Spiking Neural Network Based on Network on Chip

Final Report

Peiliang Du and Yifu Zhou

Professor Peter Beerel

EE552 - Asynchronous VLSI Design

April 24, 2023

Viterbi School of Engineering

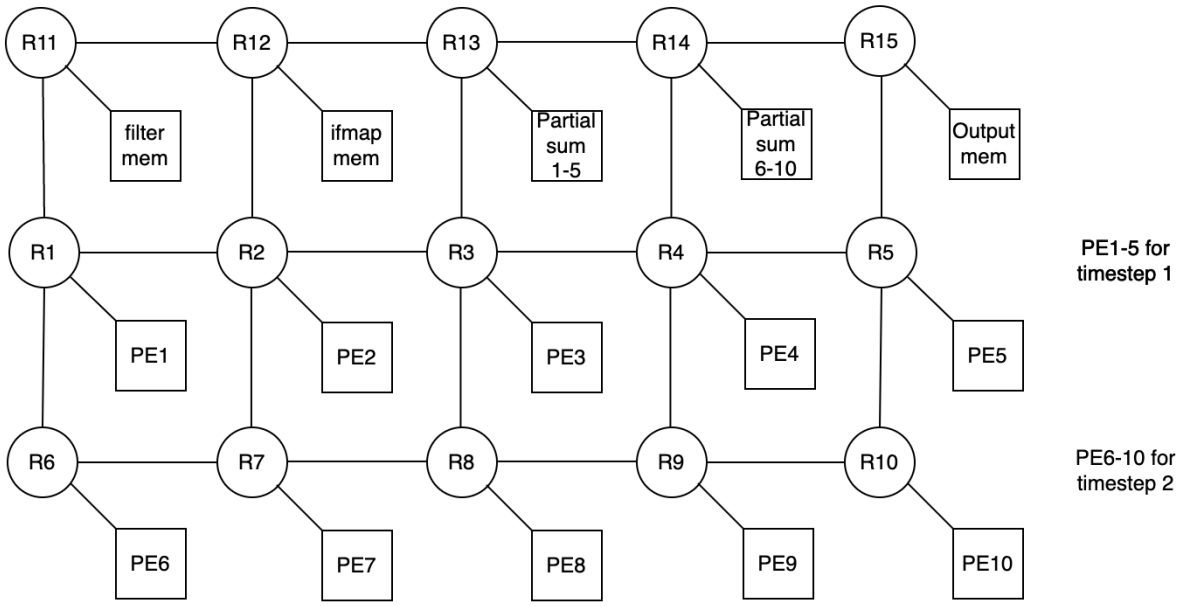
University of Southern California

Contents

GitHub: https://github.com/Chris-Du/EE552_NOC

Introduction

Network-on-chip Mesh Architecture



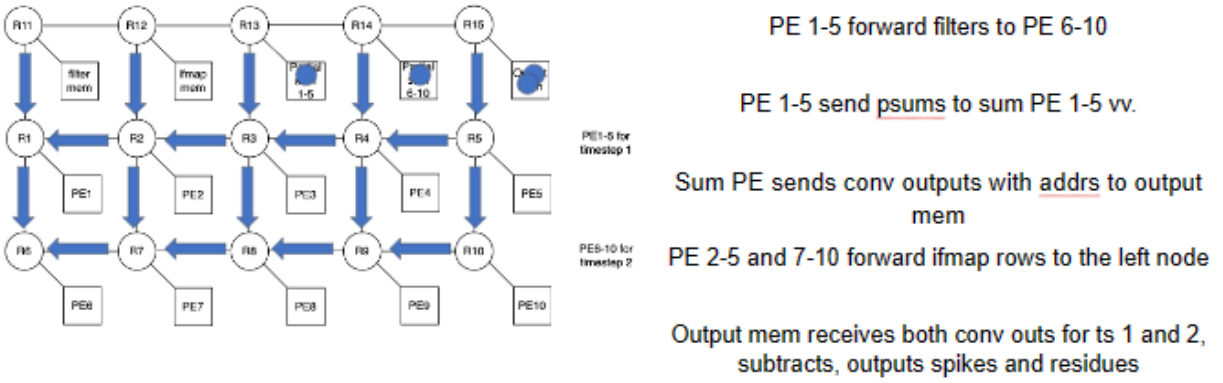
Our mesh spiking neural network consists of 10 convolution PEs, 2 partial sum PEs, an input feature map memory, a filter memory, and an output memory. In our design, PE 1-5 are responsible for calculating partial sums for timestep 1, while PE 6-10 are for timestep 2. In such an implementation, our design can generate partial sums for two timestep concurrently, and output them to corresponding partial sum modules, adding residue and subtracting threshold, and output to the memory eventually. In more detail, parietal sum 1 sends every entry of residue from timestep 1 to partial sum 2 which calculates its output spike. Memory is a likely bottleneck of the system. To ameliorate the cost of memory operation, the payload is set to 40 bits, allowing 5 input features or filters per packet. Furthermore, we forward filters from PE1-5 to PE6-10 and forward input filters between each timestep PE group. For example, PE 1 and PE 6 are using the same row of the filter data, we can forward the filter data to PE 6 right after PE 1 receives the filter from filter memory. PE 2 can forward the 2nd row of ifmap data to the PE1 after producing the partial sum output, since the ifmap data can be reused. In such a design, the

ifmap memory only needs to forward the data once in every time step to the PEs, and the filter memory only needs to send the filter to PE 1 to 5 once, which tremendously improves the performance.We use X-Y direction bits and X-hop Y-hop fields to navigate the packets through mesh.

