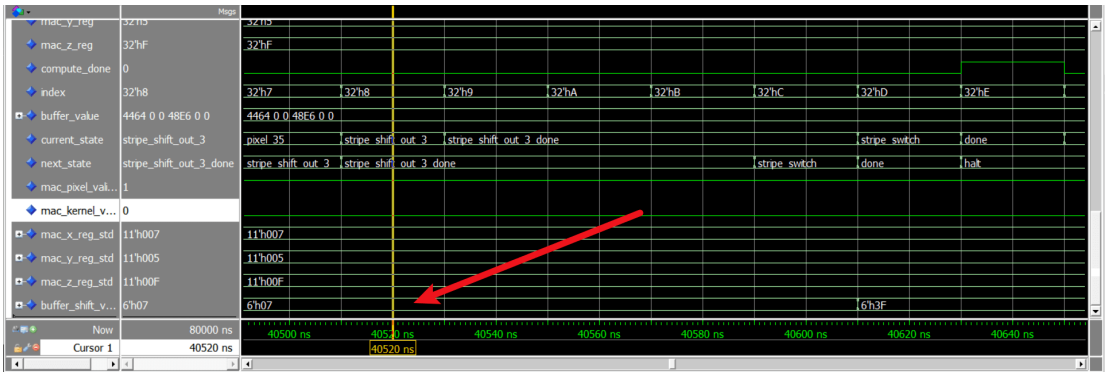

Figure 4.5: Input size 6x6: Compute Core Module finishes the convolution calculation of the last pixel in the input feature maps

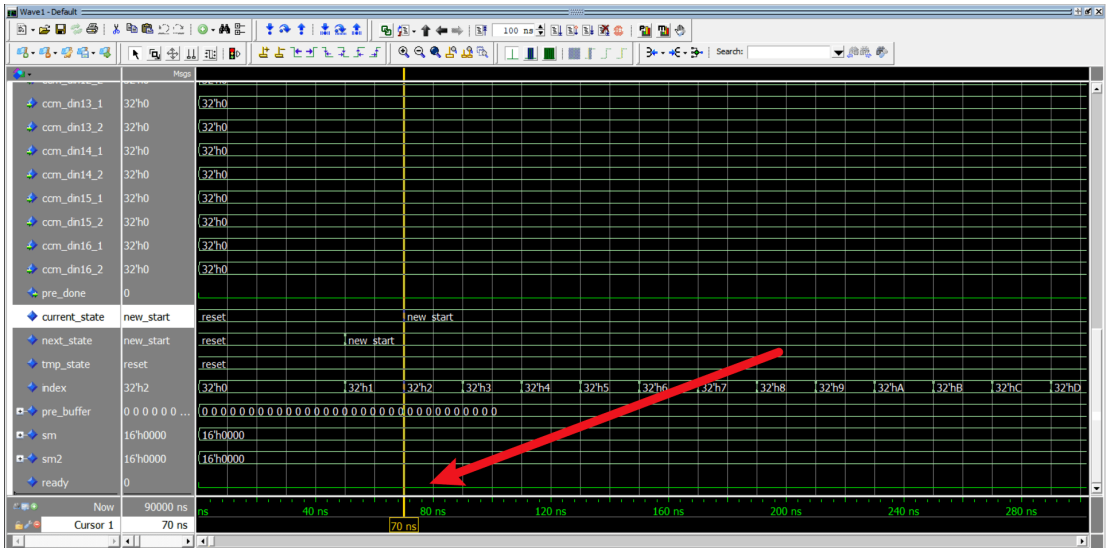

Figure 4.6: Input size 6x6: Pooling-ReLU-Encoding receives the first patch of convolution results

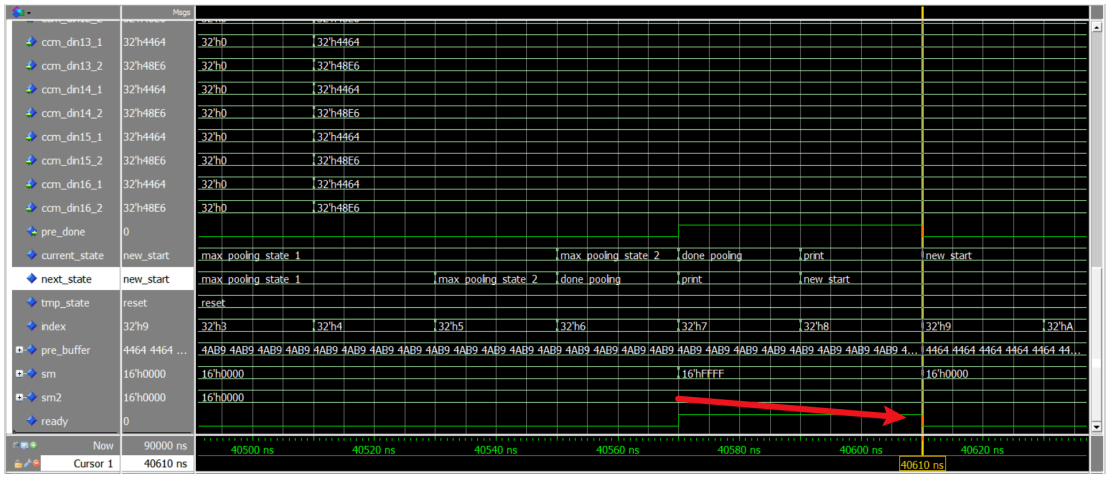

Figure 4.7: Input size 6x6: Pooling-ReLU-Encoding outputs the last result into the TXT file

