MICROARCHITECTURE FOR A SPIKING NEURAL NETWORK ACCELERATOR

by

YIMIN WANG

in the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

of the

NATIONAL UNIVERSITY OF SINGAPORE

2022

Contents

List of Figures **iii**

List of Tables **iv**

1 Introduction **1**

1.1 Motivation: Challenges and Opportunities 2

1.1.1 Programmability 2

1.1.2 Scalability 3

1.1.3 On-chip learning 3

1.1.4 FPGA vs ASIC 4

1.2 Contribution and Report Outline 4

2 Proposed Architecture for Fully Connected Network Accelerator **6**

2.1 FCN Structure 6

2.1.1 Spiking-based networks 6

2.2 Acceleration Scheme 8

2.2.1 Event-based computing paradigm 8

2.2.2 Parallelism enabled by Multi-cluster 8

2.2.3 Minimizing communication to host processor 9

2.3 Hardware Architecture 10

2.3.1 Main modules 11

2.3.2 Mapping method 13

2.3.3 Work flow 14

2.4 Instruction Set 14

2.5 Memory Organization 16

2.5.1 On-chip memory 16

2.5.2 On-board memory 17

3 Hardware Implementation **19**

3.1 General Development Flow 19

3.1.1 HLS IP development 19

3.1.2 Embedded Processor Development 20

3.2 Resource Utilization 21

3.3 Energy 22

3.4 Performance 23

3.5 Scalability 23

4 Future Work **25**

4.1 Convolutional Spiking Neural Networks 25

4.2 On-chip learning 25

5 Conclusion **26**

Bibliography **28**

List of Figures

2.1 The concept of fully connected networks: (a) matrix-based processing; (b) event-driven processing. 7
2.2 Distribution of output spike number per image in Hidden Layer 1. . . 9
2.3 Distribution of output spike number per image in Hidden Layer 2. . . 9
2.4 Distribution of output spike number per image in Output Layer. . . . 10
2.5 A general FPGA-base hardware accelerator. 11
2.6 Top-level block diagram. 11
2.7 Intra-cluster structure. 12
2.8 Method of mapping one FCN layer onto a cluster. 14
2.9 Method of mapping the whole FCN onto 3 clusters. 15

3.1 C file structure. 20
3.2 The block design diagram in Vivado for Type 1. 21
3.3 The block design diagram in Vivado for Type 2. 22

List of Tables

2.1 FCN benchmarks 7
2.2 Layer-wise sparsity enabled by spiking conversion 8
2.3 Instruction set architecture 18

3.1 Hardware Resource Consumption in HLS for one Cluster 22
3.2 Hardware Resource Consumption in HLS for One IP (Type 1) with 12 Clusters 23
3.3 Resource Utilization for the Whole Vivado Project 23
3.4 Performance 23

Chapter 1

Introduction

Neural networks have shown outstanding results in various applications such as computer vision and automation systems. Both the inference and training process have been significantly accelerated by the employment of customized hardware platforms with massively parallel architectures [31]. The utilization of traditional Artificial Neural Networks (ANNs) may become impracticable as the demand for embedded low-power solutions in edge-AI applications grows.

In recent years, Spiking Neural Network (SNN) which mimics the biological mechanism gains intensive research interest due to high energy efficiency compared to ANNs. Information is represented in SNNs as a time-domain sequence of spikes. Each neuron in an SNN has an internal membrane potential, which is updated as the neuron's input spikes. When a neuron's potential exceeds a certain threshold, it produces a spike. Because neurons do not require hardware multipliers and spikes only require single-bit communication between neurons, SNNs are well-suited for compact, low-power implementations. Furthermore, only a small percentage of neurons generate spikes at any given time, decreasing the hardware burden even further.

Many neuromorphic processors have been developed, including SpiNNaker [10], Neurogrid [3], TrueNorth [1], DYNAPs [19], and Loihi [8]. The architecture of these neuromorphic processors is defined by highly connected neurons, huge parallel processing, and co-located computation and memory. These architectures are mostly used to run SNNs. SNN is the third generation of artificial neural networks, and it closely resembles real neural networks. Biological neurons convey information and execute computation using spikes in a neuromorphic computing system, whereas

analog neurons communicate with each other using real values in a typical DNN computer system. Numerous applications have been implemented using SNNs, including electrical load forecasting [15], image processing [17] [7] [13], robot control systems [6] [2] and brain-machine interfaces [9].

Custom hardware acceleration for the neural network has always been a critical problem. GPU is a prevalent way to accelerate ANN by exploiting massive parallel computing, but this approach is not efficient for SNN acceleration. By making the most of its sparsity and temporal connections, dedicated event-driven hardware architecture shows great potential for SNN, which has been validated by many existing research. These efforts mainly fall in two folds: 1) Large-scale multi-core platforms are produced by joining numerous chips into boards and racks, however this strategy is not practicable for resource-constrained edge devices [3] [10] [18] [28] [33] [29]; 2) fine-tuned embedded system platforms, however with limited flexibility to support multiple SNN configurations [11] [12] [21] [34] [35]. Hence, it is meaningful to pave a new way on leveraging the balance between these two directions. Therefore, we are motivated to design a programmable neuromorphic accelerator that is capable of emulating an SNN with an arbitrary number of neurons, number of layers, and number of connections, compatible to both fully connected networks and convolutional networks, on the embedded systems platforms.

1.1 Motivation: Challenges and Opportunities

1.1.1 Programmability

Programmability is a key enabler toward arbitrary SNN architecture and hyper-parameters. [22] proposed an instruction set for SNN processing with high-level abstraction. It generally contains 5 instructions as *cfg*, *spre*, *spost*, *cmp* and *lrn*, where the instruction *spre* and *spost* is able to realize any connections between the neurons. This ISA is compact and convenient for programming whereas the shortcoming is that due to high-level abstraction the implementation on FPGA relies heavily on frequent data transfer between the host processor and the co-processor, which yields an extra burden on latency. Another compact instruction set is proposed in [24]. As this is an ASIC design, the energy consumption of inter-processor

data transfer is mitigated. Moreover, the instructions are generated automatically by sensing the condition of the spike queue and spike cluster, which makes the design more energy-efficient. However, the main concern for the mentioned highly abstract instructions is that the execution time of each instruction is undetermined. Comparing to a fine-grained instruction set exemplified by *mov*, *add*, etc., where each instruction will definitely consume 1 clock cycle, the coarse-grained instruction set may take more time margin.

1.1.2 Scalability

Scalability requires duplicable hardware resources and flexible mapping strategies. Prior efforts [10] [18] [28] [3] [33] [29] on specialized SNN hardware involve spatial architectures, in which each neuron is assigned into a fixed processing element, and large-scale networks are realized by connecting massive chips into a system. As exemplified by TrueNorth [1], researchers explored spatial architectures that require large transistor counts (5.4 billion in the case of Truenorth [1]). Multiple chips are connected to boards and racks to create larger networks. In the case of small or low-cost applications, this strategy is not practicable. For edge devices, on-chip scaling, termed multi-core/multi-cluster design [24] is better suited than the above-mentioned on-board scaling, termed multi-chip design. [24] proposed a method to map any given SNN to the multi-cluster architecture such that the workload is balanced across multiple clusters while pipelining across layers of the network to improve performance. However, the transferability of the existing mapping methods to different platforms is still limited. It should be optimized manually for dedicated hardware design.