Enhancement

To improve the speed and make our mesh network efficient, we implemented two enhancements. Because each PE can store an entire row of ifmap and filter data locally, we designed our mesh nodes to share resources to each other to avoid accessing memory in every convolution calculation given that the penalty to access memory is huge. To be specific, after sending 10 ifmap packets to PE 1-10 in the first round, our ifmap memory only needs to send one ifmap packet to PE 5 and PE 10 later on, because our PEs can share data among them. In such a design, the traffic load can be reduced as well. Another improvement is that our mesh network computes convolution outputs for timestep 1 and 2 concurrently. Because convolution calculation in PE will be the most time consuming operations in the network which takes around 1.2 μs, the performance can be improved by computing concurrently. At the end, our spiking neural network can compute convolution outputs for two timesteps in 28 μs.

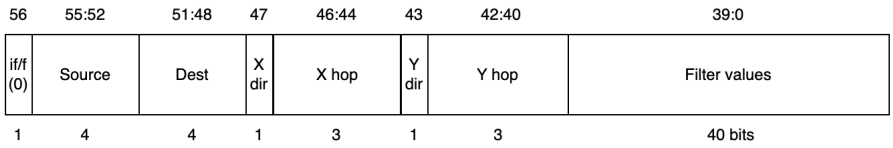
Data Flow



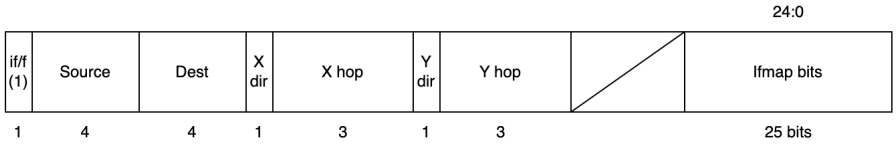
In our mesh, the filter memory will first send the filter to PE 1 to 5. Then, PE 1 to 5 will forward the filter data to PE 6 to 10. Next, the ifmap memory will start to send ifmap data for time step 1 and 2 to PE 1-5 and PE 6-10 respectively. Later on, the ifmap memory only needs to send a new row of ifmap to PE 5 and PE 10. After calculation, PE 1-5 will send 21 partial sums to summing PE at node 13, vice versa. The summing PE will add up the partial sums, 5 for each address and send the convolution outputs to the output memory. Finally, the output memory will output spikes and residues for time step 1 and 2 as soon as it receives all the convolution outputs.