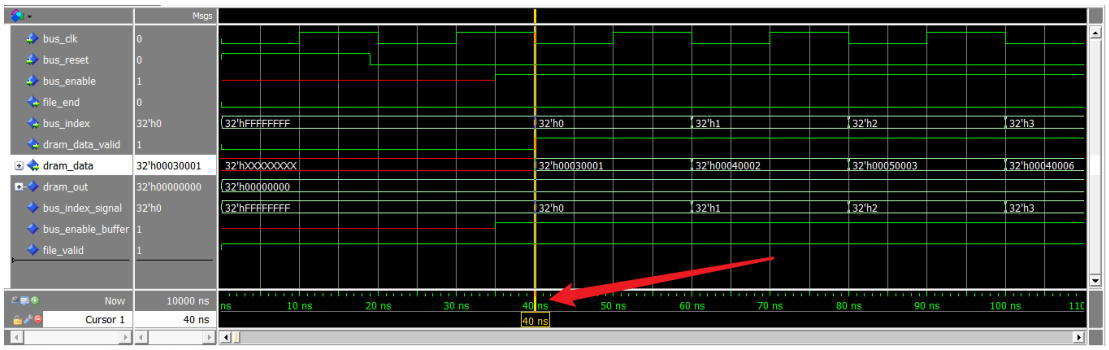

Figure 4.8: Input size 8x8: Input Bus begins to stream input data

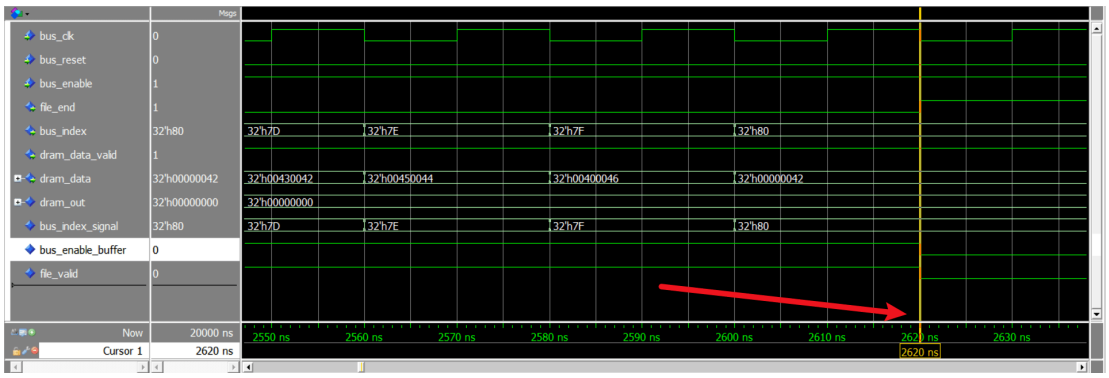

Figure 4.9: Input size 8x8: Input Bus finishes streaming

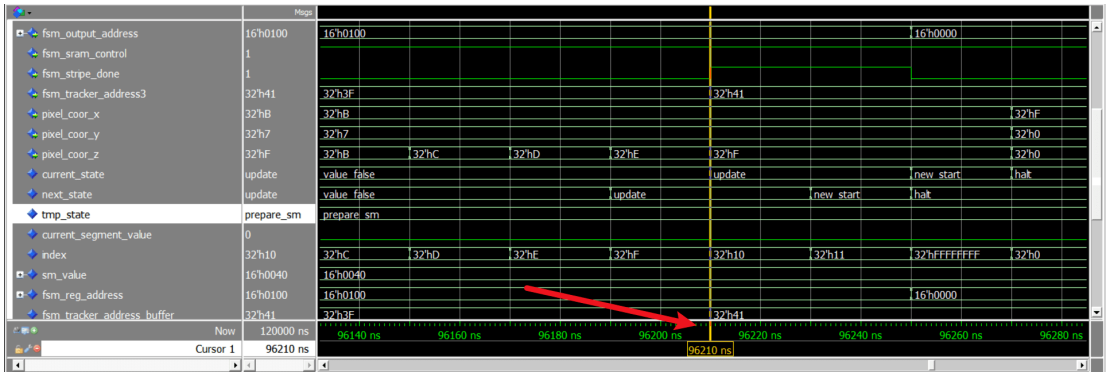

Figure 4.10: Input size 8x8: Input Data Processor streams out the last pixel in the input feature map