1.1.3 On-chip learning

SNN training which is both efficient and effective has always been a major challenge. Because the transfer functions of neurons in an SNN are non-differentiable, backpropagation cannot be employed. [32] [16]. Various approaches range from unsupervised learning and STDP rules [4] [27] [23], to supervised variations of error backpropagation [5] [20] and direct ANN to SNN conversion [25] [14]. Backpropagation and STDP-based learning has been realized on-chip be previous works [22] [26].

While the conversion-based method does not apply to on-chip learning, but the SNN generated in this way can be used for on-chip evaluation. [26] proposed a system that accelerates a SNN training process where various algorithm including supervised and unsupervised that is running on the CPU, receiving spike information feedbacked by the FPGA accelerator, where the learning method is based on a combination of the direct conversion method and a temporal variation of the backpropagation algorithm. The benchmark of these three main methods is still to be investigated further.

1.1.4 FPGA vs ASIC

A Field Programmable Gate Array (FPGA) is an integrated circuit developed to be configured by a customer or a designer in a close-to-hardware manner after manufacturing. The FPGA configuration is generally specified using the hardware description language (HDL). An Application-Specific Integrated Circuit (ASIC), is an integrated circuit particularly designed for a domain-specific usage, rather than intended for general-purpose applications. In the previous years of CNN development, the ASIC solution was preferred since it was possible to fully parallelize the whole architecture and achieve the highest energy-efficiency. But in the latest period a lot of researchers decided to switch the implementation using FPGAs [22] [26] [24]. The fundamental reason for this shift is that ASIC processors have a high non- recurrent cost, and since the neural network context is still growing, they wanted a more flexible solution. This characteristic was found in FPGA even if in some cases the performances were a little bit lower than ASIC.

1.2 Contribution and Report Outline

This report presents a set of custom instructions and corresponding hardware implementation on an FPGA platform that is capable of emulating the SNN of various sizes and depths. The proposed approach allows the SNN architecture to be defined completely in software, which negates the need to re-synthesize the hardware implementation when any parameter in the SNN architecture is changed. The following chapters will show that the proposed design is also hardware-friendly and consumes a small number of logic gates and memory resources on the FPGA

platform, which makes it suitable for edge computing applications.

The outline of this report is organized as follows: chapter 2 introduces the background of SNN, acceleration scheme, and detailed hardware architecture design; chapter 3 introduces the general development flow based on Vivado HLS and SDK environment, and the evaluation results; chapter 4 gives a brief prospect for future work, and finally, we come to the conclusion.

Chapter 2

Proposed Architecture for Fully Connected Network Accelerator

2.1 FCN Structure

2.1.1 Spiking-based networks

In a ANN, an input vector is presented at one time, and processed layer-by-layer, producing one output value. Whereas in an SNN, inputs are in a temporal sequence of events, and neurons integrate and generate spikes to send information to the subsequent layers. This brings tremendous possibilities to real-time applications.

The spiking neuron model that we used in this project is the canonical integrate-and-fire (I&F) model. The temporal behavior of the membrane voltage v_{mem} is given by,

$$\frac{dv_{mem}(t)}{dt} = \sum_{i} \sum_{s \in S_i} \omega_i \delta(t - s) \tag{2.1}$$

where ω_i is the weight of the i-th incoming synapse. $\delta(t-s)$ is the delta function, and $S_i = \{t_i^0, t_i^1, ...\}$ contains the temporal information of the i-th pre-synaptic neuron. If the membrane voltage exceeds the spiking threshold V_{thr}, a post-spike is generated and the membrane voltage is reset to a low potential V_{res}.

Neural networks is composed of a set of dependent non-linear functions where each function consists of a neuron. As to fully connected layers, the input vector is transformed into the output vector through multiplication by a weight matrix. Fully

connected layers mainly account for computational complexity in neural networks. Figure 2.1(a) shows the concept of fully connected network. When computing a fully connected layer on CPU and GPU, they perform matrix multiplication, which is resourse-consuming. Under the spike-based neural networks scenario, the fired spike is highly sparse. As shown in Figure 2.1(b), the red circles indicate the fired neurons at the current timestep, from which we can find that only a few neurons fire post-spikes. If only calculating the fired connections as illustrated in (b), the total processing time and energy can be reduced significantly.

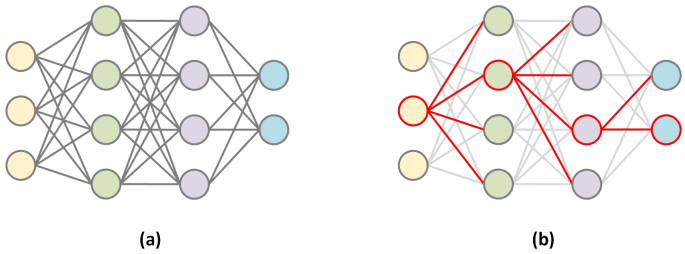

Figure 2.1: The concept of fully connected networks: (a) matrix-based processing; (b) event-driven processing.

In this chapter, the fully connected networks that we use are proposed in [30]. The parameter of this network is shown in Table 2.1. This work proposes a possible way to convert conventional neural networks to SNNs without introducing accuracy penalty.

Table 2.1: FCN benchmarks

	Type	Neurons	Connections	Synapses
Layer1	Input layer	784	/	/
Layer2	Hidden layer	1200	784×1200	784×1200
Layer3	Hidden layer	1200	1200×1200	1200×1200
Layer4	Output layer	10	1200×10	1200×10

2.2 Acceleration Scheme

2.2.1 Event-based computing paradigm

As mentioned above, the spiking conversed network has high sparsity on the spike numbers. Table 2.2 shows the average spike number in each layer. The statitics for each layer are shown in Figure 2.2, Figure 2.2, and Figure 2.2 respectively. It is easy to find that by introducing the event-based method, the total computation time can be reduced by 98.12% ideally, which is calculated by the following equation.

$$\begin{aligned} Reduction\ Rate &= 1 - \frac{\sum_{layer\ 1-4} Connection \times SpikeingProportion}{\sum_{layer\ 1-4} Connection} \\ &= 1 - \frac{784 \times 1200 \times 2.59\% + 1200 \times 1200 \times 1.41\% + 1200 \times 10 \times 1.76\%}{784 \times 1200 + 1200 \times 1200 + 1200 \times 10} \\ &= 98.12\% \end{aligned}$$

Table 2.2: Layer-wise sparsity enabled by spiking conversion

Type	Total Neurons	Average spiking neurons per image	Average spiking neurons per image per timestep	Spikeing proportion
Input layer	784	727.28	20.78	2.59%
Hidden layer	1200	592.23	16.92	1.41%
Hidden layer	1200	737.73	21.08	1.76%
Output layer	10	10.28	0.29	2.9%

2.2.2 Parallelism enabled by Multi-cluster

In the proposed architecture, there are 16 clusters in total. As far as the fully connected network is concerned, each layer requires a cluster, therefore, 16 clusters can be divided into 5 groups, where every group with 3 clusters can process a channel of input image independently. However, one constraint is the number of interfaces communicating with DDR memory, which is maximally four interfaces. Thus, 4 groups are utilized for the fully connected network to exploit the parallelism.

