Doctoral Dissertation

Multi-Grained Reconfigurable Architecture Powered by Elastic Neural Network for Approximate Computing

Yirong Kan

February 4, 2022

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Yirong Kan

Thesis Committee:

Professor Yasuhiko Nakashima	(Supervisor)
Professor Yuichi Hayashi	(Co-supervisor)
Associate Professor Renyuan Zhang	(Co-supervisor)

Multi-Grained Reconfigurable Architecture Powered by Elastic Neural Network for Approximate Computing*

Yirong Kan

Abstract

Beyond the boom of artificial intelligence, the next generation of computing architectures with high speed and low cost are always demanded. In this thesis, we proposed a multi-grained reconfigurable architecture for accelerating arbitrary functions in fully parallel with high speed and low cost. The proposed architecture is reconfigurable in fine-grained (arbitrary functions), mid-grained (flexible function feature, accuracy, and number of operands), and coarse-grained (organization of kernels). By implementing a large scale of novel bisection neural network (BNN) on hardware, the reconfiguration is conducted by partitioning entire BNN into any specific pieces without redundancy. Each piece of BNN retrieves the arbitrary function approximately. By reconfiguring the BNN topology in software, we can easily adjust dimensions of the computing kernel without rewiring, and achieve a wide range of trade-offs between accuracy and efficiency in hardware. In this manner, the multi-grained reconfigurable architecture is achieved. For proof-of-concept, a demo accelerator is built on FPGA. The processing element is designed in 16-bit fixed point scheme including two synapses and one neuron. In order to better support this architecture, we have also proposed a series of system-level optimization techniques, including design flow, on-chip interconnection, and configuration strategies, etc. Since the architecture is flexible in all grained levels, various configurations for each validation are demonstrated with rich options of performance-cost matrix. From the FPGA implementation results,

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, February 4, 2022.

compared with CPU baseline, proposed architecture achieves speedups of 5.1x to 30.3x. Compared with other traditional function approximation methods, our method provides fewer parameter storage requirements. The comparison against related works proves that our accelerator has reduced the area-latency product by at least 9.5% with a loss of accuracy by at most 8.9%.

Keywords:

Parallel computing, reconfigurable architecture, neural network, approximate computing, FPGA implementation

Contents

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

1.1 Background 1
1.2 Motivations and Contributions 2
1.3 Organization of Thesis 6

2 Related Works **7**

2.1 Neural Network-based Approximate Computing 7
2.2 Hardware Accelerator for Neural Networks 9
2.3 Reconfigurable Architectures 12

3 Neural Network Prototype of Proposed Architecture **16**

3.1 Spatial-Expanded Implementation 16
3.2 Bisection Neural Network 17
3.3 Towards Multi-Grained Reconfigurable Architecture 20
3.4 Design Flow 23
3.5 Challenges 27
3.5.1 On-Chip Interconnections 27
3.5.2 PE Utilization 29

4 Hardware Architecture Design and Optimization **31**

4.1 Overview of Proposed Accelerator 31
4.2 Design of PE Architecture 33
4.3 On-Chip Interconnection for Efficient Buffer Utilization 35
4.4 Controller Design 38

4.5 Computation Datapath 39
4.6 Configuration Strategy 42

5 Evaluation **50**
5.1 Experimental Setup 50
5.2 Results by Software 51
5.3 Results of Fixed-Point Hardware Simulation 53
5.4 Implementation Results on FPGA 55
5.5 Comparison with Other Works 57
5.6 A Case Study on Fault-Tolerant Application 61

6 Conclusion **67**
6.1 Summary 67
6.2 Future Works 68

References **71**

Publication List **85**

List of Figures

1.1 Computing architectures: (a) General computing system, (b) Heterogeneous computing system. 1
1.2 Motivation of this thesis. 6

2.1 Modeling biological neurons into artificial neurons. 7
2.2 Multilayer perceptron model. 8
2.3 Heterogeneous computing system proposed by Esmaeilzadeh et al. [42]. 10
2.4 Various neural network accelerator architectures. 11
2.5 General structure of FPGAs. 13
2.6 Typical structure of CGRAs [73]. 14

3.1 A NN performed by TDM and SEP architecture. 17
3.2 Partitioning an original NN in FC-NN and BNN style hardware. . . 19
3.3 Workflow of proposed architecture. 21
3.4 BNN-based multi-grained reconfiguration architecture. 22
3.5 Software-hardware division of the design flow. 24
3.6 Convert FC-NN to BNN using mask matrix. 25
3.7 A top view of memory architecture for spatial array accelerator. . 27
3.8 Four common NoC interconnection designs. 28
3.9 An example of improving the PE utilization. 30

4.1 Overall architecture of the proposed MuGRA system. 32
4.2 Architecture of a PE. 33
4.3 Architecture of a neuron unit. 35
4.4 Logic structure of interconnection between buffers and kernels. . . 36
4.5 Interconnection of PEs and local buffer. 37

4.6 An example of large array configuration. 37
4.7 FSM controller of the system. 39
4.8 Datapath of a computing kernel. 40
4.9 Allocating two computing kernels on a 5×5 PE array. 41
4.10 An example of mapping a computing kernel on PE array. 43
4.11 A flow diagram of NRP. 44
4.12 A flow diagram of ORP. 46
4.13 An instance of proposed configuration strategies. 47

5.1 Average accuracy with different precision. 53
5.2 Average accuracy with different fractional width by 16bit precision. 54
5.3 Resource utilization with different scale of PE arrays. 56
5.4 Speedup of TDMA and SEPA compared with CPU baseline. . . . 57
5.5 Approximate results of image segmentation. 61

List of Tables

2.1 Comparison of various reconfigurable architectures 12

5.1 Calculation performance test for one-variable functions 51
5.2 Calculation performance test for two-variable functions 52
5.3 Implementation results of various topologies on FPGA 55
5.4 Comparison of memory size for one-variable functions 58
5.5 Comparison of memory size for two-variable functions 58
5.6 Comparison of implementation results for one-variable functions with other FPGA-based works 64
5.7 Comparison of implementation results for two-variable functions with other FPGA-based works 65
5.8 Comparison of power consumption with FPGA-based works . . . 66

1 Introduction

1.1 Background

With the rapid development of big data and artificial intelligence (AI), the next generation of computing architectures with high speed and low cost is always demanded [1–5]. Emerging application scenarios such as virtual reality, robot control and the Internet of Things require massive computing capabilities [6–8]. However, the performance growth rate of general-purpose processors is too slow to follow the rapid development of applications [9]. In particular, with the bottleneck of data transfer between processing and memory units, the traditional Von Neumann architecture has been unable to meet the growing demand on computations [10, 11]. As the high quality of service (QoS) is speed-greedy at the end of application, one of the keys for speeding up is the parallelism [12]. In order to implement a huge amount of computations at the application end, hardware with massive computational cores have been widely employed as accelerators with general-purpose processors [13, 14], shown as Fig. 1.1.

During the past decades, various prototypes of Non-Von Neumann architectures have been developed for processing the big and complex datum with high paral-

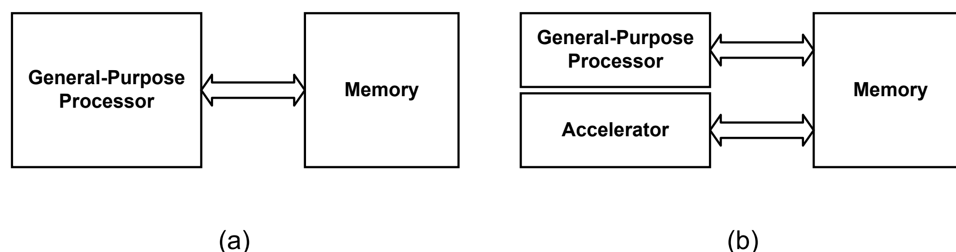


Figure 1.1: Computing architectures: (a) General computing system, (b) Heterogeneous computing system.

lelism such as general purpose graphic processing units (GPGPU) [15] and tensor processing units (TPUs) [16]. As a typical general-purpose multi-core processor, GPGPU is widely used in the hardware acceleration for machine learning applications, which are usually resource-hungry. On the other hand, domain-specific architectures (DSAs) [17] are developed to accelerate embedded applications with low power and cost, but flexible acceleration features for unpredictable tasks cannot be offered. Since the DSAs are application specific, their high performance and low cost are easily eaten up by commercial cost [18]. Fortunately, plenty of reconfigurable computing architectures have been developed to accelerate unpredictable applications [19]. Meanwhile, it has seen that reasonably approximate computing [20] processors appear the potential to achieve acceptable QoS with greatly reduced complexity of circuits and systems. Obviously, an ideal architecture is expected to win all the aspects of parallelism, flexibility, and cost.

As a powerful approximate computation model, neural networks (NNs) can effectively replace the complex modules in general programs with slight accuracy loss. Meanwhile, the NN algorithm is composed of a large number of regularized multiply accumulate (MAC) modules, which makes it easy to design a dedicated accelerator. Through configuration, the resources of the same accelerator can be reused for different applications. Therefore, by designing a NN accelerator in a general computing system, different tasks can be mapped to the accelerator for execution without adding excessive redundant computation logic. The potential flexibility and speedup of NN accelerators can greatly improve the performance and energy efficiency of general computing systems. Since the architecture of the NN accelerator will directly affect the performance of the overall system, the study in this thesis will focus on the innovative design of the NN accelerator for approximate computing, especially the design of the multi-grained reconfigurable architecture.

1.2 Motivations and Contributions

In general, the flexibility of a parallel computing architecture lies on the reconfigurability of multiple levels (seen as grains), where the fine, middle, and coarse grain indicate the function behavior, number of operands, and organization

of calculation units, respectively. The coarse-grained reconfigurable architectures (CGRAs) are widely seen as TPUs or dataflow processing units (DPUs) [21] for computer vision tasks. Without efficient function-flexibility, conventional CGRAs are not always prime especially in carrying out massive non-linear complex functions [22]. To enable (almost) arbitrary functions in-built, embedding powerful arithmetic and logic units (ALUs) onto systems is unacceptable due to the cost explosion problem. Several functional grained reconfigurable computing units have been developed by emulating arbitrary functions through rich-behavior-poor-bit ALUs [23], piece-wise linear approximation [24], and polynomial expansion [25]. However, the hardware implementations of those flexible function retrievals should be designed by specifying the performance-cost feature which is hardly re-configured post-fabric. In addition, the conventional function retrievals above hardly achieve ultra compact chip estates for massive implementation. As a result, simply migrating the function retrieve circuitry into ordinary CGRAs can not offer the real multi-grained re-configurability.

Beyond those implementation styles, the implementations of regression algorithms such as neural networks (NNs) have been reported on the aims at some specific applications [26, 27]. However, the conventional fully connection (FC) fashion of NNs leads to the hardware explosion and the remarkable redundancy during the reconfiguration. Escaping from the FC-NNs, a feasibility study of bisection topology of NN called 'DiaNet' has been reported for efficiently configuring NNs with rich trade-off between the accuracy and hardware cost [28–30]. In this sense, a single DiaNet on hardware behaves a specific computing kernel for approximate computing [31]. Unfortunately, the previous efforts on bisection neural network (BNN) are implemented by analog calculation, with problems such as variation, memory and design experiences. In particular, previous work has only explored the feasibility of a single DiaNet, while the reconfiguration and routing mechanism of multi-core parallel computing has not been developed. In this sense, not only the single-core level but also the multi-core architecture should be reconsidered for the greatly parallel and efficiently reconfigurable implementations. The motivation of this thesis is shown as Fig. 1.2.

The main contributions of this thesis are as follows:

1. This thesis proposes a multi-grained reconfigurable architecture powered by

a novel BNN topology [32]. From the perspective of neural networks, conventional fully connected neural networks cannot support the design of space expansion architecture due to the explosion of interconnection. However, the flexibility and scalability of the BNN topology allows us to build a large-scale on-chip array. This fully parallel design enables the system to achieve huge computing throughput. Meanwhile, the characteristics of BNN make it unnecessary to waste synaptic connections when allocating computing kernels on the array, thereby reducing parameter storage overhead. From the perspective of reconfigurable computing, the traditional FPGA architecture is complex to configure, while the CGRA architecture cannot achieve fine-grained functional reconfiguration. The proposed architecture is reconfigurable in fine-grained (arbitrary functions), mid-grained (flexible function feature, accuracy, and number of operands), and coarse-grained (organization of kernels). This architecture design idea brings a new perspective to the field of reconfigurable computing. From the perspective of energy-efficient computing, approximate computing based on neural networks have brought huge energy gains to general-purpose processors. By approximating a large number of complex computation modules with neural networks, application acceleration can be achieved at low cost.

2. Based on the proposed architecture, we implemented a CPU+Accelerator heterogeneous computing system on an FPGA-based SoC. At the system level, based on the pre-trained neural network model, we can efficiently run the original computing module on the proposed architecture. At the circuit level, we propose to use on-chip data and configuration buffers to smoothly process calculation and configuration information. The double-buffering design ensures that off-chip-on-chip data exchange and the computing unit read/store data at the same time to hide memory access delays. For PE design, we propose an activation function implementation method that does not use a multiplier to perform Leaky-ReLU calculations, which avoids neuron death and reduces circuit area overhead. The efficient control unit design makes the entire system run in order. Meanwhile, we propose a design flow to efficiently carry out system development and application on both software and hardware.

3. We discussed the challenges of designing interconnection networks between PE and on-chip memory in the proposed architecture. We analyzed that the

bottleneck of the proposed architecture in parallel computing is the inability to provide data for a large number of PEs at the same time. In order to provide data in parallel, the traditional approach is to use a crossbar switch to connect the PE array and the on-chip memory. For large-scale PE arrays, non-scalable crossbar switches will increase the complexity of the system. In order to solve this problem, we introduced four common NoC interconnection schemes and discussed them. Common NoC interconnects cannot be directly applied to our designs, resulting in a decrease in on-chip memory utilization or configuration flexibility. In view of the characteristics of the architecture, we designed a specific interconnection scheme to support arbitrary kernel allocation and full parallel computing, with improving the utilization of on-chip memory.

4. We propose three configuration strategies to place the computing kernels on the PE array. Efficient configuration strategies can effectively improve the utilization of PE, thereby improving the energy efficiency of the system. The naive random placement strategy simply places the kernels randomly, which has lower PE utilization in most cases. The greedy-based placement strategy assumes that placing a larger kernel can improve PE utilization. Scan the entire array and place the largest kernel first in the feasible space, with higher PE utilization. The optimized random placement strategy is based on the simulated annealing algorithm, which mutates the initial random placement solution to obtain a new solution. If the new solution is better than the original solution, the new solution is accepted, otherwise the new solution is accepted with probability. The optimized random placement strategy has the highest PE utilization rate in the instance.

5. To prove the universality of the proposed method, we have investigated multiple kinds of arithmetic functions, including one- and two-variable functions, and demonstrated their theoretical and hardware calculation results by software simulation and FPGA test, respectively. The test results of several functions show that for one-variable functions, the topology with the minimum hardware resource achieve an accuracy of over 99.75%; for two-variable functions, the topology with the minimum hardware resource achieve an accuracy of over 91.1%. By testing the average accuracy of different fixed-point implementation, we found that 16-bit is the ideal precision to design PEs that approximates the software results.

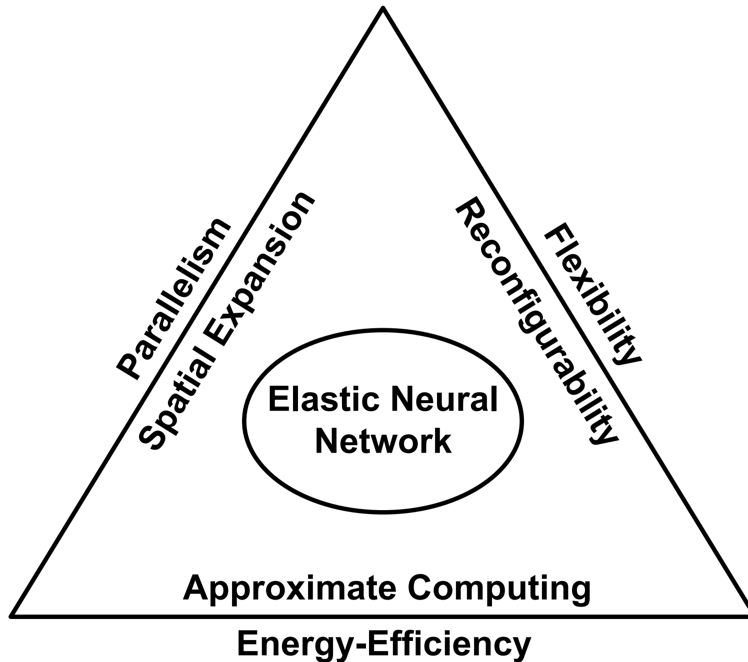


Figure 1.2: Motivation of this thesis.

Compared with other traditional function approximation methods, our method provides fewer parameter storage requirements. Compared with various one- and two-variable function generators based on FPGA implementation, the calculation latency of our accelerator has been effectively reduced with slight accuracy loss.

1.3 Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 introduced related work, including NN-based approximate computing, NN accelerator and reconfigurable architectures. Chapter 3 give the definition of BNN, concepts of BNN-based multi-grained reconfiguration architecture, a framework of design flow from software to hardware and discussion of design challenges. Chapter 4 proposed the hardware architecture designs, including overview of the MuGRA system, controller design, PE architecture, on-chip interconnection design, computation datapath and configuration strategies. Chapter 5 shows the experimental results of proposed architecture. We summarize the thesis in Chapter 6.

2 Related Works

2.1 Neural Network-based Approximate Computing

Many resource-consuming applications can tolerate approximate errors in their output results without causing significant loss in quality of results [33–35], such as image and video processing, voice recognition, web search, etc. The input of these applications itself is real-world data with noise and redundancy, such as image and audio data obtained from cameras and voice sensors. Their output is also used for human perception. The user may not be able to distinguish the subtle differences between the exact results and the approximate results, or the results may have multiple user acceptable answers [36]. For such applications, approximate computing can not only greatly reduce the complexity of circuits and systems, but also achieve rich trade-offs between cost and accuracy loss.

Artificial neural network (ANN) is a mathematical model that simulates the working principle of the nervous system in human brains. Through abstract modeling of biological neural networks, a network that realizes a certain function can be constructed artificially. For simulating nonlinear problems, ANN can

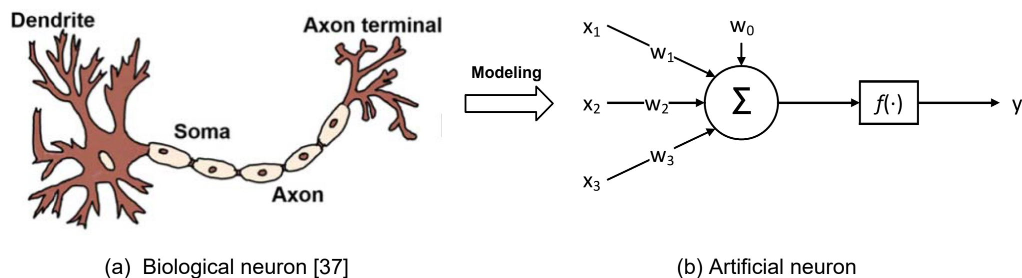


Figure 2.1: Modeling biological neurons into artificial neurons.

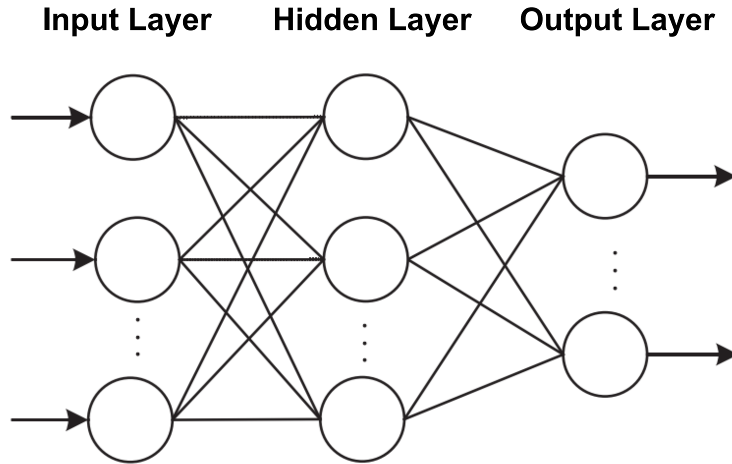


Figure 2.2: Multilayer perceptron model.

use input and output data to find the optimal parameters through self-learning capabilities without having to understand the exact model of the original problem.