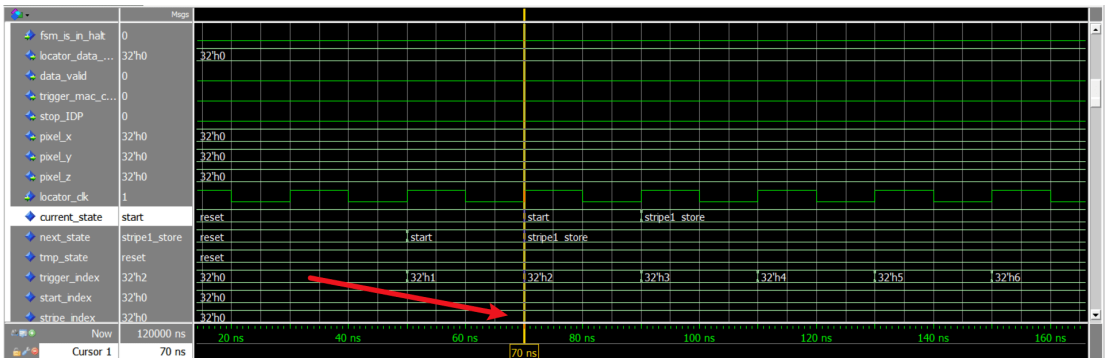

Figure 4.11: Input size 8x8: Compute Core Module begins to process decoded input feature maps

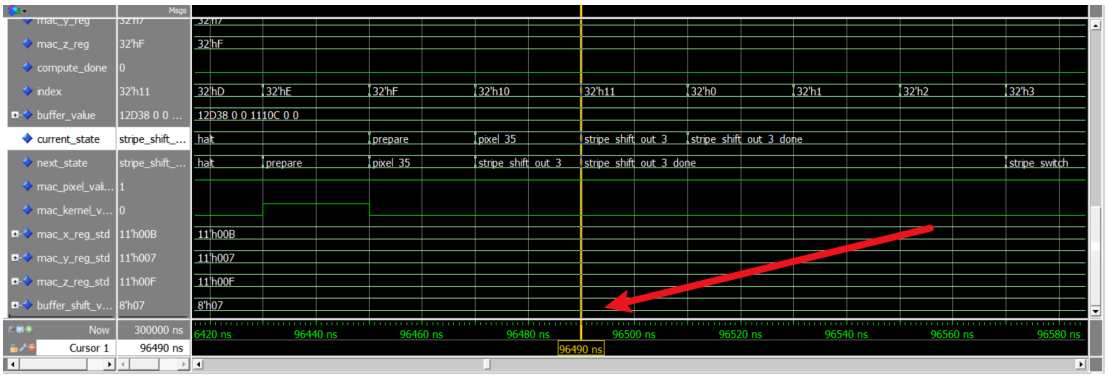

Figure 4.12: Input size 8x8: Compute Core Module finishes the convolution calculation of the last pixel in the input feature maps

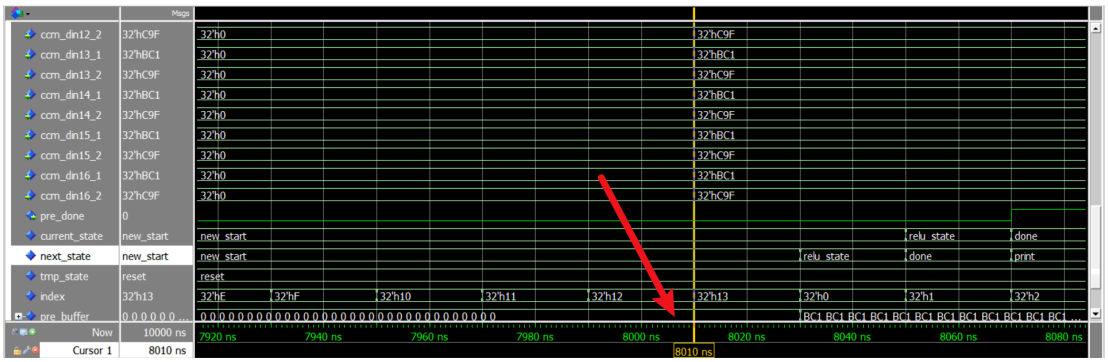

Figure 4.13: Input size 8x8: Pooling-ReLU-Encoding receives the first patch of convolution results