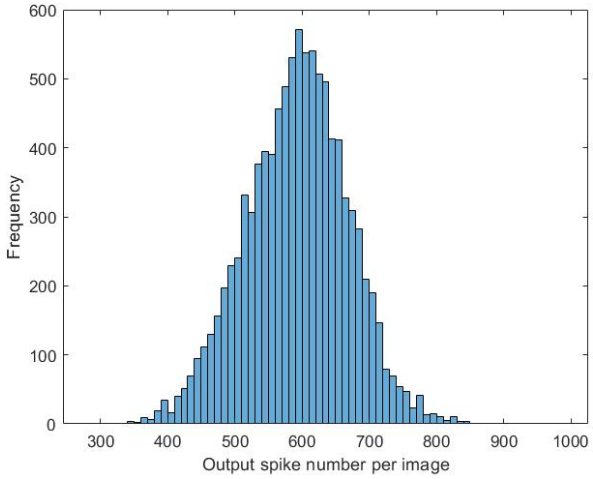

Figure 2.2: Distribution of output spike number per image in Hidden Layer 1.

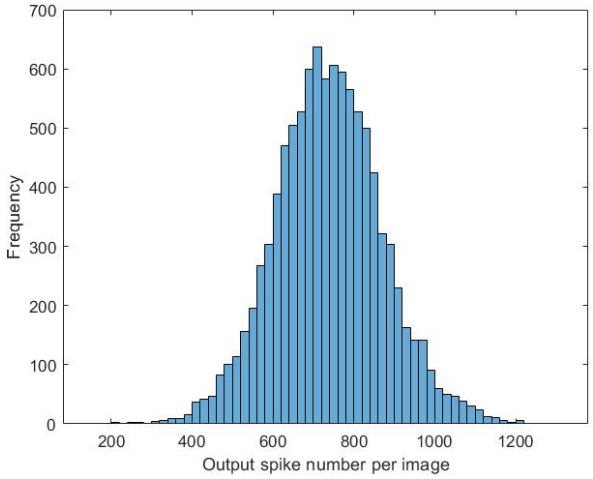

Figure 2.3: Distribution of output spike number per image in Hidden Layer 2.

2.2.3 Minimizing communication to host processor

In the proposed architecture, the PS part only needs to send the start signal to the PL part, The computations and data transfer can be operated under the control of the PL part. This can save the time spent on the communications between the host processor and co-processor.

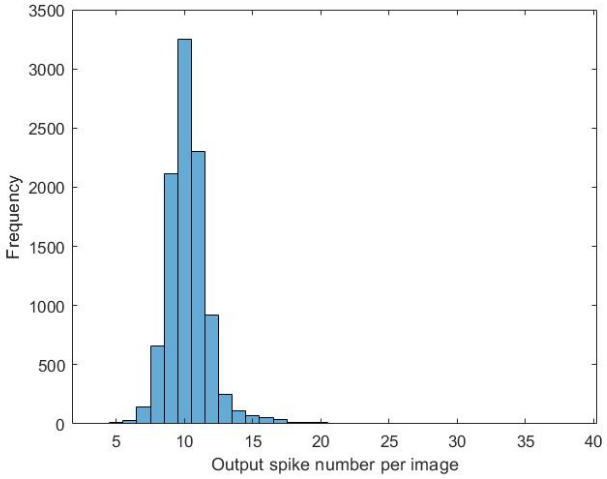

Figure 2.4: Distribution of output spike number per image in Output Layer.

2.3 Hardware Architecture

The hardware development platform used in this work is the Zynq Ultrascale+ ZCU106 Evaluation Platform. Zynq UltraScale+ devices combine real-time control hard engines for graphics, video, waveform, and packet processing capabilities in the programmable logic, allowing for 64-bit CPU scaling. The combination of an ARM-based system for advanced analytics and on-chip programmable logic for task acceleration opens up a world of possibilities for applications such as intelligent healthcare, self-driving vehicles, and the Internet of Things.

Currently reported FCNs involve a large amount of data. The on-chip block memory (BRAM) resources of the FPGA, which is 38 MByte for UltraScale system, is insufficient to store all the data, requiring megabytes of external on-board memory. Hence, a general accelerator comprises three levels: on-board external memory, on-chip buffers, and registers as shown in Figure 2.5. The general work flow is: (i) load data from external memory to on-chip buffers through the direct memory access (DMA) interface; (ii) transfer the necessary data into corresponding registers and PEs; (iii) when the processing in PEs are completed, the results are transferred backwards and used as input information to the subsequent layer.

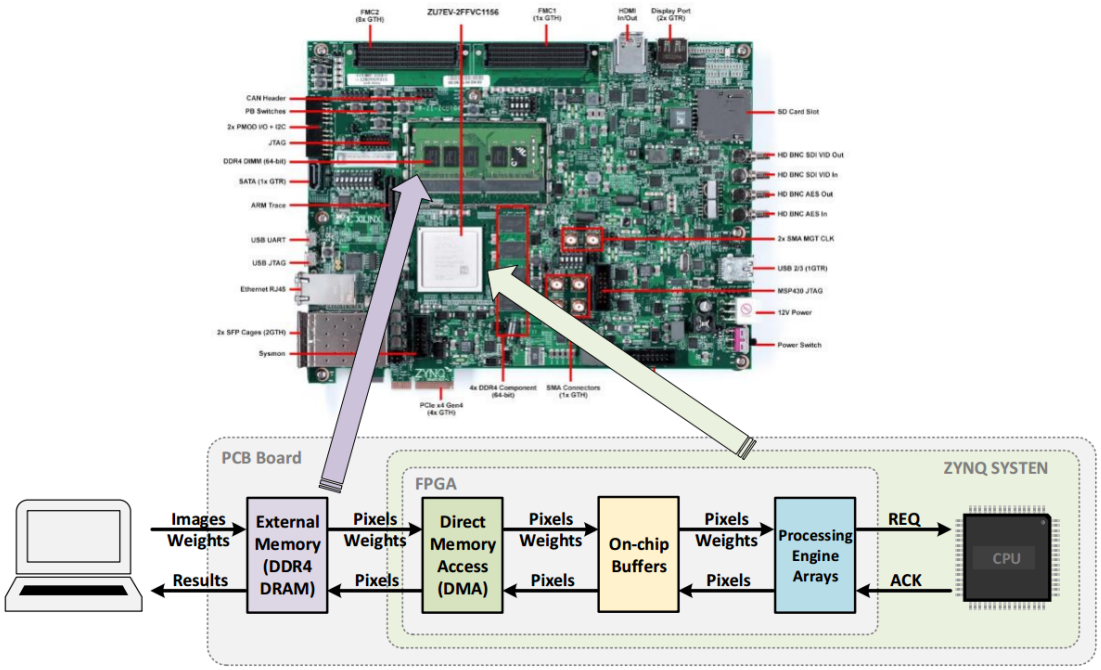

Figure 2.5: A general FPGA-base hardware accelerator.

2.3.1 Main modules

The overview of the proposed accelerator is shown in Figure 2.6. The accelerator is composed of the cluster array, the global interface, the synthesizer and the spike scheduler.

