Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
https://cas.tudelft.nl/

CAS-2022-5352177

M.Sc. Thesis

Mapping of Spiking Neural Network Architecture using VPR

Jinyun Long B.Sc.

Abstract

As the new generation of neural networks, Spiking Neural Network architectures executes on specialized Neuromorphic devices. The mapping of Spiking Neural Network architectures affects the power consumption and performance of the system. The target platform of the thesis is a hardware platform with Neuromorphic Arrays with columns for neural signal processing.

The explorations for the mapping methods are based on VPR, an open-source academic CAD tool for FPGA architecture exploration. The packing of VPR is used for mapping neurons to Neuromorphic Arrays. VPR includes two levels of mapping: pins and neurons.

An evaluation of the mapping methods is established. Based on the evaluation, the optimized mapping solution is generated. Modifications are made in VPR to adapt to SNN architectures. An Activity-Criticality input file is added to the VPR flow for the optimized mapping solution.

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Mapping of Spiking Neural Network Architecture using VPR

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jinyun Long B.Sc.
born in Chengdu, China

This work was performed in:

Circuits and Systems Group
Department of Microelectronics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology
Department of
Microelectronics

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Mapping of Spiking Neural Network Architecture using VPR"** by **Jinyun Long B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: November 28, 2022

Chairman:

prof.dr.ir. T.G.R.M. van Leuken

Advisor:

dr. N.K. Mandloi

Committee Members:

prof.dr. M. Taouil

Abstract

As the new generation of neural networks, Spiking Neural Network architectures executes on specialized Neuromorphic devices. The mapping of Spiking Neural Network architectures affects the power consumption and performance of the system. The target platform of the thesis is a hardware platform with Neuromorphic Arrays with columns for neural signal processing.

The explorations for the mapping methods are based on VPR, an open-source academic CAD tool for FPGA architecture exploration. The packing of VPR is used for mapping neurons to Neuromorphic Arrays. VPR includes two levels of mapping: pins and neurons.

An evaluation of the mapping methods is established. Based on the evaluation, the optimized mapping solution is generated. Modifications are made in VPR to adapt to SNN architectures. An Activity-Criticality input file is added to the VPR flow for the optimized mapping solution.

Acknowledgments

I would like to thank all my advisors, Prof. Rene Van Leuken, Ir. Alexander de Graaf, Dr. Neeraj K. Mandloi, and Dr. David Aledo, for all the feedback and suggestions in every weekly meeting on Tuesday. I am especially grateful to Prof. Rene van Leuken for providing me with the directions for the thesis. I was a little lost at first, but after weekly meetings, I received helpful suggestions and gradually found the right path.

My colleague master students, Jinyao Zhang and Longxing Jiang have attended weekly meetings on Tuesdays with me. Thank you for your companion in the thesis journey.

Finally, I would like to thank my beloved parents, Hua Long and Yurong Jiang, for their love and support. Without your support, my master's studies in TUDelft would not have been possible.

Jinyun Long B.Sc.
Delft, The Netherlands
November 28, 2022

Glossary

ANN Artificial Neural Network.

BLE Basic Logic Element in a CLB.

BLIF Berkeley Logic Interchange Format, the netlist description file format.

CAD Computer-Aided Design.

CLB Configurable Logic Block, the basic repeating logic resource on the hardware in VPR.

FF Flip-Flop, a circuit in BLE for FPGAs.

FPGA Field-Programmable Gate Array.

I/O Input/Output.

LIF Leaky Integrate-and-Fire, a neuron model in SNN.

LUT LookUp Table, a circuit in BLE for FPGAs.

molecule A cluster of primitive netlist blocks in VPR. In SNN mapping, one neuron is a molecule.

MUX Multiplexer, a circuit in BLE for FPGAs.

NA Neuromorphic Array, the basic repeating resource to which neurons are mapped.

NoC Network-on-Chip.

SNN Spiking Neural Network.

VPR Versatile Place and Route.

VTR Verilog-to-Routing.

XML Extensible Markup Language, the hardware architecture description file format.

Contents

List of Figures

Introduction 1

The spike-based computation model, Spiking Neural Networks (SNN), is a popular trend in new generations of machine learning. It builds models to mimic the neurons and synapses in the functioning of the brain. When executing SNN on neuromorphic processors[2][3][4] , there is a potential for reducing energy consumption, which makes it suitable for future hardware implementation for edge computing. To run Spiking Neural Networks on hardware devices, the neurons and synapses should be mapped to specific hardware resources for computation. Neuromorphic hardware processors, such as Loihi[2], TrueNorth[3], and DYNAP-SE[4], have been designed to accelerate Spiking Neural Network models. These many-core processors use crossbar-based Networks-on-Chip (NoC) for the computation. There are some algorithms for finding the best solutions for compiling and mapping SNN to neuromorphic hardware[5][6]. However, the SNN in the thesis will be mapped onto a self-defined digital and analog programmable circuit with Neuromorphic Arrays (NAs). Therefore, VPR (Versatile Place and Route), the open-source CAD tool for mapping, placement, and routing for array-based FPGA architecture, is a possible solution to the problem.

VPR is a flexible tool designed for different FPGA architectures[1]. To use the VPR design flow, a description of the FPGA architecture and the user circuit should be provided. The description of the user circuit is a logic-level netlist in BLIF (Berkely Logic Interchange Format). The FPGA architecture description file is an XML[7] file with all logic blocks and configurations listed. To use the tool, We need to build the architecture file to express logic components. In addition, the netlist of the SNN model should be described. In this thesis, the use of VPR version 8 in the Verilog-to-Routing (VTR) project will be analyzed, and some modifications will be made based on it.

1.1 Research Statement

The starting point of the thesis is to map Spiking Neural Networks to a hardware implementation. Though some existing mapping approaches exist for Spiking Neural Networks, they use multi-core neuromorphic hardware based on NoC for large-scale Spiking Neural Network models. In this thesis, an SNN mapping solution for analog and digital programmable circuits with Neuromorphic Arrays is implemented. This thesis work mainly focuses on the following challenges:

- The existing solutions for mapping Spiking Neural Networks are based on the simulation of large-scale Spiking Neural Networks. However, for a relatively small network, the CAD tool, VPR, is a possible solution.
- VPR is designed for packing, placement, and routing of FPGA architectures, so modifications should be made towards the new Spiking Neural Network systems.

Based on the challenges, the thesis is trying to find answers to the following questions:

- What is the architecture of the hardware implementation that will be used for the Spiking Neural Networks?
- What are the steps of mapping the Spiking Neural Networks to the hardware?
- How to apply VPR to the SNN architecture described in the thesis?
- What modifications in VPR have been made to adapt to SNN applications?
- How do SNN mapping solutions affect the performance? Which one is the best solution? Why does it outperform the others?

1.2 Contributions

- The optimization goal of mapping Spiking Neural Networks to Neuromorphic Arrays is presented. The evaluation of mapping methods is created.
- Modifications based on VPR 8 have been made to adapt to Spiking Neural Network architectures and Neuromorphic Arrays.
- The input file generators are added to the VPR flow for the optimized mapping solutions.

1.3 Thesis Outline

This thesis consists of the following chapters:

- Chapter 2, Background, discusses the background information for the SNN mapping problem. The related concepts: Spiking Neural Network (SNN), Neuromorphic Arrays (NAs), Mapping, and VPR are included.
- Chapter 3, VPR basics, describes the flow of using VPR for SNN mapping problems.
- Chapter 4, Mapping of pins, discusses mapping input signals to Neuromorphic Array pins.
- Chapter 5, Mapping of neurons, describes the process of mapping neurons to Neuromorphic Array columns.
- Chapter 6, Evaluation of mapping spiking neurons to NAs, presents the optimization goal of mapping methods and the implementation method.
- Chapter 7, The basic flow for mapping SNN architectures on NAs, describes the use of VPR in mapping with the input files and the activity-criticality generator.

- Chapter 8, Results and discussions, shows the results of mapping SNN to NAs using some examples.
- Chapter 9, Conclusion and Future Work, is a summary and outlook of the thesis.

Background 2

2.1 Spiking Neural Networks

Spiking Neural Networks are a kind of computation model that mimics the activities of biological neurons. It is the new generation of Artificial Neural Networks (ANN).

A traditional multilayer ANN consists of artificial neurons with connections. The neurons have connections with each other. In a trained ANN architecture, neurons are distributed in layers. The first layer is called the input layer, and the last layer is called the output layer. The layers between them are hidden layers. Neurons communicate with other neurons in other layers through connections. Each connection has a weight that proceeds the input. In the training process, the weights are updated for greater output accuracy. The output is generated after the data passes through non-linear functions. In each layer, the input data is processed by neurons, so some input features are extracted. After passing through all the layers, the network recognizes the features, creating the output. The output shows which category should the input data be classified into. The classification is based on the data used for training the network.

Spiking Neural Networks consist of spiking neurons and interconnecting synapses modeled by adjustable scalar weights [8]. The architecture is similar to that of ANN. The input layer is where input spikes enter the network. The output layer is where output generates. The layers between them are called hidden layers. The input layer does not have neuron models, and the first neurons are located in the first hidden layer. The number of outputs in the SNN equals the number of neurons in the output layer. Figure 2.1 shows a multilayer SNN with one input layer with five input spike trains, one output layer with two neurons, and two hidden layers.

As simplified biological neural networks, the inputs of Spiking Neural Networks are analog signals. The inputs are encoded into spike trains that could pass through layers.

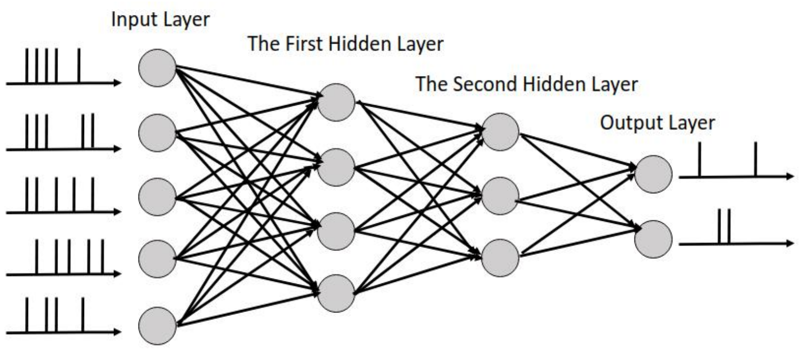


Figure 2.1: A $5 \times 4 \times 3 \times 2$ multilayer SNN architecture

The spikes are in the form of voltage changes. The spike trains pass through synaptic connections. The weights in SNN are the strength of the adaptive synapses and are updated in the learning process of SNN.

The difference between the SNN and the ANN is that the SNN processes input data in time. The spikes are sparse in time, which makes it possible to create networks with fewer spikes and higher information contents. It can reduce energy consumption, and this advantage also applies to hardware implementation[8].

For a neuron in the SNN architecture, the neurons which send spikes to it are its presynaptic neurons. The neurons which receive spikes from it are its postsynaptic neurons. Like biological neurons, each spiking neuron uses a membrane potential to decide whether it fires spikes. A spiking neuron accepts spikes from presynaptic neurons and updates the membrane potential. If the membrane potential exceeds the firing threshold, the neuron fires a spike and sends the spike to all the postsynaptic neurons. Many kinds of neuron models are created to simulate this process, and the LIF (leaky integrate-and-fire) neuron model has been implemented in many neuromorphic processors[8].

A SystemC MNIST Spiking Neural Network simulator is used as an example of architecture in the thesis. It is a trained MNIST Spiking Neural Network with weights and spike inputs. It is a $196 \times 50 \times 10$ network that classifies the input digits expressed by spikes. The spiking activities are expressed in terms of voltage changes. We use the number of spikes in the time period for signal propagation to measure the spiking activities.

2.2 Neuromorphic Arrays

Many Neuromorphic processors have been developed to implement SNN tasks. The aim is to implement an energy-efficient and high-performance spike-inspired computation platform. Loihi[2], TrueNorth[3], DYNAP-SE[4] are such Neuromorphic platforms.

The neuromorphic processors are non-von Neumann computing systems with near-memory or in-memory computing[9]. In the system, there are many neuromorphic cores to accommodate neurons. Each core can be mapped with thousands of neurons at most. For example, Loihi[2] has 128 cores, and each core can accommodate 1024 neurons. Moreover, they use Network-on-Chip for communication between different neuromorphic cores. The throughput of the interconnect largely affects the system performance.

The target hardware device of the thesis differs from the processors with many neuromorphic cores and Network-on-Chip. In the cores of the neuromorphic processors, there is a "crossbar" where neurons are placed. The crossbar's inputs are all the inputs for the neuron. The crossbar's output is at the end of a column. After signal processing in a column, a neuron's output is generated.

Our target, Neuromorphic Arrays, has similar structures to a crossbar. It has input pins where input signals should be mapped, and after signal processing, a column's output is generated. One column is used for signal processing for one neuron, so the number of columns in a Neuromorphic Array is the number of neurons that can be mapped into the NA. However, the hardware system that contains NAs is for small-

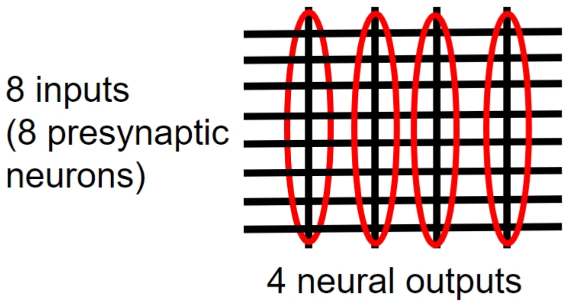


Figure 2.2: The model of a Neuromorphic Array

scale SNN tasks without a Network-on-Chip interconnect for communication. So the considerations for NoC interconnects need to be changed in the thesis.

2.3 Mapping of Spiking Neural Network

Mapping is the process of placing neurons into Neuromorphic hardware devices. Many existing mapping solutions are based on large digital neuromorphic processors with many cores and use Network-on-Chip as the communication framework. For example, NEUTRAMS[10] is an optimized runtime tool that maps neural networks onto the target hardware for better resource utilization, using the graph-partitioning algorithm to put densely communicating neurons together. PSOPART[11] partitions a given SNN into local and global synapses, where local synapses are mapped on the crossbar, and global synapses are mapped on the global synapse interconnect. SpiNeMap[12] uses Particle Swarm Optimization[13] and Kernighan-Lin[14] algorithms for SNN placement. The aim is to minimize the number of spikes on the time-multiplexed interconnect. All the solutions are to minimize the energy consumption and latency of the system.

In our target hardware device, where Network-on-Chip core communication does not exist, all the connections between different Neuromorphic Arrays are based on the interconnect design. In the design, there are fences to control the signals to and from Neuromorphic Arrays.

2.4 VPR (Versatile Place and Route)

Versatile Place and Route[1], or VPR, is a famous open-source academic CAD tool designed to explore new FPGA architectures and CAD algorithms. It has a history of over 20 years, and it has developed from a single FPGA packing, placement, and routing tool to part of the Verilog-to-Routing (VTR) program. In VPR, the structures and all parameters of the hardware device (FPGAs) are user-defined in an XML[7] file. The netlist that describes the circuit's functionality is also defined in a BLIF file. After compilation and running, the packing, placement, and routing results will be printed on the output window.

In the thesis, we start from building the BLIF file to describe the Spiking Neural Networks and the XML file to describe the Neuromorphic Arrays. The versions before

VPR v8.0 are based on C. VPR v8.0 is part of Verilog-to-Routing 8 and is based on C++. The required formats in VPR 8.0 for the XML file differ from earlier versions. After trying the earlier VPR versions, I chose to build the hardware device and SNN netlists based on the latest VPR v8.

VPR uses a set of terms to describe the netlist and hardware components. The cluster of primitive netlist blocks is called a "molecule" in packing. It is regarded as a whole group in putting it onto hardware elements. In SNN mapping tasks, one neuron is regarded as the basic element to be mapped. The basic hardware element for a neuron is called a "BLE (Basic Logic Element)". A CLB (Configurable Logic Block) is a cluster of BLEs and is the basic repeating logic resource on the hardware device. The hardware consists of input/output (I/O) pads, CLB resources in arrays, and other hardware components. So in the packing stage, the neurons are to be placed onto CLBs with sufficient BLE resources.

To use VPR for Spiking Neural Network architectures, the correspondence of the concepts needs to be clarified. The neurons in SNN are the cluster of primitive netlist blocks, so one neuron is regarded as one basic element of the original SNN. The Neuromorphic Array is the CLB in the hardware device, and it is also a repeating resource. The Neuromorphic Array consists of columns for signal processing, so the columns are the BLEs. The columns are modeled as LUTs (Lookup Tables) because of their similar structures and functionalities.

VPR Basics 3

Versatile Packing, Placement, and Routing, or VPR, is an open-source academic toolset widely used in FPGA architectures and CAD research[1]. As the name suggests, there are three steps in its basic flow: packing, placement, and routing. In the basic flow of VPR, there are two input files: a netlist file in BLIF(Berkely Logic Interchange Format) and an architecture description file of the target hardware in XML. The user specifies the files on the command line. After we run VPR with the files, the VPR packing (mapping) result can be observed.

