Neural Processor Design for Machine Learning

Final Report

Yingqiao Gou, Jingyang Cui, Giovanni Michel

COMP_ENG 393/493: Advanced Low Power Digital and Mixed-signal Integrated Circuit Design

Northwestern University
Fall 2023

Due: December 9th, 2023

Table of Content

Introductory

1. Overview

As contemporary machine learning models exhibit a growing dependence on data and an escalation in computational requirements, the significance of energy-efficient hardware capable of supporting such computing tasks becomes crucial. Numerous research endeavors have delved into alternative non-Von Neumann architectures; however, none have demonstrated sufficient promise to supplant the conventional digital architecture. Conventional CPU architecture confronts challenges such as substantial overhead for single instructions and significant overhead in data transfer.

Our study introduces a straightforward project involving the design of a neural processing unit, originally developed for the ECE-393/493 class and extended to an independent study encompassing design, fabrication, and testing. This research aims to provide a viable solution to the shortcomings inherent in current Von Neumann architectures, addressing issues and offering innovative approaches to overcome existing limitations. The goal was to optimize the convolution calculation method to achieve the computation of one convolutional layer. The metrics we used to measure the performance is evaluated based on the area, timing report, and power consumption. The processing unit performs convolution and ReLU.

2. Literature review

In recent years, extensive research has been conducted on neural processors capable of handling both inference and training tasks directly on chips. In the paper titled "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," the researchers introduced a chip featuring a two-tiered system. The first level functions as the top-level control overseeing communication between the DRAM and the global buffer, while the second level manages the traffic between the global buffer and the processing element (PE) array.

In another work, "A 45nm CMOS Neuromorphic Chip with a Scalable Architecture for Learning in Networks of Spiking Neurons," the authors proposed a neuromorphic chip design implemented on a 45nm CMOS process. The goal is to realize scalable learning algorithms tailored for networks of spiking neurons. The paper addresses challenges associated with the limited scalability of analog silicon neural circuits and the constrained learning speed due to communication demands. Noteworthy design aspects include large analog computing arrays and transposable SRAM arrays, which exhibit linear scaling with neuron numbers and quadratic

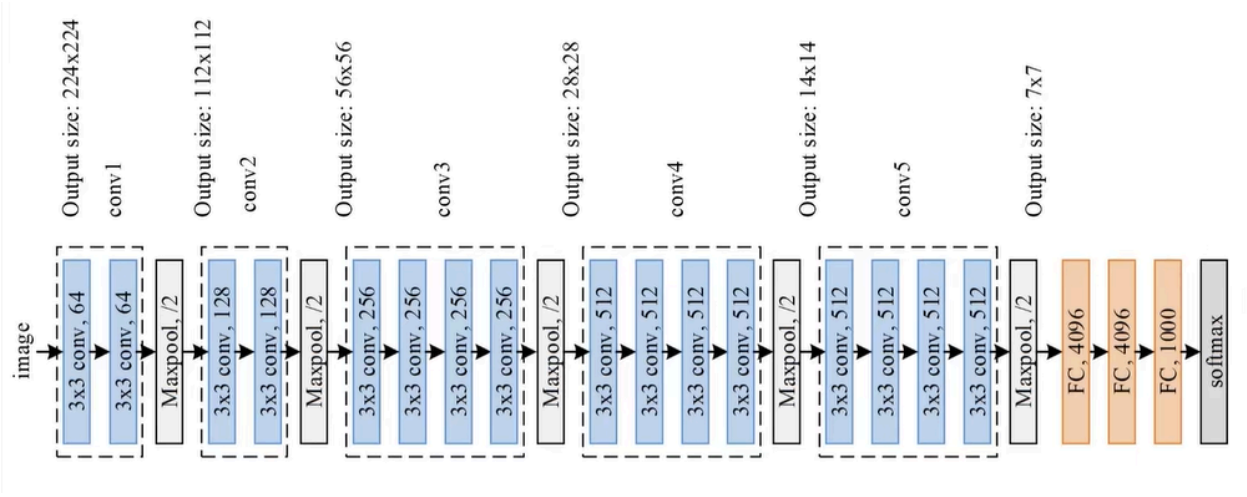
scaling with the number of synapses. The paper advocates for an analog circuit perspective in implementing neural networks.