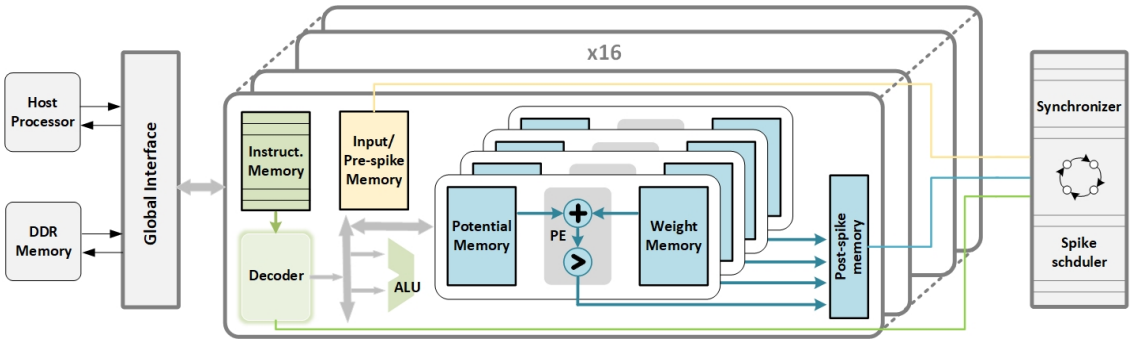

Figure 2.6: Top-level block diagram.

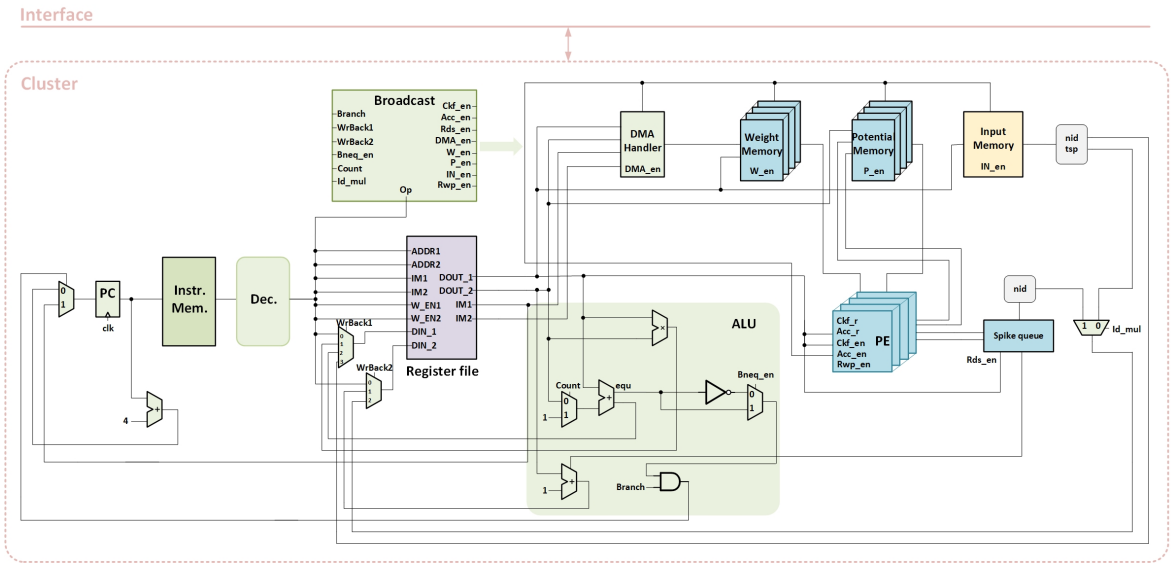

Figure 2.7: Intra-cluster structure.

2.3.1.1 Cluster

A cluster is capable to process a single lightweight layer or part of a heavyweight layer. It is composed of 4 PEs, 4 weight memory units, 4 potential memory units, an instruction memory unit, a decoder, an ALU, a post-spike memory unit, and an input/pre-spike memory (input memory is included in clusters for input layers, and pre-spike memory is included in clusters for hidden layers and output layers). Contents in input memory and instruction memory are pre-loaded, while that in weight memory is loaded through DDR interface when executing the instruction **ldw**, which will be introduced in section 2.4. The detailed network information is configurable in the program, which enables the flexibility of this architecture.

2.3.1.2 Global interface

The global interface supports communications to the host processor (which is an ARM-based core on Zynq-Ultrascale platform in this experiment), and DDR memory. The interface to access the host processor is realized by the AXI master interface on PS (Processing System), connected to AXI slave interface on PL (Processing Logic). The interface to access DDR memory is chosen as high-performance slave interface S_AXI_HP{0:3}_FPD, which supports DMA (direct memory access).

They support 32/54/128-bit data width and up to 49-bit width address width. Both individual mode data transfers and burst mode data transfers are supported.

2.3.1.3 Synchronizer and spike scheduler

The synchronizer and spike scheduler serves as a global controlling unit to update current timestep and latest pre-spike information to each cluster. When the decoder in a cluster detects the instruction **syn**, the cluster should break off the serial instruction execution, and then wait for the start signal from the synchronizer. After the processing in subsequent clusters within the current timestep is completed, the synchronizer will issue a start signal to the clusters of the first layer and update the current timestep by adding 1. This ensures the correct temporal sequence. The spike scheduler arranges the update of pre-spike memory. At the end of each timestep, the spike scheduler reset the pointer of each pre-spike memory, and store the total post-spikes of every layer in the timestep.

2.3.2 Mapping method

Figure 2.8 shows the mapping strategy of FCN onto a cluster. Contents in pre-spike memory store the neuron ID that fires spikes of the last layer in the current timestep. As illustrated in Figure 2.8, there are 1200 neurons in the layer, every time read in a fired neuron ID of the last layer, each neuron's potential is to be updated as the addition of current potential and the corresponding weight. The 1200 operations are partitioned equally to 4 PEs, i.e. 300 potential data and 300 weight data are contained in the potential memory and weight memory of each PE.

In this work, as mentioned above, the fully connected network we used contains three layers. The allocation for every layer is: (i) Cluster-0 is allocated for the first layer (Hidden layer 1) and each PE is assigned for 300 neurons; (ii) Cluster-1 is allocated for the second layer (Hidden layer 2) and each PE is assigned for 300 neurons; (iii) Cluster-2 is allocated for the third layer (Output layer) and only one PE is assigned for 10 neurons, whereas the other 3 PEs are stand-by.

Currently, the three clusters are operating in a serial sequence, since each layer's input is dependent on the output spikes of the previous layer. But this architecture has the potential to extend inter-cluster parallelism in the future.

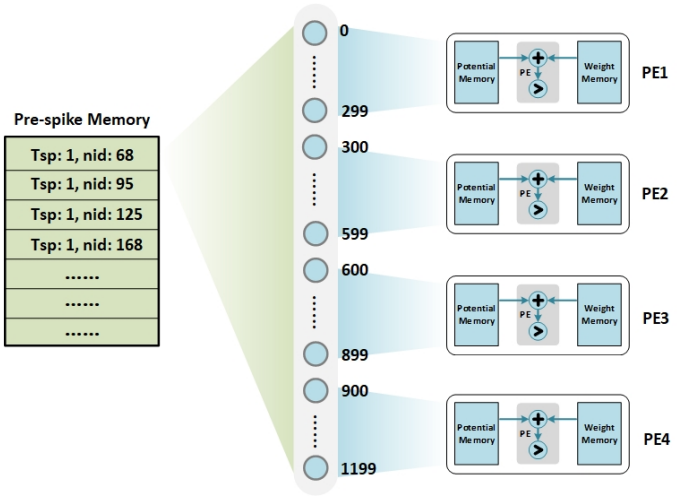

Figure 2.8: Method of mapping one FCN layer onto a cluster.