3.1 BLIF file

The BLIF file, with the file extension .blif, describes the netlist in primitive blocks, such as Lookup Tables, Flip-Flops, and latches. The use of a BLIF file started from the earliest versions of the VPR tool. Moreover, the netlist should describe blocks that match the corresponding hardware element in the XML file (the hardware architecture file), or VPR would not recognize the netlist.

The BLIF file we should build here describes a Spiking Neural Network. An SNN consists of neurons, inputs, and outputs. In the BLIF file, the inputs and outputs are listed after defining the top level(.model top). After that, the neural connections need to be described. The network can be considered fully connected in a Spiking Neural Network, where presynaptic neurons send their output signals through voltages to all the postsynaptic neurons. Every neuron in the SNN architecture receives the spiking signals from all the presynaptic neurons and sends the processed output signal to all the postsynaptic neurons. This structure is similar to the primitive block LUT (Lookup Table). The number of inputs of a LUT is constant. The LUT has one output which depends on the composition of the inputs. Similarly, neurons in the same layer have the same number of inputs, and the output activities also depend on the input activities from the presynaptic neurons.

In the BLIF file, the description of a LUT starts with ".names". In typical FPGA architectures, LUTs are the most commonly used primitive block. It is a logic gate, which is the fundamental component of FPGAs. The LUT can be considered a truth table for combinational logic behavior. The inputs and outputs are either "0" or "1".

The neurons are fully connected with adjacent layers for the Spiking Neural Network architecture. The LUT representing a spiking neuron uses all "1" to show the connectedness of inputs and outputs. Here we only think about the primary Spiking Neural Networks, with no recurrent connections or other types of neuron layers where there might be connections between non-adjacent layers. Finally, the BLIF file ends with a ".end" signal line. For a BLIF file, the description of a single primitive (in our case, a LUT) must be finished with two lines. The first line shows the names of the input and output signals. Names of input signals are separated with spaces. The output signal is at the end of this line. The following line is "1"s which describes the truth table of this logic gate. The "1" and the inputs and outputs correspond to each other.

The number of inputs and outputs can be large and causes difficulty in creating this file manually. Here a python-based BLIF file generator is created to solve the problem. Suppose there are m input signals. Then we name the input signals $i\_0, i\_1, ..., i\_(m-1)$. The output layer of input signals, the first hidden layer, has n outputs, so the signals of the first hidden layer are $out\_0[0], out\_0[1]$,..., $out\_0[n-1]$. The first number indicates that this signal is the output of the first hidden layer, and the second number is the index of this signal in this layer. Similarly, if there are p outputs for the second hidden layer, the output signals of the second hidden layer are $out\_1[0], out\_1[1]$,..., $out\_1[p]$. Based on this naming method, we can quickly create a Spiking Neural Network netlist for all signals. In VPR, for SNN architectures, the name of the signals is also used as the name of the neuron since each neuron has multiple inputs but only one output. As a result, the names of neurons in the first hidden layer (L_1) are $out\_0[0], out\_0[1], ..., out\_0[n-1]$. Furthermore, $out\_1[0], out\_1[1]$,..., and $out\_1[p]$ are the names of the neurons in the second hidden layer (L_2).

An example of a BLIF file is given in Appendix A.

3.2 XML file

The XML file describes the targeted hardware architecture. In VPR, where the targets are usually FPGA architectures, the XML file includes the description of input and output pads, soft logic blocks, memory blocks, and multiplier blocks. The critical resource blocks are soft logic blocks or Configurable Logic Blocks (CLB). CLBs are fundamental repeating resources where complex tasks are executed. There are multiple minor logic elements called Basic Logic Elements for a CLB block. The Basic Logic Element (BLE) is the smallest logical operation unit. Inside a BLE, there is one Look-up Table (LUT), one Flip-Flop (FF), and a Multiplexer (MUX)[1]. Figure 3.1 shows such a Basic Logic Element in CLB.

The Look-up Table is the truth table of the logic. The Flip Flop stores logic states between clock cycles. The multiplexer selects the output among its inputs: the output of LUT or FF. Furthermore, besides BLEs, there will be a clock input signal in a CLB to control the process of logic operations.

However, in Spiking Neural Networks hardware implementation, the Neuromorphic Arrays do not need clocks. They are self-timed and event-driven, so the corresponding hardware description file should have no input clock. In the BLIF file, we discovered that a LUT could represent a neuron in SNN. So in the XML file, Neuromorphic arrays are its basic "soft logic blocks (CLBs)"; a soft logic block still consists of multiple BLEs, and a BLE is a column in the neuromorphic array, where the neural activities happen. This BLE is a LUT for various inputs from presynaptic neurons and only one output for the neuron's activity. The input and output ports are still needed for the SNN implementation. The input signals of the input layer enter the input ports, and the output ports can connect with output processors.

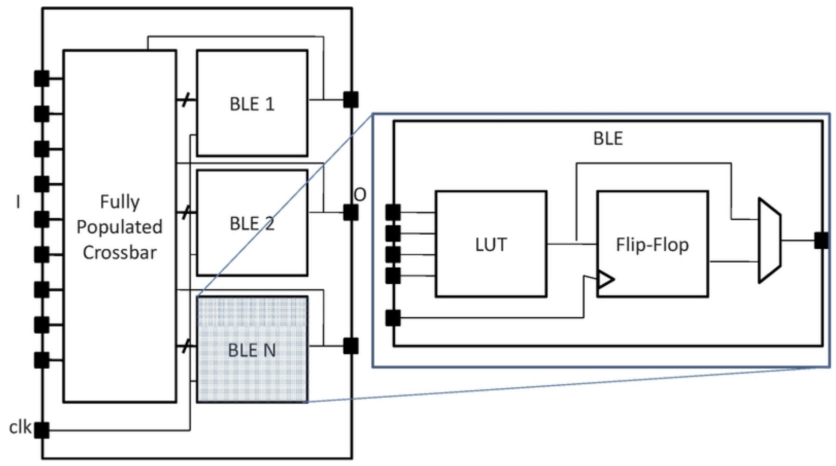


Figure 3.1: A classic BLE with 4-input LUT inside a CLB[1]

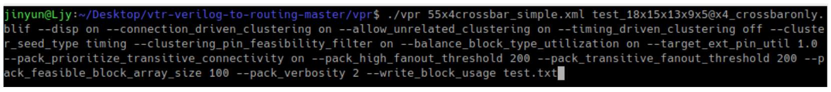
For simplicity, the SNN hardware implementation consists of input/output (I/O) ports and soft logic blocks (CLBs). The CLB in VPR is the neuromorphic array. It consists of multiple basic logic elements: Look-up Tables (LUTs). And a neuromorphic array consists of multiple columns for neural activities. So for SNN architectures, the hardware implementation consists of Neuromorphic Arrays and I/Os, and a NA consists of many columns.

This simplification of the CLB structure (CLB-LUT) is beneficial for the analysis of the mapping process in VPR. The connection between the BLIF file and the XML file is straightforward. If the CLB still uses the 3-layer elements as the original FPGA example (CLB–BLE–LUT) after deleting the input clock and MUX, there will be more pins for a CLB, and it is difficult to analyze the pin's routing rules in VPR.

The types of models, input/output ports, and CLB are defined in the XML file. In the part of the complex block list, the detailed structure of the CLB is discussed. The number of LUTs (the number of columns in the NA) is the number of neurons a CLB can accommodate. Moreover, the inputs of the CLB should be sent to all LUTs. So the number of inputs of the CLB and the LUT should be the same. The connection of the CLB inputs to the LUTs' inputs in the XML file should be specified. Likewise, the output of LUTs should connect to the CLB's outputs one by one. The total number of outputs in LUTs equals the number of outputs in the CLB.

For an I/O model, the number of input/output ports should match the number of neurons. Suppose the number of input/output signals is much larger than the number of neurons, but the number of input/output pads of I/O is small. In that case, the final mapping result of VPR will use a large area to place I/O models, which is not easy to observe the mapping results of neurons. There are also some other parameters in the XML about specific hardware features that do not need to be changed. The XML file is a model of NAs and I/O pads; those parameters will not affect the mapping result.

An example of an XML file is given in Appendix A.



```
jinyun@Ljy:~/Desktop/vtr-verilog-to-routing-master/vpr$ ./vpr 55x4crossbar_simple.xml test_18x15x13x9x5@x4_crossbaronly.
blif --disp on --connection_driven_clustering on --allow_unrelated_clustering on --timing_driven_clustering off --cluste
r_seed_type timing --clustering_pin_feasibility_filter on --balance_block_type_utilization on --target_ext_pin_util 1.0
--pack_prioritize_transitive_connectivity on --pack_high_fanout_threshold 200 --pack_transitive_fanout_threshold 200 --p
ack_feasible_block_array_size 100 --pack_verbosity 2 --write_block_usage test.txt
```

Figure 3.2: The command line to run VPR for SNN mapping

3.3 VPR Basic Flow and VPR Packing

After we have both the BLIF file and the XML file, the VPR tool can be used and observed. We can run VPR after finishing compiling. On the result window, the graph of the mapping result is presented. There are three stages: packing, placement, and routing in VPR.

Packing is the process of clustering primitive netlist blocks into larger groups called Complex Blocks. The information of the primitive netlist is in the BLIF file, and the Complex Blocks here are the models we have defined in the XML file. For SNN mapping tasks, the complex blocks are defined under the tag "$\langle complexblocklist \rangle$", including input/output ports and CLBs. So packing is to place the neurons and signals onto CLBs and input/output ports. Placement is to put the Complex Blocks placed with primitive netlist blocks into the area of the hardware device, which is the FPGA grid in most cases. Routing is the way of connection between Complex Blocks after placement.

Packing is crucial for mapping Spiking Neural Network architecture to Neuromorphic Arrays. Since the structure of the hardware device that holds all the NAs are not yet precise, the mapping problem mainly focuses on the neurons' mapping of NAs. A satisfying method that maps the SNN to NAs is expected.

As the first stage of VPR, packing decides how to map the netlist to CLBs, and the general layout of primitive blocks has been determined. There are two aspects of mapping. The signals are mapped to the input and output pins of the Complex Blocks in pin mapping. So the inputs of neurons are mapped to the input pins of NAs. In neuron mapping, the primitive blocks are clustered and placed on corresponding Complex Blocks. So the neurons are mapped to the NA columns.

The command line to run VPR is shown in figure 3.2. The names of the BLIF file and XML file are specified. The other command line options are discussed in later chapters.

4 Mapping of Pins

The mapping of pins can be observed directly from the output graph of VPR. If both the XML file and BLIF file are ready, with the default configurations on the command line and original VPR codes, how pins are connected with other pins can be observed. In VPR packing, the mapping of pins is called Routing. This routing is different from the meaning of the "Routing" stage in VPR because the routing in the "Packing" stage is within a Complex Block and the targets are netlists and pins. VPR's Routing stage, on the other hand, operates between Complex Blocks, including CLBs, I/O pads, and so on. Note that Complex Block is a more extensive concept of hardware components, including not only Complex Logic Blocks but also I/O pads.

4.1 Pin Mapping in VPR

4.1.1 Start from a simple Spiking Neural Network architecture

There is not enough information available on how VPR's routing within Complex Blocks is conducted. From the command line options provided by VTR 8.1.0, the only option for routing pins is "*–target_ext_pin_util*". It controls the ratio of pins in use and all external pins in VPR's packing. However, this option cannot control the order of using those pins. When we use the default values of VPR command line options and run VPR, the output shows the disorderly mapping of external pins. The signals from the neurons of the same layer are mapped to discontinuous pins, mixing with signals from other neurons. For the mapping problem of SNN, signals from neurons of one layer should occupy consecutive pins of the NA.

To solve the pin mapping problem, the method of pin mapping needs to be thoroughly analyzed. To begin with, in the source code "pack" folder, the costs of pins are calculated in "*cluster_router.cpp*". Using VTR's logging macros, the exact values for the costs of pins can be recorded into a log message. The analysis starts from a simple SNN mapping example: to map a 5x4x2 SNN architecture (5 inputs; the first hidden layer has 4 neurons and the second hidden layer has 2 neurons) to Neuromorphic arrays with 3 columns and 10 inputs (one NA can hold 3 neurons, and the number of input signals should be equal or less than 10). VPR will use at least 2 NAs to place 6 neurons in the hidden layers.

Figure 4.1 shows the model of such a NA. Three columns are shown as rectangular blocks. The indexed pins will be explained in 4.1.2.

The advantage of using a simple structure of Neuromorphic Arrays (CLB-LUT) is that it essentially reduces the difficulty of analyzing routing inside CLBs. In packing, there are 9 Complex Blocks: 5 input pads, 2 output pads, and 2 CLBs. For each Complex Block, the routing process decides how signals are mapped to pins.

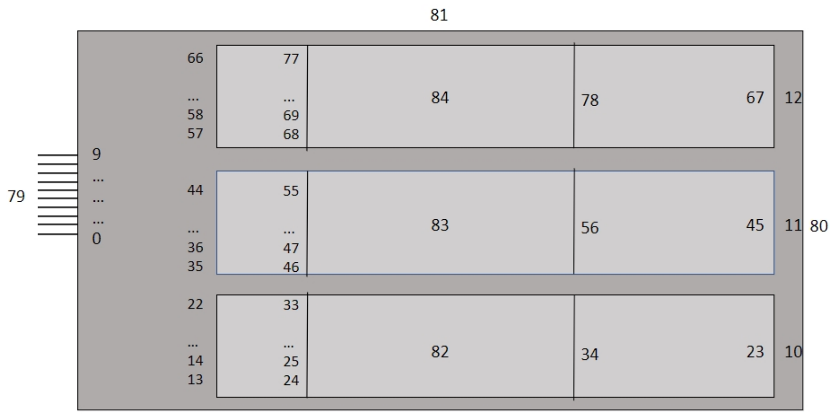


Figure 4.1: The model of a Neuromorphic Array with 10 input pins and 3 columns

4.1.2 Pin Indexing Rule of a CLB

The routing with a Complex Block uses indexes to represent the location of pins. There are not only external pins but also internal pins, such as the inputs and outputs of LUTs. The index is to tag all the pins and locations in a Complex Block.

The CLB here is a Neuromorphic Array with 10 input pins and 3 columns (so there are 3 outputs). From the log messages in VPR, the indexes of pins are set up in the following way as shown in figure 4.1

Firstly, the 10 external inputs of a Neuromorphic Array CLB are set with indexes of 0 to 9. The 3 external outputs of the NA are set with indexes from 10 to 12. There are 3 columns in the NA, so there are 3 LUTs inside the CLB. Each LUT has 10 inputs, and the inputs connect with the 10 external inputs of the CLB. The names of the LUTs are lut[0], lut[1], and lut[2]. The inputs of lut[0] are set with the indexes from 13 to 22. The LUT also has an output, and it is set with index 23.

The original FPGA CLBs have a 3-layer structure: CLB-BLE-(LUT+FF+MUX), and the corresponding indexed pins in CLB have a 3-layer structure. Here in the self-defined Neuromorphic array where there are only 2-layer structured CLBs, the LUT has a repeated layer of the indexes to maintain consistency of CLB indexes. For lut[0], the extra layer of indexes comes after index 23. As a result, indexes from 24 to 33 are the 10 repeated inputs, and index 34 is the repeated output.

For lut[1] and lut[2], the rule of setting up indexes is the same as lut[0]. So in lut[1], the input indexes are from 35 to 44, and the output index is 45. In the repeated layer, index 46 to index 55 are for LUT inputs, and index 56 is for LUT output. In lut[2], the inputs are indexed 57 to 66 for LUT and 68 to 77 for the repeated layer. Index 78 and 67 are for the outputs.

Despite the indexes for inputs and output of netlist blocks, some other indexes represent the “source and sink”. In standard FPGA devices, the sink is where the path for signals finally achieves, and the source is the beginning of a signal path. The location of the source for the whole CLB is called “cluster-external source” in VPR, and the index is 79 (right after the output index 78 of lut[2]). The sink for the whole CLB is 80.

There is one pin representing the intermediate location where the signal path passes to the output pins of one of the LUTs. After going to the output pins of the CLB, it goes back to the external inputs of the same CLB. The signal path exists when an output of a LUT in the CLB needs to go back as an input of another LUT in that CLB. All LUTs share this “cluster-internal intermediate” pin, and its index is 81. For each LUT, there is a unique sink “cluster-internal sink” pin to represent that the signal has been sent to one of the LUT to process. In total, there are 85 indexes representing the CLB pin locations.

4.1.3 Pathfinder Negotiated Congestion Algorithm

In the packing stage of VPR, the routing within Complex Blocks happens when the tool has chosen the primitive blocks for the target CLB. For the SNN architecture, pin mapping happens when the tool has found which neurons to place into the Neuromorphic Array. How VPR finds neurons to place into the NA, the mapping in primitive netlist block level, is another key component of the mapping problem, so here we only focus on mapping pins.

In the original VPR *cluster_routing.cpp*, the pathfinder negotiated congestion algorithm [1][15] is used for routing pins. The pathfinder algorithm is a well-known FPGA routing algorithm. The principle of this algorithm is to solve the routing congestion problem of FPGA. According to this algorithm, the pins are indexed in the CLB, and the mapping process of the pathfinder algorithm is to connect pins with the lowest congestion cost. The congestion cost is to evaluate the usage of the netlist on that pin. The cost updates every time a routing decision is made. There is a congestion cost when mapping to any pin, and Pathfinder chooses the pin with the lowest cost.