A large number of basic information processing units are connected by a specific structure in the ANN to form a system with nonlinear and adaptive information processing capabilities. Neuron is the most basic information processing unit of ANN. Figure 2.1 shows the biological neuron [37] and the corresponding artificial neuron model. By modeling the biological neuron, the artificial neuron performs a weighted summation of multiple input data x_i according to the weight w_i, and obtains the output y through the activation function f. Although the function of each neuron is very simple, a large number of neurons can be connected to each other through a specific topological structure to achieve very complex arithmetic functions.

The multilayer perceptron [38] (MLP) is a typical multilayer fully connected neural network, as shown in Fig. 2.2. The basic unit of the MLP model is still an artificial neuron, but the difference is that each neuron is fully connected with the output of all neurons in the previous layer to form a multilayer structure. Data enters the network from the input layer, sequentially passes through the hidden layer neurons, and finally obtains the result from the output layer. Cybenko et al. [39] proved that a 3-layer MLP is a general approximator, which can approximate any continuous function on the closed interval. Therefore, ANN is widely used

to accelerate approximate computations in general programs.

Neural network is essentially an approximate computing model, which imitates the original calculation rules by learning the mapping between input and output, to achieves approximate results instead of exact results. Despite the loss of accuracy due to the approximation, neural networks may still provide acceptable results for a wide range of applications with inherent error tolerance [40]. By replacing some complex operations in general programs, neural networks can reduce the delay and power consumption of key calculation modules, to achieve acceleration of computing. For example, in image segmentation based on the k-means algorithm, we have to calculate the similarity between each pixel, usually the Euclidean distance. For an image of size $N \times M$, we have to calculate the Euclidean distance between $N \times M$ pixels and k cluster centers in each iteration, which is unacceptable on a general-purpose CPU. By offloading this type of computing task to a large scale parallel accelerator powerd by neural networks, we can achieve a calculation acceleration of more than 10 times. In addition, hardware neural networks are also easy to implement in parallel and pipeline, so throughput and efficiency could be improved.

In summary, the reasons why neural network can be used for approximate computing are: 1. neural networks have good nonlinear mapping capabilities to learn the behavior of the replaced module in the application with extremely high accuracy; 2. neural networks have a large number of parallel processing units, which can convert operations into basic multiply and add modules, with high reconfigurability; 3. by modifying the parameters or topology, neural networks can realize various applications with different accuracy without making a lot of changes to the hardware. These reasons make neural networks very suitable as the logical basis of reconfigurable computing architecture.

2.2 Hardware Accelerator for Neural Networks

In order to utilize the approximate computing performance of neural networks in general-purpose processors, Esmaeilzadeh et al. [41–43] first proposed a heterogeneous computing system architecture of "general-purpose processors + neural network accelerators". Neural network accelerators take on important computing

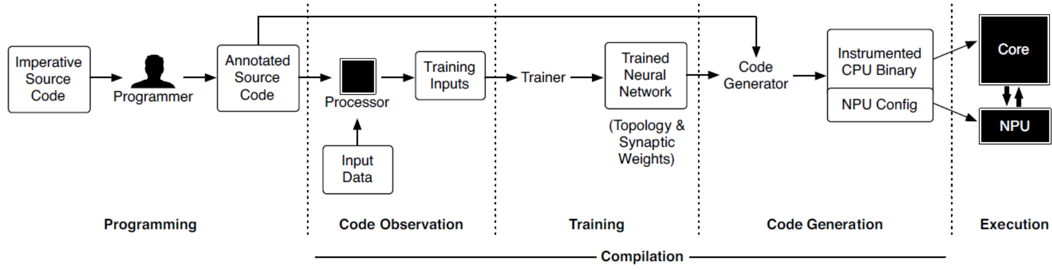
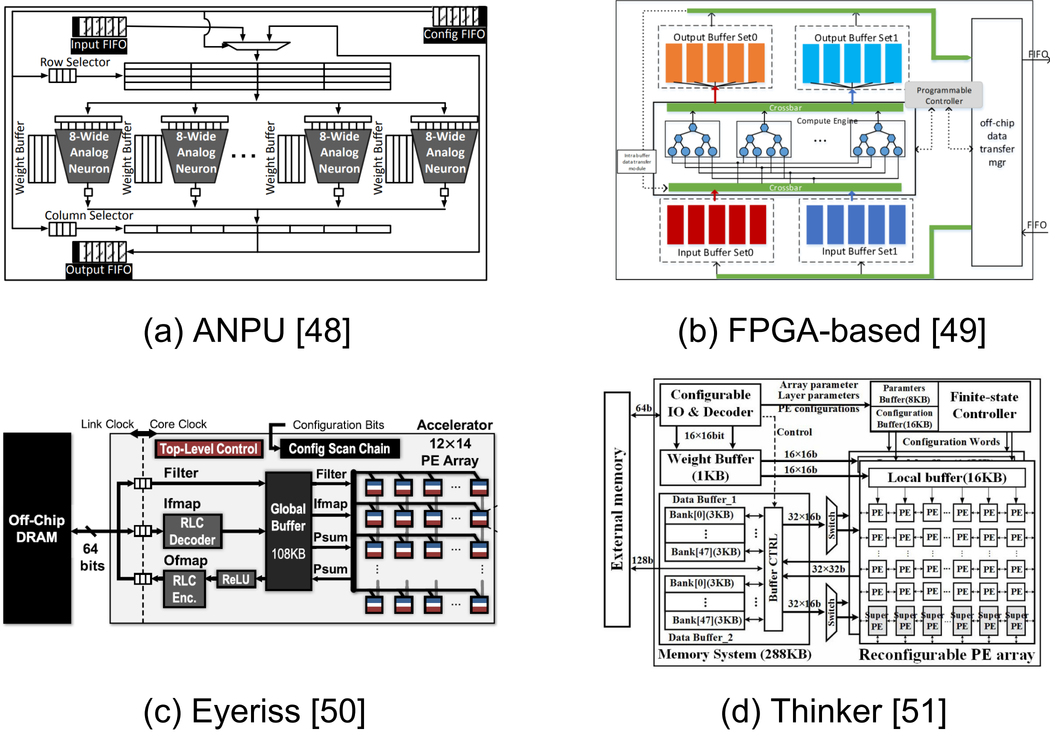


Figure 2.3: Heterogeneous computing system proposed by Esmaeilzadeh et al. [42].

tasks in general computing in the form of coprocessors to improve the energy efficiency of the entire system, as shown in Fig. 2.3. After that, neural network accelerators for general-purpose computing have developed tremendously. Moreau et al. [44] designed a neural network accelerator SNNAP based on FPGA to avoid changes to the instruction set and micro-architecture of general-purpose processors. Yazdanbakhsh et al. [45] integrated neural network accelerators and GPUs to form an NGPU architecture to implement approximate computations for CUDA applications. Eldridge et al. [26] used the general approximation performance of neural networks to design a function approximator for accelerating the calculation of transcendental functions. Wang et al. [46] integrated the neural network accelerator into the JPEG encoder to achieve high-performance and low-power JPEG encoding. Tu et al. [47] used the reconfigurability of neural networks to map neural networks on reconfigurable hardware to accelerate calculations in multimedia applications.

At the same time, the use of various hardware to realize the computational optimization of neural networks has also been extensively studied. St. Amant et al. [48] used analog circuits to realize neurons to build accelerator ANPU, which can further improve the performance and energy efficiency of general neural network approximation compared with traditional digital neurons. Zhang et al. [49] used the Roofline model to analyze the trade-off relationship between calculation and memory access, and found the design parameters with the lowest calculation delay on the FPGA platform. Chen et al. [50] designed the ASIC chip Eyeiss, which greatly improved the on-chip data reuse by using a specific calculation model. Yin et al. [51] designed a chip based on a reconfigurable architecture called



(a) ANPU [48]

(b) FPGA-based [49]

(c) Eyeriss [50]

(d) Thinker [51]

Figure 2.4: Various neural network accelerator architectures.

Thinker that supports hybrid neural networks computing. The above hardware systems are shown in Fig. 2.4. Although the principles of many optimization methods are the same on FPGA and ASIC, the implementation styles of the two are quite different, which directly affects their scope of application and optimization results. Since FPGA has the ability of reconfiguration, it is easy to construct optimization problems according to the execution goals and find the optimal design parameters. However, FPGAs have high power consumption and are not easy to optimize. FPGA-based designs usually only take performance-related indicators as optimization targets [49, 52–55]. ASIC-based designs have obvious advantages in terms of power consumption, but their reconstruction ability is weak, and they often only support a fixed calculation mode [56–58].

In addition, the use of the characteristics of algorithms for effective neural network acceleration has also been widely studied. The main idea is to use the fault-tolerant characteristics of neural networks to perform model compression, which

Table 2.1: Comparison of various reconfigurable architectures

Architecture	**PLA**	**FPGA**	**CGRA**	**EGRA**
Granularity	Fine	Fine	Coarse	Coarse
Component	Gate	LUT	ALU	ALU Clusters
Arithmatic	Boolean	Boolean	Numeric	Expression
Flexibility	High	High	Low	Low
Configuration	Hard	Hard	Easy	Easy

is divided into two types of techniques: sparsity [59–61] and quantization [62–64]. The core idea is that the neural network contains redundant information. By removing unnecessary calculations and reducing the data bit width, the amount of calculation or parameter storage is greatly reduced. Advances in algorithms have also prompted hardware researchers to design some neural network accelerators that support sparse computing [65–67] and low-bit-width computing [68–70].

2.3 Reconfigurable Architectures

The concept of reconfigurable computing was first proposed by Estrin et al. [71] in 1960. The reconfigurable hardware is controlled by the controller, and the structure is reconstructed according to the specific task, so as to realize the task-customized computation path. In 1999, DeHon et al. [72] further defined the reconfigurable processor: it can be customized post-silicon to realize the spatial mapping of tasks to the chip. According to this definition, a reconfigurable processor can have the programmability and flexibility of a general-purpose processor, as well as the high performance of spatial parallel computing on hardware.

Generally, the reconfiguration level of hardware is divided into two levels, fine-grained and coarse-grained. Table 2.1 compares various reconfigurable architectures. Programmable logic array (PLA) and FPGA are typical representatives of fine-grained reconfigurable architecture. The fine-grained architecture uses logic

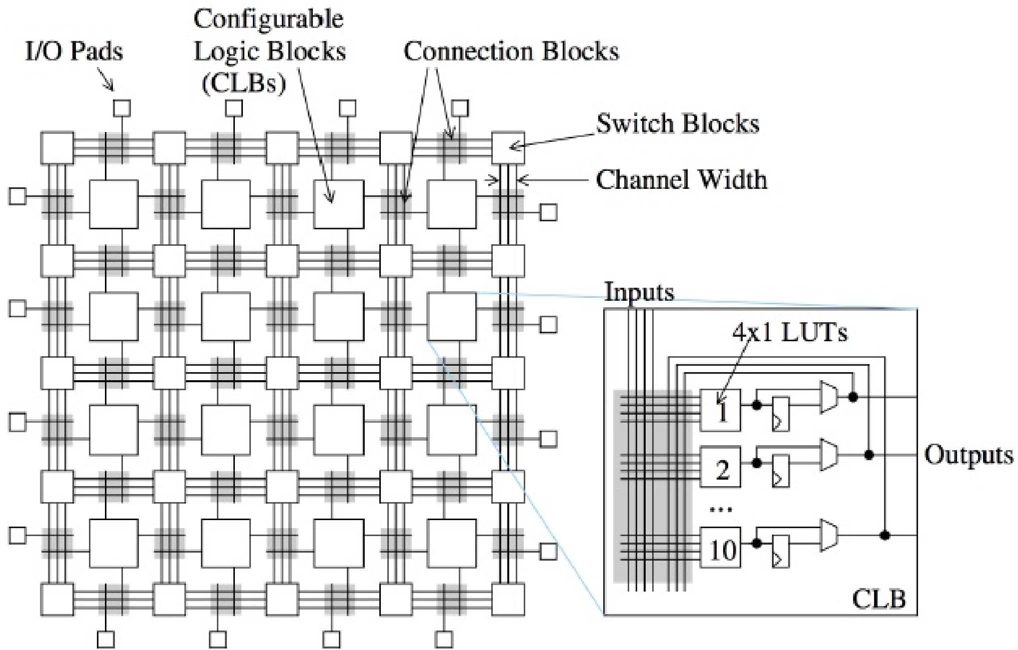


Figure 2.5: General structure of FPGAs.

gates or look-up tables (LUTs) as basic units and glues them together with a flexible interconnection network, shown as Fig. 2.5. The LUT unit is a piece of static random access memory (SRAM) with address input lines. Through different inputs to the address lines, the numbers corresponding to different addresses can be taken from the SRAM as output, thereby realizing the function of a LUT. Arbitrary logic functions can be realized by configuring the LUTs. Although the fine-grained architecture is highly flexible, its reconfiguration is very expensive, and the synthesis and layout of large-scale circuits usually take several hours.

In contrast, another representative of typical reconfigurable architecture is coarse-grained reconfigurable architectures (CGRA) [73–75]. The typical structure of CGRA is shown as Fig. 2.6. The on-chip structure of CGRA is usually composed of data buffer, configuration buffer, processing element (PE) array and configurable interconnection. Generally, the working phase of CGRA is divided into configuration phase, input phase, execution phase and storage phase. The configuration process of the on-chip structure is controlled by a controller. The PE unit is usually composed of a configurable ALU and input/output registers.

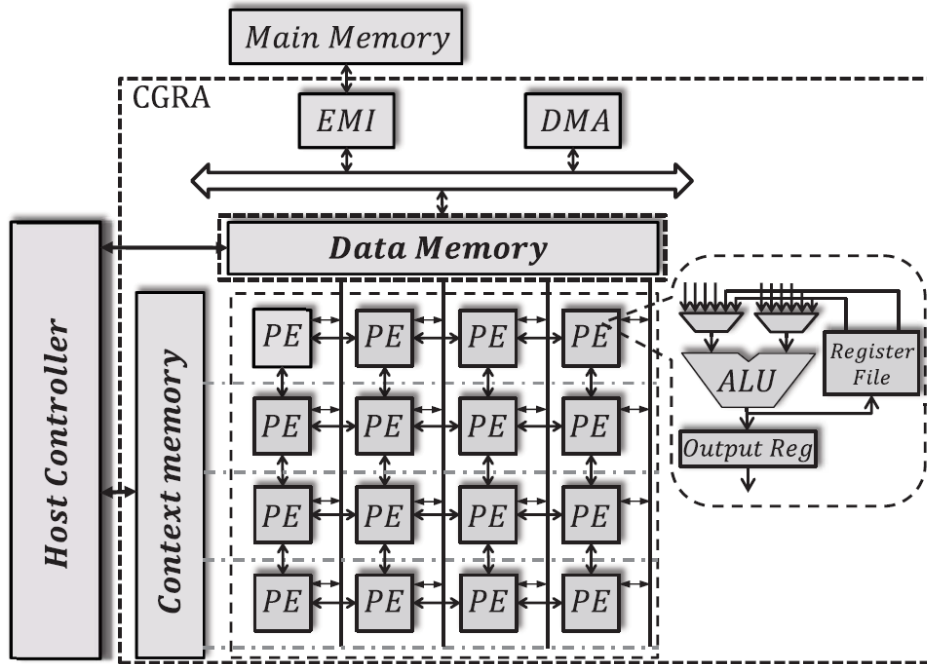


Figure 2.6: Typical structure of CGRAs [73].

According to the configuration information, the ALU in each PE unit can be configured to perform common logic operations, such as arithmetic operations, AND or NOT, shifting, etc. By using a coarser operand granularity, the typical overhead of fine-grained FPGAs can be reduced in CGRA. In addition, a coarse-grained architecture with ALU clusters as the basic unit is also reported, such as expression-grained reconfigurable array (EGRA) [76]. The basic units of these coarse-grained architectures are usually composed of ALUs, whose arithmetic logic functions have been fixed to a limited number after the design is completed, and cannot be reconfigured. In order to execute complex applications, it is usually necessary to map the computation-intensive part to the reconfigurable PE array.

In addition to the above four architectures, the concept of mixed-grained reconfigurable architecture [77, 78] has been proposed. Unlike single-grained reconfigurable architecture, mixed-grained reconfigurable architectures are expected to reconfigure hardware at multiple levels to improve efficiency and flexibility. The existing mixed-grained reconfigurable architectures are usually the integration of

CGRA and FPGA, with the purpose of increasing the flexibility of CGRA in fine-grained configuration.

3 Neural Network Prototype of Proposed Architecture

3.1 Spatial-Expanded Implementation

Many high-performance NN accelerators exploit time-division multiplexing (TDM) architecture, only arranging a small number of hardware neurons on chip, or mapping a complex NN into a small PE array by layer [79–84]. At each time step, some neurons in a layer are temporarily mapped to the network by obtaining the corresponding synapses. The intermediate output is stored to be used as input to the next layer of neurons. It takes several time steps to execute the entire NN. In this way, a large-scale NN can be effectively executed, but the challenge is how to store a large number of parameters into on-chip memory to overcome the bottleneck of data transmission. By contrast, in a NN accelerator based on spatially expanded in parallel (SEP) architecture [85], the hardware structure and operation are similar to that of a biological NN: Synapses are stored in distributed locations close to neurons; all neurons are mapped to the hardware; data flows from the input layer to the output layer. In addition to the resulting short synaptic weight access latency, the internal synaptic weight bandwidth is high. Meanwhile, distributed synaptic storage can save most of the power consumption caused by data transmission.

The architecture of TDM and SEP are shown in Fig. 3.1. A PE performs the basic functions of a neuron. The number of PEs in the TDM architecture is the same as the width of the NN, and the calculation time requires cycles of the depth of NN. In each cycle, the TDM architecture must obtains weights from the large-capacity on-chip weight memory, while the SEP architecture only have to access the internal weight register. By using pipeline technique, the SEP

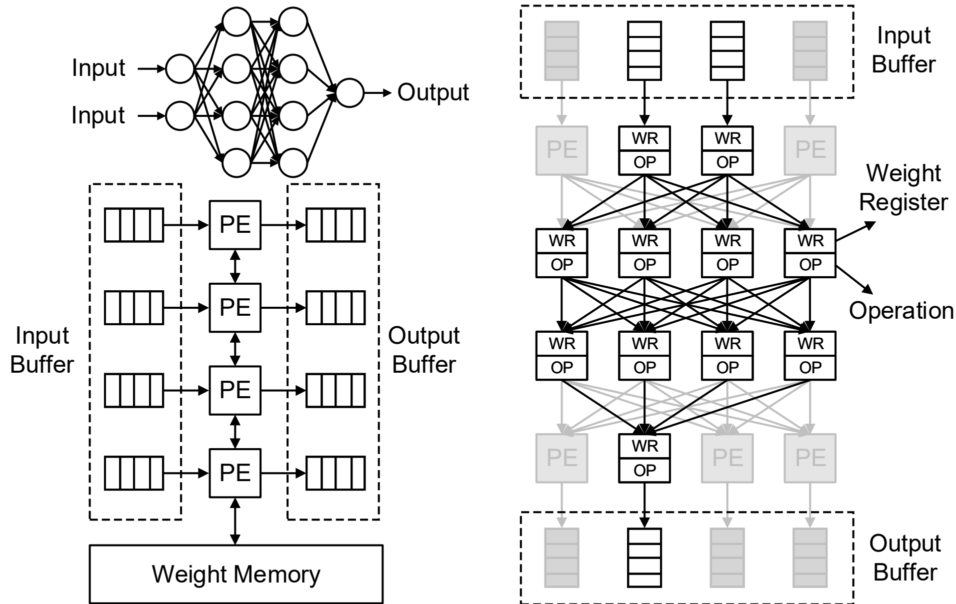


Figure 3.1: A NN performed by TDM and SEP architecture.

architecture can improve runtime throughput, but will produce a larger area. The key factor in choosing TDM or SEP to implement hardware NNs is the dimension of input and output, because it can significantly affect the depth and width of the network topology. Since the goal of our accelerator is to accelerate the functions evaluation in applications, such as the calculation of Euclidean distance in k-means clustering, to reduce the calculation delay. The input dimensions of most of this applications will not exceed 10, so the focus of this paper is on the SEP architecture rather than the TDM architecture.