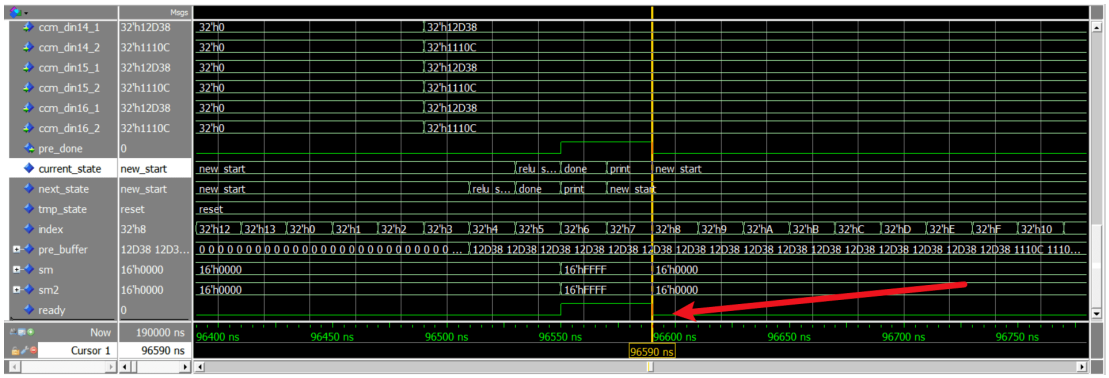

Figure 4.14: Input size 8x8: Pooling-ReLU-Encoding outputs the last result into the TXT file

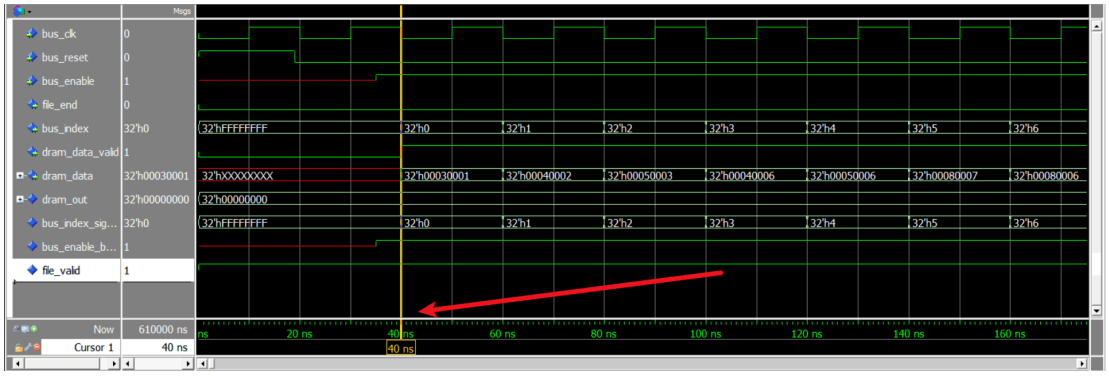

Figure 4.15: Input size 64x64: Input Bus begins to stream input data

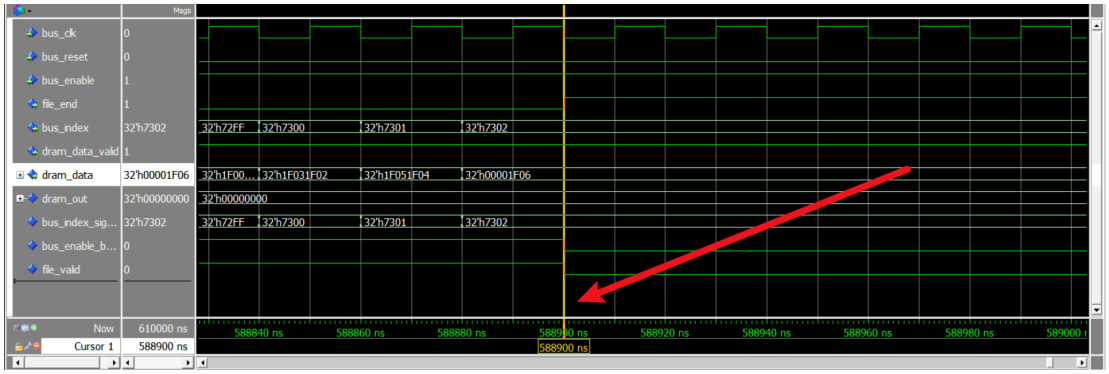

Figure 4.16: Input size 64x64: Input Bus finishes streaming

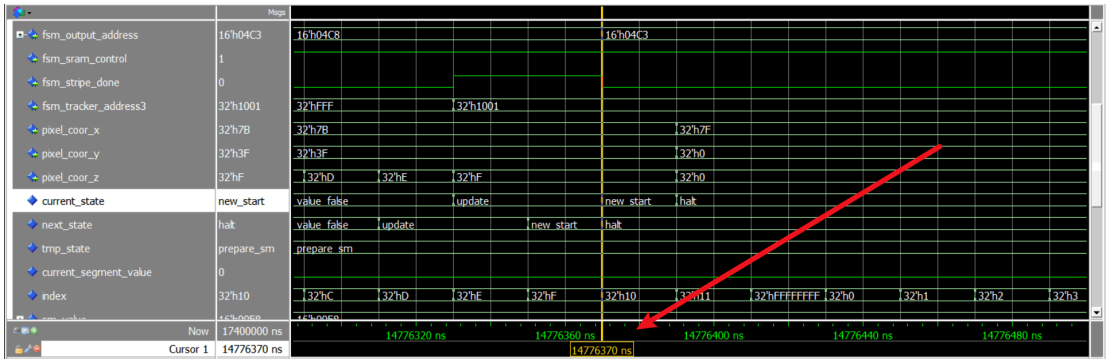

Figure 4.17: Input size 64x64: Input Data Processor streams out the last pixel in the input feature map