For the pathfinder negotiated congestion algorithm used in VPR, firstly, the intrinsic cost (*intrinsic_cost*) of each external pin is defined. The pathfinder routing will choose the signal path with the lowest cost. After a signal path has finished routing, the cost is added with a congestion cost of this signal edge’s pin (*intrinsic_cost_next_edge*). In VPR, the incremental cost (*incr_cost*) is for calculating the pin cost after a path has finished routing. The incremental cost is the product of the current congestion factor ($congestion\_fac$), the current fan-out factor ($fanout\_fac$), and the sum of the intrinsic cost of the previous pin (*intrinsic_cost*), the intrinsic cost of the currently chosen pin (*intrinsic_cost_next_edge*), and the historical usage cost (*historical_usage*) before the routing this time. The historical usage cost is the product of the historical usage and a historical factor. This equation is presented below:

$$\begin{aligned} incr\_cost = {} & (intrinsic\_cost + intrinsic\_cost\_next\_edge + historical\_usage \times hist\_fac) \\ & \times (usage \times congestion\_fac) \times fanout\_fac \end{aligned} \tag{4.1}$$

The historical usage cost records the routing result of the last iteration. The value of historical usage is the usage after the last routing iteration, while the value of usage records the possible usage of the pin after this iteration. For each iteration, the congestion factor and the fan-out factor update after one iteration, and other factors are defined in the *pack_types.h* file.

After calculating the cost, the tool iteratively routes signal paths onto the pin with the lowest congestion cost, even if the pin has already been mapped with a signal path. The tool only maps the current signal path to the pin with the lowest congestion cost. Furthermore, according to this principle, some paths that have finished routing lose the pin assigned to them. Those paths, together with the unrouted paths, enter the next iteration of routing. It is the main reason the pin mapping of the original VPR packing seems random: it is the result of iteratively selecting pins with the lowest congestion cost.

4.2 The Requirement of Mapping Pins and Modification in VPR

After investigating the process of pin routing and the Pathfinder negotiated congestion algorithm, some modifications should be made based on the need for pin routing in SNN mapping.

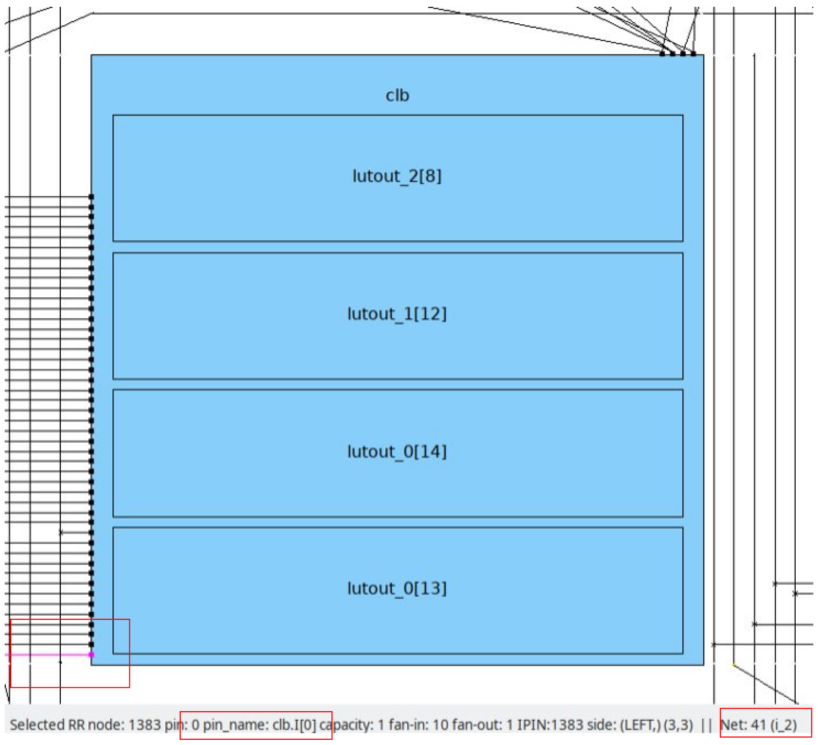
For mapping SNN architecture, the structure of NAs is quite different from the structure of FPGA CLBs. In Neuromorphic Arrays, the input signals that are the outputs from neurons on the same layer should be densely arranged according to the simplified structure of a Neuromorphic Array to minimize the distance between the input signals of one neuron. In addition, the signals should choose pins consecutively, starting from the first input pin of the CLB, so that input signals are packed as densely as possible, and the utilization of external pins can be maximized.

To meet the requirements above, the intrinsic cost of the external input pins increases with the current output edges. For example, for a signal path from the "cluster-external source" pin 79 to the external input pins 0 to 9, there could be 10 possible output edges for this signal path. The intrinsic cost of pin 0 is 0 for the first output edge. To set the priority that the signal path will be mapped on pin 0 first, the intrinsic cost of other pins (from pin 1 to pin 9) should be larger than the cost of pin 0. After finishing mapping onto pin 0, the following signal path will be mapped onto pin 1, so the intrinsic cost of pin 1 is lower than that of other pins (from pin 2 to pin 9). As a result, the intrinsic cost value increases from pin 0 to pin 9. The signal paths will prefer external pins with smaller indexes.

In addition, the cost update after finishing mapping should also meet the requirement of SNN pin mapping. In the original VPR, the costs of some pins are still lower than unmapped pins, causing signal path overlapping and rerouting of paths. The modification here is to increase the cost after mapping a pin large enough so that the mapping of new signal paths will not find the mapped pins. Among all the parameters, the update with current "usage" is the easiest and most suitable to modify.

$$incr\_cost = incr\_cost \times (usage + N) \times congestion\_fac \tag{4.2}$$

The equation describes the process of updating incremental cost with $usage$ and $congestion\_fac$. N is added to make the updated incremental cost large enough. Here $N = 200$ is used. N cannot be too large since there are intrinsic costs for using external interconnect resources (in our example in figure 4.1, the use of pin 81). In



```
Selected RR node: 1383 pin: 0 pin_name: clb.I[0] capacity: 1 fan-in: 10 fan-out: 1 IPIN:1383 side: (LEFT,) (3,3) || Net: 41 (i_2)
```

Figure 4.2: Pins mapping before the modification (1)

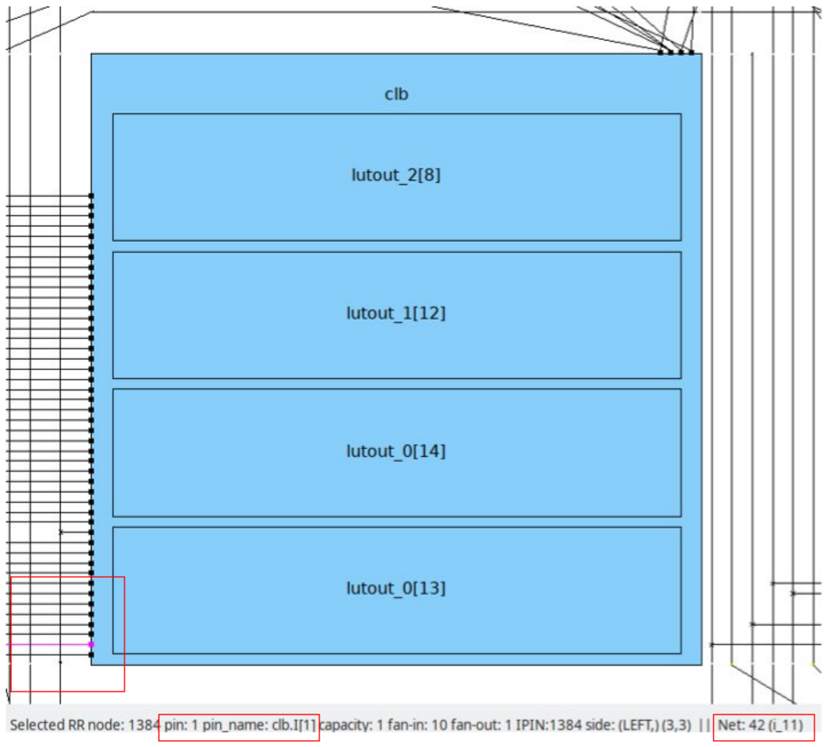
lb_type_rr_graph.cpp, the cost of using external interconnect is high enough to avoid its usage unless necessary.

After this modification, there will be only one iteration because all the signal paths have found an external input pin. There is only one routing iteration now, so the value for historical usage of pins is always 0, and the current usage value ("1"' for a mapped pin) is used in updating pin costs.

Figure 4.2 and figure 4.3 are the mapping of pins before the modification in an $18 \times 15 \times 13 \times 9 \times 5$ SNN architecture where 4 neurons ($lutout\_2[8]$, $lutout\_1[12]$, $lutout\_0[14]$, $lutout\_0[13]$) from 3 different layers are mapped onto a CLB (a Neuromorphic Array). The output of a neuron and the output itself have the same name in VPR. To distinguish them, we add a "lut" in front of the name of neurons, just as the output graph shows.

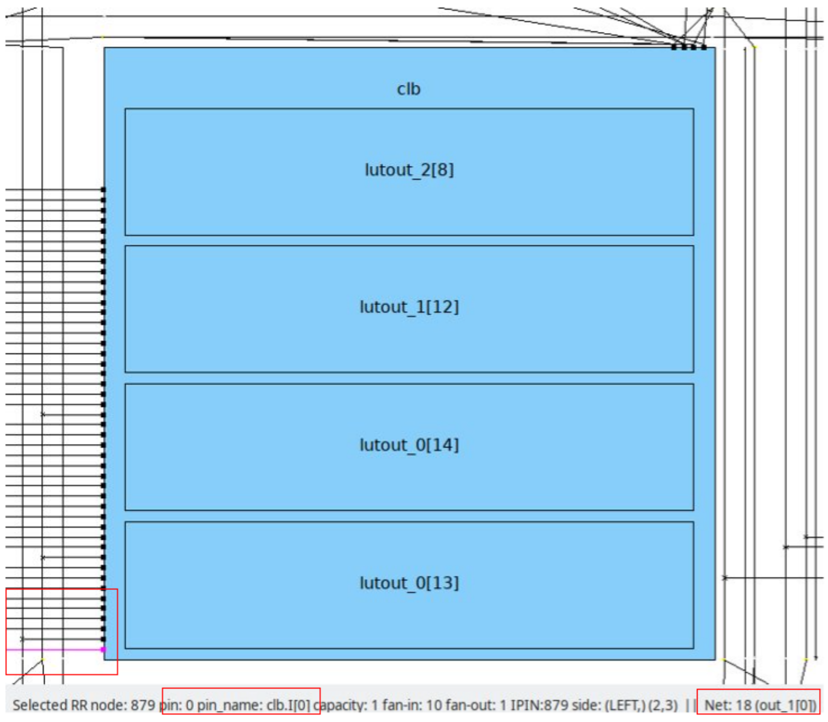
To observe the pin mapping result, in the VPR output window, we right-click the pins of NAs which are represented by black dots on the left side of the NA. In this example, the NA has 55 input pins and 4 columns, making sure that when 4 columns are all occupied, there are enough input resources for all the neurons. Once we click the pin on the left side, the **bottom sidebar** of the window shows the information of this pin. And the chosen pin is highlighted in pink.

In figure 4.2, the first pin from the bottom of the NA is chosen. The sidebar shows that it is the first input pin, $clb.I[0]$ of the NA. The last phrase "Net" indicates which signals are mapped onto the pin. In figure 4.2, i_2, the second input of the SNN is mapped to the pin. As figure 4.3 shows, i_11 is mapped to the second pin $clb.I[1]$ of the NA. So the signals and pins are not mapped in the order of inputs of neurons.



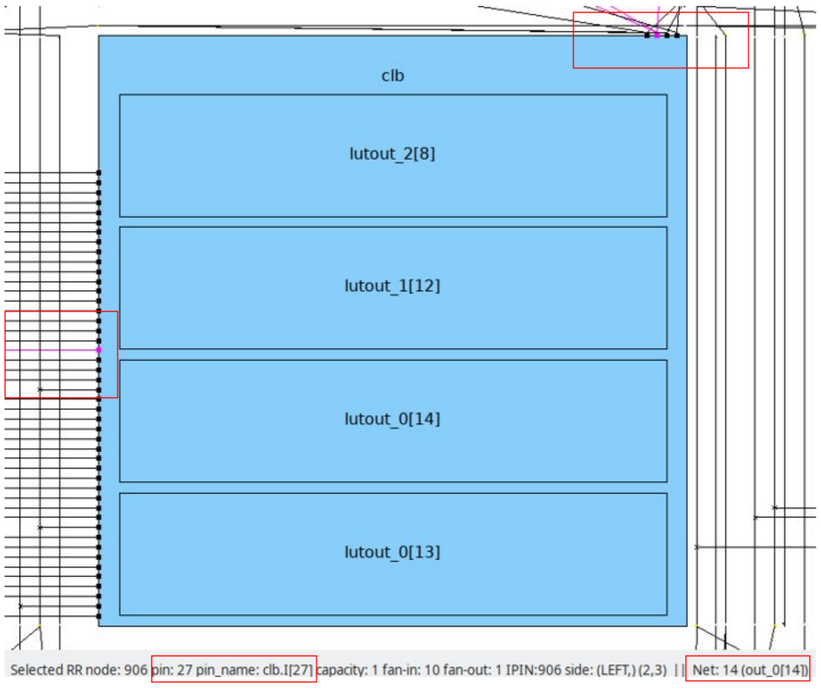
```
Selected RR node: 1384 pin: 1 pin_name: clb.I[1] capacity: 1 fan-in: 10 fan-out: 1 IPIN:1384 side: (LEFT,) (3,3) || Net: 42 (i_11)
```

Figure 4.3: Pins mapping before the modification (2)



```
Selected RR node: 879 pin: 0 pin_name: clb.I[0] capacity: 1 fan-in: 10 fan-out: 1 IPIN:879 side: (LEFT,) (2,3) || Net: 18 (out_1[0])
```

Figure 4.4: Pins mapping after the modification (1)



```
Selected RR node: 906 pin: 27 pin_name: clb.I[27] capacity: 1 fan-in: 10 fan-out: 1 IPIN:906 side: (LEFT,) (2,3) | | Net: 14 (out_0[14])
```

Figure 4.5: Pins mapping after the modification (2)

Figure 4.4 and figure 4.5 are the mapping of pins after the modification for pin mapping in the same NA. The mapping of neurons are the same, with *lutout_2*[8], *lutout_1*[12], *lutout_0*[14], and *lutout_0*[13]. The first neuron mapped into the CLB is *lutout_2*[8], so the input signal of this neuron, *out_1*[0], is mapped to the first pin *clb.I*[0]. Then *out_1*[1] is mapped to the second pin *clb.I*[1]. All the signals are mapped to the pins in order.

Figure 4.5 shows *out_0*[14] is mapped to the *clb.I*[27] pin of the NA, which is the 28th pin. This is because the neuron *lutout_1*[12] is the second neuron that mapped onto this CLB, and its inputs are signals from *out_0*[0] to *out_0*[14]. The last signal, *out_0*[14], has $(13+14)$ pins before it, and it is mapped to the 28th pin. One output pin of the NA is also highlighted because the signal *out_0*[14] is the output of the neuron *lutout_0*[14] in the same NA. So this signal is transmitted between neurons in one NA.

After the modification, the SNN pin mapping problem is solved. The modifications are made in "*cluster_router.cpp*" (for updating pin costs) and "*lb_type_rr_graph.cpp*" (for the values of intrinsic costs). The next problem to consider is mapping neurons on NAs.

Mapping of Neurons 5

The mapping of neurons is a higher-level process than mapping pins and signal paths. It decides how different neurons in the SNN architecture are placed into the Neuromorphic Arrays. To find a satisfying mapping method for neurons, we should analyze how VPR maps the primitive blocks into Complex Blocks.

5.1 Neuron Mapping in VPR

In VPR, the mapping of primitive blocks is to map primitive netlist block clusters onto the Basic Logic Elements (BLE). Such a cluster of primitive netlist blocks is called a "molecule" in VPR code. In VPR, It is a basic unit for mapping at the primitive block level. All the Complex Blocks, including CLBs, and I/O pads, are mapped with inputs, outputs, and neurons in the SNN. The mapping of CLBs is our main focus. The smallest primitive logic block cluster in SNN mapping is a spiking neuron.

In VPR, functions for this mapping are in *cluster.cpp* and *cluster_util.cpp*. AAPack is the name of the packing method.

5.2 Neuron Indexing Rule

In the packing process, each neuron has a unique ID to participate in the mapping. In SNN examples, the pins for the inputs and outputs are to be indexed first. After that, the index starts from the neurons of the first hidden layer and then the second hidden layer until it reaches the neurons in the last hidden layer. This rule applies to Spiking Neural Networks of all sizes. Take the $5 \times 4 \times 2$ SNN as an example. Figure 5.1 shows the SNN architecture and the neuron spiking paths from 5 inputs to one neuron in the first hidden layer and then to the second hidden layer and outputs. In figure 5.1, the inputs are from 0 to 5, and the outputs are 5 and 6. The indexes of neurons start from 7 in the first hidden layer. The indexes of the two neurons in the second hidden layer are 11 and 12.