3.2 Bisection Neural Network

NN-based approximate computing accelerate the application by learning the mapping of input data and output results. Many previous works have proposed the use of multi-layer fully-connected (FC) networks (i.e., multi-layer perceptron, MLP) to design accelerators. The training process includes selecting the network topology, learning synaptic weights and neuron biases. Figure 3.2 shows a typical MLP in the middle. In this topology, each neuron structure in i-th layer receives

all (i.e., N_{i-1}) inputs from the previous layer and outputs the results to all (i.e., N_{i+1}) neurons in the post layer. In this case, each neuron in i-th layer consists of N_{i-1} synapse connections, one neuron and N_{i+1} output interfaces. The output function of j-th neuron in the l-th layer can be formulated as

$$x_j^l = \sigma(\sum_{i=1}^{N_{l-1}} w_{i,j}^{l-1} x_i^{l-1} + b_j^l) \tag{3.1}$$

where x_i^{l-1} is the input from the neuron i in layer $(l-1)$, $w_{i,j}^{l-1}$ denote the weight of the edge connecting neuron i in layer $l-1$ and neuron j in layer l, b_j^l denote the bias of the neuron j in layer l. σ refers to an activation function, i.e., *sigmoid* or rectified linear unit (ReLU), denoted as follows:

$$sigmoid(x) = \frac{1}{1+e^{-x}} \tag{3.2}$$

$$ReLU(x) = max(0, x) \tag{3.3}$$

sigmoid is widely used in classification because it is easy to derivate and compress values. However, because it includes division and exponential operations, it is not suitable for the implementation of the SEP architecture, because each PE needs to be activated, which causes a lot of hardware overhead. In contrast, the easy hardware implementation of ReLU makes it more suitable for NN regression [86]. However, since the output of ReLU in the negative domain is 0, which leads to neuron death, [87] proposed to use Leaky-ReLU instead of ReLU to activate neurons, which is defined as follows:

$$LReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases} \tag{3.4}$$

where α is an adjustable negative slope parameter.

In the FC network, redundant synaptic connections may not have an impact on the final results, but they occupy a lot of hardware resources [88–90]. For example, for a neuron with N_i synaptic connections, its output function needs to perform N_i multiply accumulate (MAC) operations, and store N_i+1 parameters. NN pruning [91] and neural architecture search [92–94] are two kinds of network sparsity

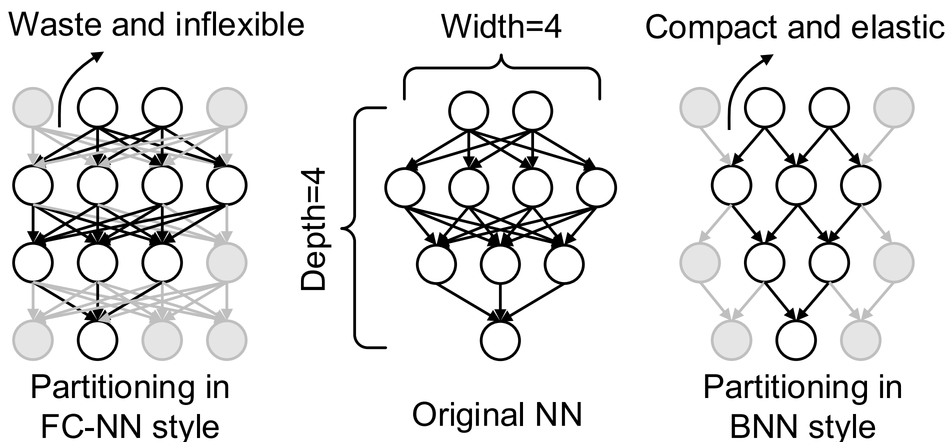


Figure 3.2: Partitioning an original NN in FC-NN and BNN style hardware.

methods for parameter compression and efficient hardware implementation of NNs. However, no matter whether it is a FC or sparsely connected network, it cannot be flexibly reconfigured on SEP hardware. Therefore, it is a feasible solution to design a hardware NN based on a predefined network structure [95,96]. BNN is a feedforward NN with a predefined bisection topology, which is used to replace the function of FC network. The definition of each neuron in BNN is as follows:

$$x_j^l = \sigma(w_{1,j}^{l-1} x_i^{l-1} + w_{2,j}^{l-1} x_{i+1}^{l-1} + b_j^l) \tag{3.5}$$

$$j = i + d \tag{3.6}$$

where d denote the number of neurons in a row. Assuming a BNN is an L-layer network, the depth is defined as the number of layers L, and the width is defined as $max(N_1, N_2, ..., N_L)$. For an M-input BNN, the minimum topology is defined by constraint $N_{i+1} = N_i - 1$ of each layer, and the width and depth of the minimum topology both are M. By modifying the constraints on the number of neurons between layers ($N_{i+1} = N_i + 1$ or $N_{i+1} = N_i - 1$), we can arbitrarily set the width and depth of an M-input BNN.

The right side of Fig. 3.2 shows a typical BNN, which is a sparse network with symmetrically connection. Specifically, each neuron in the BNN only accepts the outputs from two adjacent neurons in the previous layer; and fans out the data to

two adjacent neurons in the post layer. Compared with a fully connected hardware NN, BNN has the following advantages: (1) The sparse network structure can reduce the overhead of on-chip memory for parameter storage; (2) Dividing a large-scale BNN into multiple sub-networks avoids wasting a large number of virtual synaptic connections in the hardware; (3) It is easy to expand the NN on multiple chips without considering the complicated wiring.

3.3 Towards Multi-Grained Reconfigurable Architecture

Figure 3.3 shows the workflow of proposed architecture. The right side shows training process of neural networks, the left side shows mapping process from application codes to hardware, and the middle shows a general hardware architecture of a reconfigurable accelerator, which includes the following steps:

- Determine the dimensions of input/output data to select the suitable NN model (network topology) that balances accuracy and efficiency.
- Generate and collect input/output pairs that reflect actual execution of the target function, and run the back propagation algorithm to train parameters of the NN offline.
- Call the trained NN to generate a bitstream with model parameters and configuration files during compilation, which is used to configure the accelerator.
- Throughout execution, the accelerator is invoked to perform a NN-powered function evaluation in lieu of executing the original code region.

Unlike previous NN accelerators based on the TDM architecture, the BNN-based design is to symmetrically map all neurons and synapses in the network to hardware in a simple manner. Therefore, our purpose is not to accelerate those huge and complex popular NNs for classification, such as VGG network, etc., but to use the inherent flexibility of BNN to design a multi-grained reconfigurable accelerator for approximate computing. The efficient and compact BNN topology

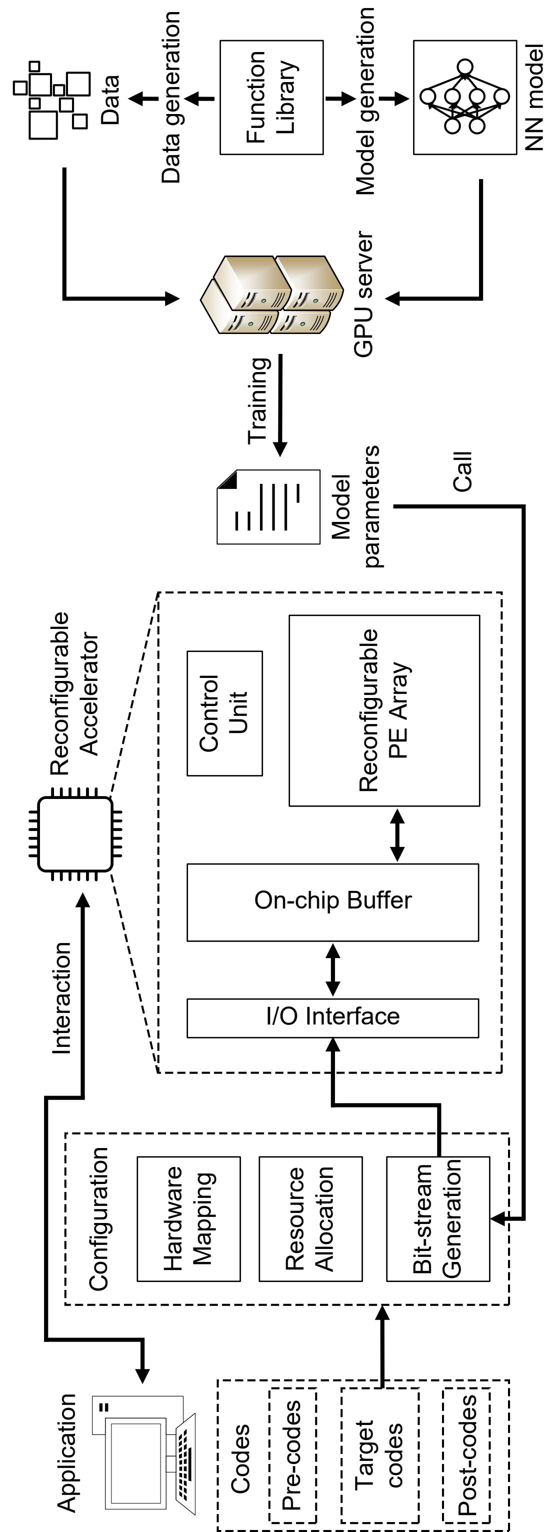


Figure 3.3: Workflow of proposed architecture.

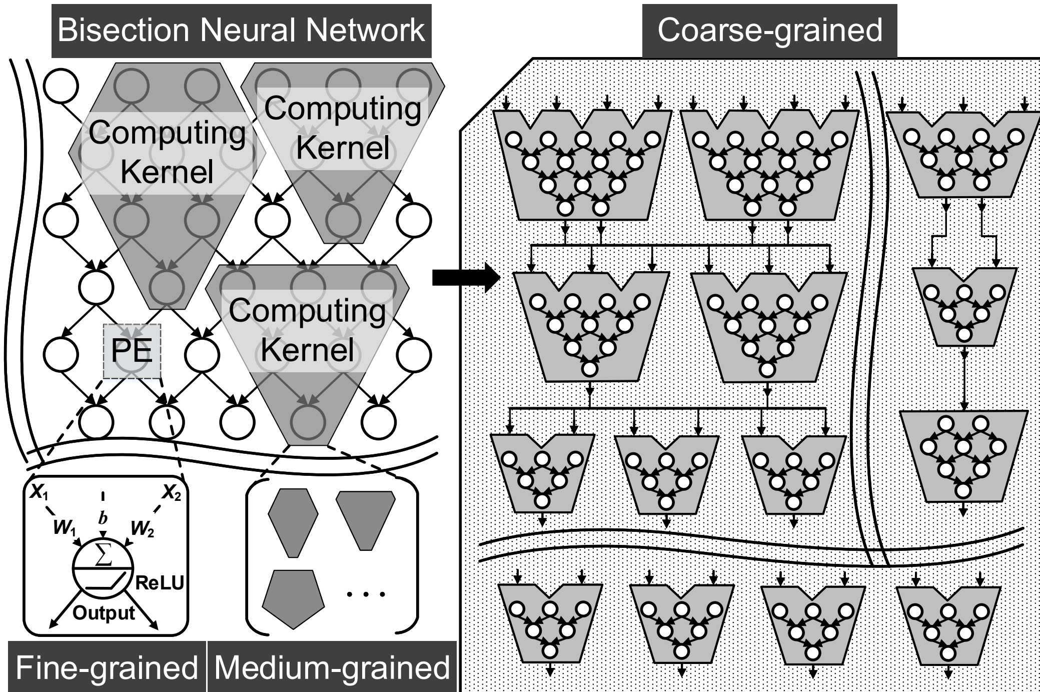


Figure 3.4: BNN-based multi-grained reconfiguration architecture.

allows us to build a large-scale network, and a single task may only occupy a small part of the network. This enables multiple tasks in applications to share the entire network without resource conflicts. Even if there is only one task in the application, we can also build multiple copies of a computing kernel for this task on the network to further improve the overall throughput. This offer a parallel hardware implementation for massive complex computations.

The basic concept of the accelerator is shown in Fig. 3.4. Specifically, we consider reconfiguring BNN-based accelerators at three levels. Fine-grained reconfiguration is based on the control of neurons and synaptic behavior. The synapse weight and bias corresponding to each neuron can be reset by modifying the parameter register to achieve the purpose of reprogramming. Medium-grained reconfiguration is based on the combination of multiple synapses and neurons to construct a computing kernel. Each neuron can be configured as an input/output layer neuron that exchanges data with on-chip memory or a hidden layer neuron that only processes data. Coarse-grained reconfiguration is based on the organi-

zation and allocation of multiple computing kernels on chip. The purpose is to minimize the number of unallocated neurons to maximize the utilization of neurons and to improve the communication efficiency between multiple computing kernels.

In order to reprogram neurons, we use a PE to describe the entire neuron that contains a neuron, synapses and its control module. PEs are organized into a two-dimensional array on-chip and interconnected by BNN topology. According to the demands of the target application, the entire PE array can be arbitrarily cut into multiple computing kernels with a small-scale bisection topology neural structure. Each computing kernel is formed by cascading input layer PE (PE_{in}), hidden layer PE (PE_{hd}) and output layer PE (PE_{out}). The computing kernel can be regarded as a multi-input function evaluator. If the computing kernel expects to support M inputs, the number of neurons in the input layer will be set to M. Starting from the first layer, the number of neurons in each hidden layer is +1/-1 of the previous layer, until the number is equal to 1, which is the output layer. This means that we can arbitrarily adjust the width and depth of the BNN until the preset accuracy and efficiency constraints are met. Each PE_{in} of input layer reads one-dimensional data of the input vector and passes it to the next layer in parallel. The data is loaded from the external memory to the local memory, which could be accessed by PE_{in} in parallel with a high bandwidth. After the calculation process, the PE_{out} writes the results back to the local memory.

3.4 Design Flow

Figure 3.5 shows the software and hardware division of the design flow. At the hardware level, after designing the Verilog HDL code for the hardware architecture, we use Synopsys Design Compiler with Renesas 65nm CMOS library to synthesize the HDL code to obtain netlist file. Then, we use Synopsys Verilog Compile Simulator to simulate the netlist file and get the optimized design parameters. If the functional verification is correct, we perform FPGA-based simulation on the Xilinx Vivado design kit, and then implement the HDL code on the ZCU102 SoC. At the software level, after given a target function, we use this function to generate training data and the corresponding minimum initial model

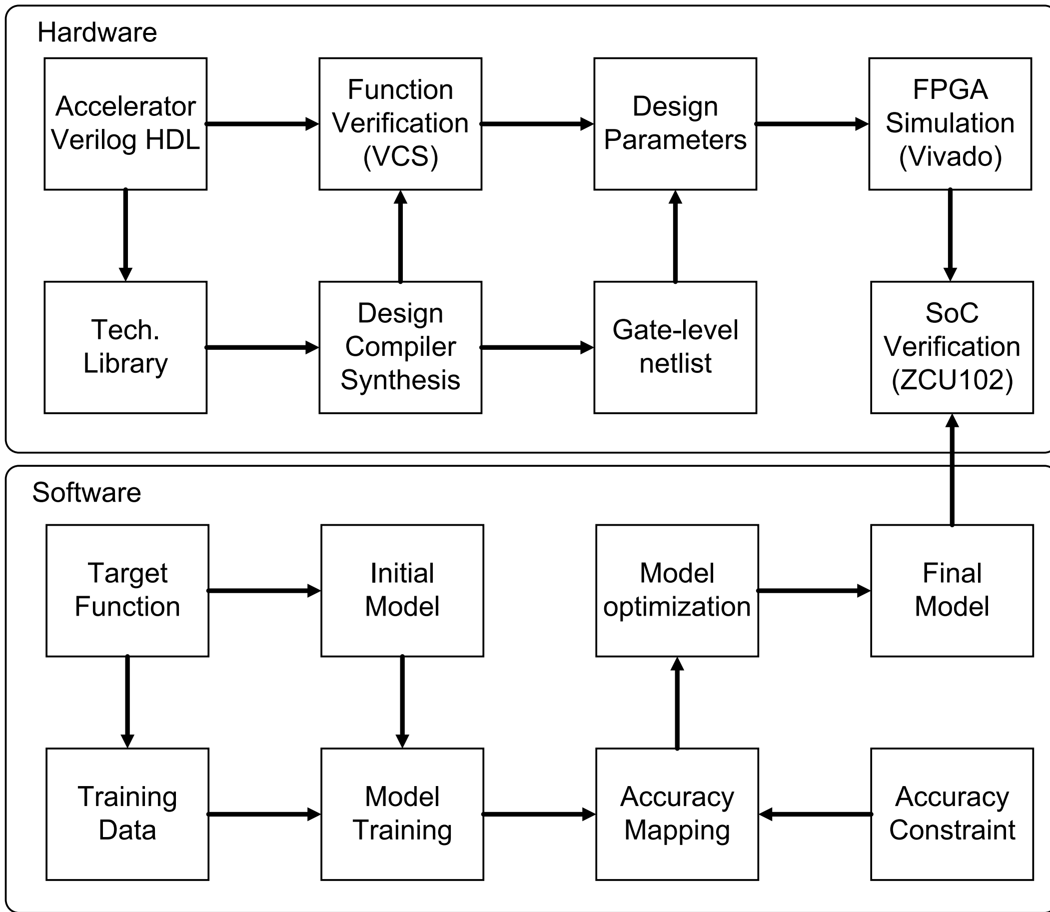


Figure 3.5: Software-hardware division of the design flow.

topology to training, then obtain model parameters and accuracy results. Generally, the initial topology cannot meet the preset accuracy constraints. Therefore, we optimize the model by modifying the topology to achieve the preset accuracy constraints. Finally, the optimal model that satisfies the accuracy constraints is output.

The topology of BNN is the algorithmic basis of the reconfigurable architecture in this research. Unlike a fully connected network, the BNN topology has a special form of sparse connection. When modeling the BNN on software, we need to generate the corresponding topological mask matrix for the BNN of a specific topology. For example, the mask matrix of a BNN as shown in Fig. 3.6. When using the backpropagation algorithm to train model parameters, masking the

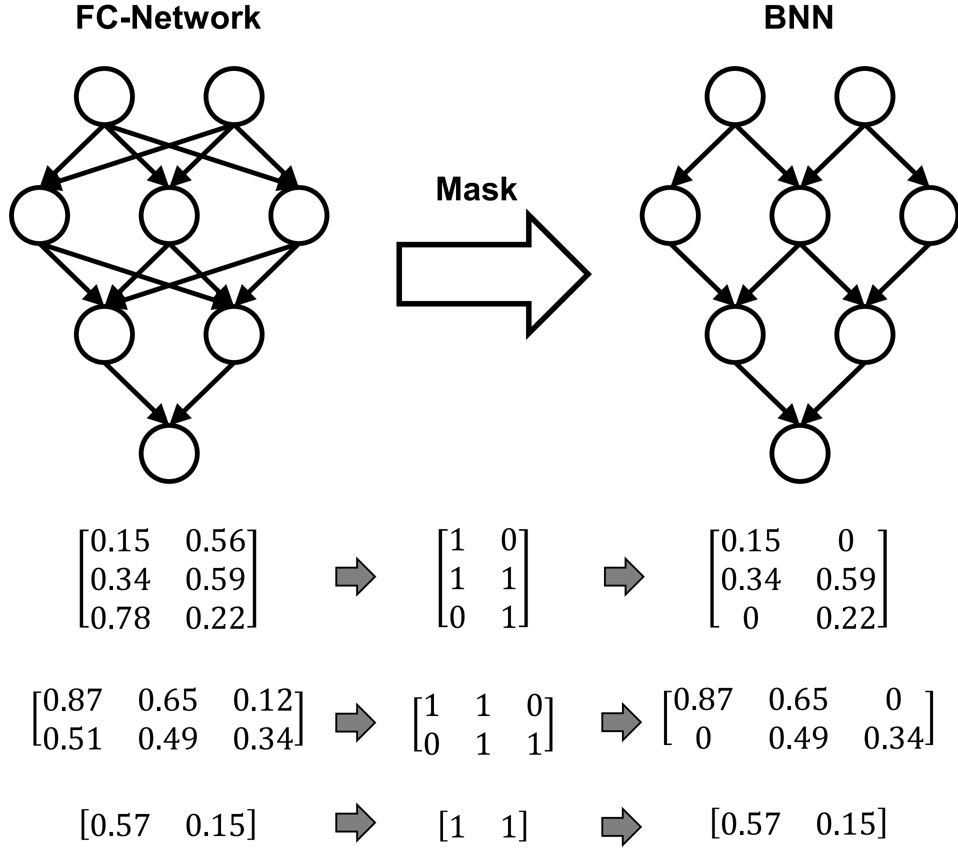


Figure 3.6: Convert FC-NN to BNN using mask matrix.

parameter matrix by the mask matrix, the fully connected network model can be constrained to the corresponding BNN topology. To efficient get BNN topology from FC-NN, we propose Algorithm 1 to generate the mask matrix of any BNN model.

In Algorithm 1, we only consider a single layer. For the mask matrix generation of the entire network, algorithm 1 is called multiple times. The number of neurons N_i in i-th layer and the number of neurons N_{i-1} in the $(i-1)$-th layer are used as input to generate a mask matrix for i-th layer. Consider two scenarios: If $N_i < N_{i-1}$, the neuron at the edge of i-th layer only has a synaptic connection with one edge neuron of the $(i-1)$-th layer; If $N_i > N_{i-1}$, all neurons in i-th layer have two synaptic connections with neurons of the $(i-1)$-th layer.