Furthermore, the literature titled "TrueNorth: Design and Tool Flow of a 65 MW 1 Million Neuron Programmable Neurosynaptic Chip" presents the design and tool flow of the TrueNorth neuromorphic chip, capable of emulating a million neurons. The development aimed to harness the benefits of cognitive computing, particularly in handling large volumes of noisy sensory data without requiring high computational power. TrueNorth, inspired by the structure and function of the brain, focuses on low-power consumption and mimics a neuron system's design with interconnected neurons and synapses. The chip operates at 65 mW, features 4096 neurosynaptic cores, and incorporates 5.4 billion transistors on a 4.2 cm^2 silicon layer, enabling real-time simulation of one million neurons.

Circuit Design

1. Algorithm:

In the development of a Neural Processing Unit (NPU), the primary algorithm to incorporate is the Convolutional Neural Network (CNN). Using the fundamental CNN model, VGG19, as an illustration, the architecture comprises various convolutional, pooling, and fully connected operations. Emphasis should be placed on identifying fixed and recurring computational patterns within the algorithm, alongside considerations of data locality and the interplay between data and computation. In light of these factors, the design of computational modules and memory access modules should be informed and tailored accordingly.



a. FC layer

The computational characteristics of the fully connected layer are: vector inner product, vector element operations, and no complex control flow. Here is a HLS code description:

```
// x is the input vector, y is the output vector, W is the weight
y all = 0 // Initialize all output vector values to 0
for j = 0, j < No, j++:
 for i = 0, i < Ni, i++:
  y[j] += W[j][i] * x[i]
  if i == Ni:
   y[j] = G(y[j] + b[j])
```

b. Convolution layer

The computational characteristics of the fully connected layer are: matrix inner product, vector element operations, and no complex control flow. Here is a HLS code description:

```
nor = 0
for r=0, r<Nir, r+=s:  // Move in row-major order
   noc = 0
   for c=0, c<Nic, c+=sc:  // Move in column-major order
      for j=0, j<Nof, j++:
         sum[j] = 0
      for kr=0, kr<Kr, kr++:
         for kc=0, kc<Kc, kc++:
            for j=0, j<Nof, j++:
               for i=0, i<Nif, i++:
                  sum[j] += W[kr][kc][j][i] * X[r+kr][c+kc][i]
   for i=0, i<Nof, i++:
      // Remaining part of the code seems to be missing
```

c. Pooling layer

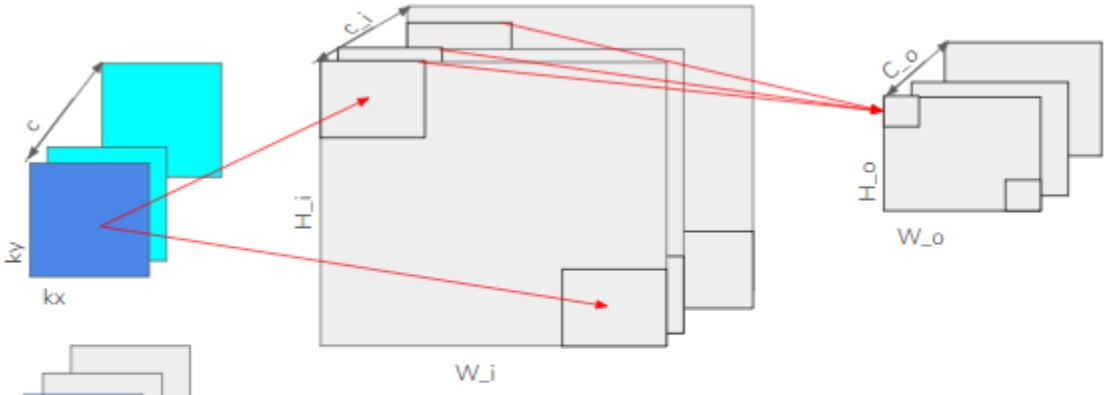
The computational characteristics of the fully connected layer are: vector element operations, and no complex control flow. Here is a HLS code description:

```
nor = 0
 for r = 0, r < Nir, r += sr: // Move in row-major order
  noc = 0
  for c = 0, c < Nic, c += sc: // Move in column-major order
   for i = 0, i < Nif, i++:
    value[i] = 0
```

```
for kr = 0, kr < Kr, kr++:
  for kc = 0, kc < Kc, kc++:
   for i = 0, i < Nif, i++: // for average pooling
    value[i] += X[r + kr][c + kc][i] // for max pooling
    value[i] = max(value[i], X[r + kr][c + kc][i])
```