2.3.3 Work flow

Firstly, the host processor sends a start signal to the accelerator. Then the processing in the clusters will be controlled by executing the instructions serially. By sensing certain flags, the synthesizer and spike scheduler can control the cooperation between clusters. Finally, when the whole processing of the network is done, the accelerator will send the done signal to the host processor.

2.4 Instruction Set

Table 2.3 shows the detailed format and function of each instruction used in this work. There are three types of instructions: 1) Move instruction: **mov**; 2) ALU-related instructions: **add**, **mul**, **sft**, and **mod**; 3) Flow control instructions: **beq**, **bneq**, **jmp**, and **syn**; 4) Pre-spike instructions: **rdi**, and **rds**; 5) PE-related instructions: **rwp**, **acc**, and **ckf**; 6) DDR-based instruction: **ldw**.

Algorithm 1 gives a simple example of program using the proposed instruction set to emulate an SNN on our hardware implementation. This sample program is suitable for clusters processing the first layer. In the program, (i) firstly, it reads in the neuron id and corresponding timestep of the pre-spikes (line 1); (ii) then judge whether this pre-spike should be processed in the current timestep or not, if the timestep matches, the program proceeds to step iii (line 2); (iii) load in

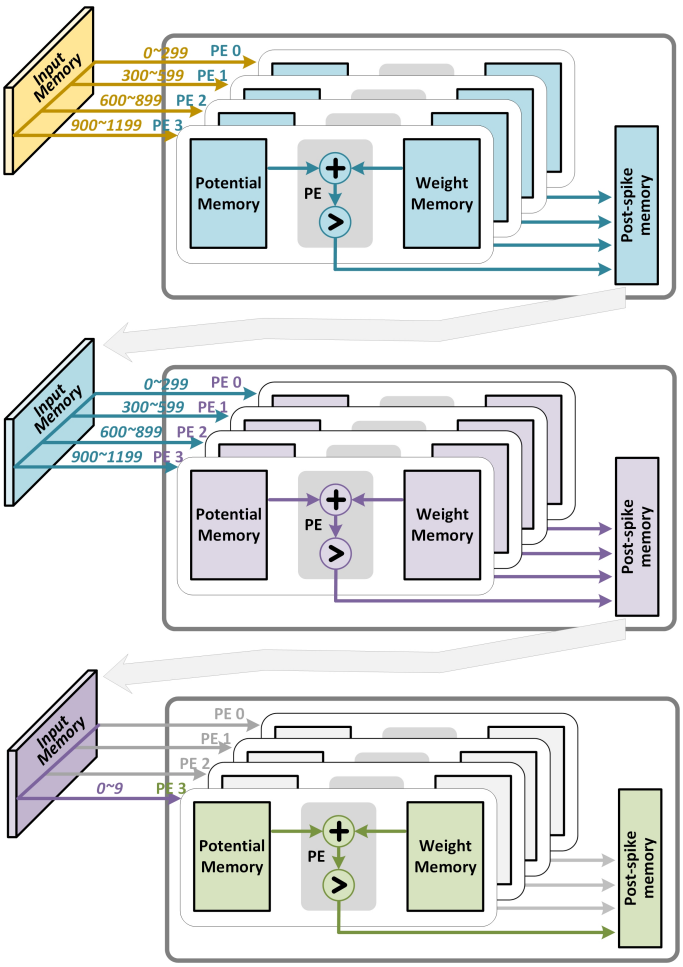

Figure 2.9: Method of mapping the whole FCN onto 3 clusters.

the corresponding weight contents from DDR to weight memory (line 11-14); (iv) read the current potential and weight into PE, and calculate the latest potential; repeat this step until all the neuron potentials in a PE are updated (line 15-19); (v) jump to the step i (line 20-21); (vi) in step 2, if the timestep unmatch, which means the processing of pre-spikes in the current timestep is completed, thus the program proceeds to step vii; (vii) compare the latest potential with the threshold, if exceeding the threshold, generate the post-spikes, repeat this step until all the neuron potential in a PE are checked (line 4-6); (viii) wait until the other subsequent clusters to complete; when completed, increase the timestep and jump again to step i (line 6-10).

The detailed structure inside a cluster is illustrated in Figure 2.7.

Algorithm 1: An example of a program using the proposed instruction set to emulate an SNN on our hardware implementation.

```
 1: L_read:     rdi src_id, spike_step, input_spike_idx
 2:             beq spike_step, time_step, !L_PE_init
 3:             mov potential_idx, #00, #0
 4: L_ckf:      ckf potential_idx, #0100000000000000, #1111
 5:             add potential_idx, potential_idx, #00, #1
 6:             bneq potential_idx, num, !L_ckf
 7:             syn
 8:             beq spike_step, r31, !endimg
 9:             add time_step, time_step, #00, #1
10:             jmp !L_read
11: L_PE_init:mov potential_addr, #00, #0
12:             mul weight_ddr_addr, src_id, #00, d2
13:             mov weight_addr, #00, #0
14:             ldw weight_addr, weight_ddr_addr, num, #1111
15: L_PE_on: rwp weight_addr, potential_addr, #1111
16:             acc potential_addr, #1111
17:             add potential_addr, potential_addr, #00, #1
18:             add weight_addr, weight_addr, #00, #1
19:             bneq potential_addr, num, !L_PE_on
20:             add input_spike_idx, input_spike_idx, #00, #1
21:             jmp !L_read
22: endimg:    mov done, #00, #1
```

2.5 Memory Organization

2.5.1 On-chip memory

Zynq Ultrascale+ ZCU106 Evaluation Platform includes on-chip memory in the form of block RAM (BRAM), distributed RAM (DRAM), and UltraRAM (URAM). The main on-chip memory in the PL part is Block RAM (BRAM). There are overall 32MB BRAM available.

For the proposed architecture, five types of buffer are used on the PL side: instruction memory, input memory, potential memory, weight memory, and spike memory.

Instruction memory: Each cluster contains a block of instruction memory. The width of instruction memory is 32-bit, and the depth is 64. The storage pattern please refer to section 2.4.

Input memory: Only clusters assigned for the first layer require a block of

input memory. The width of input memory is 32-bit, the storage pattern is [23:16] stores the timestep, and [15:0] stores the neuron ID. The depth of input memory is 2048, which depends on the maximum spike number in the input layer for one image. As to the dataset used in this work, the maximum number is 1801.

Potential memory: Each cluster contains four blocks of potential memory, assigned for each PE respectively. The width of potential memory is 32-bit. In this work, the original float-point type weight and potential are translated into fixed-point type by multiplying 2^{11}. Based on the statistics, 24-bit is enough for the fixed-point type potential and weight. However, since the weight potential data should be loaded from off-chip DDR, the interface requires the data width should be the power of 2. So, the width of potential memory and weight memory is supposed to be rounded up to 32-bit. The depth of potential memory is 1024 since the maximum neurons allocated to one PE is maximally 300 for the network used.

Weight memory: Each cluster contains four blocks of weight memory, assigned for each PE respectively. The width of the weight memory is 32-bit. The consideration of width is the same as that described for potential memory. Even if 24-bit is enough for fixed-point weight, we have to choose 32-bit due to the constraint of DDR interface width. The depth of potential memory is 1024 since the maximum neurons allocated to one PE is maximally 300 for the network used.