Algorithm 1: Mask Matrix Generation for i-th Layer

```
   Input: N_{i-1}, N_i
   Output: mask_i
1  if N_{i-1} < N_i then
2  |  mask_i(0, 0) = 1;
3  |  mask_i(N_i - 1, N_{i-1} - 1) = 1;
4  |  x = 1;
5  |  y = 1;
6  |  while x < N_i - 2 do
7  |  |  mask_i(x, y) = 1;
8  |  |  mask_i(x, y + 1) = 1;
9  |  |  x = x + 1;
10 |  |  y = y + 1;
11 |  end
12 else
13 |  x = 0;
14 |  y = 0;
15 |  while x < N_i - 1 do
16 |  |  mask_i(x, y) = 1;
17 |  |  mask_i(x, y + 1) = 1;
18 |  |  x = x + 1;
19 |  |  y = y + 1;
20 |  end
21 end
```

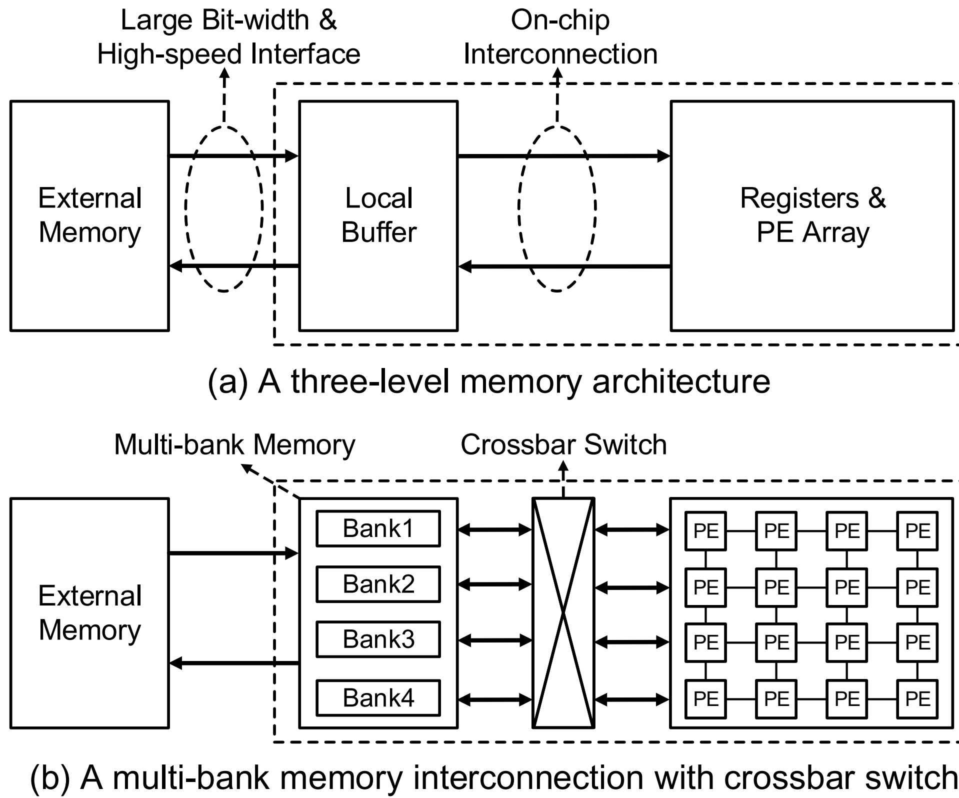


Figure 3.7: A top view of memory architecture for spatial array accelerator.

3.5 Challenges

3.5.1 On-Chip Interconnections

Although the spatial parallel architecture can effectively use data-level parallelism and fully pipeline to accelerate calculations, the data transmission between PE array and memory is the main bottleneck that limits the parallelism of the MuGRA [97]. In different configurations, the status of PE may switch from the hidden layer of the computing kernel to the input layer or the output layer, which requires each PE to have the ability to access local memory. In addition, there is a data dependency between the layers of the BNN, that is, the input data of the neurons in the post layer is the results of the neurons in the previous layer. According to this feature, the data flow of the computing kernel is calculated layer by layer from the input layer through the hidden layer to the output layer.

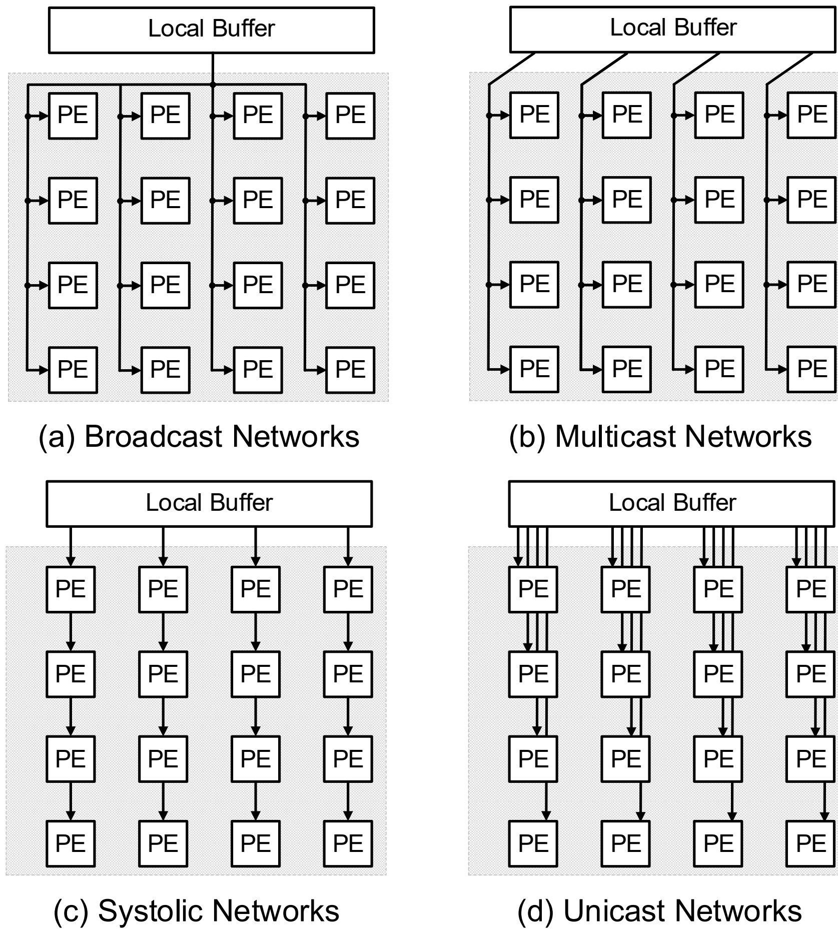


Figure 3.8: Four common NoC interconnection designs.

Therefore, pipelined computing kernel can improve the utilization of PE and the throughput of accelerator. The maximum throughput under this mapping is the calculation of a layer in each clock cycle. In order to achieve this throughput to save energy, the computing kernel must be allocated non-conflict access local memory ports to provide a continuous data stream.

Many spatial array accelerators (e.g., CGRA) adopt a three-level memory architecture including large-capacity external memory, local on-chip memory and PE internal memory, shown as Fig. 3.7(a). The arithmetic data generated by the main processor is stored in the external memory, and the data transmission between off-chip and on-chip memory is carried out at a block granularity. The local

buffer and PE are interconnected through a flexible NoC to support continuous data stream transmission.

The local memory is a unit that the PE array can directly access, and can be designed as a single bank or multiple banks. Figure 3.7(b) shows a typical multi-bank memory interconnect architecture. In this architecture, the local data memory is composed of multiple banks, and each bank has a single port/multi-port. For multi-bank memory, the programmer/compiler is responsible for ensuring that the data accessed by the PE exists in the bank it can access. The memory banks and the PE array are interconnected through crossbar switches to ensure that each PE can flexibly access any bank. However, the complexity of the crossbar switch increases sharply with the growth of the number of ports, which causes the cost of memory access for large-scale PE arrays become expensive.

Figure 3.8 shows several common NoC designs used in interconnections between PE arrays and local buffer [98]. Due to the property of BNN-based accelerator, using a single NoC design will not be able to balance bandwidth of data transmission and consumption of on-chip memory resource. For example, a broadcast network can use the least amount of on-chip memory bank to send data, but its low source bandwidth can limit the throughput when data reuse is low. Using a multicast network at the expense of consuming more on-chip memory banks can increase throughput, but PEs located in different rows still need to wait several clock cycles to obtain data. Combining the systolic network and the pipeline, the PE can obtain data from the neighboring PE without directly accessing the local buffer, but the flexibility of the PE array configuration is reduced. Unicast network is the most extreme way of interconnection. Each PE has a dedicated channel and independent connection to the memory bank, providing maximum throughput and flexibility for PE array. However, the cost of its design increases with the number of PEs, and therefore is difficult to support the expansion of scale of the PE array.

3.5.2 PE Utilization

In the case of a fixed array size, placing as many computing kernels on the PE array as possible can increase the utilization of the PEs to improve the energy efficiency of the system. Arbitrarily placing computing kernels on the array may

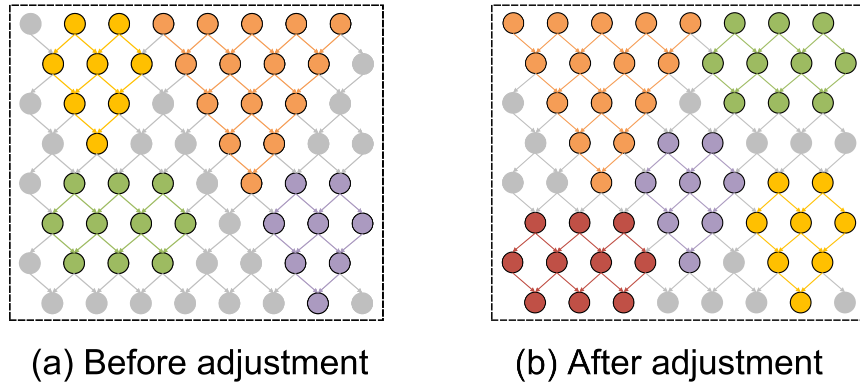


Figure 3.9: An example of improving the PE utilization.

result in reduced PE utilization. Figure 3.9 shows an example of placing computing kernels on an 8*8 PE array. The colored ones are PEs used by the computing kernels, and the gray ones are idle PEs. Figure 3.9(a) randomly placed 4 computing kernels on the array, a total of 41 PEs were used, and the utilization rate was $41/64 = 64.1\%$. In Fig. 3.9(b), after careful adjustment, while keeping the original 4 computing kernels on array, one more computing kernel is added, which increases the PE utilization rate to $51/64 = 79.7\%$. This example proves that if an effective configuration strategy is used, the efficiency of the entire system can be improved.

4 Hardware Architecture Design and Optimization

4.1 Overview of Proposed Accelerator

Figure 4.1 shows the overall architecture of the proposed MuGRA system. It is a CPU+FPGA heterogeneous architecture used to accelerate the approximate calculation of arithmetic functions. The architecture includes the host processor, FPGA coprocessor and external memory. A PE array, a finite state machine (FSM) controller and an on-chip local memory system are implemented on the FPGA coprocessor. We implemented the MuGRA system on the commercial system-on-chip (SoC) Xilinx ZCU102. The SoC provides a quad-core ARM Cortex-A53, an FPGA fabric, DRAM controller and 32.1Mb on-chip memory. The application program running on the host processor communicates with the coprocessor through the Advanced Extensible Interface (AXI), which can be used to implement memory-registers mapping.

After synthesizing the accelerator on the programmable logic, the host processor can configure it through the communication interface without reprogramming the FPGA. The PEs are interconnected in a bisection topology, and each PE can access the local buffers. Local buffers are all realized by BRAM, which can be set to FIFO or RAM mode. The configuration file includes synaptic weights, neuron biases and control field of each PE. In the static configuration phase, the host processor sends the configuration file to the configuration buffer (CB) on the FPGA for temporary buffering. Then, the FSM controller smoothly fanning massive configuration files from the CB to configuration registers inside PEs, which helps to reduce the transmission latency and achieve data reuse. In the execution phase, each PE accesses the parameters in its own configuration register instead

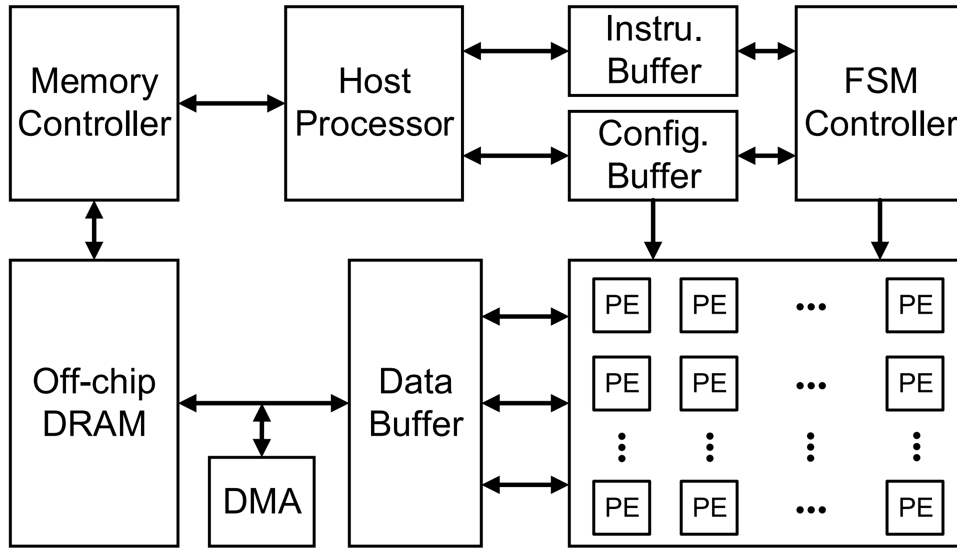


Figure 4.1: Overall architecture of the proposed MuGRA system.

of the CB.

The operation data generated by the application of the host processor is stored in the off-chip DRAM. Under the command of the host processor, direct memory access (DMA) is used to transfer data between the local data buffer (DB) and the off-chip DRAM. The DB is composed of an input buffer (IB) and an output buffer (OB). The IB is responsible for storing the data transmitted by the off-chip DRAM, and the OB is responsible for storing the calculation results of the PE array. There are multiple memory banks in each buffer, which can provide data for the PE array in parallel. Each memory bank is implemented based on BRAM and works in a double-buffered manner. Through this mechanism, the data exchange between the PE array and the DB and the data prefetching between the DB and the off-chip DRAM are carried out simultaneously, which is used to hide memory access delays. The instructions generated by the host processor are stored in the instruction buffer (InB) through the AXI interface. The FSM-based control unit is used to read, analyze and execute the instructions generated by the host processor to schedule the hardware accelerator system on the FPGA.

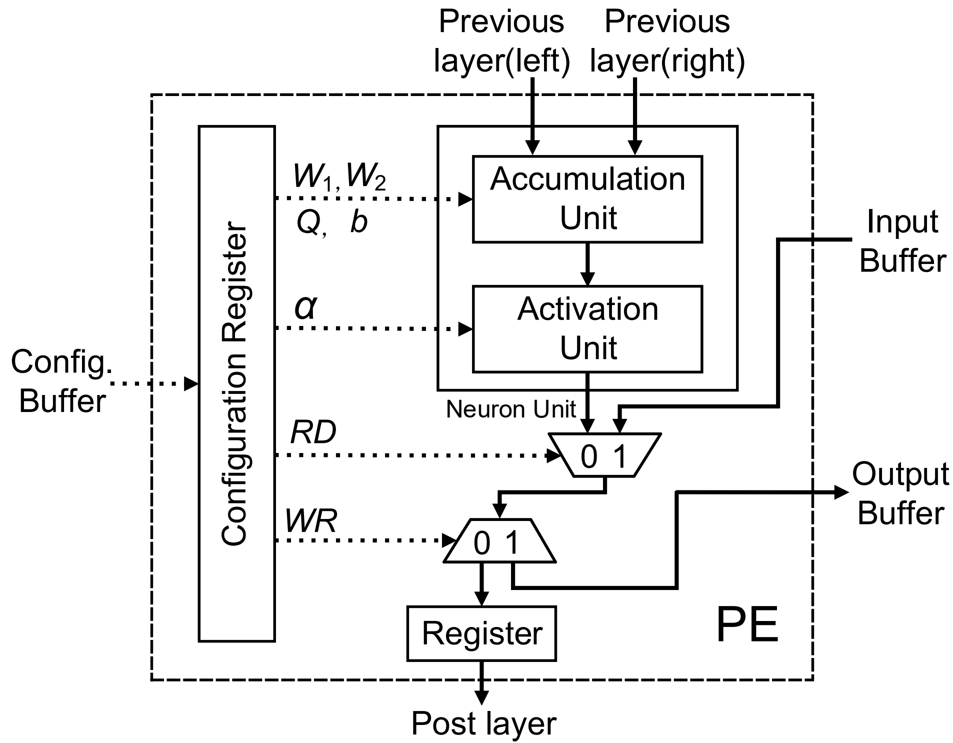


Figure 4.2: Architecture of a PE.

4.2 Design of PE Architecture

The architecture of PE is shown in Fig. 4.2. It consists of a neuron unit (NU), a multiplexer (MUX), a demultiplexer (DeMUX), a configuration register and an output register. Each PE is identical, and its operating mode is controlled by the programmable configuration register. The configuration register loads data from the CB and can only be modified during the static configuration phase. MUX is used to select output of PE from NU or IB, indicated by the *RD* field. When working as a input neuron in the network, the PE is in PE_{in} mode, its input comes from the IB, and the NU does not perform any processing. In other cases, the input of PE comes from previous layer of the PE, and output calculation results of the NU to post layer or OB. The output path is controlled by DeMUX, indicated by the WR field. The results of the PE_{in} and PE_{hd} are output to the next layer, while the results of PE_{out} are output to the OB.

The NU is composed of an accumulation unit and an activation unit, as shown in Fig. 4.3. Input/output data, synapse weights and biases are all in N_w-bit signed fixed-point format, with 1-bit sign, Q-bit fraction and $(N_w - Q)$-bit integer. A reasonable value of N_w will be tested in Section V. The value of Q is stored in the configuration register. By modifying the value of Q, the range and accuracy of neuron operands can be modified according to the application. The two input data X_1, X_2 from the adjacent PE of the previous layer are passed into the accumulation unit as operands, and the fixed-point multiplication operation $X_1 * W_1$, $X_2 * W_2$ is performed in parallel with the synaptic weights W_1, W_2 passed in from the configuration register respectively. The result after fixed-point multiplication is expanded to $2N_w$-bit, and truncated to N_w-bit original fixed-point format after right shifting by Q bits. Then, perform two addition operations serially to output the result $X_1 * W_1 + X_2 * W_2 + b$, where b is the bias of the neuron. A configuration register stores two N_w-bit synapse weights, one N_w-bit bias, and an 8-bit control field, for a total of $3N_w + 8$ bits. For FPGA implementation, it is possible to synthesize a configuration register of this size near the computation logic by using the on-chip flip-flops to achieve distributed storage of parameters. Specifically, in the description of Verilog syntax, we define the weight registers inside the "PE module", and synthesize them as flip-flop (FF) instead of latch through sequential logic description. Then, the register and computation logic will be synthesized in a same PE block, by using the FF and LUT within the basic cells (slice) of FPGA respectively. It should be noticed that the number of FF and LUT in each slice is very limited (Each slice contains 8 FFs and 4 LUTs). Our strategy is feasible only if the scale of weights in each PE is small, which is difficult to achieve with conventional FC-NN. Fortunately, with 16bit implementation, each PE of the BNN merely stores two 16-bit weights, one 16-bit bias and an 8-bit control field, for a total of 56 bits. Therefore, connecting 56 FFs in 7 slices in parallel is sufficient as a register to store all the parameters of a PE block, and LUTs in these slices can be used as computation logic.