5.3 Search for Neurons

To map neurons on CLBs, the first neuron to be put onto an empty NA is called a "seed". All the other neurons that are searched are based on this seed neuron. The first step of mapping primitive netlist blocks is finding a seed. Then the tool finds other unpacked neurons based on that seed and fills the empty BLEs with them until all BLEs are mapped.

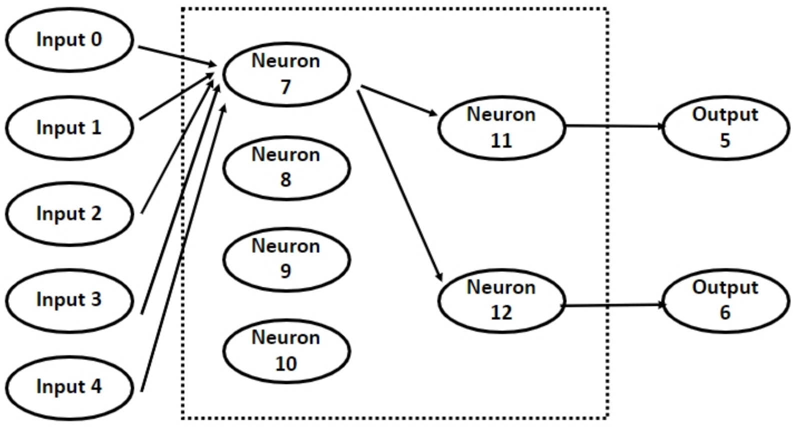


Figure 5.1: The indexes of neurons, inputs, and outputs in a $5 \times 4 \times 2$ SNN example

The function that defines how VPR finds a seed is "*initialize_seed_atoms*" in *cluster_util.cpp*. The user can change the way of selecting seed on the command line of "*cluster_seed_type*". There are many seed selection methods, including two main types and their weighted selections. For the selections, each neuron has a selection gain. The neuron with the largest gain among all unpacked neurons will be selected as the seed. The first type of selection is to choose the unpacked neuron with the most timing-critical connections. Each neuron has a "criticality", and the neuron with the largest criticality will be selected as the seed. The second type is to select neurons with the most number of pins as the seed. Other selection methods are based on the weighted sum of the gains in those two selection methods.

After a seed is mapped to a NA, the next step is to find feasible unpacked neurons for columns in the NA. In VPR, the critical function to describe the neuron selection is "*get_molecule_for_cluster*" in *cluster_util.cpp*. The algorithm used is in the function "*get_highest_gain_molecule*". There are four types of ways to find unpacked neurons:

- 1) The function helps find unpacked neurons connected directly with the current seed. For example, it will choose the neuron that uses the present seed neuron's output as its input.
- 2) it will find unpacked neurons in the architecture with no direct connection but transitive connections with the current neurons.
- 3) It will then find unpacked neurons with weak connections with the current neuron. For such neurons, there is a threshold to control the number of selected neurons in the command line.
- 4) if there are attraction groups for the current neuron, it will find them. An attraction group is a group of primitive netlist clusters that are highly desirable to be packed together.

Here the current neuron means not only the seed but also the neuron that has been found in this turn of finding neurons. The reason for using this order of finding is that in

VPR primitive blocks which are close to each other in the netlist will be placed together in the CLB. The order of 2) and 3) can be switched by the option of command line "*pack_prioritize_transitive_connectivity*", to decide which neurons should be searched first. In the original AAPack, methods 2) and 3) are used only when method 1) cannot find feasible neurons.

Searching for all feasible neurons that can be mapped into the CLB is not enough. The tool calculates the gain of mapping each neuron into the BLE. In the original VPR, this calculation is based on the neuron's sharing gain, connection gain, and timing gain.

Sharing gain is the number of input pins the unpacked neurons shares with the packed neuron(s) in the CLB. The larger the sharing gain, the closer the neuron is to the currently packed neurons.

Connection gain is a weighted sum value for the current unpacked neuron with the last neuron that has been mapped into the NA. It is a non-zero value only if there is a direct connection between them. This gain updates in every mapping process of a NA and the connection gain value will be retained and not cleared for each search of unpacked neurons in one NA.

Timing gain is 1 if the current unpacked neuron is on the timing-critical path of the mapped neurons. If it is not, then the timing gain is 0.

The total gain is a weighted sum of those three gains, and the weights can be modified in the command line options. In the command line, the user can select the way of mapping neurons onto CLBs–timing-driven clustering or connection-driven clustering. Beta is to control the proportion of connection gain and sharing gain. Alpha is used to manage the proportion of timing gain and the other two gains and is available only if it is timing-driven clustering.

After the total gain of each neuron is calculated, the packer chooses the neuron with the largest gain from a list of feasible unpacked neurons. In the list, the neurons are ranked according to the gain, from the lowest to the highest. The length of the list is defined as a constant, and if the list cannot hold all the feasible neurons, the packer in VPR will dump the neurons with the smallest gains. This strategy has no negative influence on the correctness of the algorithm because AAPack will choose the neurons with the largest gain. In AAPack, the neuron represented by the last item (the largest gain) in the list will be selected. However, suppose some other algorithm prefers the neuron with the lowest "gain" (the neuron represented by the first item in the list). In that case, there could be errors if the list has deleted neurons with low "gains" due to insufficient list length. So if the way of choosing neurons from the list is modified, the list length for all feasible neurons should be large enough.

For the mapping problem of the SNN architecture, a pre-experiment on the methods of selecting neurons has been made.

In our problem, the CLB is a Neuromorphic Array. A BLE is a column in the Neuromorphic Array. In SNN architecture, a neuron connects directly with all the presynaptic neurons and all the postsynaptic neurons. The calculation of connection gain only happens between neurons from adjacent layers. For unpacked neurons from non-adjacent layers, the connection gain is zero. For neurons from the same layer, where neurons have no direct connections, the connection gain between them is also 0.

In the pre-experiment, only 2 of the 4 methods for searching for feasible neurons are used: 1) and 2). There are no weak connection neurons and attraction groups in a Spiking Neural Network architecture. To find unpacked neurons in adjacent layers (the layers of its presynaptic neurons and postsynaptic neurons) and the same layer as the mapped neuron, method 1) is applied. To find unpacked neurons in non-adjacent layers where neurons have no direct connections, method 2) is applied. Method 2) is used only when method 1) cannot find feasible neurons, so the packer preferentially maps the neurons in adjacent and the same layers.

6 Evaluation of Mapping Spiking Neurons to Neuromorphic Arrays

The problem of mapping neurons in SNN architecture is to put neurons into Neuromorphic Arrays. As there are more neurons, there are more ways to put neurons onto NAs. The critical problem is that there is no existing standard to measure a method for mapping Spiking Neurons to Neuromorphic Arrays. For mapping Spiking Neural Networks to Neuromorphic Processors, there are some existing research results evaluating a mapping method. However, the hardware device they generally use is Digital Neuromorphic Processors, which use Network-on-Chip (NoC) to transport the communication between cores with packet messages, such as Loihi[2] and TrueNorth[3]. However, the Neuromorphic Arrays models discussed here do not have such interconnect architecture as communication between neuron cores. Moreover, most importantly, it is self-timed and event-driven, without system clocks. Our Neuromorphic Array has a smaller scale than the digital neuromorphic chips. The mixed signals between different Neuromorphic arrays in the system pass a ”fence” that controls the signal transmission.

The goal is to reduce the spike messages, which helps to reduce energy consumption and spike latency. A large number of spike messages on the interconnect causes congestion and affects the system's throughput.

The first optimization is to minimize the number of crossbars (the arrays that accommodate neurons in digital neuromorphic devices) utilized in the system or to maximize the utilization of a single crossbar (such as PACMAN[16] and NEUTRAMS[10]). The second optimization is to minimize the number of spikes on time-multiplexed interconnects (such as PSOPART[11] and SpiNeMap[12]). Those two optimizations are the basis for many optimization methods.

Among all those methods, the topology-aware mapping method called NeuToMa[9] has built a signal transport model similar to our Neuromorphic Arrays. Though in NeuToMa an NoC-based chip is used, it divides communication into two categories: the first is "Neuron-to-neuron", and the second is "Neuron-to-core". Neuron-to-neuron communication is the activity of sending spikes between two neurons. Neuron-to-core communication[2][17] is a kind of mechanism that processors use to reduce NoC spike messages. When a neuron in one core spikes, the destination core of this spike message receives the spikes. As long as the core can be mapped with the neurons, this single spike activity is shared by all the neurons in that core. It is only one spike activity from the presynaptic neuron to the postsynaptic neurons. This "Neuron-to-core" mechanism is very similar to the working mechanism of the Neuromorphic arrays. In a Neuromorphic array, there are columns for signal processing from input activities to an output of a neuron. All the columns can receive the signals from inputs, so the input spike activity is shared by all the neurons mapped in the Neuromorphic Array. As long as the number of neurons does not exceed the number of columns, the amount of spiking activity that travels across the neuron and the destination Neuromorphic Array is counted once. The

Spiking activities that travel across different Neuromorphic arrays pass the interconnect structures. If too many spike activities travel, there might be congestion, and the system throughput and performance might be negatively affected. So the optimization goal of our SNN mapping problem is to minimize the spike activities that traverse between NAs.

6.1 Optimization Goal

An evaluation of the mapping method can be established based on the above theory of Neuromorphic Arrays.

In the Spiking Neural Network, the input layer is the Virtual Layer because the inputs are not real neurons. This input layer is marked as L_0, and the number of input signals is n_0. The first hidden layer, L_1, is where the inputs are processed, and the outputs are sent to the second hidden layer. The number of neurons in L_1 is n_1. The neurons from this layer are Spiking Neurons. So L_1 is also "the first neuron layer". Similarly, L_2 is the second hidden layer, and there are n_2 neurons in L_2. It is "the second neuron layer". Suppose there are x layers with Spiking Neurons in total. The x^{th} hidden layer is L_x, with n_x neurons in it. The number of outputs should be the same as the number of outputs in L_x, so the number of outputs of the SNN architecture is n_x. Because of the features of Spiking Neurons, the spiking activity from L_0, L_1, to L_{x-1}, L_x is reducing as more layers are passed. For a neuron in layer $L_n(2 \leq n \leq x)$, its presynaptic neurons are in layer L_{n-1}. For a neuron in layer $L_m(1 \leq m \leq x-1)$, its postsynaptic neurons are in layer L_{m+1}.

S_{L_n} is the spiking activities of the Spiking Neurons over a period of time. We use S to express the Spiking activities that traverse between different NAs.

For the first neuron in layer L_1, we use $S_{L_1}(1)$ to represent its Spiking activity. For the second neuron in layer L_1, the corresponding spiking activity is marked as $S_{L_1}(2)$. In general, the spiking activity in the time period for the k^{th} neuron in layer L_q is $S_{L_q}(k)$. The values of the spiking activities are measures of the spiking activities of neurons' interrelationships so that they can be normalized to a particular range. So the values are normalized between 0 and 1 for convenience.

Suppose there is a mapping method for a Spiking Neural Network architecture on Neuromorphic Arrays. The neurons in L_1 are divided into N_{L_1} parts, and each part is placed into a Neuromorphic Array. So the neurons in L_1 are placed into N_{L_1} Neuromorphic Arrays. In general, the neurons in $L_n(1 \leq n \leq x)$ are divided into N_{L_n} parts, so the neurons are placed into N_{L_n} NAs.

Figure 6.1 is a general model for SNN architectures used in the thesis. The spiking activities are represented by S_{L_n}.

A_{spike} is the total amount of spike activities traversing between NAs. One mapping method of the simple example of mapping a $5 \times 4 \times 2$ SNN architecture (5 inputs; the first hidden layer has 4 neurons and the second hidden layer has 2 neurons) to Neuromorphic arrays (each with 3 columns and 10 inputs) is used to illustrate the expression of A_{spike}.

In this mapping method, as figure 6.2 shows, two Neuromorphic arrays are used. Each NA is mapped with two neurons from the first hidden layer and one neuron from

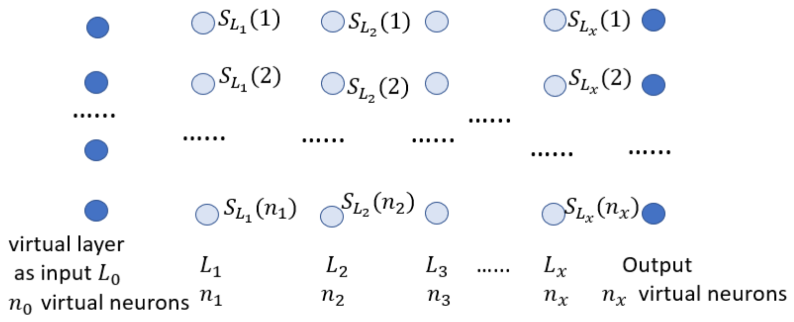


Figure 6.1: The SNN architecture model

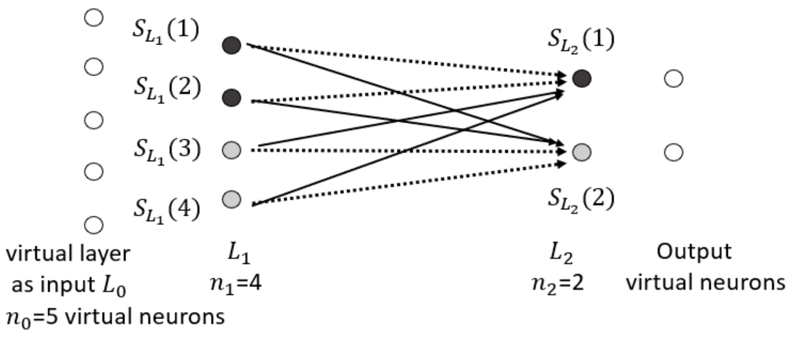


Figure 6.2: The spike activities in a $5 \times 4 \times 2$ SNN architecture

the second hidden layer. The neurons in gray are in one NA, while the neurons in black are in another NA. The dotted line represents spiking activities inside the NAs, while the solid line represents spiking activities that traverse between NAs. In this situation:

$$\begin{aligned} A_{spike} &= (S_{L_1}(1) + S_{L_1}(2)) \times 1 + (S_{L_1}(3) + S_{L_1}(4)) \times 1 \\ &= (S_{L_1}(1) + S_{L_1}(2) + S_{L_1}(3) + S_{L_1}(4)) \times 2 \\ &- (S_{L_1}(1) + S_{L_1}(2)) - (S_{L_1}(3) + S_{L_1}(4)) \end{aligned} \tag{6.1}$$

For more general SNN architectures:

$$\begin{aligned} A_{spike} &= \sum[\sum S_{L_i}(k) \times N_{postsynaptic}(S_{L_i}(k))] \\ &= \sum_{hidden\ layers} [N_{L_{i+1}} \sum S_{L_i}(k) - S_{L_i}{}'] \\ &= \sum_{hidden\ layers} N_{L_{i+1}} \sum S_{L_i}(k) - \sum_{hidden\ layers} S_{L_i}{}' \end{aligned} \tag{6.2}$$

In this equation, the sum of all spiking activities in a layer is known and can be calculated by counting the number of spikes in the time period. To minimize A_{spike}, the number of NAs that the postsynaptic neurons are mapped ($N_{L_{i+1}}$) should be minimized, and the spike activities inside one NA ($S_{L_i}{}'$) should be as large as possible. According to the equations, our optimization goal for mapping SNN to NAs can be broken down into 2 targets:

- 1) to minimize the number of Neuromorphic Arrays used by postsynaptic neurons;
- 2) to maximize the spike activities inside Neuromorphic Arrays.

6.2 Implementation Method

To reach the target of minimizing the number of NAs used by postsynaptic neurons and maximizing the spike activities inside NAs, three steps are needed for the mapping of SNN architecture. The steps also consider the characteristics of mapping in VPR and the feasibility of the method.

Step 1:

Rank the neurons in each layer according to their spiking activities, from the neuron with the lowest activities to the neuron with the largest activities. It is to leave the neurons with the largest activities unpacked, which might contribute to more $S_{L_i}{}'$ in later steps.

After that, cluster spiking neurons layer by layer. Put the neurons of a layer in as few Neuromorphic Arrays as possible. The spiking activities we discussed here traverse through different NAs, and are received by NAs where the postsynaptic neurons are mapped. The first layer, which can be a layer of postsynaptic neurons, is the second hidden layer L_2. The neurons there are the postsynaptic neurons of spiking neurons in L_1. So the clustering starts from the second hidden layer and then to the third hidden layer until the neurons from the last hidden layer (the output layer) are clustered. The number of neurons in each cluster equals the number of columns in the Neuromorphic Arrays. This ensures that a Neuromorphic array is fully utilized by as many postsynaptic neurons as possible. This is the first strategy for minimizing the number of NAs used by postsynaptic neurons. After this clustering, there may be some unpacked neurons in each layer because if they are mapped into a Neuromorphic array, there are still empty columns in the NA where other neurons (from different layers) can be mapped. The neurons in L_1 are prepared for further filling processes, so they are not clustered here. Figure 6.3 shows this process in an $18 \times 15 \times 13 \times 9 \times 5$ SNN architecture mapping to NAs with 4 columns. Most of the neurons in L_2, L_3, and L_4 are mapped to NAs, and the 6 NAs are mapped with neurons from the same layers.