2. Design Specification:

Our design uses stride = 1 without padding to simplify the design. The size of the output feature map will be calculated based on the size of the input feature map as well as the algorithm. We need a total of 8 filters that are 4x4. The size of input feature map is 64 * 64, Input feature map's channel size= kernel's channel size, which is 8, and the output feature map channel is the same as the input channel. An example of the convolution calculation is shown below.



$$O[z][y][x] = ReLU\left(B[z] + \Sigma_{k=0}^{c_i-1}\Sigma_{j=0}^{ky-1}\Sigma_{i=0}^{kx-1} I[k][Uy+j][Ux+i] \cdot W[z][k][j][i]\right)$$

$$0 \le z \le C_o, 0 \le y \le H_o, 0 \le x \le W_o$$

U is stride. B is bias.

$$ReLU(x) = \begin{cases} x, & x > 0 \\ 0, & x \le 0 \end{cases}$$

3. Modeling and Analysis:

The modeling and analysis involves doing the modeling of how the NPU will be made. We will make a 4x5 processing element array that will be used for the convolution. We also implemented a weight buffer and input feature map buffer.

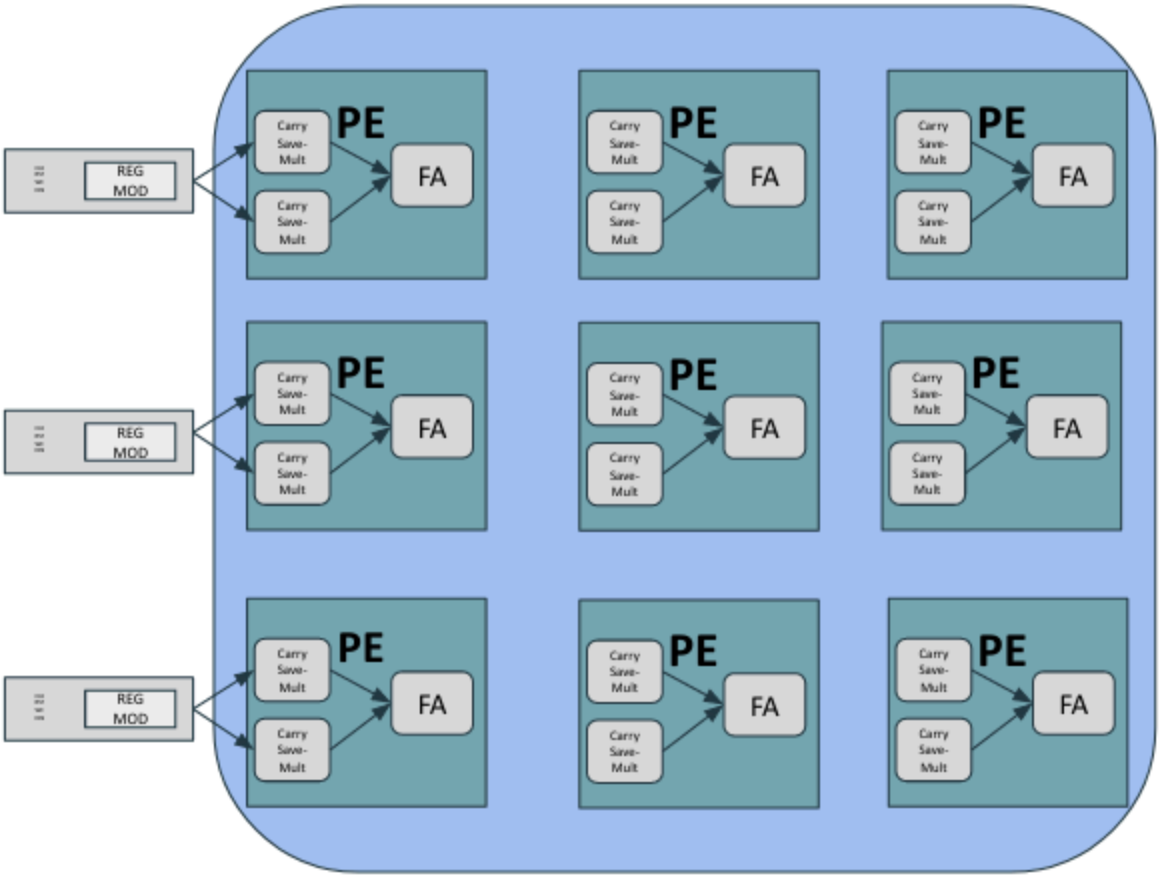


Figure 1: The PE array we implemented for our NPU.