Packet Format

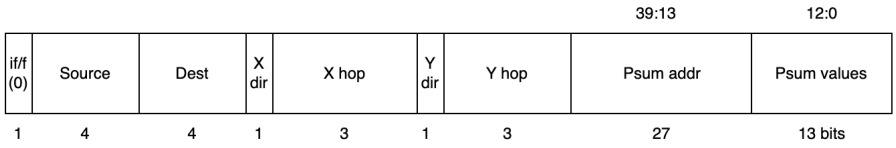
Filter packet



Ifmap packet



Psum packet



There are 3 types of packets in our network: filter, ifmap and partial sum packets. Each packet has a length of 57 bits, 40 bits belong to the data field, and the rest 17 bits are for the headers. A filter packet can hold an entire row of the filter data that's 8 bits each, and the ifmap packet also holds the entire row of the ifmap bits.

Format

- [56] if/f type:

  defines a packet whether it's ifmap or filter. The receiving PE will forward data based on the iff_type. Since the partial sum module only receives psum packets, the module doesn't care what the iff_type is.

- [55:52] 4-bit source:

  indicates where the packet is sent, whether it is from the memory or PE.

- [51:48] 4-bit dest:

  indicates which node the packet is heading toward. For receiving PEs, this 4-bit data is also used as PE id in the state machine, so that each PE operates accordingly.

- [47] X-direction [46:44] X-Hop value:

  indicates the X-direction, 1 for +X (East), and 0 for -X (West). 3-bit indicates the hop value in X-direction, and it will be manipulated by the router. An “000” means that the packet reaches its destination column.

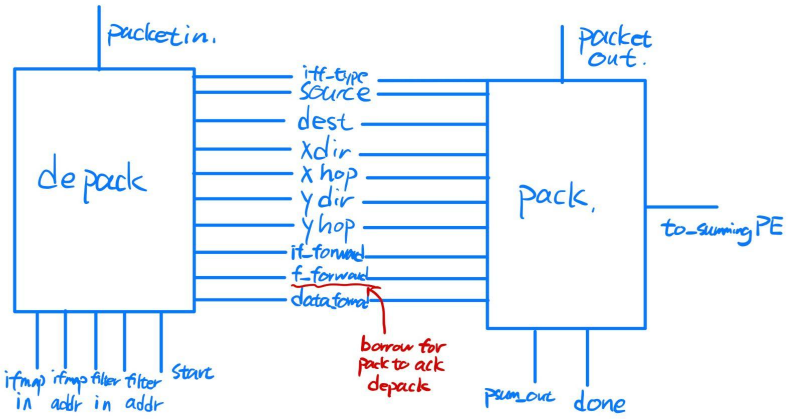
- [43] Y-direction [42:40] Y-Hop value:

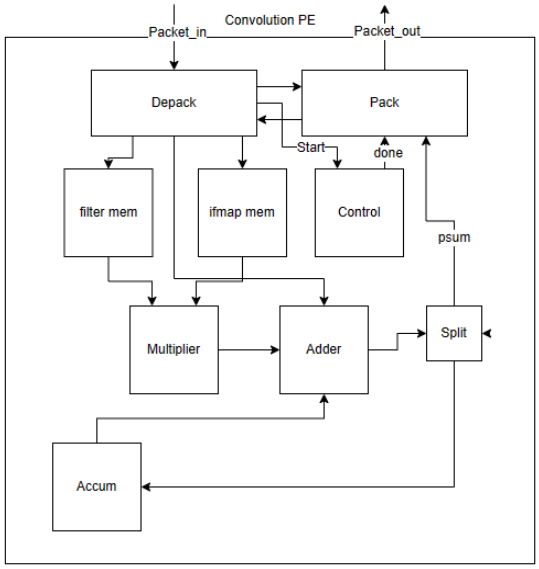
  indicates the Y-direction, 1 for +Y (North), and 0 for -Y (South). 3-bit indicates the hop value in Y-direction, and it will be manipulated by the router. An “000” means that the packet reaches its destination row.