Step 2:

Map the unpacked neurons in the second hidden layer (L_2). Find all the possible mapping solutions and calculate the corresponding A_{spike} and $S_{L_i}{}'$. According to Step 1, the neurons with the largest activities are unpacked and could contribute to a larger value of $S_{L_i}{}'$.

The reason for starting to map the unpacked neurons in L_2 is that the spiking activities are reducing as the number of layers increases. The mapping of the unpacked neurons in the second hidden layer has a relatively larger effect on the value of $S_{L_i}{}'$ than other layers.

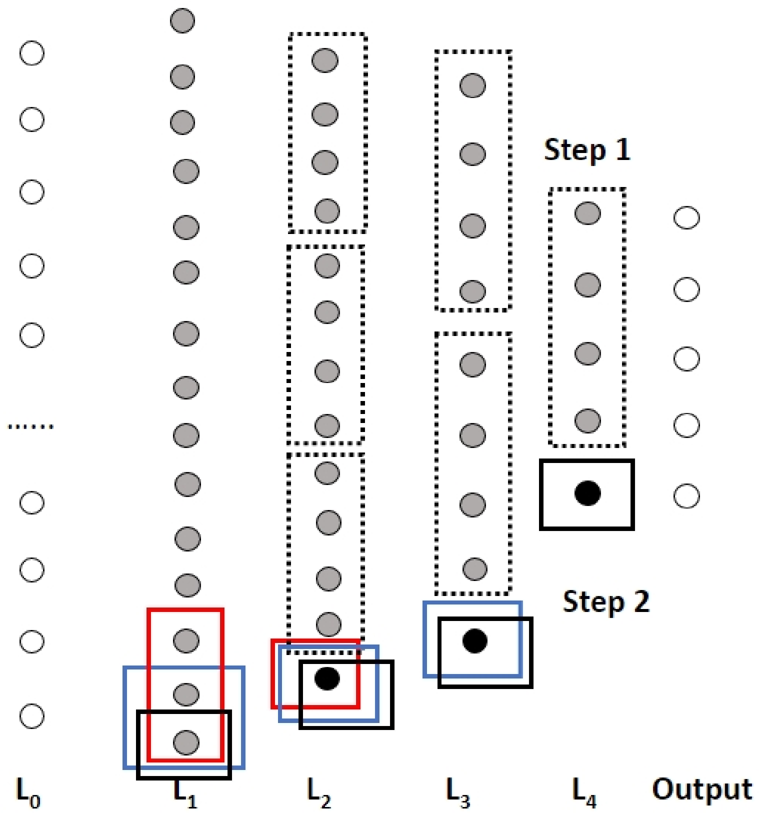


Figure 6.3: An $18 \times 15 \times 13 \times 9 \times 5$ SNN architecture

The target that the number of NAs used by postsynaptic neurons should be minimized still exists, so the unpacked neurons from one layer must stay together and cannot be split into multiple parts to fill in empty columns. Apparently, even though breaking the unpacked neurons increases the spike activity within Neuromorphic Arrays (${S_{L_i}}'$), it increases the number of NAs, which increases the total spiking activity of that presynaptic neuron layer. The increase of the spiking activity ($N_{L_{i+1}} \sum S_{L_i}(k)$) is always more than the spiking activity increased within NAs (${S_{L_i}}'$). So the rule should be followed.

To find the possible methods for the unpacked neurons in L_2, we should discover the deepest layer that the cluster of unpacked neurons from L_2 can reach. Suppose it can reach the unpacked neurons from L_k for the deepest, which means that the unpacked neurons from L_2 can be clustered with unpacked neurons from L_3,..., and finally, L_k. All of them can be mapped to one Neuromorphic Array, and some neurons from L_1 can be filled in the empty columns of the NA. The neurons from L_1 are also ranked according to their spiking activities from small to large. Furthermore, the neurons we choose from L_1 to fill into the empty columns is the one with the largest spiking activities among all neurons in L_1. It ensures that the largest spiking activities are prioritized to be categorized as spiking activities within NAs, increasing the value of ${S_{L_i}}'$.

At the same time, there are more possible methods of mapping: the unpacked neurons from L_2 can be clustered with those from L_3,..., and finally, L_{k-1}. In this situation, more neurons from L_1 need to be filled in. Still, the neurons from L_1 are picked, starting from the neuron with the largest activities and then the neuron with smaller activities.

In this example, there are $(k-1)$ kinds of methods in total. The next step is to calculate ${S_{L_i}}'$ for each method. For example, in a method that unpacked neurons of L_2, L_3, $\ldots L_q$ are mapped into one NA and fill the empty columns with L_1 neurons, there are 2 situations:

- 1) If neurons from L_1 are filled into the NA, then $\sum {S_{L_i}}' = {S_{L_1}}' + {S_{L_2}}' + {S_{L_3}}' + ... + {S_{L_{(q-1)}}}'$
- 2) If no neurons from L_1 are filled in, then $\sum {S_{L_i}}' = {S_{L_2}}' + {S_{L_3}}' + ... + {S_{L_{(q-1)}}}'$

${S_{L_n}}'(1 \leq n \leq q-1)$ is the sum of the spiking activities of unpacked neurons in L_n.

In figure 6.3, the possible mapping method in this example is highlighted in different rectangles. The red rectangle shows the unpacked neuron in L_2 is mapped directly with 3 neurons in L_1. The blue one shows the unpacked neuron in L_2 is mapped with the unpacked neuron in L_3 and 2 other neurons in L_1. The black rectangle shows the unpacked neurons in L_2, L_3, and L_4 are mapped together, with an extra neuron in L_1.

Step 3:

After listing and calculating $\sum {S_{L_i}}'$ for mapping unpacked neurons in L_2, there might be some other unpacked neurons. For the rest of the unpacked neurons, map as much of them as possible to a NA. Those neurons have no direct spike activity connection from the neurons in L_1, so how neurons in L_1 are mapped does not influence $\sum {S_{L_i}}'$ of their NA. In this case, the unpacked neurons are mapped as much as possible if the requirement of minimizing the number of NA for postsynaptic neurons is met. After all unpacked neurons are mapped into a NA, the total $\sum_{hidden\ layers} {S_{L_i}}'$ in all NAs can be calculated. The method with the largest value of $\sum {S_{L_i}}'$ is the method with the lowest spiking activities that traverse NAs, and this is our final solution for the mapping problem.

7 The Basic Flow for Mapping Spiking Neural Network architectures on Neuromorphic Arrays

The basic flow for mapping Spiking Neural Network architectures on Neuromorphic Arrays in VPR is shown in figure 7.1 below.

7.1 Input Files

There are 3 inputs for the process of mapping:

- 1) BLIF file, the description for SNN architecture. The LUT model is used to build SNN neurons. And the SNN network is fully connected. The number of neurons in each layer can be found in the BLIF file.
- 2) XML file, the description for the hardware device with Neuromorphic Arrays and input/output pads. The number of columns of a NA is the maximum number of neurons mapped on the NA. It is defined in the XML file.

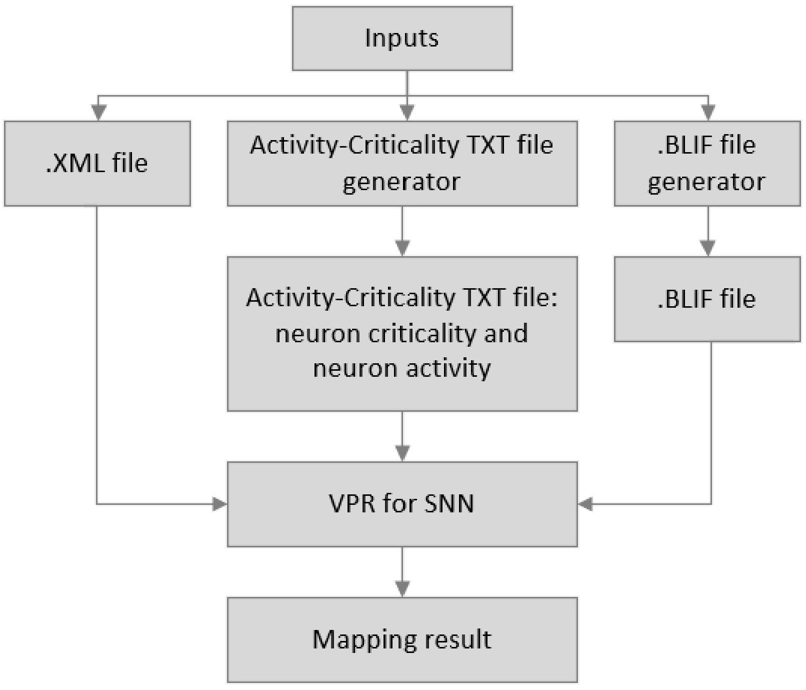

Inputs
.XML file
Activity-Criticality TXT file generator
.BLIF file generator
Activity-Criticality TXT file: neuron criticality and neuron activity
.BLIF file
VPR for SNN
Mapping result

Figure 7.1: Basic flow for mapping Spiking Neural Networks in VPR

- 3) The activity-criticality TXT file describes neuron spiking activities in SNN architecture. It tells VPR how to map the neurons to the NAs. This file contains an array that stores all information about neuron spiking activities and the choice of mapping method. In that array, criticalities for neurons tell VPR the order of choosing neurons to put into a new NA. In addition, there are block activities that tell VPR the way of selecting neurons inside a NA.

For XML files, the description for CLB inputs and the size of LUTs should be modified to change the sizes of Neuromorphic Arrays. In addition, the delays for each input of LUTs and the delays for CLB (Neuromorphic Array) inputs to each LUT's inputs should also be modified. A .py file is created for them to print all the delay values, with the number of columns and the number of CLB input pins as input information.

For BLIF files, the python-based BLIF file generator is used to print various kinds of SNN architectures. The generator is modified based on the number of layers in the SNN architecture. The inputs of the generator are the number of neurons in each layer.

The activity-criticality TXT file, which does not exist in the original VPR, is a modification based on the mapping method of the SNN architectures to NAs. Moreover, this file is adaptive to the mapping process of VPR's packing. A python-based activity-criticality TXT file generator adds values to the array.

The activity-criticality TXT file is an array of the mapping relationship between two blocks. Suppose the SNN architecture has L_x hidden layers: in layer $L_k (1 \leq k \leq x)$, there are n_k neurons. In the input layer L_0, where there are no spiking neurons, we use n_0 to represent the number of virtual neurons–the number of inputs of the SNN architecture. Since the last hidden layer is L_x with n_x neurons, the number of outputs of the SNN architecture is also n_x. Each neuron has a unique ID according to the rule of indexing neurons in VPR. The block IDs in this example are as follows:

0 to $(n_0 - 1)$ are the inputs of the SNN; n_0 to $(n_0 + n_x - 1)$ are the outputs of the SNN; $(n_0 + n_x)$ to $(n_0 + n_x + n_1 - 1)$ are the neurons in the first hidden layer L_1. And the following neurons and their indexed block ID follow the same rule. This is how VPR recognizes the neurons in the SNN architecture.

The inputs and outputs are also regarded as virtual "neurons" in VPR. There are n_0 input virtual neurons, and n_x output virtual neurons. The difference between actual neurons and virtual neurons is the location they are mapped to: virtual neurons are mapped to I/O ports, and actual neurons are mapped to NAs. In total, there are $(n_0 + n_x + n_1 + n_2 + n_3 ... + n_x)$ neurons (including the input and output virtual neurons) in VPR, so the array in the activity-criticality TXT file is a $(n_0 + n_x + n_1 + n_2 + n_3 ... + n_x) \times (n_0 + n_x + n_1 + n_2 + n_3 ... + n_x)$ square matrix.

It is easy for the VPR to find the values for every neuron in the array. Suppose A is the array in the activity-criticality TXT file, and the array size is $N \times N$, meaning there are N neurons (including virtual neurons) in the SNN architecture. We only focus on the neurons that need to be mapped to NAs, so the rows and lines that the indexes are among the inputs and outputs are all zeros. For a neuron with a block ID m, the value of $A[m][m]$ is stored with the criticality of this neuron. According to the analysis of VPR's mapping processes, the criticality decides the order of mapping this neuron into a new NA if it is a "seed" neuron according to the mapping method. VPR's packing will find the neurons with the largest criticality value among all feasible blocks and make it the "seed".

If there is a neuron with a ID n, then the value of $A[m][n]$ represents the relative spiking activities from the neuron with the ID m to the neuron with the block ID n, if n is larger than m. If n is smaller than m, this value means the relative spiking activities from the neuron with ID n to the neuron with ID m. This sounds as if the array is a symmetric matrix; for some locations, it is indeed symmetric. However, to apply the mapping method in VPR, the activity-criticality TXT generator changes the values. It makes the mapping method adaptable to VPR's packing, so the array is not symmetric. This means some values of the "relative spiking activities" are not the same as the original ones, but VPR can read this modified array and output a satisfying mapping result.

7.2 Activity-Criticality Generator

In the activity-criticality TXT generator, the inputs are the number of neurons in each layer and are stored in an array. Another important input is the number of spikes of each spiking neuron in the SNN architecture. Here we choose to count the number of spikes in a time period for a number of input signals, and this number of spikes can represent the average spiking activities of a neuron. For real SNN examples, the three steps for SNN mapping are applied.

According to **Step 1**, the neurons are ranked from the smallest number of spiking activities to the largest. And all the numbers are normalized to a number between 0 and 1. The sorted array is divided by the largest number of spikes. In the input array, the spiking activities are placed from largest to smallest, so the largest number of spikes is the first number. Using NumPy functions, the original position of each activity value can be found. As **Step 1** has described, the criticality of the parts of neurons in a layer that can be fully placed into a NA should be larger than the left unpacked neurons that need to fill in more other neurons. So their criticality is set up, and their lowest criticality should be larger than the unpacked neurons.

At the same time, the way of filling empty columns of NAs with neurons should be decided. For the NAs that can be mapped with neurons from the first layer, and even for the unpacked neurons in **Step 2**, after selecting a seed neuron, the neurons from the same layer should be mapped first. In calculating neuron gains, the neurons from the same layer have no direct connection with each other, so their connection gain is 0. Using this feature, the method of choosing neurons with the largest gain first is changed to choosing neurons with the smallest gain first. This change requires that the list used to store feasible neurons should be large enough to accommodate all feasible neurons and not miss any possible neurons.

Step 2 is critical for the evaluation of different mapping methods. There are possibly some unpacked neurons in the second hidden layer L_2. All the mapping methods for unpacked neurons in L_2 are presented, and their value of ${S_{L_i}}'$ is calculated. In the activity-criticality file generator, the deepest layer that the NA with unpacked neurons of L_2 is first found. After that, the values of ${S_{L_i}}'$ are calculated based on whether the NA is filled with neurons from L_1 in that method.

Step 3 is for the rest of the unpacked neurons. The unpacked neurons should be packed as much as possible if all neurons from the same layer can be mapped. This process starts from lower layers because the spiking activities are generally larger there. If there are empty columns, fill them with unpacked neurons from L_1.

After finishing this step, each method's values of $\sum_{hidden\ layers} S_{L_i}{}'$ can be calculated, and the method with the largest $\sum_{hidden\ layers} S_{L_i}{}'$ value will be chosen. It is because, in equation 6.2, the $\sum_{hidden\ layers} N_{L_{i+1}} \sum S_{L_i}(k)$ of all the methods are the same, so the largest $\sum_{hidden\ layers} S_{L_i}{}'$ value means a smallest A_{spike}.

After the decision on which mapping method has been made, the activity-criticality TXT generator will change some of the values in the input spiking activity array and build the final activity-criticality array A based on the modified activity array.

The first modification is associated with mapping unpacked neurons in L_2. The criticality of the deepest layer that the NA reaches is set with higher values. The direction of mapping is from the deepest layer neurons towards neurons in L_2 or even neurons in L_1. On the other hand, the spiking activities that travel from the neurons in the deepest layer to even deeper layers are set to 0 because we do not want VPR to find neurons in that direction when searching for all feasible neurons.

The second modification is about the order of adding neurons to the NA for the mapping of unpacked neurons in L_2. Here we explain how the connection gains for the spiking activities are calculated. The connection gain only exists if the two neurons are connected directly with each other. So the values on the corresponding places where neurons are not connected (for instance, neurons from the same layer) do not influence the mapping result. However, for the neurons that connect directly, the connection gain affects the result of mapping: the order of placing neurons onto the NA columns. The rule of finding neurons with the lowest gain still applies to those neurons. Because of this, we keep connection gain as the gain for a neuron and delete all the other gains.

In addition, the neurons with larger spiking activities should have lower connection gain so that the neuron will be picked and mapped before other neurons.