We only implemented convolution and ReLU, so we used random input feature maps and random weights to initialize the NPU with data. To show the functionality that the input feature maps and weights were instantiated here is our waveform showing the initialization of the data.

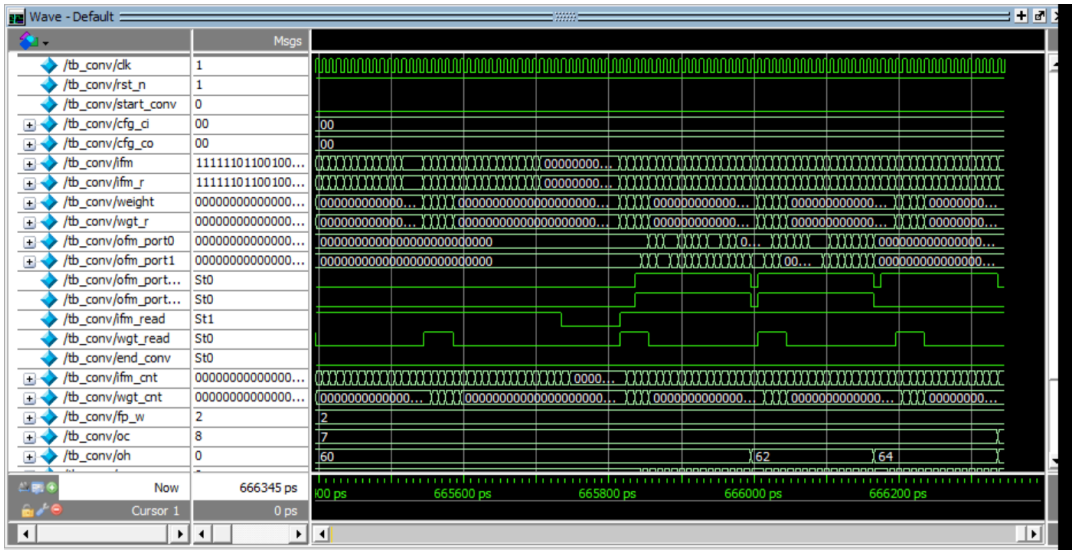


Figure 2: The waveform of weights and IFM being initialized.

```
------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | in2reg  | reg2out | in2out  | default |
+--------------------+---------+---------+---------+---------+---------+---------+
|           WNS (ns):|  0.017  |  0.017  |  6.497  |  3.176  |   N/A   |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |   N/A   |    0    |
|          All Paths:|  11913  |  11861  |   100   |   52    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+
|slow                |  0.017  |  0.017  |  6.497  |  3.176  |   N/A   |  0.000  |
|                    |  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|                    |    0    |    0    |    0    |    0    |   N/A   |    0    |
|                    |  11913  |  11861  |   100   |   52    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+

+---------------+------------------------------+------------------+
|               |             Real             |       Total      |
|    DRVs       +------------------+-----------+------------------|
|               |  Nr nets(terms)  | Worst Vio |  Nr nets(terms)  |
+---------------+------------------+-----------+------------------+
|   max_cap     |      0 (0)       |   0.000   |      0 (0)       |
|   max_tran    |      0 (0)       |   0.000   |      0 (0)       |
|   max_fanout  |    695 (695)     |    -55    |    958 (958)     |
|   max_length  |      0 (0)       |     0     |      0 (0)       |
+---------------+------------------+-----------+------------------+

Density: 76.121%
Total number of glitch violations: 0
------------------------------------------------------------
```

Figure 3: The timing summary and timing report.