- [39:0] 40-bit data:

  For the filter packet, the data field holds 5 filter values with 8 bits each. For the ifmap packet, the data field holds 25 input spike bits. In the psum packet, the 27-bit (don’t necessarily use all 27 for address) psum address shows the memory location the psum value needed to be stored, and the 13-bit psum field stores the psum output.

Processing Element Module





Our PE module is built based on the traditional PE provided. On top of that, we added a depacketizer and a packetizer. Our PE will take 5 filter values and 25 ifmap bits in each row of calculation, and output 21 partial sums. In our design, only PE 5 and 10 need to receive a new row of ifmap data, so our PE module needs to forward the ifmap and filter data to each other to avoid accessing memory frequently.

Data Sharing

To support the forwarding, we built a state machine inside the depacketizer such that the PE can operate based on its PE id. For example, PE 2 will forward filter data to PE 7 as soon as PE 2 receives the packet, because PE 2 and PE 7 use the same row of filter data. Then, the PE module will start convolution calculation and send 21 partial sums to the summing PE. Eventually, it will forward the ifmap data to the left node that is PE 1, since the PE 1 will use the same row of ifmap data as well.

PE	R/S	dest addr	data type	x dir	y dir	x hop	y hop	filter forward	ifmap forward
1	R	1							
	S	13	psum	1	1	010	01		
	S	6	filter	0	0	000	01	1	0
2	R	2							
	S	13	psum	1	1	001	01		
	S	7	filter	0	0	000	01	1	1
3	R	3							
	S	13	psum	0	1	000	01		
	S	8	filter	0	0	000	01	1	1
4	R	4							
	S	13	psum	0	1	001	01		
	S	9	filter	0	0	000	01	1	1
5	R	5							
	S	13	psum	0	1	010	01		
	S	10	filter	0	0	000	01	1	1
6	R	6							
	S	14	psum	1	1	011	10		
								0	0
7	R	7							
	S	14	psum	1	1	010	10		
								0	1
8	R	8							
	S	14	psum	1	1	001	10		
								0	1
9	R	9							
	S	14	psum	1	1	000	10		
								0	1
10	R	10							
	S	14	psum	0	1	001	10		

Since each PE has a different X Y distance to their target node, we build the state machine based on the table above. The state machine operates using the PE id and assigns new source and dest addresses, and X Y hop values to the packetizer.

PE Mem Overwrite Protection

The ifmap and filter memory will send out filter data and ifmap data quickly. However, the PE requires a large amount of time for convolution calculation, and we need to prevent the

incoming ifmap data from overwriting the PE mem while the PE is still in the previous convolution calculation. To achieve that, we added a channel between packetizer and depacketizer. Only when the PE finishes sending out all 21 partial sums and forwarding data, the packetizer will acknowledge the depacketizer to receive a new data packet.