Using the method of choosing neurons with the smallest gain first, the neurons from the same layer can be searched first. For the neurons from the other layers, those with large spiking activities should be mapped first. So we hope VPR recognizes the neurons with large spiking activities as the neurons with small connection gains. The conversion formula for a neuron's spiking activities and its connection gain in VPR is as follows:

$$gain = 1 - activity \tag{7.1}$$

The activity has been normalized, so the value range is between 0 and 1. Surely, all values are positive between 0 and 1.

In the mapping of L_2, this feature could cause trouble when deeper layers are mapped into the NA because the neurons in L_1 with large activities will be picked even before the unpacked neurons from the same layer. To solve this problem, modifications have been made to the value of spiking activities of neurons in that NA. The values of the spiking activities of the neurons in L_1 are reduced so that the packer picks the unpacked neurons from the same layer first. Here we should make sure that the values of reduced activities are also between 0 and 1.

After the modifications, the mapping solution of unpacked neurons in L_2 finally gets the right result.

To map the rest of the unpacked neurons in **Step 3**, there are some controls over finding feasible neurons. We have mentioned that for mapping SNN architecture to NAs, there are two ways of finding feasible neurons:

- 1) Find the unpacked neurons that have direct activity connections with the current neuron or the neurons that are in the same layer;
- 2) Find the unpacked neurons that have no direct connection with the current neuron but there are transitive connections. For example, there may be some other neurons between them.

The packer first finds feasible unpacked neurons in 1), and if no neurons are available, it will search on a larger scale in 2).

The packer first finds feasible unpacked neurons in 1), and if no neurons are available, it will search on a larger scale in 2). For the rest of the unpacked neurons in **Step 3**, the neurons in adjacent layers are packed first, and then it will pack neurons in non-adjacent layers. If the rest of the unpacked neurons are mapped together in one NA, then only 1) will be used to search feasible neurons. However, if the neurons cannot be packed together, we need to use 2) to add neurons from the first hidden layer to fill in the empty columns. The control of using 2) can be done using a control signal for 2). As we have designed before, we know the index of the CLB where the neurons in **Step 3** will be mapped. In our design, the neurons mapped first are those in **Step 1** and **Step 2**. So the number of CLBs (Neuromorphic Arrays) used in **Step 1** and **Step 2** can be calculated. The index of CLB that will be used for neurons in **Step 3** follows them. Through the control over the use of 2), the mapping for the rest of the neurons in **Step 3** is finished.

Results and Discussions 8

This chapter presents the mapping results of Spiking Neural Networks to Neuromorphic Arrays using VPR.

8.1 Mapping of small Spiking Neural Networks

Some self-defined small SNNs are used for the testing experiment. The number of columns in the Neuromorphic Arrays is 4. In each of the networks, alternatives for mapping solutions are created. The choice of a solution depends on the value of A_{spike} and $\sum_{hidden\ layers} S_{L_i}{}'$. The Activity-Criticality file generator creates the TXT file based on the values. If the method has the largest $\sum_{hidden\ layers} S_{L_i}{}'$, meaning the value of A_{spike} is the smallest, that method will be chosen, and the corresponding TXT file will be created. In the following small network examples, all the methods have the possibility of being selected as the best solution. So the mapping outputs for each of them are presented.

8.1.1 An $18 \times 15 \times 13 \times 9 \times 5$ network

In this example, $(15 + 13 + 9 + 5) = 42$ neurons will be mapped onto NAs. Each NA can accommodate at most 4 neurons, so there will be at least 11 NAs.

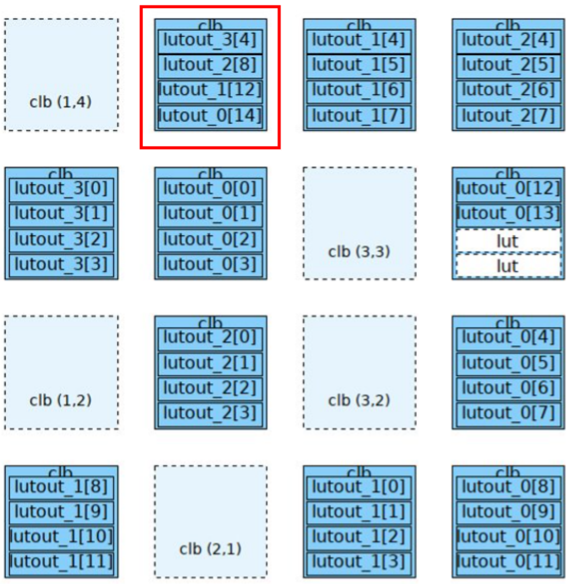


Figure 8.1: The mapping result of $18 \times 15 \times 13 \times 9 \times 5$: option 0

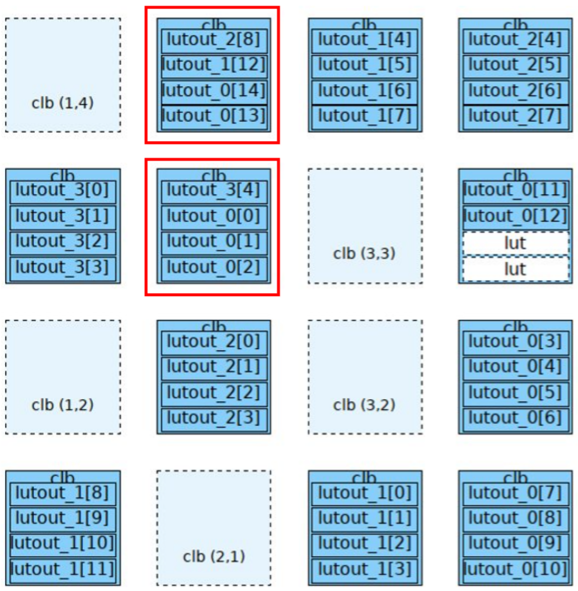


Figure 8.2: The mapping result of 18 × 15 × 13 × 9 × 5: option 1

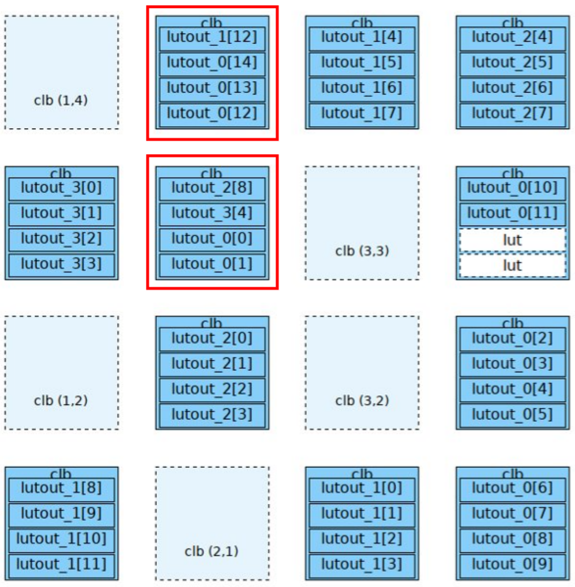


Figure 8.3: The mapping result of 18 × 15 × 13 × 9 × 5: option 2

The figures show how neurons are mapped to NAs. Each NA is represented by a square, and "clb" suggests the name of NAs in VPR. There are 4 rectangles in each clb square, representing 4 columns of the NAs. To observe the mapping of neurons, the pin mapping result is canceled in the graphs.

Figure 8.1 shows the mapping result of $18 \times 15 \times 13 \times 9 \times 5$ option 0, which means the unpacked neurons in L_2 are mapped with the deepest layer that the NA could reach. So the neurons *lutout*_3[4], *lutout*_2[8], and *lutout*_1[12] are mapped together in a NA, and a neuron *lutout_*0[14] is added to fill the empty column. In the other NAs, they are mapped with neurons from the same layer. So the number of NAs used by each layer is minimized.

Figure 8.2 is the mapping result of option 1, which means the unpacked neurons in L_2 are mapped with ($the\ deepest - 1$) layer, so *lutout_*1[12] is only mapped with *lutout_*2[8] and two other neurons in the first hidden layer are filled into that NA. The rest of the unpacked neuron, *lutout_*3[4], is mapped together with three neurons in the first hidden layer in a NA.

The option N for mapping means the unpacked neurons in L_2 are mapped with ($the\ deepest - N$) layer.

Figure 8.3 shows *lutout_*1[12] only mapped with neurons from the first hidden layer, so *lutout_*0[14], *lutout_*0[13], and *lutout_*0[12] are added together. Moreover, to maximize $\sum_{hidden\ layers} {S_{L_i}}'$, the two unpacked neurons, *lutout_*3[4] and *lutout_*2[8] can be mapped together. So they are mapped together with another two neurons from the first hidden layer.

The difference between these mapping methods is the mapping of unpacked neurons in each layer, after mapping neurons in the same layer in as few NAs as possible.

According to function 6.2, the total amount of spike activities traversing between NAs is calculated by the total amount of spike activities in the network minus the activities that traverse between neurons inside one NA. For all the mapping options 1,2, and 3 in this example, the number of NAs that the neurons in each layer are mapped to is minimized. The neurons in L_2, L_3, and L_4 are mapped to 4, 3, and 2 NAs respectively. So the total amount of spike activities is the same for the three mapping options. The difference is the spike activities inside each NA.

In figure 8.1, the activities inside a NA are the spikes of signals *out_*0[14], *out_*1[12], and *out_*2[8] to the neurons *lutout_*1[12], *lutout_*2[8], and *lutout_*3[4], respectively.

In figure 8.2, the activities inside a NA are the spikes of signals *out_*0[14], *out_*0[13], and *out_*1[12] to the neurons *lutout_*1[12] and *lutout_*2[8].

In figure 8.3, the activities inside a NA are the spikes of signals *out_*0[14], *out_*0[13], and *out_*0[12] to the neuron *lutout_*1[12], together with *out_*2[8] to neuron *lutout_*3[4] because *lutout_*2[8] is mapped together with *lutout_*3[4].

The activity-criticality file generator can calculate the values of the activities inside one NA with unpacked neurons in L_2 above. By adding the activities inside one NA with more unpacked neurons, the generator will choose the option with the highest value of activities inside a NA, so the amount of activities between different NAs is the lowest in that mapping option. The corresponding activity-criticality TXT file will be generated, so VPR shows the result of the mapping method according to the file.

In the above example, it is very likely that option 2 shown in figure 8.3 has the lowest spiking activities between NAs, because part of the spikes of 3 neurons in L_1 are inside one NA, and the spikes of *lutout_*2[8] are also inside one NA for the neuron *lutout_*3[4].

8.1.2 An $18 \times 16 \times 14 \times 10 \times 6$ network

In this example, $(16 + 14 + 10 + 6) = 46$ neurons will be mapped onto NAs. Each NA can accommodate at most 4 neurons, so there will be at least 12 NAs.

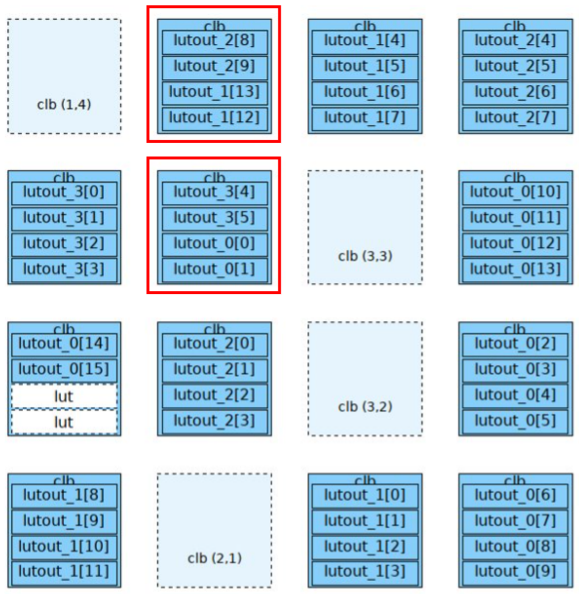


Figure 8.4: The mapping result of 18 × 16 × 14 × 10 × 6: option 0

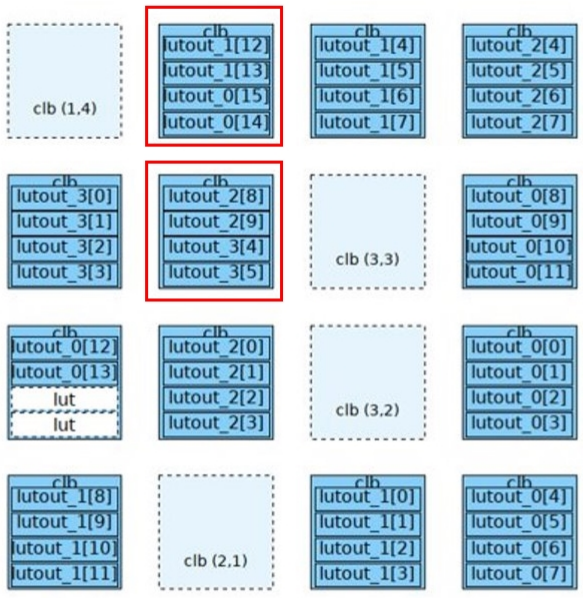


Figure 8.5: The mapping result of 18 × 16 × 14 × 10 × 6: option 1

Figure 8.4 shows the mapping result that unpacked neurons in L_2 and L_3, $lutout\_2$[8], $lutout\_2$[9], $lutout\_1$[13], $lutout\_1$[12] are mapped together. The NA is full so that no other neurons can be filled in. However, in the former example of 8.1.1, some unpacked neurons in L_1 are needed to be mapped into the empty columns. The unpacked neurons in L_4, $lutout\_3$[4] and $lutout\_3$[5] are filled with two other neurons $lutout\_0$[0], $lutout\_0$[1] from L_1.

Figure 8.5 is the mapping result that unpacked neurons in L_2, $lutout\_1$[12] and

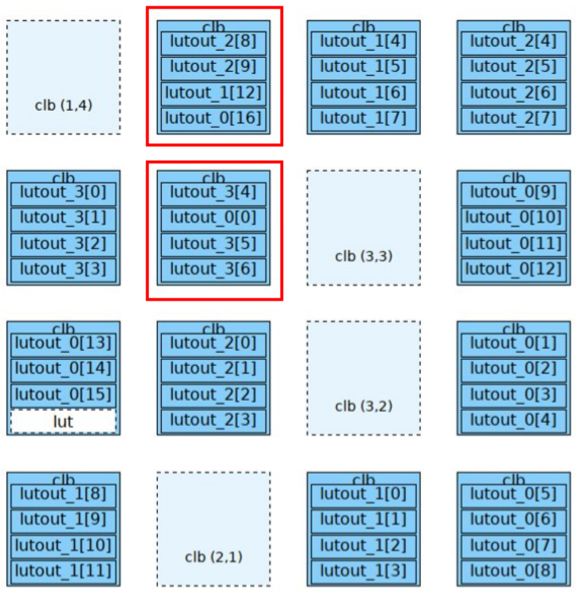


Figure 8.6: The mapping result of $18 \times 17 \times 13 \times 10 \times 7$: option 0

$lutout\_1$[13] are mapped directly with two neurons $lutout\_0$[15] and $lutout\_0$[14] in L_1. The rest of the unpacked neurons in L_3 and L_4 are mapped together to maximize $\sum_{hidden\ layers} S_{L_i}{}'$.

In figure 8.4, the activities inside a NA are the spikes of signals out_1[12] and out_1[13] to the neurons $lutout\_2$[8] and $lutout\_2$[9].

In figure 8.5, the activities inside a NA are the spikes of signals out_0[14], out_0[15] to the neurons $lutout\_1$[12] and $lutout\_1$[13], together with spikes of out_2[8] and out_2[9] to neurons $lutout\_3$[4] and $lutout\_3$[5], because the neurons are mapped together in one NA.

The result of mapping depends on which option has the lowest spike activities between different NAs. For the two options, the neurons in L_2, L_3, and L_4 are mapped to 4, 3, and 2 NAs respectively. So the amount of total spike activities is the same. According to function 6.2, the activity-criticality file generator chooses the method with the largest activities inside one NA as the optimized solution.

8.1.3 An $18 \times 17 \times 13 \times 10 \times 7$ network

In this example, $(17 + 13 + 10 + 7) = 47$ neurons will be mapped onto NAs. Each NA can accommodate at most 4 neurons, so there will be at least 12 NAs.

Figure 8.6 shows that the unpacked neuron in L_2 is mapped with two unpacked neurons in L_3 in a NA. One neuron from L_1 is filled into the empty column.

Figure 8.7 is the mapping result that the unpacked neuron in L_2 is mapped directly with three neurons from L_1. The rest of the unpacked neurons cannot fill into one NA, so each group is filled with neurons from L_1.

In figure 8.6, the activities inside a NA are the spikes of signals out_0[16] and out_1[12] to the neurons $lutout\_2$[8] and $lutout\_2$[9].