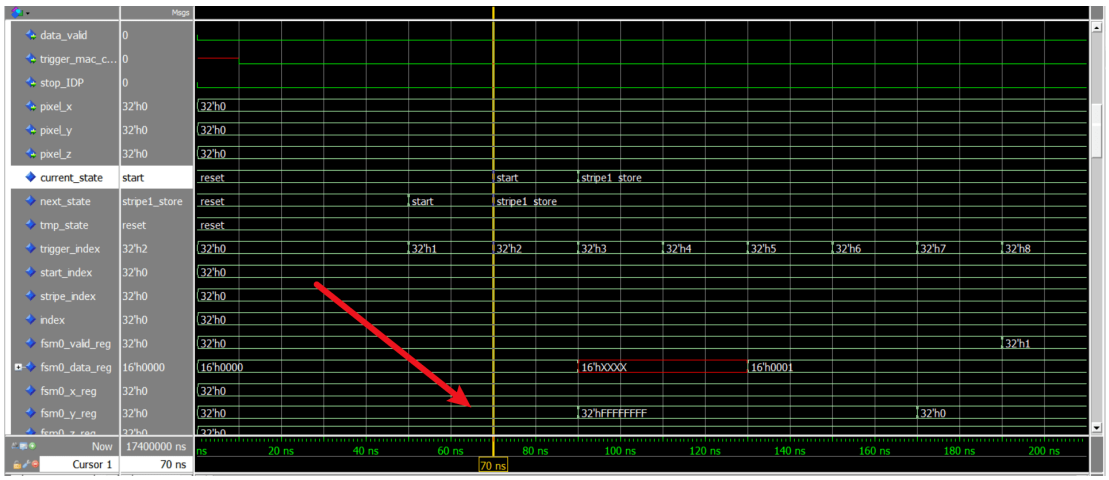

Figure 4.18: Input size 64x64: Compute Core Module begins to process decoded input feature maps

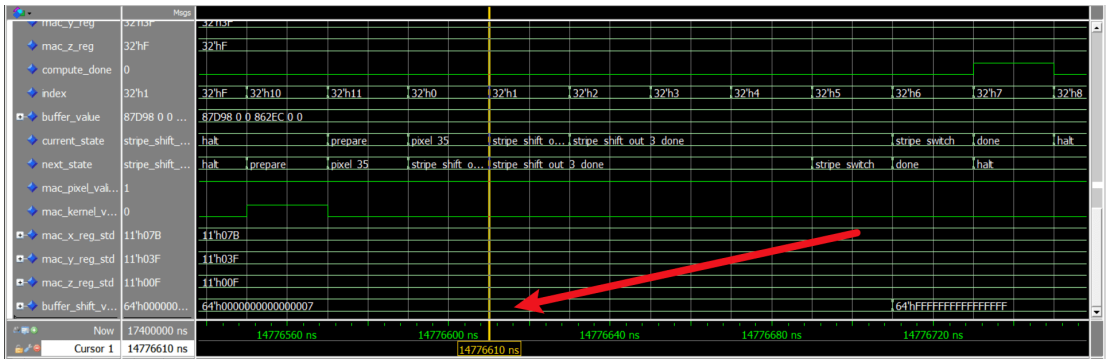

Figure 4.19: Input size 64x64: Compute Core Module finishes the convolution calculation of the last pixel in the input feature maps

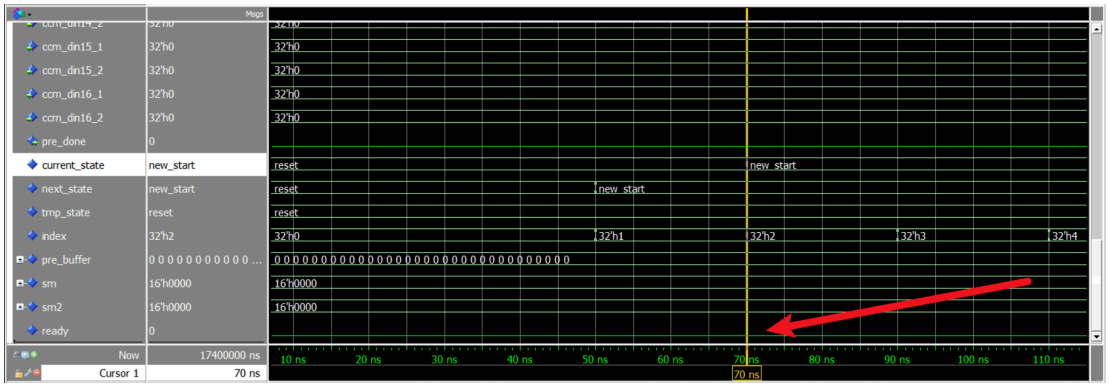

Figure 4.20: Input size 64x64: Pooling-ReLU-Encoding receives the first patch of convolution results