The activation unit provides the NN with non-linear approximation for transcendental functions regression. We recommend using Leaky-ReLU to activate neurons in BNN-based regression, since the output of ReLU in negative domain is zero, which leads to neuron death and non-negative output. For efficiently

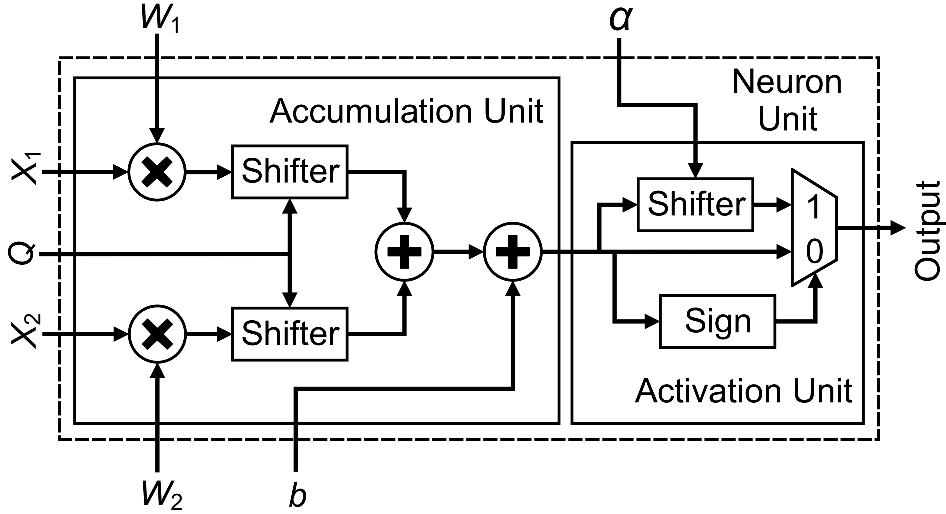


Figure 4.3: Architecture of a neuron unit.

implementing Leaky-ReLU on hardware, we set the parameter of negative part slope to $\alpha = 2^{-p}$ when pre-training the model, where p is a positive integer. For example, if $\alpha = 0.125$, then $p = 3$. Therefore, by shifting the negative input to the right by p bits, the result of Leaky-ReLU can be quickly calculated without a multiplier. The value of p is adjusted according to the target function, so it is stored as a parameter in the configuration register. The MUX in the activation unit is used for the segmented output of Leaky-ReLU, controlled by the sign bit of the input data. The operation of PE is completed in one clock cycle, and calculation result of the PE is stored in the output register and transmitted to the adjacent PE of the post layer in next clock cycle. The PE is then fully pipelined across layers that exploits the inter-layer data correlation of the NN.

4.3 On-Chip Interconnection for Efficient Buffer Utilization

The memory access conflict between the PE array and the local DB is the main bottleneck that limits the efficiency of MuGRA. After configuration, the PE array is divided into multiple computing kernels. The computing kernels work in

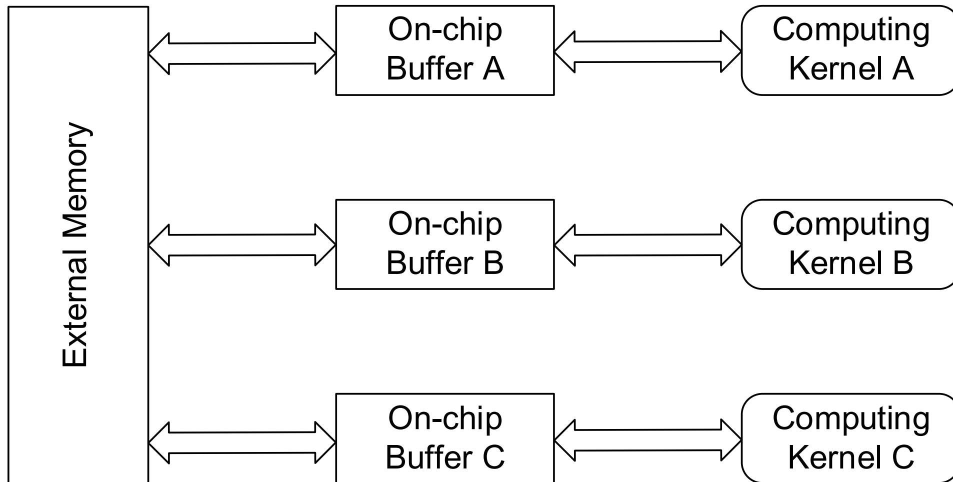


Figure 4.4: Logic structure of interconnection between buffers and kernels.

parallel and independently access the local DB. Multiple PEs$_{in}$ inside the computing kernel load data from the local DB at the same time to achieve data-level parallelism. Therefore, the local DB is designed as a multi-bank architecture, which can provide input data for the PE array in parallel. The logic structure of interconnection between buffers and kernels is shown as Fig. 4.4.

In the MuGRA architecture, we hope that all PEs have the opportunity to be configured as PE$_{in}$ that can access local DB without conflict. A straightforward approach is to configure independent memory bank for all PEs. If a PE is configured as PE$_{in}$, the input data will be buffered in the memory bank directly connected to it. Although this design ensures the conflict-free memory access of the PE$_{in}$, it causes huge overhead and redundancy of the memory unit, because only a few memory banks may be used in a configuration. Another way to reduce memory redundancy is to use a crossbar switch to connect the memory bank and the PE array. Only the PE set as the PE$_{in}$ can communicate with the memory bank. However, the inextensibility and complexity of the crossbar switch makes it only suitable for small-scale PE arrays. Therefore, it is necessary to design the interconnection between the DB and the PE array according to the features of the MuGRA architecture.

In a computing kernel, the positions of all input PEs are in different columns of the first layer (i.e., the input layer), and the output PEs are in the last layer (i.e.,

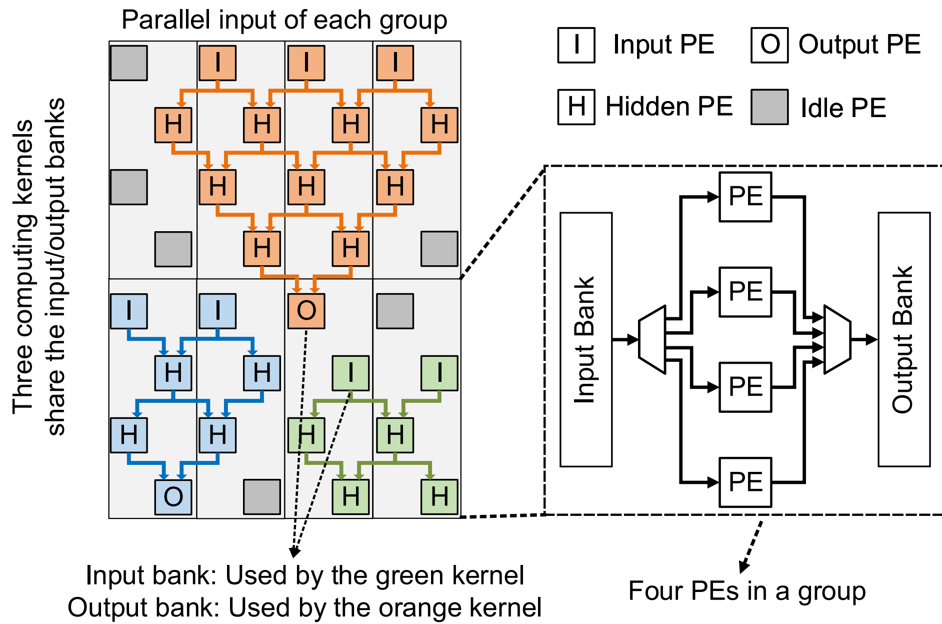


Figure 4.5: Interconnection of PEs and local buffer.

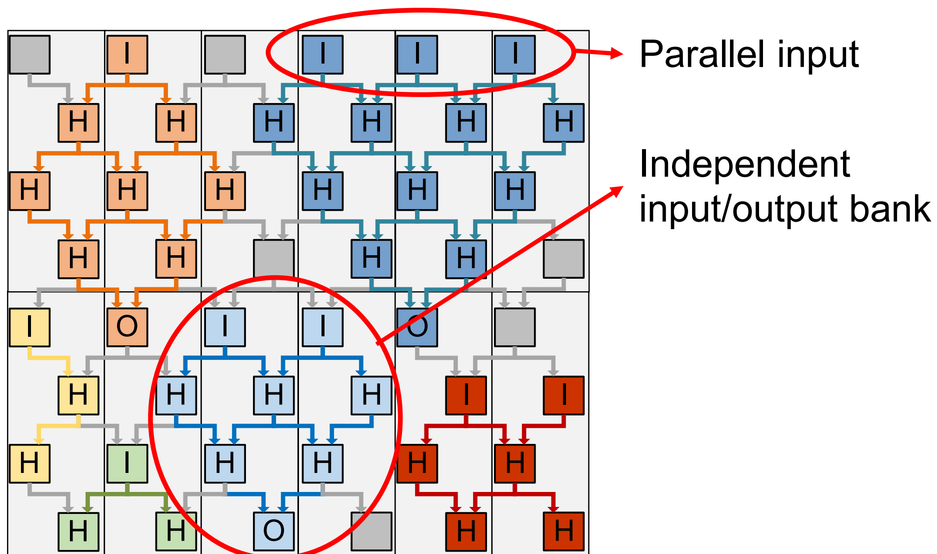


Figure 4.6: An example of large array configuration.

the output layer), and the middle layer (i.e., the hidden layer) does not access the memory bank. In order to reduce the use of memory banks, we interconnected PEs and memory banks by memory-sharing, as shown in Fig. 4.5. According to the features of BNN, the minimum computing kernel is a 4-layer (1 input layer, 2 hidden layers, 1 output layer) network, so we connect 4 PEs in the same column to 1 input bank and 1 output bank as a group. Under a certain configuration, there is only one PE accessing the input bank and one PE accessing the output bank in a group. This sharing method has the following advantages: 1. The data-level parallelism of the computing kernel will not be destroyed since the PE of each column can access the memory bank without conflict; 2. The overhead of the memory bank is reduced by 4x compared with unicast networks; 3. Scaling up the PE array is easy since the implementation cost increases linearly. Although the flexibility of PE array configuration is undermined, the features of the BNN can be used to cover up. For example, the three input PEs of the orange computing kernel in Fig. 4.5 are in different groups (columns), so the inputs are parallel. Since the hidden PEs do not access the memory banks, it will not cause a memory access conflict with the input PE in the same group. The output PE of the orange kernel and the leftmost input PE of the green kernel use the output bank and input bank of a same group respectively, while the output PE of the green kernel will use an output bank of another group (the green kernel is not fully shown). In the example of Fig. 4.5, 7 out of 8 input banks are used (three in orange kernel, two in blue kernel and two in green kernel). Therefore, in this example, the utilization rate of input bank of this design is 7/8*100%=87.5%, while the unicast network is 7/56*100%=12.5%, since each PE of the unicast network is configured with one input bank. An example of large array configuration is shown as Fig. 4.6.

4.4 Controller Design

The controller of the proposed architecture consists of a FSM and an instruction decoder. The state transition of the FSM is shown in Fig. 4.7, which has 6 states: Idle, Load Config, Run Config, Load Data, Execution and Store Data. After the system is started, the state machine automatically enters the Idle state, and reads

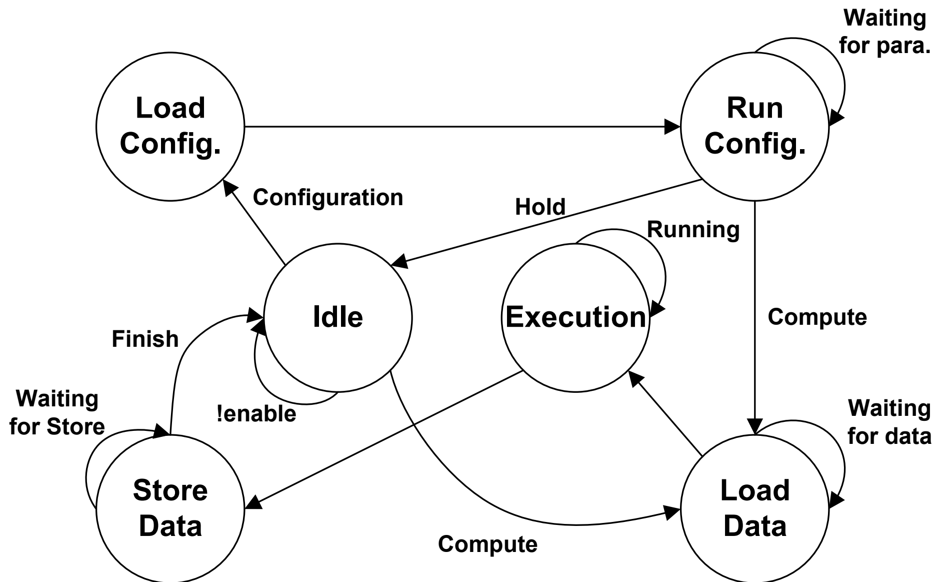


Figure 4.7: FSM controller of the system.

instructions from the first address of the instruction buffer. If the configuration instruction is read, the system enters the configuration mode and waits for the configuration information loading to configuration buffer. In the configuration mode, the controller loads the configuration information from the configuration buffer and broadcasts it to the PE array. After finishing the configuration, jump back to the Idle state or enter the execution mode according to the instruction. In the execution mode, the controller loads data from the data buffer, and the PE stores the result back to the data buffer after performing the computation. After the execution is completed, the system automatically enters the Idle state. The controller supports one configuration and multiple executions. Therefore, the execution mode can be entered from the Idle state through compute instructions without executing the configuration mode.

4.5 Computation Datapath

The configuration of the PE array includes the setting of internal data path of the computing kernel and allocation of multiple tasks on the PE array. For computing kernels, each of which can be regarded as a multi-dimensional function evalua-

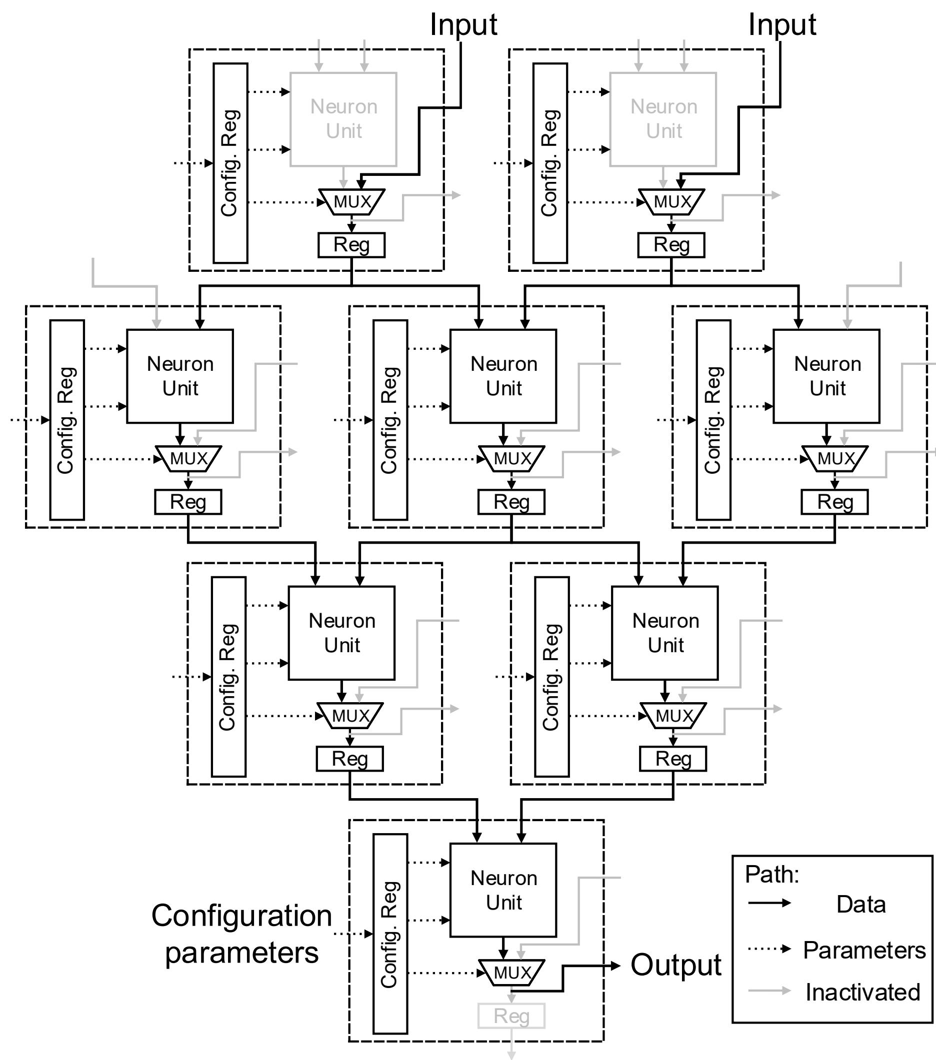


Figure 4.8: Datapath of a computing kernel.

Status Table

PE	Task	Mode
0	0	2'b00
1	1	2'b01
2	0	2'b00
3	2	2'b01
4	2	2'b01
5	1	2'b10
6	1	2'b10
...	...	...
24	0	2'b00

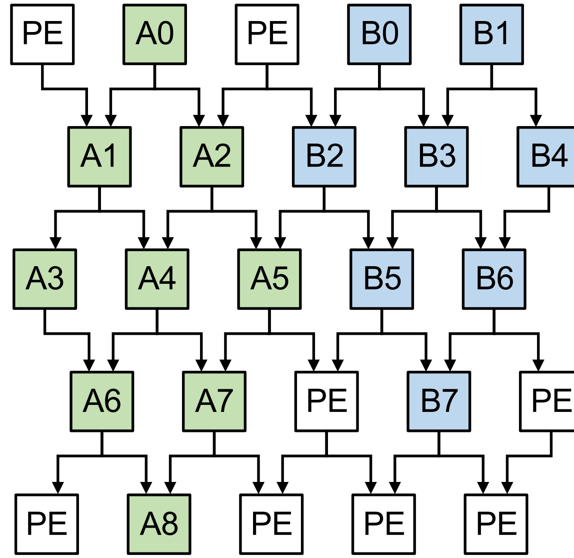


Figure 4.9: Allocating two computing kernels on a 5×5 PE array.

tor with SIMD feature to perform approximate calculations of specific arithmetic function. PE can be reused in different computing kernels is the key to reconfiguration. Therefore, we design the PE as a triple mode of PE_{in}, PE_{hd} and PE_{out}, which can be reconfigured by modifying parameters in PE. Figure 4.8 shows an example of a data path for a computing kernel with two PE_{in}, five PE_{hd}, and one PE_{out}. The PEs of the input layer are all PE_{in} mode, and they simultaneously load input data from the IB to exploit the intra-layer parallelism of the NN. The PE in the hidden layer is set to PE_{hd} mode, receives data from the PEs of the previous layer, and feeds the calculation result to the subsequent PEs. The PE in the output layer is set to PE_{out} mode, and its output is stored in the OB instead of the PE in the post layer. All PEs have been pipelined, that is, new data is acquired at each input of each clock cycle and new results are sent at each output of each clock cycle.

The running mode of all PEs in the array is maintained by a scheduling status table. The compiler can generate hardware configuration parameters based on the information in the table, so as to allocate idle PE resources to the incoming tasks. Figure 4.9 shows an example of allocating two computing kernels named A and B on a 5×5 PE array. Since all parameters are set to zero when the accelerator

is initialized, the hardware isolation of multiple computing kernels exploit the nature of the BNN without additional hardware control logic. For example, as shown in Fig. 4.9, A_5 is physically connected to A_2 and B_2. The output of A_5 can be expressed as $PE_{out}(A_5) = \sigma(W_1(A_5) * PE_{out}(A_2) + W_2(A_5) * PE_{out}(B_2) + b(A_5))$. Even if the input data of A_5 contains the output from B_2, since $W_2(A_5)$ is set to zero during initialization and has not been modified during the configuration phase, the value of $W_2(A_5) * PE_{out}(B_2)$ is always equal to zero.

4.6 Configuration Strategy

Efficiently mapping computing kernels to the PE array is a key technology for coarse-grained reconfiguration. For large-scale PE arrays, the higher the utilization rate of PE, the higher the energy efficiency of the entire system. Even if there is only one type of computing kernel, we can copy this computing kernel multiple times and place them on the PE array for parallel acceleration. Therefore, it is necessary to explore efficient configuration strategies for effective PE array configuration. In order to facilitate the description, we give the following definitions.

Definition 1. A PE array is denoted as a two-dimensional matrix $Array(x, y)$, where x and y indicate the coordinate of a PE in the x-th row and the y-th column, respectively.

Definition 2. Suppose that there are k types of computing kernel $Kernel_1$, $Kernel_2$, ..., $Kernel_k$. Let $Kernel_i(j)(1 \leq i \leq k, 1 \leq j \leq Kernel_i.len)$ indicated the number of neurons in j-th layer of i-th kernel, where $Kernel_i.len$ is the depth of i-th kernel.