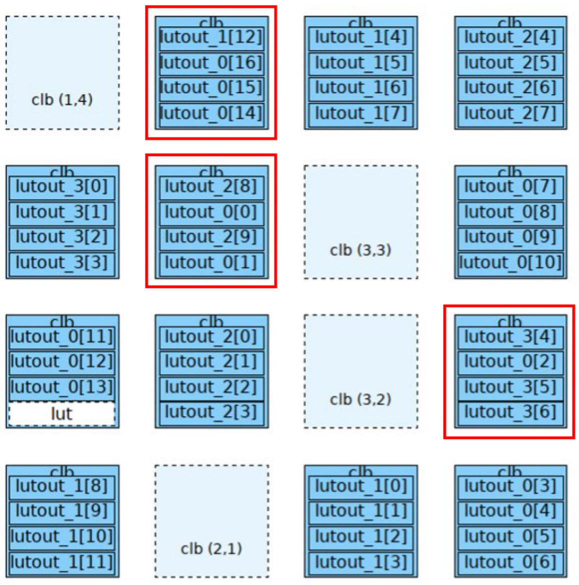


Figure 8.7: The mapping result of $18 \times 17 \times 13 \times 10 \times 7$: option 1

In figure 8.7, the activities inside a NA are the spikes of signals $out\_0[16]$, $out\_0[15]$ and $out\_0[14]$ to the neuron $lutout\_1[12]$.

According to function 6.2, the total spiking activities of the options are the same, so the mapping option with the highest spiking activities inside a NA is the optimized solution with the lowest spiking activities between NAs.

8.2 Mapping of a 3-layer MNIST Spiking Neural Network

The SystemC simulator has a trained $196 \times 50 \times 10$ Spiking Neural Network with the number of spikes for each neuron. The target NA has 256 input pins and 15 columns. So these input pin resources are enough to map neurons from 3 different layers. The number of spikes is counted using a SystemC counter module. In the activity-criticality TXT file generator, the numbers of spikes are normalized to numbers between 0 and 1. The neurons are reordered from a small number of spikes to larger ones. Using NumPy functions, the original positions of the neurons can be found. The spike activities of 50 neurons in the hidden layer are critical.

In this example, $(50 + 10) = 60$ neurons will be mapped onto NAs. Each NA can accommodate at most 15 neurons, so there will be 4 NAs at least.

As figure 8.8 shows, the 10 neurons in L_2 in the red rectangle are mapped together with 5 neurons from L_1, to maximize $\sum_{hidden\ layers} S_{L_i}{'}$. This is the only mapping method made by the activity-criticality file generator according to function 6.2

In figure 8.8, the activities inside a NA are the spikes of signals $out\_0[45]$, $out\_0[46]$, $out\_0[47]$, $out\_0[48]$ and $out\_0[49]$ to the 10 neurons from L_2. $out\_0[45]$, $out\_0[46]$, $out\_0[47]$, $out\_0[48]$ and $out\_0[49]$ are the signals with largest spiking activities, so this method has the lowest amount of spiking activities between neurons.

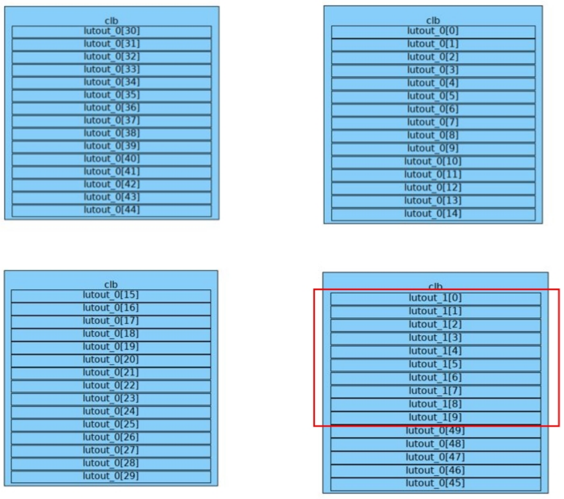


Figure 8.8: The mapping result of $196 \times 50 \times 10$

8.3 Mapping of large Spiking Neural Networks

There are large SNNs with more neurons in each layer, so there could be situations where the number of neurons in layers exceeds the total number of inputs of the Neuromorphic Array. In this situation, some modifications are made to solve the problem of insufficient input pin resources.

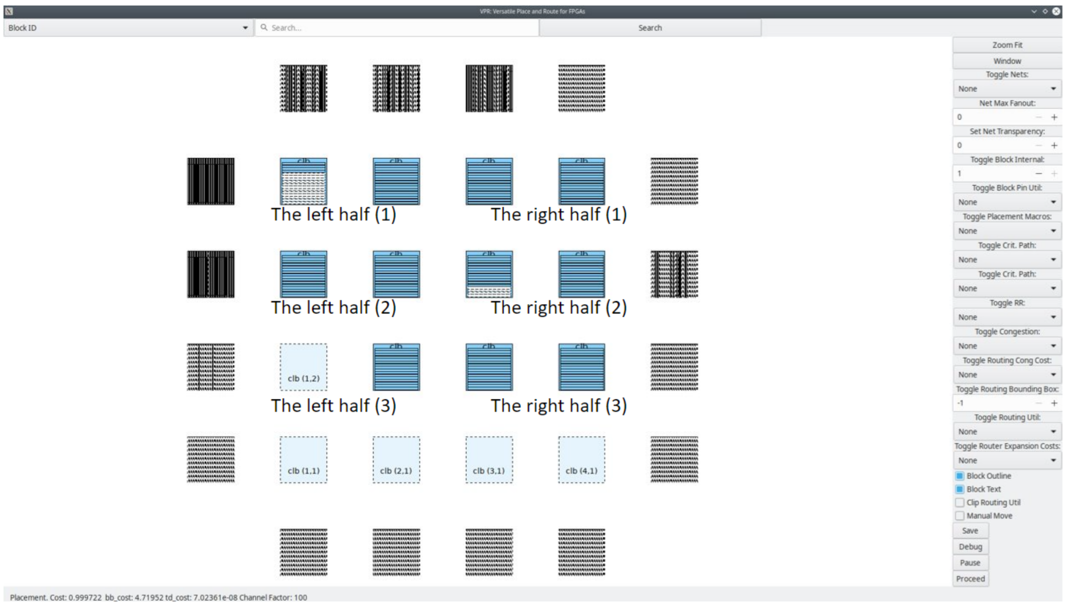
Firstly, according to the possible methods of the unpacked neurons in the second hidden layer (L_2), the option is chosen if the NA input pin resources are enough for the SNN.

In our example, where a $196 \times 100 \times 50 \times 10$ SNN is to be mapped onto a NA with 256 inputs and 16 columns, the neurons in the first hidden layer cannot be mapped with the neurons from the second hidden layer. In this example, the option of mapping we choose for unpacked neurons in L_2 is to map as many neurons as possible with the unpacked neurons in L_3. The modification is that we increase the criticality for mapping the unpacked neurons in L_3 so that some other neurons in L_2 occupy the columns that used to belong to neurons in L_1, which increases the total spikes that happen inside a NA. Without this modification, the columns for L_1 will stay empty. If the empty columns are placed with more neurons in L_2, the spike activities inside a NA will increase.

In this example, $(100 + 50 + 10) = 60$ neurons will be mapped onto NAs. Each NA can accommodate at most 16 neurons. However, the constraints of input pins make the neurons from L_1 unable to be mapped together with neurons from L_2. After the criticalities of the unpacked neurons are modified, a new solution based on the constraints is generated.

Figure 8.9 the output window of VPR. Figure 8.10 shows all the NAs and signals of the mapping of the $196 \times 100 \times 50 \times 10$ network. To have a clearer understanding, we will only show the mapping of neurons without the pin mapping results.

Figure 8.11, 8.12, 8.13 are the enlarged view of the left half of figure 8.10. Figure



Toggle Nets: None
Net Max Fanout: 0
Set Net Transparency: 0
Toggle Block Internal: 1
Toggle Block Pin Util: None
Toggle Placement Macros: None
Toggle Crit. Path: None
Toggle Crit. Path: None
Toggle RR: None
Toggle Congestion: None
Toggle Routing Cong Cost: None
Toggle Routing Bounding Box: -1
Toggle Routing Util: None
Toggle Router Expansion Costs: None
- [x] Block Outline
- [x] Block Text
- [ ] Clip Routing Util
- [ ] Manual Move

```
Placement. Cost: 0.999722  bb_cost: 4.71952 td_cost: 7.02361e-08 Channel Factor: 100
```

Figure 8.9: The mapping result of 196 × 100 × 50 × 10: VPR output window

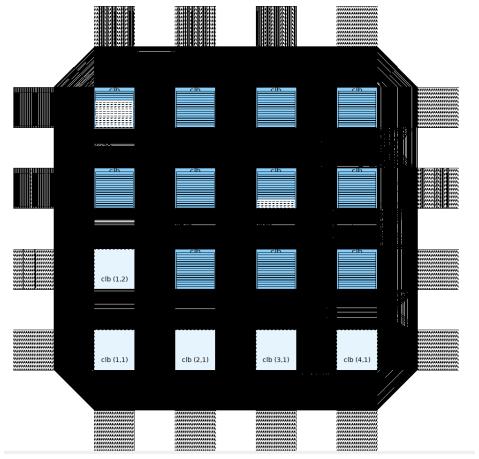


Figure 8.10: The mapping result of 196 × 100 × 50 × 10: overview

8.14, 8.15, 8.16 are the enlarged view of the right half of figure 8.10. After the modification, the unpacked neurons in L_3 have higher criticalities to pack as many neurons

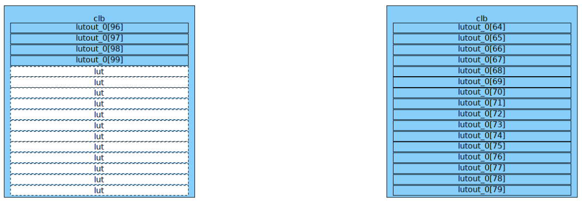


Figure 8.11: The mapping result of 196 × 100 × 50 × 10: the left half (1)

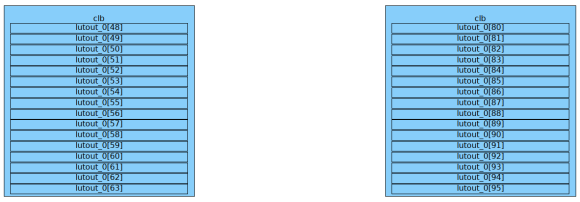


Figure 8.12: The mapping result of 196 × 100 × 50 × 10: the left half (2)

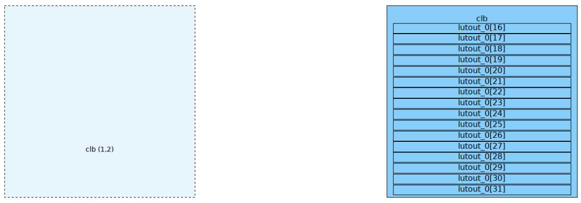


Figure 8.13: The mapping result of 196 × 100 × 50 × 10: the left half (3)

in L_2 as possible to maximize $\sum_{hidden\ layers} {S_{L_i}}'$ and minimize A_{spike}.

The number of input pins in the NA is 256, so the neurons in L_1 and L_2 cannot be mapped together because of the lack of input resources. Though neurons in L_1 can be mapped with neurons in other layers, there is no contribution to the activities inside a NA. The optimization method is still based on function 6.2.

Each NA has 16 columns. According to the Implementation method in chapter 6, in **Step 1**, the neurons in the same layer are mapped to as few NAs as possible. The architecture of the SNN is 196 × 100 × 50 × 10, so there are 2 unpacked neurons in L_2 and 10 unpacked neurons in L_3. This is the result of **Step 1** if there are enough input pins. In this case, the unpacked neurons in L_2 and L_3 are mapped together, and neurons from L_1 fill in the empty columns. However, due to limited input pins on the NA, the neurons from L_1 cannot be mapped together with the unpacked neurons of L_2 and L_3. In this situation, a modification on mapping criticality and activity array is made to map more neurons in L_2 with the unpacked neurons in L_3. The number of NAs that neurons are mapped to is not changed in the modification, so the total spike activities in the system are not changed.

Without the modification, the 10 neurons in L_3 are mapped with 2 neurons from L_2, and 4 empty columns would have been mapped with neurons in L_1 if there are enough input pins. The spike activities inside the NA are the sum of spikes of the 2

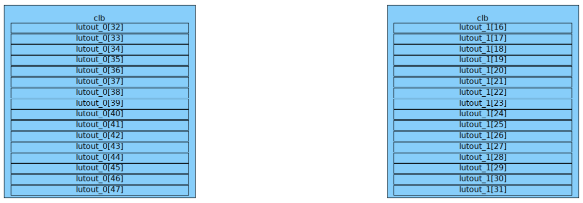


Figure 8.14: The mapping result of 196 × 100 × 50 × 10: the right half (1)

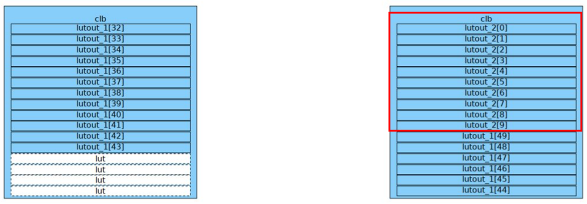


Figure 8.15: The mapping result of 196 × 100 × 50 × 10: the right half (2)

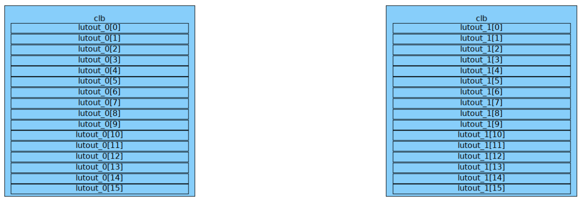


Figure 8.16: The mapping result of 196 × 100 × 50 × 10: the right half (3)

neurons from L_2.

After the modification, the 10 neurons in L_3 are mapped with 6 neurons from L_2. There is no empty column, and the spike activities inside the NA is the sum of spike activities of the 6 neurons from L_2, which is larger than the activity before modification. According to function 6.2, the total amount of spike activities between NAs is smaller than the former method. And this is the optimized solution for the SNN and NAs without enough input pin resources. The NA with the 10 neurons from L_3 and 6 neurons from L_2 is in figure 8.15.

Conclusion and Future Work 9

9.1 Conclusion

The mapping of Spiking Neural Network architectures has been discussed in the thesis. The design flow is based on the VPR, which conducts the mapping tasks according to the input files.

The analysis of the mapping problem starts with mapping signals and input pins of the Neuromorphic Array. In solving pin mapping problems, the understanding of VPR is deepened. Then the mapping of primitive blocks is analyzed. These are the basic knowledge needed for VPR usage.

Among all the SNN architecture parameters, neurons' spiking activities are used to evaluate different mapping methods. Learning from some existing mapping solutions for neuromorphic processors, the optimization goal is to minimize the spike activities between different Neuromorphic Arrays. This includes two aspects:

1) to use as few Neuromorphic Arrays as possible;

2) to maximize the spike activities inside a NA.

An implementation method is presented according to the optimization goal.

To apply the implementation in VPR flow, an activity-criticality generator is created to print the input files of an optimized mapping method. Since there are different sizes of SNNs and NAs, modifications are made for different situations.

The answers to the questions in Section 1.1 are as follows:

- In the thesis, we use a hardware device model with I/O ports and Neuromorphic Arrays. A Neuromorphic Array consists of columns, where inputs of a neuron are processed, and an output is generated. Therefore, a neuron in the SNN can be mapped onto a column. The number of columns in a Neuromorphic Array is the maximum number of neurons that can be mapped to it.
- To map the SNN to NAs, firstly, which groups of neurons are to be mapped together in one NA is considered. Secondly, the input of mapped neurons should match the input pins of the NA.
- To use VPR in SNN mapping, the architecture description file (XML file) and the netlist description file (BLIF file) are used for the hardware platform with Neuromorphic Arrays and the Spiking Neural Network architecture, respectively. In the XML file, a CLB is a Neuromorphic Array. The BLEs, which are LUTs, are columns in the Neuromorphic Array. The clusters of primitive logic blocks (molecules) are the neurons in the SNN. In addition, an Activity-Criticality file, which is the optimized mapping solution, is added to the inputs of VPR.
- The modifications in VPR are mainly in two aspects:

1) An activity-criticality TXT file is added to the VPR flow. To get the optimized result, VPR maps the neurons in the same layer together to use as few NAs as possible. For the unpacked neurons, VPR tries to map them so that spike activities inside a Neuromorphic Array are maximized.

2) For mapping inputs of neurons to input pins of NAs, the inputs for one neuron are densely mapped together in consecutive pins instead of using the routing algorithm for FPGA.

- For different mapping solutions, the amount of spike activities transmitted from a neuron to other neurons in other Neuromorphic Arrays is different. The spike activities between different NAs pass through interconnect hardware components, influencing the system throughput. The mapping solution with the lowest spike activities between NAs (and the largest spike activities inside NAs) has the fewest spikes in the interconnect, causing the lowest congestion, so it is the best mapping solution.

9.2 Future Work

The thesis is just a minor step toward mapping Spiking Neural Networks on Neuromorphic Arrays. The directions for future improvement are as follows.

- Currently, the Spiking Neural networks are only basic feed-forward architectures in which input signals pass through layers, and the output is generated. There are more kinds of Spiking Neural Networks with more complex architectures, for example, a recurrent Spiking Neural Network with feedback synapses. The network architecture is more complex.