Processing Element Transcript

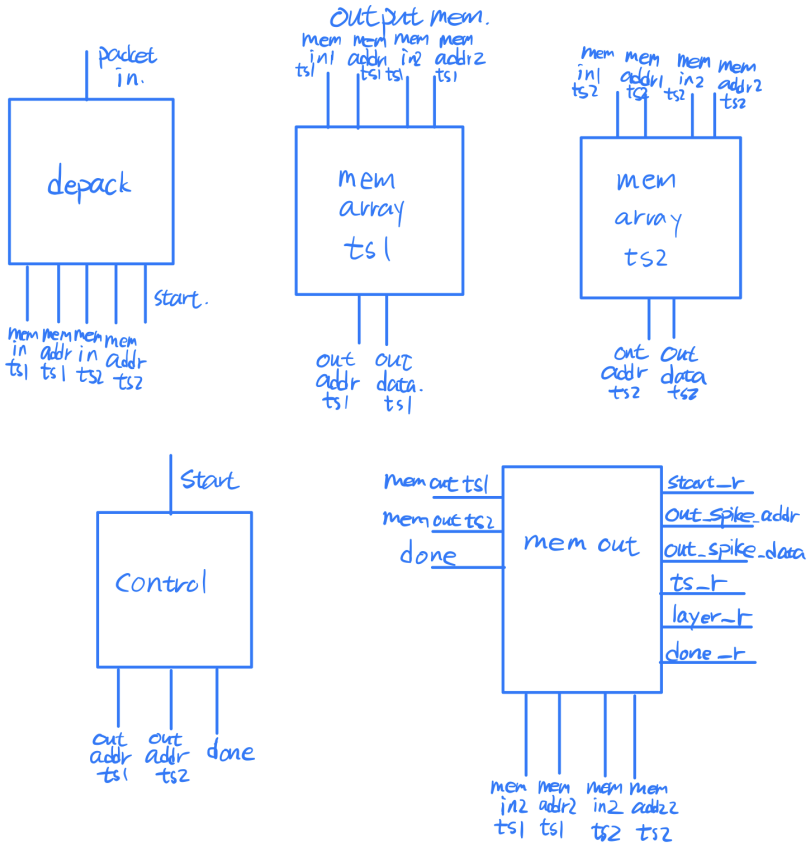
```
# Sending forwarding data
# iff type = 0
# data = 00000101000001000000001100000001000000001
# source =  2
# dest =  7
# x_dir = 0
# y_dir = 0
# x_hop = 0
# y_hop = 1
#
# packet = 00010011100000001000001010000010000000011000000100000000 1
```

```
# Sending psum
# iff type = 0
# psum_out =    1
# psum_addr =         4
# source =  2
# dest = 13
# x_dir = 1
# y_dir = 1
# x_hop = 1
# y_hop = 1
#
# packet = 00010110110011001000000000000000000000000100000000000001
#
```

```
# Sending forwarding data
# iff type = 1
# data = 00000000000000000000000000000000011111
# source =  2
# dest =  1
# x_dir = 0
# y_dir = 0
# x_hop = 1
# y_hop = 0
#
# packet = 10010000100010000000000000000000000000000000000000011111
```

From the above PE transcript, we could see that the PE will first forward filter data with new header information, then it will calculate partial sum and send the packet to the summing module. Finally, the PE module will forward the ifmap data, and each packet type has distinct header information.

Output Memory



After storing all the convolution outputs for time step 1 and 2, the output memory will first output the spikes and residues for ts 1, then it will output spikes and residues for ts 2. In our spiking neural mesh network, two timesteps are calculated concurrently, and we need to add the residues from ts1 to the convolution outputs from ts2 to calculate the spikes and residues for ts2. In such a case, we need to store the convolution outputs for both timesteps properly. Hence, there are two memory arrays in our output memory and each memory array has two write ports, so that the residues can be written back to the memory array for future use. Basically, the control module only needs to send addresses to two memory arrays for reading and writing. Our output memory can support multi-timesteps as well, it simply operates on two memory arrays alternatively.

Router

The router consists of five input control units and five output control units. Each input control unit models one input channel to the router (from four direction + from PE), and each output control unit models one output channel as well (to four direction + to PE).

Upon receiving a packet, the input control unit first checks if x hop field equals to 0. If the x hop field does not equal to 0, the x direction bit gets read and the packet is passed to the corresponding output channel (west or east) with its x hop field decremented by 1. If x hop field equal to 0, the module then checks y hop field. If y hop equals to 0, the packet has arrived at its destination and is then sent to an output control unit named PE_output for arbitration. If y hop does not equal to 0, y direction gets read and the packet is passed to the corresponding output control unit with y hop field decremented.