Definition 3. Let $P(i)$ denotes the allocated number of i-th kernel, $N(i)$ denotes the number of neurons of i-th kernel. Thus, we can obtain

$$N(i) = \sum_{j=1}^{Kernel_i.len} Kernel_i(j) \tag{4.1}$$

The utilization rate of the PE array can be defined by

$$PEU = \frac{\sum_{i=1}^{k} P(i) \cdot N(i)}{H \cdot W} \times 100\% \tag{4.2}$$

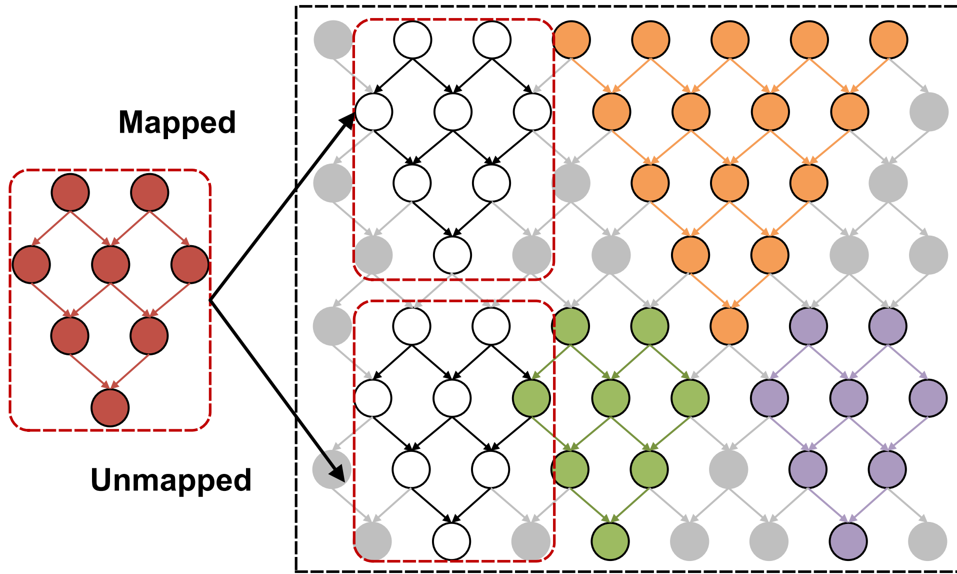


Figure 4.10: An example of mapping a computing kernel on PE array.

Where H and W are the height and width of the PE array, respectively. For a computing kernel of arbitrary shape, a sequence can be used to describe the topological structure. For example, the topology 1-2-3-2-1 is a 5-layer network with 9 neurons, while the topology 1-2-3-4-3-2-1 is a 7-layer network with 16 neurons. In the case that the PE array has been partially allocated, a new computing kernel may not be placed, as shown in Fig. 4.10. Therefore, we propose Algorithm 2 to determine whether any computing kernel can be placed on the PE array. In Algorithm 2, the input is the topological description of the computing kernel and the placement coordinates of the neurons in the upper left corner, and the output is a Boolean value indicating whether the computing kernel can be placed legally. The PE array is described by a two-dimensional matrix and updated after placement.

For configuration strategies, we propose three solutions: naive random placement (NRP), optimized random placement (ORP) and greedy-based placement (GBP). For the NRP, we try t times, each time we randomly select coordinates and computing kernels on the PE array for placement, and take the one with the largest PE utilization rate as the result, as shown in Algorithm 3. The flow diagram of NRP is shown as Fig. 4.11. In this strategy, the number of trials has

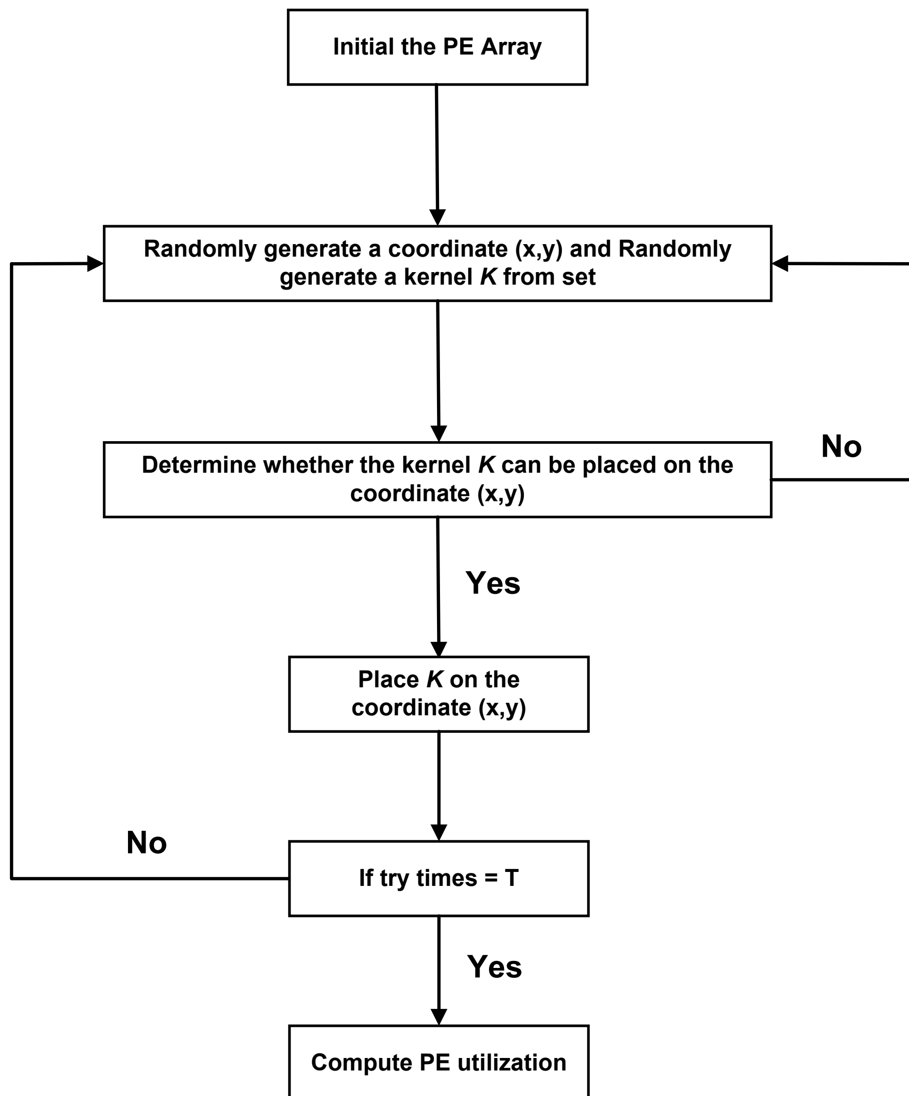


Figure 4.11: A flow diagram of NRP.

Algorithm 2: Allocate a computing kernel

```
   Input: Kernel, Array, x, y
   Output: Match, Array
 1 dx = 0; dy = 0;
 2 if (Array.H-x+1) < Kernel.len then
 3 |   Return(Match = 0);
 4 end
 5 for i = 1 to Kernel.len do
 6 |   if i == 1 then
 7 |   |   dx = x; dy = y;
 8 |   else
 9 |   |   if (dx+i-1) mod 2 == 0 then
10 |   |   |   if Kernel(i) > Kernel(i-1) then
11 |   |   |   |   dy = dy − 1;
12 |   |   |   |   if dy < 1 then
13 |   |   |   |   |   Return(Match = 0);
14 |   |   |   |   end
15 |   |   |   end
16 |   |   else
17 |   |   |   if Kernel(i) < Kernel(i-1) then
18 |   |   |   |   dy = dy + 1;
19 |   |   |   end
20 |   |   end
21 |   end
22 |   if (Array.W - dy + 1) < Kernel(i) then
23 |   |   Return(Match = 0);
24 |   end
25 |   for j = 1 to Kernel(i) do
26 |   |   if Array(dx+i-1,dy+j-1) != 0 then
27 |   |   |   Return(Match = 0);
28 |   |   end
29 |   |   Array(dx + i − 1, dy + j − 1) = Kernel.number;
30 |   end
31 end
32 Return(Match = 1); Return(Array);
```

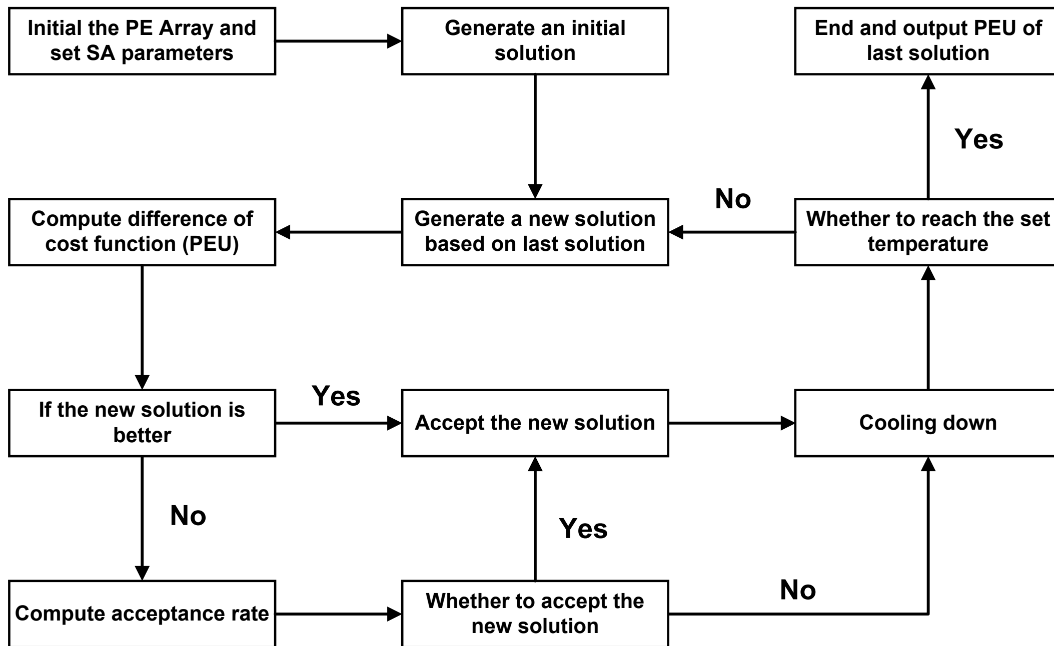
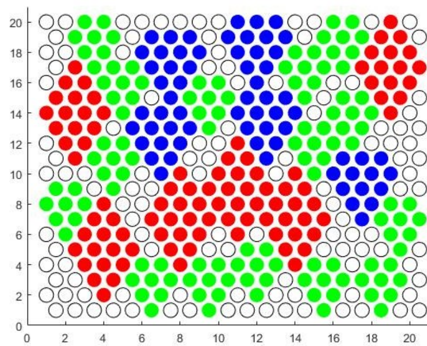
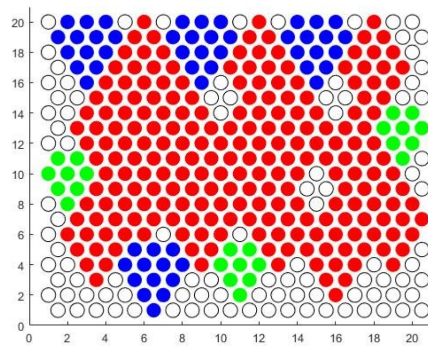


Figure 4.12: A flow diagram of ORP.

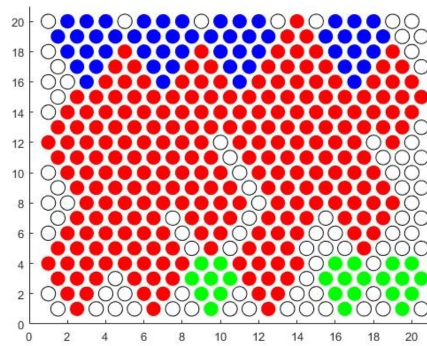
a great influence on the final result. Try as much as possible to get better results. For the ORP, based on the idea of simulated annealing, we mutate the result of initial random placement to obtain a relevant result. If the PE utilization of the mutation result is better than the initial result, then the result is accepted, otherwise the result is accepted with probability. Repeat this operation t times, and take the one with the largest PE utilization rate as the result. The flow diagram of ORP is shown as Fig. 4.12 and idea is shown in Algorithm 4. For the GBP, we scan all idle PEs in turn and place the computing kernel with the most neurons first. If it cannot be placed, try to place the computing kernel with the second most neurons until all computing kernels are traversed. The idea is shown in Algorithm 5. Figure 4.13 shows an instance of the three configuration strategies on a 20×20 PE array with three computing kernels.



(a) NRP(70.25%)



(b) GBP(75.00%)



(c) ORP(79.00%)

Figure 4.13: An instance of proposed configuration strategies.

Algorithm 3: Naive random placement (NRP)

```
   Input: T, Array, Kernels
   Output: Array, MaxU
 1 MaxU = 0;
 2 for t = 1 to T do
 3     Match = 0;
 4     A = Initial(Array);
 5     for i = 1 to size(A) do
 6         x = random(A.H);
 7         y = random(A.W);
 8         if A(x,y) == 0 then
 9             K = random(Kernels);
10             Match = Allocate(K, A, x, y);
11         end
12         if Match == 1 then
13             P(i) = P(i) + 1;
14             Match = 0;
15         end
16     end
17     PEU = (sum_{i=1}^{k} P(i)·N(i)) / (A.H·A.W) × 100%;
18     if PEU>MaxU then
19         MaxU = PEU;
20         Array = A;
21     end
22 end
```

Line 17: $PEU = \frac{\sum_{i=1}^{k} P(i) \cdot N(i)}{A.H \cdot A.W} \times 100\%$;

Algorithm 4: Optimized random placement (ORP)

Input: T_0, T_{end}, L, q, $Array$, $Kernels$

Output: $Array$, $MaxU$

```
1  T = log_q(T_end/T_0);
2  MaxU = 0;
3  A = RandomSolution(Array, Kernels);
4  for t = 1 to T do
5  |   for k = 1 to L do
6  |   |   NS = NewSolution(A);
7  |   |   NewA = Metropolis(A, NS);
8  |   |   if PEU(NewA)>MaxU then
9  |   |   |   MaxU = PEU(NewA);
10 |   |   |   Array = A;
11 |   |   end
12 |   end
13 end
```

Algorithm 5: Greedy-based placement (GDP)

Input: $Array$, $Kernels$

Output: $Array$, $MaxU$

```
1  A = Initial(Array);
2  for i = 1 to A.H do
3  |   for j = 1 to A.W do
4  |   |   for k = MAX(Kernels) to MIN(Kernels) do
5  |   |   |   if A(i,j) == 0 then
6  |   |   |   |   Match = Allocate(k, A, i, j);
7  |   |   |   end
8  |   |   end
9  |   end
10 end
11 MaxU = PEU(A); Array = A;
```

5 Evaluation

5.1 Experimental Setup

In this section, we evaluate the approximate calculation results of several functions based on BNN regression and the FPGA implementation results of the MuGRA system. To prove the universality of the proposed method, we have investigated multiple kinds of arithmetic functions, including one-variable functions and two-variable functions, and demonstrated their theoretical calculation and fixed-point representation results respectively. All the parameters of the BNN model are generated in Python, using the open source NN framework based on Pytorch. We manually generate equal interval discrete sampling points according to the target function as the training, validation and testing dataset of BNN. For example, assuming that the function $f(x)$ is approximated on the range $[M, N]$, we sample k data on $[M, N]$, then input and output set can be defined as $X = \{M, M+1(N-M)/k, M+2(N-M)/k, ..., N\}$ and $Y = f(X)$, respectively. For $f(x)$, we generate 1000, 256 and 384 points on input range as the training/validation/testing set, where the batch size is 1000. For $f(x, y)$, we generate 100*100, 45*45 and 55*55 points on input range as the training/validation/testing set, where the batch size is 10000. We set the maximum epoch = 50000, loss function is L1loss, and optimizer is Adam. We save the parameters with the minimum loss in all epochs as results.

The proposed hardware accelerator architecture is written using RTL-level V erilog HDL, and is synthesized and implemented on the Xilinx Zynq UltraScale+ ZCU102 Evaluation Board using Vivado (v2018.3). The ZCU102 board contains a quad-core ARM Cortex-A53 and XCZU9EG-2FFVB1156 FPGA devices, which can be used to verify our proposed CPU+FPGA heterogeneous architecture. Then, we verified the resource consumption and acceleration performance of

Table 5.1: Calculation performance test for one-variable functions

Function	Input range	Topology	MAE	MRE
$sin(x)$	$x \in [0, \frac{\pi}{4}]$	1-2-3-2-1	0.0006	0.16%
		1-2-3-4-3-2-1	0.0006	0.16%
$tanh(x)$	$x \in [0, 1]$	1-2-3-2-1	0.0011	0.25%
		1-2-3-4-3-2-1	0.0007	0.15%
2^x	$x \in [0, 1]$	1-2-3-2-1	0.0024	0.17%
		1-2-3-4-3-2-1	0.0013	0.09%
$log_2(1 + x)$	$x \in [0, 1]$	1-2-3-2-1	0.0010	0.19%
		1-2-3-4-3-2-1	0.0008	0.14%

the accelerator under different configurations through synthesize results and real hardware test. Finally, we compared our method with other methods in terms of memory size, latency, and resource consumption to prove the superiority of our method.

5.2 Results by Software

Table 5.1 and 5.2 show the theoretical calculation results of 4 one-variable functions and 3 two-variable functions based on BNN regression, respectively. The theoretical calculation results are based on software simulation on Python to verify the approximate performance of the BNN. We have chosen four representative one-variable functions $sin(x)$, $tanh(x)$, 2^x and $log_2(1+x)$, the input range of $sin(x)$ is [0,$\frac{\pi}{4}$], and the rest of the functions is [0,1]. Even if the input range of the verified elementary function is limited, we can also calculate any input by scaling, with simple addition and multiplication. We calculate the regression error by mean absolute error (MAE) and mean relative error (MRE), which are

Table 5.2: Calculation performance test for two-variable functions

Function	**Input range**	**Topology**	**MAE**	**MRE**
$\sqrt{x^2+y^2}$	$x, y \in [0,1]$	2-3-2-1	0.0085	1.10%
		2-3-4-3-2-1	0.0035	0.45%
$\sqrt[3]{x^3+y^3}$	$x, y \in [0,1]$	2-3-2-1	0.0143	2.00%
		2-3-4-3-2-1	0.0057	0.79%
$e^x sin(\pi y)$	$x, y \in [0,1]$	2-3-2-1	0.0965	8.90%
		2-3-4-3-2-1	0.0410	3.78%

defined as:

$$MAE(X,h) = \frac{1}{k}\sum_{i=1}^{k}|h(x_i) - y_i| \tag{5.1}$$

$$MRE(X,h) = \frac{\sum_{i=1}^{k}|h(x_i) - y_i|}{\sum_{i=1}^{k}|y_i|} \times 100\% \tag{5.2}$$

where $h(x_i)$ is a regression result, y_i is a real value and k is the number of test data.

For one-variable functions, we tested MAE and MRE of two different topologies, the length of which indicates the depth of network layers, and the number indicates the number of neurons in each layer. For example, the topology 1-2-3-2-1 is a 5-layer network with 9 neurons, while the topology 1-2-3-4-3-2-1 is a 7-layer network with 16 neurons. Compared with the 5-layer network, the number of neurons in the 7-layer network has increased by 77.8% for one-variable functions. As can be seen from Table 1, for $sin(x)$, compared with the 5-layer network, the performance of the 7-layer network is not significantly improved. Since $sin(x)$ is highly linear in $[0, \frac{\pi}{4}]$, a 5-layer network is sufficient to approximate the exact result of the function. For more complex functions, such as $tanh(x)$, 2^x and $log_2(1+x)$, compared with the 5-layer network, the MRE of the 7-layer network

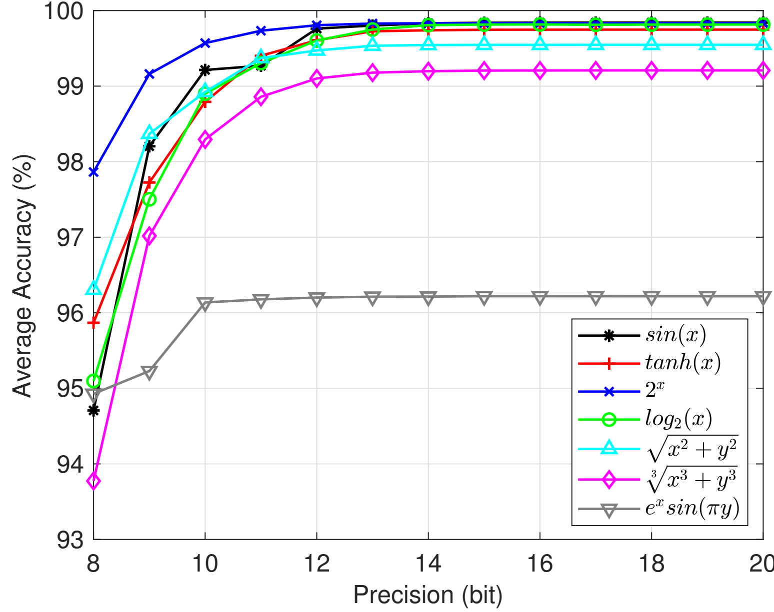


Figure 5.1: Average accuracy with different precision.

has decreased by 40.0%, 47.1% and 26.3%, respectively. For the two-variable functions $\sqrt{x^2+y^2}$, $\sqrt[3]{x^3+y^3}$ and $e^x sin(\pi y)$, we also tested the theoretical calculation results of the 4-layer network and the 6-layer network with input range [0,1]. Compared with the 4-layer network, the number of neurons in the 6-layer network has increased by 87.5% for two-variable functions. It can be seen from Table 2 that compared with the 4-layer network, the MRE of the 6-layer network has decreased by 59.1%, 60.5% and 57.5%, respectively.