4. System/Schematic Design and Simulations:

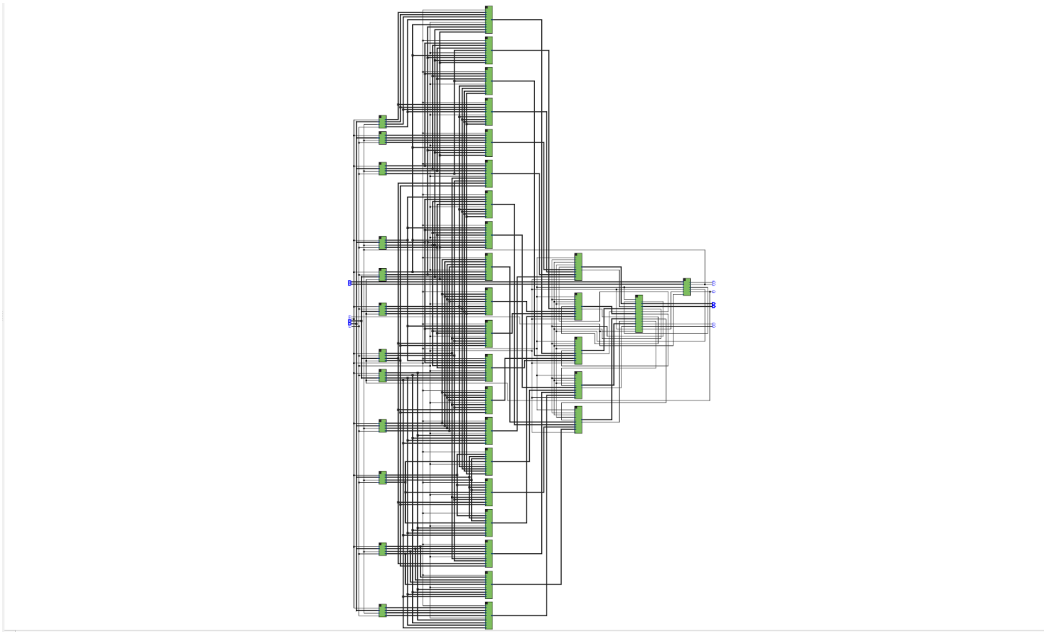


Figure 4: The top level schematic for our design.

5. Back end Design (Innovus Part):

```
#====================================Env Vars====================================
set RST_NAME                            rst_n
set CLK_NAME                            clk

# 100 MHz
set freqMHz                   125

set CLK_PERIOD                [expr 1000.0/$freqMHz]
set CLK_SKEW                  [expr $CLK_PERIOD*0.05]
set CLK_SOURCE_LATENCY        [expr $CLK_PERIOD*0.1]
set CLK_NETWORK_LATENCY       [expr $CLK_PERIOD*0.1]
set CLK_TRAN                  [expr $CLK_PERIOD*0.01]


set INPUT_DELAY_MAX           [expr $CLK_PERIOD*0.3]
set INPUT_DELAY_MIN           [expr $CLK_PERIOD*0.1]
set OUTPUT_DELAY_MAX          [expr $CLK_PERIOD*0.3]
set OUTPUT_DELAY_MIN          [expr $CLK_PERIOD*0.1]

set MAX_FANOUT                20
set MAX_TRAN                  2
set MAX_CAP                   2

set ALL_INPUT_EX_CLK [remove_from_collection [all_inputs] {$RST_NAME $CLK_NAME}]


#==================================Define Design Environment=========================
#GUIDANCE: use the default
```

```
#==================================Define Design Environment=========================
#GUIDANCE: use the default

set_max_area 0
set_max_transition  $MAX_TRAN     [current_design]
set_max_fanout      $MAX_FANOUT   [current_design]
set_max_capacitance $MAX_CAP      [current_design]
set_drive 0                       [get_ports $CLK_NAME]
set_drive 0                       [get_ports $RST_NAME]


#============================ Set Design Constraints=========================
#------------------------------Clock and Reset Definition--------------------------

# create clock
create_clock -name $CLK_NAME -period $CLK_PERIOD [get_ports $CLK_NAME]
set_dont_touch_network [get_ports $CLK_NAME]

#set_drive 0 [get_ports $CLK_NAME]

set_clock_uncertainty $CLK_SKEW [get_clocks $CLK_NAME]
set_clock_transition  $CLK_TRAN [all_clocks]
set_clock_latency -source $CLK_SOURCE_LATENCY [get_clocks $CLK_NAME]
set_clock_latency -max $CLK_NETWORK_LATENCY   [get_clocks $CLK_NAME]

#rst_ports
set_dont_touch_network                          [get_ports $RST_NAME]
set_false_path -from                            [get_ports $RST_NAME]
#set_ideal_network -no_propagate                 [get_ports $RST_NAME]

#------------------------------I/O Constraint-----------------------------
set input delay   -max $INPUT DELAY MAX   -clock $CLK NAME   $ALL INPUT EX CLK
```