The output control module consists of three 2to1 arbiters. In effect, the module models a 4to1 arbiter. The four input channels can be arbitrary combinations of the four directions and PE. For example, the north output control module reads input channels of south, east, west and PE input control units and stores whoever won the arbitration to the north output channel.

Gate Level Implementation: 2 to 1 Arbiter

The aforementioned 2to1 arbiter is implemented at the gate level, so is the output control unit (since output control is literally made of 2to1 arbiters). We successfully integrated the gate level arbiter into our SNN NoC and passed the testbench. As indicated in line 104 of router.sv, north_output control unit is instantiated as a “output_ctrl_gate” module.

```
104    output_ctrl_gate north_output(
105        .in1(south_to_north),
106        .in2(east_to_north),
107        .in3(west_to_north),
108        .in4(PE_to_north),
109        .out(north_out)
110    );
```

The output_ctrl_gate module is implemented with gate level arbiters.

```
`timescale 1ns/1ns

module output_ctrl_gate (
    interface in1, in2, in3, in4, out
);
    parameter WIDTH_packet = 57;
    parameter FL = 0, BL = 0;
    logic [WIDTH_packet-1:0] in_packet;
    logic out1_req,out2_req,out3_req;
    logic out1_ack,out2_ack,out3_ack;
    logic [WIDTH_packet-1:0] out1_data, out2_data,out3_data, interface_out_data;

    arbiter_2to1_gate arb1(in1.req, in1.ack,in1.data,in2.req,in2.ack,in2.data,out1_req,out1_ack,out1_data);
    arbiter_2to1_gate arb2(in3.req, in3.ack,in3.data,in4.req,in4.ack,in4.data,out2_req,out2_ack,out2_data);
    arbiter_2to1_gate final_out(out1_req,out1_ack,out1_data,out2_req,out2_ack,out2_data, out3_req,out3_ack,out3_data);

    always begin
        out3_ack = 0;
        wait(out3_req);
        #FL;
        out3_ack = 1;
        out.Send(out3_data);
        wait(!out3_req);
        #BL;
    end

endmodule
```

Please refer to the arbiter_2to1_gate.sv for gate level implementation.

Verification, Analysis and Future Work

We wrote testbenches for most submodules including the mesh itself. However, when channeled together, two issues came up, as we indicated in our presentation slides. The first issue is convolution PE output arrives at the partial sum module out of order, causing the calculation to be incorrect. The second issue is a deadlock. To achieve as much as concurrency as possible, we decided to have the input feature map memory node keep sending packets down the NoC until finished. At some point all channels from R12 to R5 (in a X-Y routing scheme fashion) are occupied with input feature packets to be received by R5 or R10. However, these packets are not received by the PE nodes because the PE nodes are waiting for partial sum nodes to acknowledge PE's output packet. The partial sum nodes cannot be acknowledged because it is actively trying to send packets to output memory nodes through channel R13-14 and channel R14-15 which are already occupied by input feature map packets. Thus a deadlock is created.

Our temporary solution at presentation day was to remove partial sum and output memory nodes from the NoC and connect the channels directly. R13, R14 and R15 are simply empty nodes with a router to allow passage for the packets. This solution did resolve the deadlock and out of order arrival issues and pass the demo. However, we did not go far from there. We tried to resolve the two issues mentioned without sacrificing the concurrency but failed. So in short, our SNN NoC worked but not in the most ideal situation.

For future work and based on the feedback from Professor Beerel, we can definitely fix the deadlock issue by reducing the concurrency in NoC. For example, convolution PE only sends packets to partial sum nodes after an entire row of input feature map is received, and input feature memory node only sends when an entire row of residue is received by output memory node.

Reference

1. Beerel Peter, Ozdag RO, Ferretti M. A Designer's Guide to Asynchronous VLSI. Cambridge University Press; 2010.