5.3 Results of Fixed-Point Hardware Simulation

To verify the approximate performance of the hardware accelerator under various bit-width configurations, we tested the average accuracy of different fixed-point number formats. Input/output data, synaptic weights and biases of neuron are all represented in a set fixed-point format to approximate the results of software. Figure 5.1 shows the average accuracy of each function when the fixed-point precision (i.e. N_w) is increased from 8bit to 20bit. The results of the one-variable

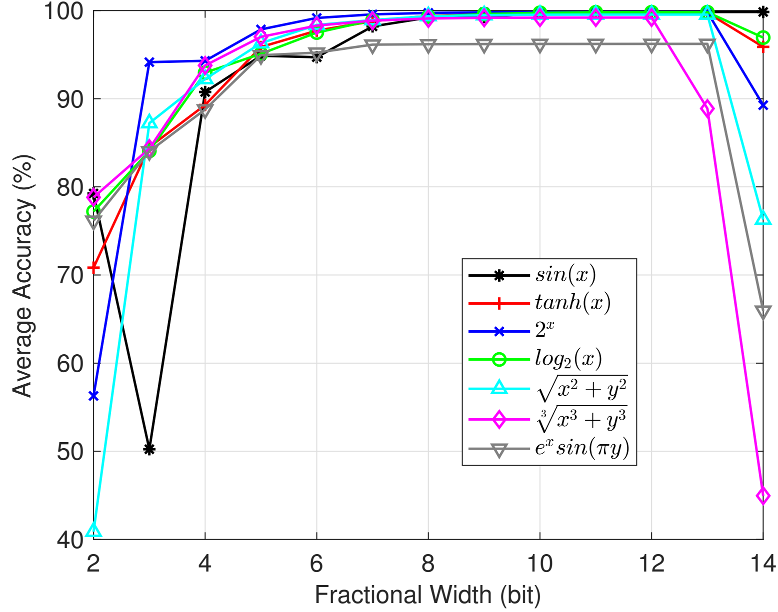


Figure 5.2: Average accuracy with different fractional width by 16bit precision.

function is based on a 5-layer network, and the two-variable function is calculated based on a 6-layer network. In the experiment in Fig. 5.1, we set the fractional width under each precision to $N_w - 3$, which ensures that the integer bits of the intermediate data and calculation results will not overflow during the calculation process. It can be seen from Fig. 5.1 that the average accuracy of the functions increases with the precision increases. For most objective functions, when the precision is greater than 14bit, the average accuracy rate reaches the theoretical value and no longer increases. After repeated function verification, we think it is reasonable to set the precision to 16bit, because this value ensures that the average accuracy of the results will not be lower than the theoretical value, and it occupies exactly two bytes. In addition, Figure 5.2 shows the average accuracy of each function when the fractional width is changed under 16bit precision. It can be seen from Fig. 5.2 that when the fractional width is below 4bit, the average accuracy of the function is very unstable, especially the average accuracy of $sin(x)$ oscillates around 3bit. The reason is that the calculation result deviates from the exact value, because of the low bit-width representation of the fractional bits.

Table 5.3: Implementation results of various topologies on FPGA

Topology	**Pipeline**	**Synthesis**	**LUT**	**FF**	**DSP**	**Power**	**Max Freq.**
1-2-3-2-1	4-stage	LUT-only	1810	128	0	52mW	238.1MHz
		DSP-enable	345	64	12	30mW	265.9MHz
1-2-3-4-3-2-1	6-stage	LUT-only	3958	240	0	104mW	217.4MHz
		DSP-enable	770	144	26	49mW	252.5MHz
2-3-2-1	3-stage	LUT-only	1002	80	0	28mW	243.9MHz
		DSP-enable	201	48	7	18mW	294.1MHz
2-3-4-3-2-1	5-stage	LUT-only	3030	192	0	74mW	229.4MHz
		DSP-enable	666	64	25	37mW	259.1MHz

With the improvement of the fractional bits representation, the average accuracy tends to stabilize and reach the theoretical value between 8bit and 12bit. We continue to increase the fractional width to 14bit, then the integer bit width is 1bit. Even if the output results of the objective function only need an integer width of 1bit, the parameters and intermediate data in the calculation process may need an integer width of at least 2bit. The overflow of integer bits causes a sharp drop in the average accuracy when the 14bit fractional width is used.

5.4 Implementation Results on FPGA

On the basis of the hardware architecture proposed in Chapter 4, we evaluated the results of the FPGA-based implementation. All results are derived from the report of Vivado after implementation. Table 5.3 shows the hardware resource consumption, power and maximum frequency of the four different topologies, with pure LUT and DSP-enable synthesis respectively. For a d-layer network, the pipeline stage is $d-1$, since the input layer does not participate in the operation.

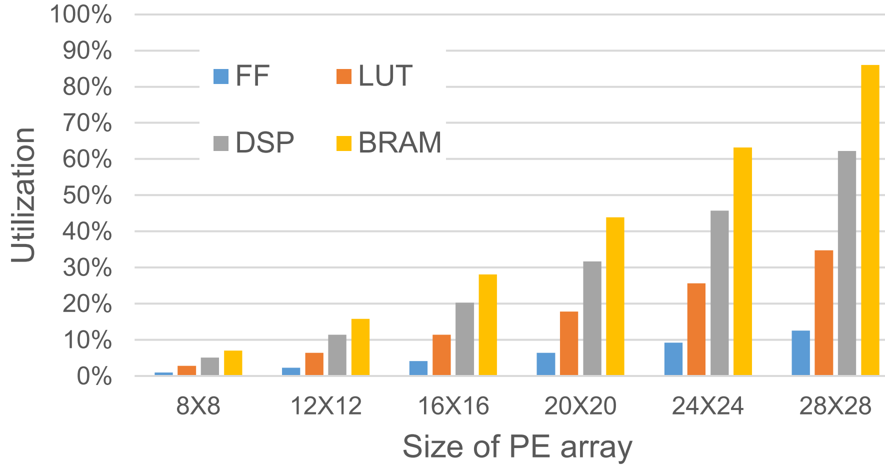


Figure 5.3: Resource utilization with different scale of PE arrays.

It can be seen that for the same topology, using DSP can achieve higher frequency and lower power than pure LUT synthesis; for different topologies with same synthesis, hardware consumption increases with the increase of network depth.

In addition, we evaluated the system-level resource consumption of the accelerator by modifying the size of the PE array. Figure 5.3 shows the resource utilization with different scale of PE arrays. Among them, BRAM is used to build on-chip buffer, each individual BRAM module is configured with $16 * 2304$ bits, and each PE takes up exactly one BRAM36K slice. It can be seen from Fig. 5.3 that the consumption of hardware resources increases as the size of the array increases. BRAM resources limit the maximum array size on ZCU102 to 28x28. This bottleneck can be alleviated by using the on-chip interconnection proposed in Chapter 4.

Finally, we compare the speedup of the time division multiplexing architecture (TDMA) and the spatially expanded in parallel architecture (SEPA). Using the CPU baseline on ZCU102, we calculated 10 basic functions provided by the glib library, and each function used 10^6 samples to calculate the total calculation time. The TDM architecture uses one layer of neurons to perform operations multiple times, and the intermediate data and parameters of each operation need

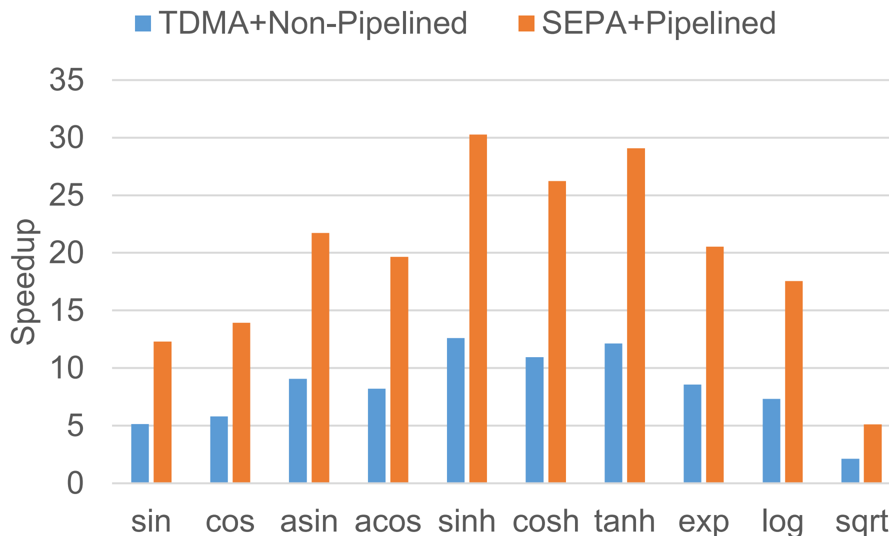


Figure 5.4: Speedup of TDMA and SEPA compared with CPU baseline.

to be repeatedly loaded from outside the PE. The SEP architecture is constructed using a neuromorphic approach of multilayer neurons whose parameters are stored inside the PE. Through fully pipelining, each layer processes different samples, which greatly speeds up the calculation. As can be seen from Fig. 5.4, compared with the CPU baseline, the TDM architecture achieves a speedup of 2.1x-12.1x, while the SEP architecture achieves a speedup of 5.1x-30.3x.

5.5 Comparison with Other Works

Many function generation methods require storage of parameters, such as function values or polynomial coefficients. The scale of the parameters will affect the circuit area, especially the overhead of LUTs. For one-variable functions, Table 5.4 shows the comparison between the proposed method and the existing three typical methods in terms of memory size. For a fair comparison, we implemented the same functions as the reference for the same input range. For the $sin(x)$ function, we evaluate the memory size with a 5-layer network, and evaluate the $ln(1+x)$ function with a 7-layer network. For the proposed method, the parameters refer to the synaptic weights and biases directly involved in the calculation.

Table 5.4: Comparison of memory size for one-variable functions

Method	Mult.	Function	Width	Input	Memory
Table-based [99]	No	$sin(x)$	16bit	$x \in [0, \frac{\pi}{4}]$	6272 bits
		2^x	16bit	$x \in [0, 1]$	6656 bits
Polynomial [100]	Yes	$ln(1+x)$	18bit	$x \in [0, 1)$	1664 bits
		$sin(x)$	24bit	$x \in [0, \frac{\pi}{4}]$	4480 bits
Interpolation [100]	Yes	$ln(1+x)$	18bit	$x \in [0, 1)$	1462 bits
		$sin(x)$	24bit	$x \in [0, \frac{\pi}{4}]$	3773 bits
Proposed	Yes	$sin(x)$	16bit	$x \in [0, \frac{\pi}{4}]$	384 bits
		$ln(1+x)$	16bit	$x \in [0, 1)$	720 bits

Table 5.5: Comparison of memory size for two-variable functions

Method	Mult.	Function	Width	Input	Memory
Polynomial [101]	Yes	$\sqrt{x^2+y^2}$	8bit	$x, y \in [0, 1)$	9334 bits
		$\sqrt[3]{x^3+y^3}$	8bit	$x, y \in [0, 1)$	9658 bits
Interpolation [102]	Yes	$\frac{x}{x^2+y^2}$	8bit	$x, y \in [0, 1)$	4708 bits
		$e^x sin(\pi y)$	8bit	$x, y \in [0, 1)$	5200 bits
Proposed	Yes	$\sqrt{x^2+y^2}$	16bit	$x, y \in [0, 1)$	288 bits
		$e^x sin(\pi y)$	16bit	$x, y \in [0, 1)$	624 bits

In [99], a table-based function generator without multipliers is realized by using a table addition method based on hierarchical multipartite (HMP). Although this

method does not require expensive multiplication, it has a huge memory overhead and increases exponentially as the function range increases. [100] evaluated two typical multiplication-driven function generation methods: polynomial-based and interpolation-based methods. The difference between the two methods is that the polynomial method uses stored coefficients to calculate the function value, while the interpolation method uses the stored function value to calculate the coefficient on-the-fly. Polynomial methods usually use degree-1 or degree-2 for piecewise approximation, which leads to large-scale segmentation and coefficient storage overhead. Compared with the polynomial method, the interpolation-based method requires less memory, but at the cost of additional calculations. Obviously, compared with other methods, the proposed method requires less memory because it does not use large-scale segmentation of the functions to achieve sufficient accuracy. Even if the application requires higher accuracy results, our method can be implemented with fewer segments. Table 5.5 shows the memory size comparison between the proposed method and the two existing two-variable function generation methods. We implemented the $\sqrt{x^2+y^2}$ function using a 4-layer network and the $e^x sin(\pi y)$ function using a 6-layer network. [101] Solved the calculation problem of two-variable function based on polynomial method. The polynomial method of the two-variable function must use a specific plane segmentation method to achieve. If the curvature of the two-variable function is large, a large number of plane divisions are needed to approximate the original surface, which leads to a large segmented storage overhead. The method in [102] regards complex functions as two-variable real functions, and uses two-dimensional interpolation to solve the calculation problem of two-variable functions. Similar to polynomial methods, interpolation-based methods usually only use low-degree interpolation, which results in segmentation overhead.

In addition, we compared the hardware implementation results against existing works. For fairness, the related works retrieved are based on Xilinx FPGA platform design. By evaluating LUT, frequency and latency, we can compare the resource consumption and performance of each work in general. In our work, the BNN is implemented in a pipelined manner, and its critical path is the internal operations of a neuron. Therefore, the minimum area and delay of a neuron are $A_{min} = 2A_{mul} + 2A_{add} + 3A_{shift} + A_{mux} + A_{reg}$ and $T_{min} =$

$T_{mul}+2T_{add}+2T_{shift}+T_{mux}+T_{reg}$, respectively. For a d-layer network, the number of pipeline stages and clock cycle are both $d-1$. Considering that each work uses a different frequency, we use normalized latency as a measurement of processing time, defined as $latency = delay * cycles$, where $delay = 1/frequency$, expressed in nanoseconds. In order to compare the core area (number of LUT) fairly, we have synthesized various BNN topologies using pure LUTs. Since [106] uses 4 DSPs, we use 1DSP=196LUT to convert to an equivalent LUT, refer to [105]. Considering that [104] and [105] use larger bit widths, we use the ALP/bit as a metric, where ALP denotes area-latency product, to make comparisons fairly between various widths, since the hardware resource increases along the bit-length. Table 5.6 and 5.7 show the FPGA implementation results of several one- and two-variable function generators, respectively. In general, increasing the depth/width of BNN helps to improve retrieve quality; but in most cases, the minimum topology is sufficient for error tolerant applications. Since the topology is optional, we show the implementation results of the four typical topologies, where 5 and 7-layer networks are used for one-variable functions, 4 and 6-layer networks are for two-variable functions. The related works in the tables report implementation results of particular functions, while our method is general for arbitrary functions. Compared with the CORDIC method for $sin(x)$ in [103], the 5-layer network consumes more area and lower latency, while reducing the ALP/bit by 42.4%. Compared with the polynomial method for $cos(x)$ in [104], the 5-layer network consumes less area and lower latency, while reducing the ALP/bit by 69.0%. Compared with the polynomial method for $log_2(1+x)$ in [105], the 5-layer network consumes more area and lower latency, while reducing the ALP/bit by 22.1%. Compared with the polynomial method for $sigmoid(x)$ and $tanh(x)$ in [106], the 5-layer network consumes more area and lower latency, while reducing the ALP/bit by 81.6% and 74.8%, respectively. As the 7-layer network achieves a larger area, this leads to a disadvantage in ALP/bit. However, it still achieves a lower latency than other works, and reduces the ALP/bit than two functions in [106] by 33.9% and 9.5%, respectively. For two-variable functions, due to less work based on Xilinx FPGA platform, we can merely retrieve recent works [107] and [108], which are based on interpolation methods, as shown in Table 5.7. Compared with [107] and [108], the ALP/bit of the 4-layer network

Figure 5.5: Approximate results of image segmentation.

and the 6-layer network are reduced by 20.4% and 63.4%, respectively. Since FPGA-based designs rarely report their power consumption, but as a reference, we still compare the power consumption with the two works [103] and [109] in Table 5.8. Generally, DSP-based implementations produce lower power and higher frequencies than LUT-based implementations. Compared with the PFR in [103], the LUT-based in our work is higher, but the DSP-based is lower. Compared with the PFR in [109], both LUT-based and DSP-based are lower in our work. Although the above related works can approximate the function with the best accuracy, our work is a general method for any function, and it is flexible in both software and hardware. The comparison against related works proves that the proposed accelerator can approximate various functions efficiently.

5.6 A Case Study on Fault-Tolerant Application

Approximate computing has potential in fault-tolerant applications such as video and image processing. In this section, we take the K-means-based image segmentation task as a case to illustrate the real application of the proposed approximate computing method. The K-means algorithm is an unsupervised clustering algorithm that treats similar samples as one class by calculating the similarity between samples. For color images, each pixel has three color values $(r, g, b) \in [0, 255]$. The K-means algorithm can effectively divide pixels with similar color values into a same cluster, so as to achieve image segmentation. In the image segmentation

task based on the K-means algorithm, since the pixel color value is a scalar, we usually use the Euclidean distance

$$d(p_1, p_2) = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2} \tag{5.3}$$

to measure the similarity between pixels, where (r_1, g_1, b_1) and (r_2, g_2, b_2) are the color values of pixels p_1 and p_2, respectively. In order to prevent the color channel with a larger value range in the pixel from affecting other channels with a smaller value range, we use

$$\bar{p_i} = (p_i - min(p_i))/(max(p_i) - min(p_i)) \tag{5.4}$$

to normalize the color value to the range [0,1] and then calculate the Euclidean distance. Thus, BNN-based methods only need to approximate the Euclidean distance function in the input range of [0,1]. The image segmentation method based on K-means is described as follows:

(1) Input image, normalize pixel color value to [0,1] based on Eq. 5.4;

(2) Randomly extract k pixels from all pixels as the initial cluster center;

(3) Calculate the similarity between each pixel (r_i, g_i, b_i) and the cluster center (r_j, g_j, b_j) based on the Eq. 5.3, and assign each pixel to the corresponding cluster c_j according to the principle of being closest to the cluster center;

(4) Calculate the mean value of all pixels in each cluster as the new cluster center;

(5) Go back to step (3) and execute in a loop until all cluster centers no longer change.

In this algorithm, a large number of Euclidean distance calculations are performed in multiple iterations, which is the main bottleneck of computational efficiency. Meanwhile, in one iteration, the similarity between each pixel of the image can be calculated in parallel, so that the calculation efficiency can be improved. Therefore, the architecture proposed in this paper can be effectively used in this application. Figure 5.5 shows the results of the proposed approximate computing method on the K-means-based image segmentation task: the original image is Fig. 5.5 (a); the segmentation results based on exact and approximate Euclidean distance calculations are Fig. 5.5 (b) and (c), respectively. In this application, we use 3-4-3-2-1 as the BNN topology for Euclidean distance function approximation. As can be seen from Fig. 5.5, the BNN-based method achieves high-quality

approximation results on this task. The MAE and MRE of the three-variable Euclidean distance function are 0.0325 and 3.38%, respectively. The relative error of the segmented image is 8.2% by calculating the image differences.

Table 5.6: Comparison of implementation results for one-variable functions with other FPGA-based works

Function	Approach	Platform	Bit-width	LUT	FF	DSP	eq. LUT	Cycle@Freq.	Lat.(ns)	ALP/bit
$sin(x)$	CORDIC [103]	XC6SLX9	16bit(FxP)	951	94	0	951	6@108.1MHz	55.49	3298.19
$cos(x)$	Polynomial [104]	XC4VLX100	32bit(FP)	3077	/	0	3077	/	63.84	6138.62
$log_2(1+x)$	Polynomial [105]	XC6VLX240T	36bit(FxP)	1780	/	0	1780	19@385MHz	49.35	2440.08
$sigmoid(x)$	Polynomial [106]	XC7VX690T	16bit(FP)	744	1258	4	1528	68@628.5MHz	108.19	10332.15
$tanh(x)$	Polynomial [106]	XC7VX690T	16bit(FP)	508	1399	4	1292	62@663.6MHz	93.43	7544.47
Arbitrary	5-layer BNN [Our]	XCZU9EG	16bit(FxP)	1810	128	0	1810	4@238.1MHz	16.80	1900.50
Arbitrary	7-layer BNN [Our]	XCZU9EG	16bit(FxP)	3958	240	0	3958	6@217.4MHz	27.60	6827.55

a. Use 1DSP=196LUT to convert to equivalent LUT, refer to [105].
b. ALP denotes Area-Latency Product, where Area is the number of equivalent LUT.
c. FxP and FP denote fixed-point and floating-point, respectively.