Spike memory: Each cluster contains a block of spike memory. The width of spike memory is 32-bit, the storage pattern is [23:16] stores the timestep, and [15:0] stores the neuron ID. The depth of spike memory is 1024, which depends on the maximum spike number in a layer for one image. As to statistics, the spike number in the first layer and the second layer is ranging from 600 to 800, whereas that in the output layer is around 10-20.

2.5.2 On-board memory

In this design, we need DDR to store the dataset and pre-trained weight data. We used 2 segments in DDR: (i) starting from 0x0 with 2.39 MB size for pre-trained weight data; (ii) starting from 0x989680 with 28.4 MB size for MNIST dataset.

Table 2.3: Instruction set architecture

Instruction	Function
mov <tar>, flag, <imm>/<src>	Move the content of <imm>/<src> to <tar>
add <tar>, <src1>, flag, <imm>/<src2>	Add content in <src1> and <imm>/<src2>, and store results in <tar>
mul <tar>, <src1>, flag, <imm>/<src2>	Multiply content in <src1> and <imm>/<src2>, and store results in <tar>
sft <tar>, <src>, <imm>	Shift the content in <src> by <imm> bits to the left, and store results in <tar>.
mod <tar>, <src>, <imm>	Calculate the content in <src> modulo <imm>, and store results in <tar>.
beq <src1>, <src2>, <imm>	Comparing contents in <src1> and <src2>, if equal, then jump to pc location <imm>.
bneq <src1>, <src2>, <imm>	Comparing contents in <src1> and <src2>, if unequal, then jump to pc location <imm>.
jmp <imm>	Jump to pc location <imm>.
rdi <nid>, <tsp>, <addr>	Read the content in input memory with location <addr>, and store the neuron id and timestep into <nid> and <tsp> respectively.
rds <nid>, <tsp>, <addr>	Read the content in pre-spike memory with location <addr>, and store the neuron id and timestep into <nid> and <tsp> respectively.
ldw <wgt_addr>, <ddr_addr>, <length>, pe_mask	Move the data from start address <ddr_addr> in DDR to start address <wgt_addr> in weight memory by length <length>.
rwp <wgt_addr>, <ptn_addr>, pe_mask	Move the content with address <wgt_addr> in weight memory and <ptn_addr> in potential memory to corresponding registers in PE.
acc <idx>, pe_mask	Add the current potential and weight, and store result into potential memory corresponding to location <idx>.
ckf <idx>, cluster_mask, pe_mask	Compare the latest potential with threshold, if exceeding, then store nid and timestep into post-spike memory corresponding to location <idx>.
syn	Wait for completing of subsequent clusters in the current timestep.

Chapter 3

Hardware Implementation

3.1 General Development Flow

3.1.1 HLS IP development

3.1.1.1 Setup

Open the Vivado HLS Graphical User Interface (GUI) and create a new project.

3.1.1.2 Validate the C source code

Realize the aforementioned hardware functions in C programming. The C file architecture is shown in Figure 3.1. Then run the C simulation to confirm that the functionality of the C code is correct. The C testbench will be reused in RTL validation. It is noted that the whole functionality of the IP relies on the input from the DDR channel, so it is impossible to validate the whole functionality by C simulation and RTL simulation. For example, we cannot simulate the instruction **ldw** by C simulation. To simulate other instructions such as **rwp**, and **ckf**, we need to initialize the weight memory by directly writing the initial content in the C code.

3.1.1.3 High-level synthesis

Run C synthesis and the estimated performance and utilization and interface report will be generated.

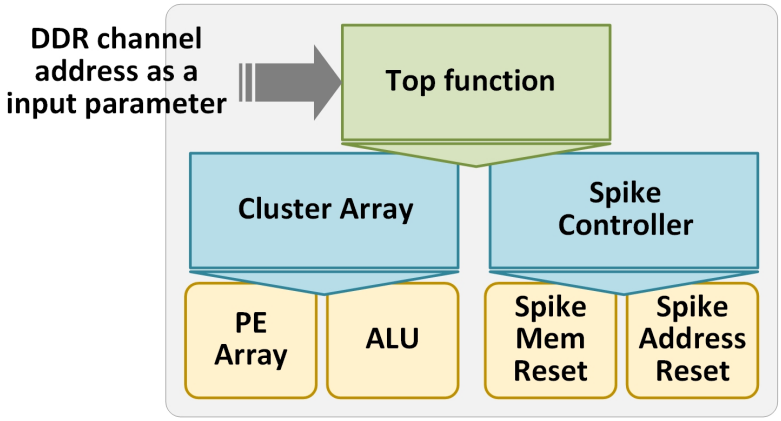

Figure 3.1: C file structure.

3.1.1.4 RTL verification

High-Level Synthesis can re-use the C test bench to verify the RTL using simulation. Run C/RTL Cosimulation and it will check whether the synthesized results have the same function as what is described in C codes.

3.1.1.5 IP creation

The final step in the High-Level Synthesis flow is to package the design as an IP block for use with other tools in the Vivado Design Suite.

3.1.2 Embedded Processor Development

3.1.2.1 Setup

Firstly, start the Vivado IDE and create a Project. Then create an IP Integrator Design. Add in the IPs that we are going to use: Zynq_Ultrascale+_MPSoC, and the HLS-IP that we have already exported. Run automatic connections, then AXI_SmartConnect, AXI_Interconnect, and Processor_System_Reset will be generated automatically. An illustration is shown in Figure 3.2, where the Global_Controller is the HLS-IP that we specified. Modify the address for each IP in the Address Editor, especially ensuring that the address of the four DDR channels is not overlapped with each other.

3.1.2.2 Implement Design and Generate Bitstream

Run synthesis, implementation, and generate the bitstream. Then export hardware to SDK.

3.1.2.3 Create a Software Application

Launch Vivado SDK environment. Create an application project in SDK. Write the C code to initialize the platform and correspond with the PL part using the driving function already packaged in the IP solution. To run the C code, we choose System Debugger. To restore the DDR memory content, select Xilinx Tools > Dump/Restore Memory.

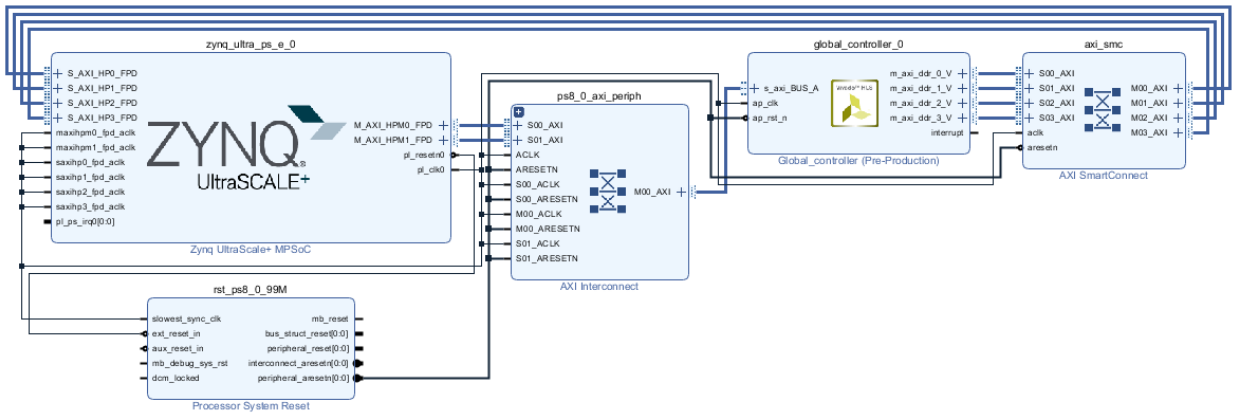

Figure 3.2: The block design diagram in Vivado for Type 1.