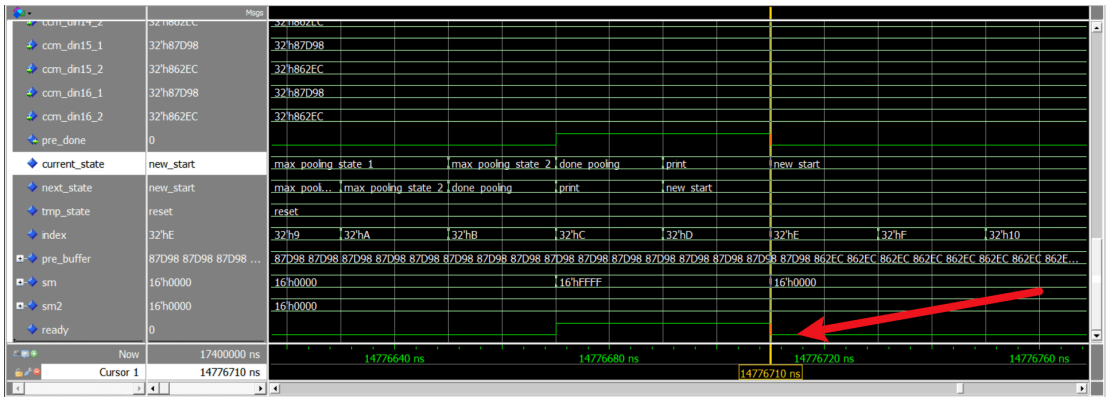

Figure 4.21: Input size 64x64: Pooling-ReLU-Encoding outputs the last result into the TXT file

Chapter 5

Conclusion and Future Works

5.1 Thesis Conclusion

This thesis aimed to implement a hardware accelerator for CNNs in VHDL based on the sparse representations of feature maps. The benefits of this accelerator come from four aspects:

1. The input feature maps are always compressed. This could reduce enormously the required memory size.

2. Zero pixels are not forwarded into the accelerator(except for very few special purpose pixels). This mechanism brings a vast energy pavement and an enormous reduction of computation cycles.

3. The ReLU, max pooling and Encoding stages are done on the fly. The PRE unit begins to work as soon as the results shifted out from the CCM unit. And the encoding stage compresses the output feature map again for further convolutional layers.

4. The output channel depth of the accelerator could be easily extended by duplicating more MACs. Meanwhile, the latency in the IDP and CCM unit won't increase.

5.2 Future Work

The accelerator implemented in this thesis could be further optimized in two aspects:

1. The accelerator is fully functional with input channel depth 16. In the case of an actual RGB image, an off-chip C/Python program is needed to extend the input channel depth from 3 to 16 by zero-padding another 13 input channels.

2. The accelerator is designed to let one MAC calculate one output channel. In future work, the MAC Controller could be upgraded to enable simultaneous multiple pixels calculation. The accelerator could split the computation of the current output channel over several clusters of MACs. One cluster is in charge of computing one output channel. And each cluster has the same number of MACs—every MAC stores the partial convolution results in their individual MAC Buffer. Afterward, when the MAC Buffers are shifting out the results, there needs an additional stage to merge the results from the MAC Buffers to produce the output feature map.