Table 5.7: Comparison of implementation results for two-variable functions with other FPGA-based works

Function	**Approach**	**Platform**	**Bit-width**	**LUT**	**FF**	**DSP**	**eq. LUT**	**Cycle@Freq.**	**Lat. (ns)**	**ALP/bit**
$atan2(x, y)$	Interpolation [107]	XC7K325T	16bit(FxP)	893	81	0	893	3@173MHz	17.34	967.79
$e^x sin(\pi y)$	Interpolation [108]	XC4VLX60	16bit(FP)	3325	/	0	3325	/	54.26	11275.91
Arbitrary	4-layer BNN [Our]	XCZU9EG	16bit(FxP)	1002	80	0	1002	3@243.9MHz	12.30	770.29
Arbitrary	6-layer BNN [Our]	XCZU9EG	16bit(FxP)	3030	192	0	3030	5@229.4MHz	21.79	4126.48

Table 5.8: Comparison of power consumption with FPGA-based works

	[103]	Proposed		[109]	Proposed	
		LUT	DSP		LUT	DSP
Function	$sin(x)$			$atan2(x, y)$		
Freq. (MHz)	108.1	238.1	265.9	93.6	243.9	294.1
Power (mW)	14	52	30	24.2	28	18
PFR (mW/MHz)	0.13	0.22	0.11	0.26	0.11	0.06

[a.] PFR denotes Power-Frequency Ratio.

6 Conclusion

6.1 Summary

This thesis mainly studies the architecture design of a novel multi-grained reconfigurable accelerator and its supporting optimization technologies. Neural network-based accelerators have gradually become an important part of computing systems in the past decade. As a new type of computing system, the heterogeneous computing system of "General Purpose Processor + Neural Network Accelerator" brings new hope to conventional computing models and architectures. Offloading complex calculations on general-purpose processors to neural network accelerators for approximation can significantly improve the energy efficiency of the entire system. Although a large number of architectural solutions have been proposed by the academic community and the industry, there are still various problems in flexibility, parallelism, and cost. Obviously, an ideal architecture is expected to win all the aspects of parallelism, flexibility, and cost. Therefore, this article attempts to start from the above three points to discuss the realization of this ideal architecture.

In this thesis, we proposed a multi-grained reconfigurable architecture powered by BNN for approximate computing. By implementing a large scale of bisection mesh of NN on hardware, the entire network-on-chip (NoC) is expected to be partitioned and configured into pieces for any specific applications (almost) without redundancy. By configuring BNN into massive pieces on-chip, where each small piece performs as a computing kernel through NN regressions, arbitrary acceleration architecture can be achieved with flexible features. In this manner, the computing kernels (fine-grained), their performance-scale-cost features (mid-grained), and organization scheme are re-configurable. In this work, the FPGA-based solution of MuGRA is offered along with a special on-chip memory

organization to optimize PE communication throughput and on-chip Block-RAM (BRAM) overhead. For proof-of-concept, a demo accelerator is built on FPGA in the scale up to 28×28 process elements (PEs) which is capable for accommodating more than a hundred computing kernels. The proposed MuGRA system is implemented on the Xilinx Zynq UltraScale+ ZCU102 Evaluation Board. To prove the universality of the proposed method, we have investigated multiple kinds of arithmetic functions, including one- and two-variable functions, and demonstrated their theoretical and hardware calculation results by software simulation and FPGA test, respectively. The test results of several functions show that for one-variable functions, the topology with the minimum hardware resource achieve an accuracy of over 99.75%; for two-variable functions, the topology with the minimum hardware resource achieve an accuracy of over 91.1%. By testing the average accuracy of different fixed-point implementation, we found that 16-bit is the ideal precision to design PEs that approximates the software results. From the FPGA implementation results, compared with CPU baseline, proposed architecture achieves a speedup of 5.1x to 30.3x. Compared with other traditional function approximation methods, our method provides fewer parameter storage requirements. Compared with various one- and two-variable function generators based on FPGA implementation, the area-latency product of our accelerator has been reduced at least 9.5% with a loss of accuracy by at most 8.9%.

6.2 Future Works

In future work, the following points are worth exploring:

1. For more input applications, the proposed architecture loses its advantages. Since the current architecture can only input data in the first layer of the neural network, the increase in dimensionality will greatly increase the depth of the topology, resulting in the inability to train an effective neural network model. Therefore, it is inevitable to explore the next generation of reconfigurable neural network topology that supports more inputs.

2. Explore the in-memory computing architecture. With the network scale increases, a large amount of data and weights can only be stored in off-chip memory. The separation of storage and computing has led to an increase in the

energy consumption of data movement and has become the main bottleneck of the system. In the in-memory computing architecture, the computing units and the memory units are coupled together, with greatly reduced on the cost of memory access.

3. Computing architecture based on new devices. With the end of Moore's Law, CMOS-based devices have been difficult to upgrade. As a promising alternative, AQFP-based superconducting quantum computing technology can be used as the infrastructure of data centers and supercomputing centers. Compared with CMOS, AQFP only needs to be powered by alternating current, and the static power consumption is close to zero, which further reduces the current required by the switching device.

Acknowledgements

Upon the completion of this doctoral thesis, I would like to express my utmost gratitude to important people.

I would like to express my deepest appreciation to Prof. Yasuhiko Nakashima, who is my supervisor, for his guidance and encouragement throughout my research period. I would like to extend my deepest gratitude to Associate Prof. Renyuan Zhang, who is also my supervisor, for his guidance and encouragement during the whole period of my Ph.D course in NAIST. I have learned a lot of important things from him including academic thinking, technical writing, how to proceed with a research project, and education. Their kindness and academic ability encourage me not only in my study but also in my future work. I would also like to thank all the member of Computing Architecture Laboratory.

I would like to thank my parents and family members for their kind support.

This research was partly supported by JST, PRESTO Grant Number JP-MJPR18M7, Japan.

References

[1] J. Lee, S. Kang, J. Lee, D. Shin, D. Han, and H.-J. Yoo, "The hardware and algorithm co-design for energy-efficient dnn processor on edge/mobile devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 10, pp. 3458–3470, 2020.

[2] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique, "X-cgra: An energy-efficient approximate coarse-grained reconfigurable architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2558–2571, 2019.

[3] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "Diannao family: energy-efficient hardware accelerators for machine learning," *Communications of the ACM*, vol. 59, no. 11, pp. 105–112, 2016.

[4] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[5] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 940–953.

[6] H. Kwon, L. Lai, M. Pellauer, T. Krishna, Y.-H. Chen, and V. Chandra, "Heterogeneous dataflow accelerators for multi-dnn workloads," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 71–83.

[7] S. Lian, Y. Han, X. Chen, Y. Wang, and H. Xiao, "Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment," in *2018*

55th ACM/ESDA/IEEE Design Automation Conference (DAC). IEEE, 2018, pp. 1–6.

[8] A. Ardakani, C. Condo, and W. J. Gross, "Fast and efficient convolutional accelerator for edge computing," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 138–152, 2019.

[9] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, 2012.

[10] D. Shin and H.-J. Yoo, "The heterogeneous deep neural network processor with a non-von neumann architecture," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1245–1260, 2019.

[11] S. Dutta, H. Jeong, Y. Yang, V. Cadambe, T. M. Low, and P. Grover, "Addressing unreliability in emerging devices and non-von neumann architectures using coded computing," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1219–1234, 2020.

[12] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC).* IEEE, 2018, pp. 1–6.

[13] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 370–380.

[14] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri *et al.*, "Blackparrot: An agile open-source risc-v multicore for accelerator socs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020.

[15] D. Peroni, M. Imani, H. Nejatollahi, N. Dutt, and T. Rosing, "Arga: Approximate reuse for gpgpu acceleration," in *2019 56th ACM/IEEE Design Automation Conference (DAC).* IEEE, 2019, pp. 1–6.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[17] B. Zamanlooy and M. Mirhassani, "Efficient vlsi implementation of neural networks with hyperbolic tangent activation function," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 1, pp. 39–48, 2013.

[18] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[19] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[20] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[21] C. Nicol, "A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing," *Wave Computing White Paper*, 2017.

[22] H. Sun, Y. Luo, Y. Ha, Y. Shi, Y. Gao, Q. Shen, and H. Pan, "A universal method of linear approximation with controllable error for the efficient implementation of transcendental functions," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 177–188, 2019.

[23] T. Filippov, M. Dorojevets, A. Sahu, A. Kirichenko, C. Ayala, and O. Mukhanov, "8-bit asynchronous wave-pipelined rsfq arithmetic-logic unit," *IEEE transactions on applied superconductivity*, vol. 21, no. 3, pp. 847–851, 2011.

[24] H. Dong, M. Wang, Y. Luo, M. Zheng, M. An, Y. Ha, and H. Pan, "Plac: Piecewise linear approximation computation for all nonlinear unary functions," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 2014–2027, 2020.

[25] K. K. Parhi and Y. Liu, "Computing arithmetic functions using stochastic logic by series expansion," *IEEE Transactions on Emerging Topics in Computing*, vol. 7, no. 1, pp. 44–59, 2016.

[26] S. Eldridge, F. Raudies, D. Zou, and A. Joshi, "Neural network-based accelerators for transcendental function approximation," in *Proceedings of the 24th edition of the great lakes symposium on VLSI*, 2014, pp. 169–174.

[27] P. Wijesinghe, C. M. Liyanagedera, and K. Roy, "Fast, low power evaluation of elementary functions using radial basis function networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 208–213.

[28] R. Zhang, Y. Chen, T. Nakada, and Y. Nakashima, "Dianet: An efficient multi-grained re-configurable neural network in silicon," in *2019 32nd IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2019, pp. 132–137.

[29] M. Wu, Y. Chen, Y. Kan, T. Nomura, R. Zhang, and Y. Nakashima, "An elastic neural network toward multi-grained re-configurable accelerator," in *2020 18th IEEE International New Circuits and Systems Conference (NEWCAS)*. IEEE, 2020, pp. 218–221.

[30] M. Wu, Y. Kan, T. Erlina, R. Zhang, and Y. Nakashima, "Dianet: An elastic neural network for effectively re-configurable implementation," *Neurocomputing*, vol. 464, pp. 242–251, 2021.

[31] Y. Kan, M. Wu, R. Zhang, and Y. Nakashima, "A multi-grained reconfigurable accelerator for approximate computing," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2020, pp. 90–95.

[32] Y. Kan, M. Wu, R. Zhang, and Y. Nakashima, "Mugra: A scalable multi-grained reconfigurable accelerator powered by elastic neural network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 1, pp. 258-271, 2022.

[33] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 957–960.

[34] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, "Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 2856–2868, 2018.

[35] L. Chen, J. Han, W. Liu, and F. Lombardi, "On the design of approximate restoring dividers for error-tolerant applications," *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2522–2533, 2015.

[36] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel, "Cross-layer approximate computing: From logic to architectures," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[37] S. Li, Z. Zhang, R. Mao, J. Xiao, L. Chang, and J. Zhou, "A fast and energy-efficient snn processor with adaptive clock/event-driven computation scheme and online learning," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 4, pp. 1543–1552, 2021.

[38] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[39] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[40] A. Yasoubi, R. Hojabr, and M. Modarressi, "Power-efficient accelerator design for neural networks using computation reuse," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 72–75, 2016.

[41] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Towards neural acceleration for general-purpose approximate computing," in *Proceedings of the 4th Workshop on Energy Efficient Design, ISCA*, vol. 12.

[42] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, 2012, pp. 449–460.

[43] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," *Communications of the ACM*, vol. 58, no. 1, pp. 105–115, 2014.

[44] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "Snnap: Approximate computing on programmable socs via neural acceleration," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2015, pp. 603–614.

[45] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for gpu throughput processors," in *Proceedings of the 48th international symposium on microarchitecture*, 2015, pp. 482–493.

[46] Z. Wang, S. Yin, F. Tu, L. Liu, and S. Wei, "An energy efficient jpeg encoder with neural network based approximation and near-threshold computing," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS).* IEEE, 2018, pp. 1–5.

[47] F. Tu, S. Yin, P. Ouyang, L. Liu, and S. Wei, "Reconfigurable architecture for neural approximation in multimedia computing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 29, no. 3, pp. 892–906, 2018.

[48] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 505–516, 2014.

[49] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[50] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[51] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, S. Zheng, T. Lu, J. Gu, L. Liu, and S. Wei, "A high energy efficient reconfigurable hybrid neural network processor for deep learning applications," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 4, pp. 968–982, 2017.

[52] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[53] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 16–25.

[54] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[55] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.

[56] L. Cavigelli and L. Benini, "Origami: A 803-gop/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2016.

[57] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[58] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2017, pp. 240–241.

[59] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *Advances in neural information processing systems*, vol. 28, pp. 1135–1143, 2015.

[60] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 806–814.

[61] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5687–5695.

[62] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.

[63] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[64] P. Wang, Q. Hu, Y. Zhang, C. Zhang, Y. Liu, and J. Cheng, "Two-step quantization for low-bit neural networks," in *Proceedings of the IEEE Conference on computer vision and pattern recognition*, 2018, pp. 4376–4384.

[65] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[66] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[67] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[68] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 688–698.

[69] P. Guo, H. Ma, R. Chen, P. Li, S. Xie, and D. Wang, "Fbna: A fully binarized neural network accelerator," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 51–513.

[70] Y. Cai, T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang, "Low bit-width convolutional neural network on rram," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, pp. 1414–1427, 2019.

[71] G. Estrin, "Organization of computer systems: the fixed plus variable structure computer," in *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, 1960, pp. 33–40.

[72] A. DeHon and J. Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 610–615.

[73] S. Yin, X. Yao, D. Liu, L. Liu, and S. Wei, "Memory-aware loop mapping on coarse-grained reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 5, pp. 1895–1908, 2015.

[74] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of cgra," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 9–16.

[75] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[76] G. Ansaloni, P. Bonzini, and L. Pozzi, "Egra: A coarse grained reconfigurable architectural template," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 6, pp. 1062–1074, 2010.

[77] C. M. Diniz, M. Shafique, S. Bampi, and J. Henkel, "Run-time accelerator binding for tile-based mixed-grained reconfigurable architectures," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–4.

[78] L. Liu, Z. Li, C. Yang, C. Deng, S. Yin, and S. Wei, "Hrea: An energy-efficient embedded dynamically reconfigurable fabric for 13-dwarfs processing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 3, pp. 381–385, 2017.

[79] P. O. Domingos, F. M. Silva, and H. C. Neto, "An efficient and scalable architecture for neural networks with backpropagation learning," in *International Conference on Field Programmable Logic and Applications, 2005.* IEEE, 2005, pp. 89–94.

[80] M. Pietras, "Hardware conversion of neural networks simulation models for neural processing accelerator implemented as fpga-based soc," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–4.

[81] D. Ferrer, R. González, R. Fleitas, J. P. Acle, and R. Canetti, "Neurofpga-implementing artificial neural networks on programmable logic devices," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 3. IEEE, 2004, pp. 218–223.

[82] C. Latino, M. A. Moreno-Armendariz, and M. Hagan, "Realizing general mlp networks with minimal fpga resources," in *2009 International Joint Conference on Neural Networks*. IEEE, 2009, pp. 1722–1729.

[83] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 267–278.

[84] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G.-Y. Wei, "14.3 a 28nm soc with a 1.2 ghz 568nj/prediction sparse deep-neural-network engine with> 0.1 timing error rate tolerance for iot applications," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2017, pp. 242–243.

[85] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 356–367.

[86] D. Yarotsky, "Error bounds for approximations with deep relu networks," *Neural Networks*, vol. 94, pp. 103–114, 2017.

[87] A. L. Maas, A. Y. Hannun, A. Y. Ng *et al.*, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1. Citeseer, 2013, p. 3.

[88] G. Yang, J. Yang, Z. Lu, and D. Liu, "A convolutional neural network with sparse representation," *Knowledge-Based Systems*, vol. 209, p. 106419, 2020.

[89] J. Yang and J. Ma, "Feed-forward neural network training using sparse representation," *Expert Systems with Applications*, vol. 116, pp. 255–264, 2019.

[90] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu *et al.*, "Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 3, pp. 644–656, 2018.

[91] H.-J. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 7, pp. 2093–2103, 2019.

[92] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[93] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *International Conference on Machine Learning.* PMLR, 2018, pp. 4095–4104.

[94] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 19–34.

[95] S. Dey, K.-W. Huang, P. A. Beerel, and K. M. Chugg, "Pre-defined sparse neural networks with hardware acceleration," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 332–345, 2019.

[96] S. Kundu, M. Nazemi, M. Pedram, K. M. Chugg, and P. A. Beerel, "Pre-defined sparsity for low-complexity convolutional neural networks," *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 1045–1058, 2020.

[97] A. Tretter, G. Giannopoulou, M. Baer, and L. Thiele, "Minimising access conflicts on shared multi-bank memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–20, 2017.

[98] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[99] S.-F. Hsiao, C.-S. Wen, Y.-H. Chen, and K.-C. Huang, "Hierarchical multipartite function evaluation," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 89–99, 2016.

[100] D.-U. Lee, R. Cheung, W. Luk, and J. Villasenor, "Hardware implementation trade-offs of polynomial approximations and interpolations," *IEEE Transactions on computers*, vol. 57, no. 5, pp. 686–701, 2008.

[101] S. Nagayama, T. Sasao, and J. T. Butler, "Programmable architectures and design methods for two-variable numeric function generators," *IPSJ Transactions on System LSI Design Methodology*, vol. 3, pp. 118–129, 2010.

[102] D. Wang, M. D. Ercegovac, and Y. Xiao, "Complex function approximation using two-dimensional interpolation," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 2948–2960, 2013.

[103] F. Salehi, E. Farshidi, and H. Kaabi, "Novel design for a low-latency cordic algorithm for sine-cosine computation and its implementation on fpga," *Microprocessors and Microsystems*, vol. 77, p. 103197, 2020.

[104] K. Long, H. Chen, and X. Li, "Analysis and optimization for hardware implementation of sine/cosine with faithful rounding and monotonicity through piecewise quadratic polynomial," *IEICE Electronics Express*, pp. 18–20 210 158, 2021.

[105] S. Xu, S. A. Fahmy, and I. V. McLoughlin, "Square-rich fixed point polynomial evaluation on fpgas," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 99–108.

[106] S. M. Ho and H. K.-H. So, "Nncore: A parameterized non-linear function generator for machine learning applications in fpgas," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 160–167.

[107] V. Torres and J. Valls, "A fast and low-complexity operator for the computation of the arctangent of a complex number," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 9, pp. 2663–2667, 2017.

[108] A. Hosseiny and G. Jaberipur, "Complex exponential functions: A high-precision hardware realization," *Integration*, vol. 73, pp. 18–29, 2020.

[109] R. Gutierrez and J. Valls, "Low-power fpga-implementation of atan (y/x) using look-up table methods for communication applications," *Journal of Signal Processing Systems*, vol. 56, no. 1, pp. 25–33, 2009.

Publication List

Journal Papers

[1] Y. Kan, M. Wu, R. Zhang, and Y. Nakashima, "MuGRA: A Scalable Multi-Grained Reconfigurable Accelerator Powered by Elastic Neural Network," IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 69, no. 1, pp. 258-271, Jan. (2022)

Corresponds to Chapter 4 and 5

[2] M. Wu, Y. Kan, T. Erlina, R. Zhang, and Y. Nakashima, "DiaNet: An Elastic Neural Network for Effectively Re-Configurable Implementation," Neurocomputing 464, pp. 242-251, Aug. (2021)

Conference Papers

[1] Y. Kan, M. Wu, R. Zhang, and Y. Nakashima, "A Multi-Grained Reconfigurable Accelerator for Approximate Computing," IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 90-95, Aug. (2020)

Corresponds to Chapter 3 and 5

[2] M. Wu, Y. Chen, Y. Kan, T. Nomura, R. Zhang, and Y. Nakashima, "An Elastic Neural Network toward Multi-Grained Re-configurable Accelerator," 18th IEEE International New Circuits and Systems Conference (NEWCAS), pp. 218-221, Jun. (2020)