- The Neuromorphic Arrays are just simple models in the thesis, without detailed parameters about the delays and performances. If the structure of the hardware system is clearer with more hardware components, these components can also be defined in the XML file. The mapping would be more complex. Inside VPR, more information can be analyzed about the implementation, including the usage of resources, the timing paths, and the power estimation of the hardware device. The other stages of VPR, the placement, and routing, are not fully discovered. With more detailed hardware descriptions, the placement and routing can also be analyzed and utilized.

File Examples

A

This appendix demonstrates some examples of files used in VPR.

A.1 The BLIF file for a $18 \times 15 \times 13 \times 9 \times 5$ SNN

A.1.1 The BLIF file

```
.model top
.inputs i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17
.outputs out_3[0] out_3[1] out_3[2] out_3[3] out_3[4]

.names unconn
0



.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[0]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[1]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[2]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[3]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[4]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[5]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[6]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[7]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[8]
111111111111111111 1
```

```
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[9]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[10]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[11]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[12]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[13]
111111111111111111 1
.names i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14
   i_15 i_16 i_17 out_0[14]
111111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[0]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[1]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[2]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[3]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[4]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[5]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[6]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[7]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
   out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
   out_0[14] out_1[8]
```

```
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
    out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
    out_0[14] out_1[9]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
    out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
    out_0[14] out_1[10]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
    out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
    out_0[14] out_1[11]
111111111111111 1
.names out_0[0] out_0[1] out_0[2] out_0[3] out_0[4] out_0[5] out_0[6]
    out_0[7] out_0[8] out_0[9] out_0[10] out_0[11] out_0[12] out_0[13]
    out_0[14] out_1[12]
111111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[0]
1111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[1]
1111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[2]
1111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[3]
1111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[4]
1111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[5]
1111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[6]
1111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[7]
1111111111111 1
.names out_1[0] out_1[1] out_1[2] out_1[3] out_1[4] out_1[5] out_1[6]
    out_1[7] out_1[8] out_1[9] out_1[10] out_1[11] out_1[12] out_2[8]
1111111111111 1
.names out_2[0] out_2[1] out_2[2] out_2[3] out_2[4] out_2[5] out_2[6]
    out_2[7] out_2[8] out_3[0]
111111111 1
.names out_2[0] out_2[1] out_2[2] out_2[3] out_2[4] out_2[5] out_2[6]
    out_2[7] out_2[8] out_3[1]
111111111 1
.names out_2[0] out_2[1] out_2[2] out_2[3] out_2[4] out_2[5] out_2[6]
    out_2[7] out_2[8] out_3[2]
111111111 1
```

```
.names out_2[0] out_2[1] out_2[2] out_2[3] out_2[4] out_2[5] out_2[6]
   out_2[7] out_2[8] out_3[3]
111111111 1
.names out_2[0] out_2[1] out_2[2] out_2[3] out_2[4] out_2[5] out_2[6]
   out_2[7] out_2[8] out_3[4]
111111111 1

.end
```

A.1.2 The python script

This python script is used to generate the BLIF file

```python
import math

layer0 = 18
layer1 = 15
layer2 = 13
layer3 = 9
layer4 = 5


with open("test_18x17x13x10x7@x4_crossbaronly.txt","w") as f:
    f.write(".model top\n")
    f.write(".inputs")
    for i in range(layer0):
                f.write(" i_"+str(i))
    f.write("\n.outputs")
    for i in range(layer4):
                f.write(" out_3["+str(i)+"]")
    f.write("\n\n")
    f.write(".names unconn\n0\n\n")


    f.write("\n\n")

    #crossbars
    for n in range(layer1):
        f.write(".names ")
        for i in range(layer0):
                f.write("i_"+str(i)+" ")
        f.write("out_0["+str(n)+"]\n")
        for m in range(layer0):
            f.write("1")
        f.write(" 1")
        f.write("\n")
    for n in range(layer2):
        f.write(".names ")
        for i in range(layer1):
                f.write("out_0["+str(i)+"] ")
        f.write("out_1["+str(n)+"]\n")
        for m in range(layer1):
            f.write("1")
        f.write(" 1")
        f.write("\n")
    for n in range(layer3):
        f.write(".names ")
        for i in range(layer2):
                f.write("out_1["+str(i)+"] ")
        f.write("out_2["+str(n)+"]\n")
        for m in range(layer2):
            f.write("1")
        f.write(" 1")
        f.write("\n")
    for n in range(layer4):
        f.write(".names ")
        for i in range(layer3):
                f.write("out_2["+str(i)+"] ")
        f.write("out_3["+str(n)+"]\n")
        for m in range(layer3):
            f.write("1")
        f.write(" 1")
        f.write("\n")
    f.write("\n.end\n\n")
```

A.2 The XML file for a hardware platform with NAs that have 10 inputs and 3 columns

```xml
<architecture>
  <models>
  </models>
  <tiles>
    <tile name="io" area="0">
      <sub_tile name="io" capacity="4">
        <equivalent_sites>
          <site pb_type="io" pin_mapping="direct"/>
        </equivalent_sites>
        <input name="outpad" num_pins="1"/>
        <output name="inpad" num_pins="1"/>
        <fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
        <pinlocations pattern="custom">
          <loc side="left">io.outpad io.inpad</loc>
          <loc side="top">io.outpad io.inpad</loc>
          <loc side="right">io.outpad io.inpad</loc>
          <loc side="bottom">io.outpad io.inpad</loc>
        </pinlocations>
      </sub_tile>
    </tile>
    <tile name="clb" area="53894">
      <sub_tile name="clb">
        <equivalent_sites>
          <site pb_type="clb" pin_mapping="direct"/>
        </equivalent_sites>
        <input name="I" num_pins="10" equivalent="none"/>
      <output name="O" num_pins="3" equivalent="none"/>
      <fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
      <pinlocations pattern="custom">
        <loc side="left">clb.I</loc>
        <loc side="top">clb.O</loc>
      </pinlocations>
    </sub_tile>
  </tile>
</tiles>
<layout>
  <auto_layout aspect_ratio="1.0">
    <!--Perimeter of 'io' blocks with 'EMPTY' blocks at corners-->
    <perimeter type="io" priority="100"/>
    <corners type="EMPTY" priority="101"/>
    <!--Fill with 'clb'-->
    <fill type="clb" priority="10"/>

  </auto_layout>
</layout>
<device>
  <sizing R_minW_nmos="8926" R_minW_pmos="16067"/>
  <area grid_logic_tile_area="0"/>
  <chan_width_distr>
    <x distr="uniform" peak="1.000000"/>
    <y distr="uniform" peak="1.000000"/>
  </chan_width_distr>
  <switch_block type="wilton" fs="3"/>
  <connection_block input_switch_name="ipin_cblock"/>
</device>
<switchlist>
  <switch type="mux" name="0" R="551" Cin=".77e-15" Cout="4e-15" Tdel="
     58e-12" mux_trans_size="2.630740" buf_size="27.645901"/>
  <!--switch ipin_cblock resistance set to yeild for 4x minimum drive
     strength buffer-->
  <switch type="mux" name="ipin_cblock" R="2231.5" Cout="0." Cin="1.47e
     -15" Tdel="7.247000e-11" mux_trans_size="1.222260" buf_size="auto"
     />
</switchlist>
<segmentlist>
  <segment freq="1.000000" length="4" type="unidir" Rmetal="101" Cmetal
     ="22.5e-15">
    <mux name="0"/>
    <sb type="pattern">1 1 1 1 1</sb>
    <cb type="pattern">1 1 1 1</cb>
  </segment>
</segmentlist>
<complexblocklist>
  <!-- Define I/O pads begin -->
  <!-- Capacity is a unique property of I/Os, it is the maximum number
     of I/Os that can be placed at the same (X,Y) location on the FPGA
     -->
  <!-- Not sure of the area of an I/O (varies widely), and it's not
     relevant to the design of the FPGA core, so we're setting it to 0.
```

```
        -->
    <pb_type name="io">
      <input name="outpad" num_pins="1"/>
      <output name="inpad" num_pins="1"/>
      <!-- IOs can operate as either inputs or outputs.
         Delays below come from Ian Kuon. They are small, so they should
             be interpreted as
         the delays to and from registers in the I/O (and generally I/Os
             are registered
         today and that is when you timing analyze them.
         -->
      <mode name="inpad">
        <pb_type name="inpad" blif_model=".input" num_pb="1">
          <output name="inpad" num_pins="1"/>
        </pb_type>
        <interconnect>
          <direct name="inpad" input="inpad.inpad" output="io.inpad">
            <delay_constant max="4.243e-11" in_port="inpad.inpad"
                out_port="io.inpad"/>
          </direct>
        </interconnect>
      </mode>
      <mode name="outpad">
        <pb_type name="outpad" blif_model=".output" num_pb="1">
          <input name="outpad" num_pins="1"/>
        </pb_type>
        <interconnect>
          <direct name="outpad" input="io.outpad" output="outpad.outpad">
            <delay_constant max="1.394e-11" in_port="io.outpad" out_port=
                "outpad.outpad"/>
          </direct>
        </interconnect>
      </mode>
      <power method="ignore"/>
    </pb_type>
    <pb_type name="clb">
      <input name="I" num_pins="10" equivalent="none"/><!--none or full--
         >
      <output name="O" num_pins="3" equivalent="none"/><!--none or
         instance-->
          <!-- Define LUT -->
      <pb_type name="lut" blif_model=".names" num_pb="3" class="lut">
        <input name="in" num_pins="10" port_class="lut_in"/>
        <output name="out" num_pins="1" port_class="lut_out"/>
          <delay_matrix type="max" in_port="lut.in" out_port="lut.out">
            261e-12
            261e-12
            261e-12
            261e-12
            261e-12
            261e-12
            261e-12
            261e-12
        261e-12
        261e-12
      </delay_matrix>
    </pb_type>
    <interconnect>
      <!--increasing delays for the inputs of the crossbar-->
      <complete name="crossbar" input="clb.I" output="lut[2:0].in">
        <delay_constant max="20e-12" in_port="clb.I[0]" out_port="lut[0].in"/>
        <delay_constant max="25e-12" in_port="clb.I[0]" out_port="lut[1].in"/>
        <delay_constant max="30e-12" in_port="clb.I[0]" out_port="lut[2].in"/>
        <delay_constant max="40e-12" in_port="clb.I[1]" out_port="lut[0].in"/>
        <delay_constant max="45e-12" in_port="clb.I[1]" out_port="lut[1].in"/>
        <delay_constant max="50e-12" in_port="clb.I[1]" out_port="lut[2].in"/>
        <delay_constant max="60e-12" in_port="clb.I[2]" out_port="lut[0].in"/>
        <delay_constant max="65e-12" in_port="clb.I[2]" out_port="lut[1].in"/>
        <delay_constant max="70e-12" in_port="clb.I[2]" out_port="lut[2].in"/>
        <delay_constant max="80e-12" in_port="clb.I[3]" out_port="lut[0].in"/>
        <delay_constant max="85e-12" in_port="clb.I[3]" out_port="lut[1].in"/>
        <delay_constant max="90e-12" in_port="clb.I[3]" out_port="lut[2].in"/>
        <delay_constant max="100e-12" in_port="clb.I[4]" out_port="lut[0].in"/>
        <delay_constant max="105e-12" in_port="clb.I[4]" out_port="lut[1].in"/>
        <delay_constant max="110e-12" in_port="clb.I[4]" out_port="lut[2].in"/>
        <delay_constant max="120e-12" in_port="clb.I[5]" out_port="lut[0].in"/>
        <delay_constant max="125e-12" in_port="clb.I[5]" out_port="lut[1].in"/>
        <delay_constant max="130e-12" in_port="clb.I[5]" out_port="lut[2].in"/>
        <delay_constant max="140e-12" in_port="clb.I[6]" out_port="lut[0].in"/>
        <delay_constant max="145e-12" in_port="clb.I[6]" out_port="lut[1].in"/>
        <delay_constant max="150e-12" in_port="clb.I[6]" out_port="lut[2].in"/>
        <delay_constant max="160e-12" in_port="clb.I[7]" out_port="lut[0].in"/>
        <delay_constant max="165e-12" in_port="clb.I[7]" out_port="lut[1].in"/>
```

```
          <delay_constant max="170e-12" in_port="clb.I[7]" out_port="lut
            [2].in"/>
          <delay_constant max="180e-12" in_port="clb.I[8]" out_port="lut
            [0].in"/>
          <delay_constant max="185e-12" in_port="clb.I[8]" out_port="lut
            [1].in"/>
          <delay_constant max="190e-12" in_port="clb.I[8]" out_port="lut
            [2].in"/>
          <delay_constant max="200e-12" in_port="clb.I[9]" out_port="lut
            [0].in"/>
          <delay_constant max="205e-12" in_port="clb.I[9]" out_port="lut
            [1].in"/>
          <delay_constant max="210e-12" in_port="clb.I[9]" out_port="lut
            [2].in"/>
        </complete>
        <direct name="clbouts1" input="lut[0].out" output="clb.O[0]"/>
        <direct name="clbouts2" input="lut[1].out" output="clb.O[1]"/>
        <direct name="clbouts3" input="lut[2].out" output="clb.O[2]"/>
      </interconnect>
    </pb_type>
    <!-- Define general purpose logic block (CLB) ends -->
  </complexblocklist>
  <power>
    <local_interconnect C_wire="2.5e-10"/>
    <mux_transistor_size mux_transistor_size="3"/>
    <FF_size FF_size="4"/>
    <LUT_transistor_size LUT_transistor_size="4"/>
  </power>
  <clocks>
    <clock buffer_size="auto" C_wire="2.5e-10"/>
  </clocks>
</architecture>
```

Bibliography

[1] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. ElDafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "Vtr 8: High performance cad and customizable fpga architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, 2020.

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

[3] M. V. DeBole, B. Taba, A. Amir, F. Akopyan, A. Andreopoulos, W. P. Risk, J. Kusnitz, C. Ortega Otero, T. K. Nayak, R. Appuswamy, P. J. Carlson, A. S. Cassidy, P. Datta, S. K. Esser, G. J. Garreau, K. L. Holland, S. Lekuch, M. Mastro, J. McKinstry, C. di Nolfo, B. Paulovicks, J. Sawada, K. Schleupen, B. G. Shaw, J. L. Klamo, M. D. Flickner, J. V. Arthur, and D. S. Modha, "Truenorth: Accelerating from zero to 64 million neurons in 10 years," *Computer*, vol. 52, no. 5, pp. 20–29, 2019.

[4] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, "A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps)," *IEEE Transactions on Biomedical Circuits and Systems*, vol. PP, 08 2017.

[5] S. Song, A. Balaji, A. Das, N. Kandasamy, and J. Shackleford, "Compiling spiking neural networks to neuromorphic hardware," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 38–50. [Online]. Available: https://doi.org/10.1145/3372799.3394364

[6] S. Song, H. Chong, A. Balaji, A. Das, J. Shackleford, and N. Kandasamy, "Dfsynthesizer: Dataflow-based synthesis of spiking neural networks to neuromorphic hardware," *ACM Trans. Embed. Comput. Syst.*, aug 2021, just Accepted. [Online]. Available: https://doi.org/10.1145/3479156

[7] J. Luu, J. H. Anderson, and J. S. Rose, "Architecture description and packing for logic blocks with hierarchy, modes and complex interconnect," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 227–236. [Online]. Available: https://doi.org/10.1145/1950413.1950457

[8] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, vol. 111, pp. 47–63, mar 2019. [Online]. Available: https://doi.org/10.1016%2Fj.neunet.2018.12.002

[9] C. Xiao, Y. Wang, J. Chen, and L. Wang, “Topology-aware mapping of spiking neural network to neuromorphic processor,” *Electronics*, vol. 11, no. 18, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/18/2867

[10] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, “Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[11] A. Das, Y. Wu, K. Huynh, F. Dell’Anna, F. Catthoor, and S. Schaafsma, “Mapping of local and global synapses on spiking neuromorphic hardware,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 1217–1222.

[12] A. Balaji, A. Das, Y. Wu, K. Huynh, F. G. Dell’Anna, G. Indiveri, J. L. Krichmar, N. D. Dutt, S. Schaafsma, and F. Catthoor, “Mapping spiking neural networks to neuromorphic hardware,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 76–86, 2020.

[13] R. Eberhart and J. Kennedy, “A new optimizer using particle swarm theory,” in *MHS’95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, 1995, pp. 39–43.

[14] B. W. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[15] Y. Zha and J. Li, “Revisiting pathfinder routing algorithm,” in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 24–34. [Online]. Available: https://doi.org/10.1145/3490422.3502356

[16] F. Galluppi, S. Davies, A. Rast, T. Sharp, L. A. Plana, and S. Furber, “A hierachical configuration system for a massively parallel neural hardware platform,” in *Proceedings of the 9th Conference on Computing Frontiers*, ser. CF ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 183–192. [Online]. Available: https://doi.org/10.1145/2212908.2212934

[17] P. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, “A million spiking-neuron integrated circuit with a scalable communication network and interface,” *Science*, vol. 345, pp. 668 – 673, 2014.