Appendix A

Input Data

The following data is used as input during simulation. Code line 1 to Code line 126 is the testing input for 6x6x16 simulation, with sparsity 84.375%. Code line 1 to Code line 258 is the testing input for 8x8x16 simulation, with sparsity 81.05%.

```
0000000000000001
0000000000000011
0000000000000010
0000000000000100
0000000000000011
0000000000000101
0000000000000110
0000000000000100
0000000000000110
0000000000000101
0000000000000111
0000000000001000
0000000000000110
0000000000001000
0000000000001001
0000000000000111
0000000000001001
0000000000001010
0000000000001011
0000000000001000
0000000000001010
0000000000001001
0000000000001011
0000000000001100
0000000000001010
0000000000001100
0000000000001101
0000000000001011
```

```
0000000000001101
0000000000001110
0000000000001111
0000000000001100
0000000000001110
0000000000001111
0000000000001101
0000000000001111
0000000000010000
0000000000010001
0000000000001110
0000000000010000
0000000000010001
0000000000010010
0000000000001111
0000000000010001
0000000000010010
0000000000010011
0000000000010100
0000000000010000
0000000000010010
0000000000010001
0000000000010011
0000000000010100
0000000000010010
0000000000010100
0000000000010101
0000000000010011
0000000000010101
0000000000010110
0000000000010111
0000000000010100
0000000000010110
0000000000010111
0000000000010101
0000000000010111
0000000000011000
0000000000011001
0000000000010110
0000000000011000
0000000000011001
0000000000011010
0000000000010111
0000000000011001
0000000000011010
0000000000011011
0000000000011100
0000000000011000
0000000000011010
```

```
0000000000011011
0000000000011001
0000000000011011
0000000000011100
0000000000011101
0000000000011010
0000000000011100
0000000000011101
0000000000011110
0000000000011011
0000000000011101
0000000000011110
0000000000011111
0000000000100000
0000000000011100
0000000000011110
0000000000011111
0000000000100000
0000000000011101
0000000000011111
0000000000100000
0000000000100001
0000000000100010
0000000000011110
0000000000100000
0000000000100001
0000000000100010
0000000000100011
0000000000011111
0000000000100001
0000000000100010
0000000000100011
0000000000100100
0000000000100101
0000000000100000
0000000000100010
0000000000100001
0000000000100011
0000000000100100
0000000000100010
0000000000100100
0000000000100101
0000000000100011
0000000000100101
0000000000100110
0000000000100111
0000000000100100
0000000000100110
0000000000100111
```

```
0000000000100101
0000000000100111
0000000000101000
0000000000101001
0000000000100110
0000000000101000
0000000000101001
0000000000101010
0000000000100111
0000000000101001
0000000000101010
0000000000101011
0000000000101100
0000000000101000
0000000000101010
0000000000101011
0000000000101001
0000000000101011
0000000000101100
0000000000101101
0000000000101010
0000000000101100
0000000000101101
0000000000101110
0000000000101011
0000000000101101
0000000000101110
0000000000101111
0000000000110000
0000000000101100
0000000000101110
0000000000101111
0000000000110000
0000000000101101
0000000000101111
0000000000110000
0000000000110001
0000000000110010
0000000000101110
0000000000110000
0000000000110001
0000000000110010
0000000000110011
0000000000101111
0000000000110001
0000000000110010
0000000000110011
0000000000110100
0000000000110101
```

```
0000000000110000
0000000000110010
0000000000110011
0000000000110001
0000000000110011
0000000000110100
0000000000110101
0000000000110010
0000000000110100
0000000000110101
0000000000110110
0000000000110011
0000000000110101
0000000000110110
0000000000110111
0000000000111000
0000000000110100
0000000000110110
0000000000110111
0000000000111000
0000000000110101
0000000000110111
0000000000111000
0000000000111001
0000000000111010
0000000000110110
0000000000111000
0000000000111001
0000000000111010
0000000000111011
0000000000110111
0000000000111001
0000000000111010
0000000000111011
0000000000111100
0000000000111101
0000000000111000
0000000000111010
0000000000111011
0000000000111100
0000000000111001
0000000000111011
0000000000111100
0000000000111101
0000000000111110
0000000000111010
0000000000111100
0000000000111101
0000000000111110
```

```
0000000000111111
0000000000111011
0000000000111101
0000000000111110
0000000000111111
0000000001000000
0000000001000001
0000000000111100
0000000000111110
0000000000111111
0000000001000000
0000000001000001
0000000000111101
0000000000111111
0000000001000000
0000000001000001
0000000001000010
0000000001000011
0000000000111110
0000000001000000
0000000001000001
0000000001000010
0000000001000011
0000000001000100
0000000000111111
0000000001000001
0000000001000010
0000000001000011
0000000001000100
0000000001000101
0000000001000110
0000000001000000
0000000001000010
0000000000000000
```

Bibliography

[1] Alessandro Aimar et al. «NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps». In: *CoRR* abs/1706.01406 (2017). arXiv: 1706.01406. URL: http://arxiv.org/abs/1706.01406 (cit. on pp. ii, 2, 10, 11, 16–18, 20).

[2] Jonathan Schmidt, Mário RG Marques, Silvana Botti, and Miguel AL Marques. «Recent advances and applications of machine learning in solid-state materials science». In: *npj Computational Materials* 5.1 (2019), pp. 1–36 (cit. on p. 5).

[3] Vinay Williams, Vasileios Argyriou, Peter Shaw, Christoph Montag, Georg Herdrich, Aaron Knoll, and Maximilian Moertl. «Development of PPTNet a Neural Network for the Rapid Prototyping of Pulsed Plasma Thrusters». In: Sept. 2019 (cit. on p. 5).

[4] Iqbal H Sarker. «Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions». In: *SN Computer Science* 2.6 (2021), pp. 1–20 (cit. on p. 7).

[5] Yang Xin, Lingshuang Kong, Zhi Liu, Yuling Chen, Yanmiao Li, Hongliang Zhu, Mingcheng Gao, Haixia Hou, and Chunhua Wang. «Machine learning and deep learning methods for cybersecurity». In: *Ieee access* 6 (2018), pp. 35365–35381 (cit. on p. 7).