3.2 Resource Utilization

The accelerator based on the proposed instruction set is implemented on Zynq Ultrascale+ ZCU106 Evaluation Platform. Here we show two types of implementation. Type 1 is illustrated in Figure 3.2, where one HLS-IP is instantiated, while there are 12 clusters inside the IP. Type 2 is illustrated in Figure 3.3, where four HLS-IPs are instantiated, while there are 3 clusters inside each IP. Introducing these two types of implementation, we want to explore the parallelism between IPs. The hardware resource consumption of one cluster (the same for both Type 1 and Type 2) generated by HLS is shown in Table 3.1. The hardware resource

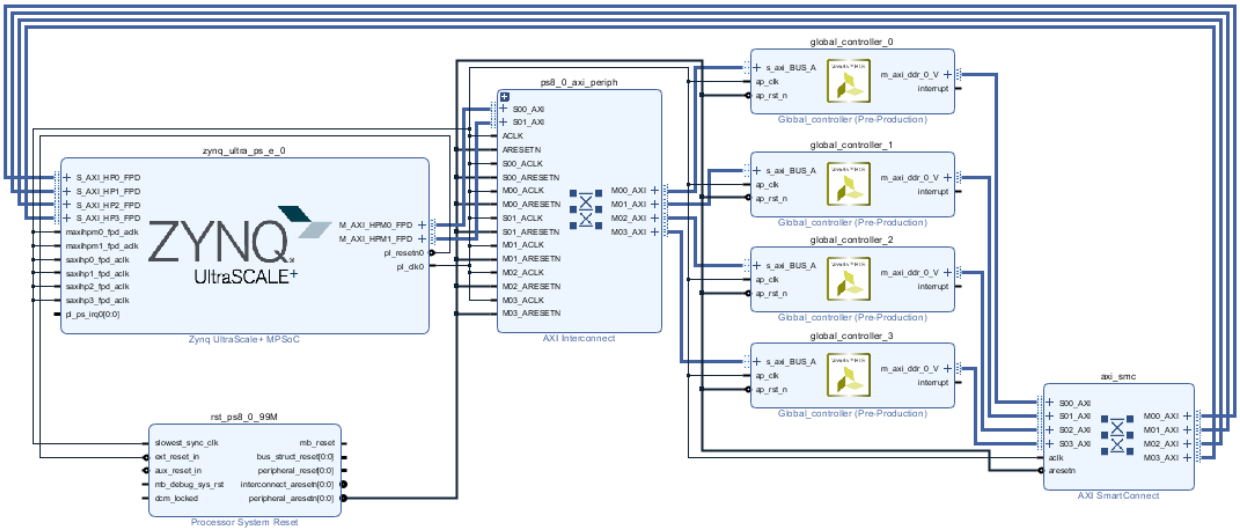

Figure 3.3: The block design diagram in Vivado for Type 2.

consumption of twelve clusters generated by HLS (Type 1) is shown in Table 3.2. As to the implementation results of the Vivado project, the resource utilization is shown in Table 3.3. It shows that our neuromorphic implementation consumes a small number of hardware resources and memory storage. This shows that our hardware architecture is suitable to be adopted in edge IoT applications, which have tight constraints on the memory resource consumption.

Table 3.1: Hardware Resource Consumption in HLS for one Cluster

Type	First layer	Non-first layer
BRAM(18kBits)	14	21
DSP	10	10
LUT	5175	5207
FF	7003	7166

3.3 Energy

The total on-board power is 4.357 W when the junction temperature is 29.2°C as reported in the Vivado Project Manager.

Table 3.2: Hardware Resource Consumption in HLS for One IP (Type 1) with 12 Clusters

Type	Total used	Available	Utilization
BRAM(18kBits)	390	1824	21%
DSP	136	2520	5%
LUT	105962	274080	38%
FF	71355	548160	13%

Table 3.3: Resource Utilization for the Whole Vivado Project

Type	Type 1	Type 2
BRAM(18kBits)	59%	60%
DSP	3%	3%
LUT	41%	38%
LUTRAM	3%	3%
FF	14%	13%
BUFG	1%	1%

Table 3.4: Performance

Type	Latency	Throughput
One IP (Type 1 or Type 2)	116.95ms	8.55fps
Four IPs (Type 2 only)	116.95ms	27.04fps

3.4 Performance

The performance of this design is shown in Table 3.4. The average time for processing one image in an IP is 116.95 ms. If we only use one IP, the throughput is 8.55 fps (frame per second). If the total utilized IP number is 4, the throughput is 27.04 fps (frame per second), speeded up by around 3.5 times. The total time for processing the whole dataset containing 10000 images is 418620 ms.

3.5 Scalability

From the performance results, we can see that the total throughput can be increased with more cores. But the throughput cannot be increased unlimitedly.

One main limitation is the on-board resources: (i) As the memory resource is

limited, based on the memory size indicated in section 2.5, the UltraScale Platform can maximally support around 18 clusters; (ii) There are 4 HP channels in total, which means even if more than 4 IPs are available, the throughput cannot be scaled up proportionally due to DDR channel multiplexing.

Another main limitation is the flexibility of the instruction set. If we introduce Convolutional Spiking Neural Networks to the current architecture, the following things need to be considered: (i) The total weight data size is smaller than FCN which is affordable for on-chip memory, thereby no instruction **ldw** will not be involved. But the challenge is, as the on-chip memory is evenly distributed into each cluster, we need to allocate each layer's neurons onto multiple clusters. Only in this way, all the weights and potentials can be accommodated in on-chip memory. (ii) Due to (i), indexing the weight and potential data in on-chip memory would be more complicated, since there are concepts like kernel, filter, sliding, and padding in CNN. Thus we may introduce more range in instruction **rwp** and **ckf**.

Chapter 4

Future Work

4.1 Convolutional Spiking Neural Networks

The main challenge for Convolutional Spiking Neural Networks is the mapping strategy. The structure of CNN is much more complicated than FCN, where CNN has concepts such as kernel, filter, sliding, padding, etc. This will also make the neuron index more difficult, thus requiring modifications from the FCN accelerator version, which has been discussed in section 3.5 already.

4.2 On-chip learning

Efficient SNN training has always been a major challenge. Various approaches range from unsupervised learning and STDP rules to supervised variations of error backpropagation and direct ANN to SNN conversion. Backpropagation and STDP-based learning has been realized on-chip be previous works. While the conversion-based method does not apply to on-chip learning, the SNN generated in this way can be used for on-chip evaluation. The benchmark of these three main methods is still to be investigated further.

Chapter 5

Conclusion

The objective of this work is to exploit the acceleration scheme for FPGA-based SNN processing and to design the corresponding hardware architecture to evaluate the proposed schemes.

This report first analyzed the sparsity of the used spiking-based fully connected neural networks, which shows that maximally 98.12% acceleration can be achieved when making full use of the sparsity by introducing the event-driven computing architecture. Then we analyzed the required memory resources that are needed to process SNN based on an FPGA. Thus, we come up with a multi-cluster architecture. What's more, in a common FPGA platform, the communications between the host processor (CPU) and co-processor (FPGA) take a main part of the latency. Hereby, it is motivated to design an instruction set for the clusters within FPGA that can execute instructions independently.

Thereafter, we propose the instruction set and the corresponding hardware architecture. The hardware architecture is realized in C language, which can be translated into hardware implementation by Vivado High-Level Synthesis (HLS). In the final design, there are in total 12 clusters, where every 3 clusters are grouped together. Therefore, 4 input images can be processed simultaneously each by a group of clusters. As to evaluation results, the throughput of the proposed architecture is 27.04 fps, and the total on-board power is 4.357 W. The resource utilization rate is 59%, 3%, 41%, and 14% for BRAM, DSP, LUT, and FF respectively. The proposed approach allows the SNN architecture to be defined completely in the instruction set, and our neuromorphic implementation consumes a small number of hardware resources and memory storage. This shows that our hardware architecture is suitable

to be adopted in edge IoT applications, which have tight constraints on the memory resource consumption.

In the future, we will expand the remaining architecture to spiking-based convolutional neural networks, which requires higher flexibility and scalability on the hardware architecture. The on-chip learning mechanism is also to be explored. Potential learning algorithms are backpropagation and STDP-based learning.

Bibliography

[1] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[2] R. Batllori, C. B. Laramee, W. Land, and J. D. Schaffer, "Evolving spiking neural networks for robot control", *Procedia Computer Science*, vol. 6, pp. 329–334, 2011.

[3] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations", *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[4] M. Beyeler, N. D. Dutt, and J. L. Krichmar, "Categorization and decision-making in a neurobiologically plausible spiking network using a stdp-like learning rule", *Neural Networks*, vol. 48, pp. 109–124, 2013, ISSN: 0893-6080.

[5] S. M. Bohte, J. N. Kok, and J. A. La Poutré, "Spikeprop: Backpropagation for networks of spiking neurons." In *ESANN*, Bruges, vol. 48, 2000, pp. 419–424.

[6] H. Burgsteiner, "Training networks of biological realistic spiking neurons for real-time robot control", in *Proceedings of the 9th international conference on engineering applications of neural networks, Lille, France*, 2005, pp. 129–136.

[7] S. Chaturvedi, A. A. Khurshid, and S. S. Dorle, "Reconfiguration of spiking neural network for optimization with applications to image processing", in *2013 6th International Conference on Emerging Trends in Engineering and Technology*, IEEE, 2013, pp. 191–192.

[8] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning", *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[9] J. Dethier, P. Nuyujukian, S. I. Ryu, K. V. Shenoy, and K. Boahen, "Design and validation of a real-time spiking-neural-network decoder for brain–machine interfaces", *Journal of neural engineering*, vol. 10, no. 3, p. 036 008, 2013.

[10] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project", *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, 2014.

[11] J. Han, Z. Li, W. Zheng, and Y. Zhang, "Hardware implementation of spiking neural networks on fpga", *Tsinghua Science and Technology*, vol. 25, no. 4, pp. 479–486, 2020.

[12] M. Heidarpur, A. Ahmadi, M. Ahmadi, and M. Rahimi Azghadi, "Cordic-snn: On-fpga stdp learning with izhikevich neurons", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 7, pp. 2651–2661, 2019.

[13] Z. Huang, H. Luo, and D. Guo, "Application of locally connected spiking neural network in image processing", in *2019 IEEE 13th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, IEEE, 2019, pp. 108–111.

[14] E. Hunsberger and C. Eliasmith, "Training spiking deep networks for neuromorphic hardware", *arXiv preprint arXiv:1611.05141*, 2016.

[15] S. Kulkarni, S. P. Simon, and K. Sundareswaran, "A spiking neural network (snn) forecast engine for short-term electrical load forecasting", *Applied Soft Computing*, vol. 13, no. 8, pp. 3628–3635, 2013, ISSN: 1568-4946.

[16] L. Maguire, T. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin, "Challenges for large-scale implementations of spiking neural networks on fpgas", *Neurocomputing*, vol. 71, no. 1, pp. 13–29, 2007, Dedicated Hardware Architectures for Intelligent Systems Advances on Neural Networks for Speech and Audio Processing, ISSN: 0925-2312.

[17] B. Meftah, O. Lézoray, S. Chaturvedi, A. A. Khurshid, and A. Benyettou, "Image processing with spiking neuron networks", in *Artificial Intelligence, Evolutionary Computing and Metaheuristics*, Springer, 2013, pp. 525–544.

[18] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface", *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[19] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, "A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps)", *IEEE Transactions on Biomedical Circuits and Systems*, vol. 12, no. 1, pp. 106–122, 2018.

[20] E. O. Neftci, C. Augustine, S. Paul, and G. Detorakis, "Event-driven random back-propagation: Enabling neuromorphic deep learning machines", *Frontiers in neuroscience*, vol. 11, p. 324, 2017.

[21] D. Neil and S.-C. Liu, "Minitaur, an event-driven fpga-based spiking network accelerator", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, 2014.

[22] T. Nguyen, B. Veeravalli, and X. Fong, "Instruction set for a neuromorphic co-processor with on-chip learning", in *2021 International Conference on Neuromorphic Systems (ICONS)*, 2021.

[23] F. Ponulak and A. Kasiński, "Supervised learning in spiking neural networks with resume: Sequence learning, classification, and spike shifting", *Neural computation*, vol. 22, no. 2, pp. 467–510, 2010.

[24] A. Roy, S. Venkataramani, N. Gala, S. Sen, K. Veezhinathan, and A. Raghunathan, "A programmable event-driven architecture for evaluating spiking neural networks", in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2017, pp. 1–6.

[25] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification", *Frontiers in neuroscience*, vol. 11, p. 682, 2017.

[26] V. Sakellariou and V. Paliouras, "An fpga accelerator for spiking neural network simulation and training", in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[27] D. J. Saunders, H. T. Siegelmann, R. Kozma, *et al.*, "Stdp learning of image patches with convolutional spiking neural networks", in *2018 international joint conference on neural networks (IJCNN)*, IEEE, 2018, pp. 1–7.

[28] T. Schoenauer, S. Atasoy, N. Mehrtash, and H. Klar, "Neuropipe-chip: A digital neuro-processor for spiking neural networks", *IEEE Transactions on Neural Networks*, vol. 13, no. 1, pp. 205–213, 2002.

[29] S. Sen, S. Venkataramani, and A. Raghunathan, "Approximate computing for spiking neural networks", in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 193–198.

[30] "Spiking relu conversion", https://github.com/dannyneil/spiking_relu_conversion.

[31] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey", *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[32] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks", *Neural Networks*, vol. 111, pp. 47–63, 2019, ISSN: 0893-6080.

[33] D. Thomas and W. Luk, "Fpga accelerated simulation of biologically plausible spiking neural networks", in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 45–52.

[34] Q. Wang, Y. Li, B. Shao, S. Dey, and P. Li, "Energy efficient parallel neuromorphic architectures with approximate arithmetic on fpga", *Neurocomputing*, vol. 221, pp. 146–158, 2017, ISSN: 0925-2312.

[35] A. Yousefzadeh, E. Stromatias, M. Soto, T. Serrano-Gotarredona, and B. Linares-Barranco, "On practical issues for stochastic stdp hardware with 1-bit synaptic weights", *Frontiers in Neuroscience*, vol. 12, p. 665, 2018, ISSN: 1662-453X.