[6] Mathias Kraus, Stefan Feuerriegel, and Asil Oztekin. «Deep learning in business analytics and operations research: Models, applications and managerial implications». In: *European Journal of Operational Research* 281 (Sept. 2019). DOI: 10.1016/j.ejor.2019.09.018 (cit. on p. 7).

[7] Sakshi Indolia, Anil Kumar Goswami, Surya Prakesh Mishra, and Pooja Asopa. «Conceptual understanding of convolutional neural network-a deep learning approach». In: *Procedia computer science* 132 (2018), pp. 679–688 (cit. on p. 7).

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «Imagenet classification with deep convolutional neural networks». In: *Advances in neural information processing systems* 25 (2012) (cit. on p. 8).

[9] Xingcheng Luo, Ruihan Shen, Jian Hu, Jianhua Deng, Linji Hu, and Qing Guan. «A deep convolution neural network model for vehicle recognition and face recognition». In: *Procedia Computer Science* 107 (2017), pp. 715–720 (cit. on p. 8).

[10] Tara N Sainath, Brian Kingsbury, Abdel-rahman Mohamed, George E Dahl, George Saon, Hagen Soltau, Tomas Beran, Aleksandr Y Aravkin, and Bhuvana Ramabhadran. «Improvements to deep convolutional neural networks for LVCSR». In: *2013 IEEE workshop on automatic speech recognition and understanding*. IEEE. 2013, pp. 315–320 (cit. on p. 8).

[11] Harry Pratt, Frans Coenen, Deborah M. Broadbent, Simon P. Harding, and Yalin Zheng. «Convolutional Neural Networks for Diabetic Retinopathy». In: *Procedia Computer Science* 90 (2016). 20th Conference on Medical Image Understanding and Analysis (MIUA 2016), pp. 200–205. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2016.07.014. URL: https://www.sciencedirect.com/science/article/pii/S1877050916311929 (cit. on p. 8).

[12] Ayşegül Uçar. «Deep Convolutional Neural Networks for facial expression recognition». In: *2017 IEEE International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*. IEEE. 2017, pp. 371–375 (cit. on p. 8).

[13] Luiz Zaniolo and Oge Marques. «On the use of variable stride in convolutional neural networks». In: *Multimedia Tools and Applications* 79 (May 2020). DOI: 10.1007/s11042-019-08385-4 (cit. on pp. 8, 9).

[14] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. «Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead». In: *IEEE Access* 8 (2020), pp. 225134–225180. DOI: 10.1109/ACCESS.2020.3039858 (cit. on pp. 8, 12, 15).

[15] Karen Simonyan and Andrew Zisserman. «Very deep convolutional networks for large-scale image recognition». In: *arXiv preprint arXiv:1409.1556* (2014) (cit. on pp. 12, 13).

[16] Bibo Shi et al. «Learning better deep features for the prediction of occult invasive disease in ductal carcinoma in situ through transfer learning». In: Feb. 2018, p. 98. DOI: 10.1117/12.2293594 (cit. on p. 13).

[17] Pornntiwa Pawara, Emmanuel Okafor, Olarik Surinta, Lambert Schomaker, and Marco Wiering. «Comparing Local Descriptors and Bags of Visual Words to Deep Convolutional Neural Networks for Plant Recognition». In: Feb. 2017. DOI: 10.5220/0006196204790486 (cit. on p. 14).

[18] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. «Going Deeper With Convolutions». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015 (cit. on p. 14).

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep residual learning for image recognition». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on pp. 14, 15).

[20] Thorsten Hoeser and Claudia Kuenzer. «Object Detection and Image Segmentation with Deep Learning on Earth Observation Data: A Review-Part I: Evolution and Recent Trends». In: *Remote Sensing* 12 (May 2020). DOI: 10.3390/rs12101667 (cit. on pp. 14, 15).

[21] Moritz B. Milde, Daniel Neil, Alessandro Aimar, Tobi Delbrück, and Giacomo Indiveri. «ADaPTION: Toolbox and Benchmark for Training Convolutional Neural Networks with Reduced Numerical Precision Weights and Activation». In: *CoRR* abs/1711.04713 (2017). arXiv: 1711.04713. URL: http://arxiv.org/abs/1711.04713 (cit. on p. 16).

[22] Evangelos Stromatias, Daniel Neil, Michael Pfeiffer, Francesco Galluppi, Steve B Furber, and Shih-Chii Liu. «Robustness of spiking deep belief networks to noise and reduced bit precision of neuro-inspired hardware platforms». In: *Frontiers in neuroscience* 9 (2015), p. 222 (cit. on p. 16).

[23] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. «Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks». In: *IEEE journal of solid-state circuits* 52.1 (2016), pp. 127–138 (cit. on p. 18).