Figure 5: SDC Setup.

```
#rst_ports
set_dont_touch_network                          [get_ports $RST_NAME]
set_false_path -from                            [get_ports $RST_NAME]
#set_ideal_network -no_propagate                 [get_ports $RST_NAME]

#--------------------------------I/O Constraint-----------------------------
set_input_delay   -max $INPUT_DELAY_MAX   -clock $CLK_NAME   $ALL_INPUT_EX_CLK
set_input_delay   -min $INPUT_DELAY_MIN   -clock $CLK_NAME   $ALL_INPUT_EX_CLK -add
set_output_delay  -max $OUTPUT_DELAY_MAX  -clock $CLK_NAME   [all_outputs]
set_output_delay  -min $OUTPUT_DELAY_MIN  -clock $CLK_NAME   [all_outputs] -add
set_load          1                       [all_outputs]
set_driving_cell -lib_cell    INVX2        $ALL_INPUT_EX_CLK

####################
# group path
####################
group_path -name reg2reg -weight 10 -critical_range 0.5 -from [all_registers] -to [all_registers]
group_path -name reg2out -weight 2  -critical_range 0.5 -to   [all_outputs]
group_path -name in2reg  -weight 2  -critical_range 0.5 -from [all_inputs]
group_path -name in2out  -weight 1  -critical_range 0.5 -from [all_inputs]    -to [all_outputs]
```

Figure 6: SDC Results.

During DC synthesis, a 20% margin must be reserved for the back-end. Therefore, the synthesis should be performed at 125MHz. And we set some design constraints of the clock such as clock transition and latency in the script we write. We can just source the total run script to run all the script At the end we get the needed file for Innovus part(netlist file and sdc file)

```
module PE_FSM ( clk, rst_n, start_conv, start_again, cfg_ci, cfg_co, ifm_read,
       wgt_read, p_valid_output, last_chanel_output, end_conv );
  input [1:0] cfg_ci;
  input [1:0] cfg_co;
  input clk, rst_n, start_conv, start_again;
  output ifm_read, wgt_read, p_valid_output, last_chanel_output, end_conv;
  wire   n23, n24, n25, n26, n28, n30, n32, n34, n36, n38, n40, n42, n44, n45,
         n46, n47, n48, n49, n50, n51, n53, n55, n69, n1, n2, n3, n4, n5, n6,
         n7, n8, n9, n10, n11, n12, n13, n14, n15, n16, n17, n18, n19, n20,
         n21, n22, n27, n29, n31, n33, n35, n37, n39, n41, n43, n52, n54, n56,
         n57, n58, n59, n60, n61, n62, n63, n64, n65, n66, n67, n68, n70, n71,
         n72, n73, n74, n75, n76, n77, n78, n79, n80, n81, n82, n83, n84, n85,
         n86, n87, n88, n89, n90, n91, n92, n93, n94, n95, n96, n97, n98, n99,
         n100, n101, n102, n103, n104;
  wire   [2:0] current_state;
  wire   [5:0] ci;
  wire   [8:0] cnt2;
  wire   [5:0] cnt1;

  DFFSXL \cnt2_reg[6]  ( .D(n36), .CK(clk), .SN(rst_n), .Q(n99), .QN(cnt2[6])
       );
  DFFSXL \last_chanel_i_reg[2]  ( .D(n101), .CK(clk), .SN(rst_n), .Q(n89) );
  DFFSXL \ci_reg[5]  ( .D(n55), .CK(clk), .SN(rst_n), .Q(n94), .QN(ci[5]) );
  DFFSXL \ci_reg[4]  ( .D(n53), .CK(clk), .SN(rst_n), .QN(ci[4]) );
  DFFSXL \ci_reg[3]  ( .D(n51), .CK(clk), .SN(rst_n), .Q(n96), .QN(ci[3]) );
  DFFSXL \cnt1_reg[0]  ( .D(n50), .CK(clk), .SN(rst_n), .QN(cnt1[0]) );
  DFFSXL \current_state_reg[0]  ( .D(n69), .CK(clk), .SN(rst_n), .Q(n97), .QN(
       current_state[0]) );
  DFFSXL \cnt1_reg[1]  ( .D(n49), .CK(clk), .SN(rst_n), .QN(cnt1[1]) );
  DFFSXL \cnt1_reg[2]  ( .D(n48), .CK(clk), .SN(rst_n), .QN(cnt1[2]) );
  DFFSXL \cnt1_reg[3]  ( .D(n47), .CK(clk), .SN(rst_n), .QN(cnt1[3]) );
  DFFSXL \cnt1_reg[4]  ( .D(n46), .CK(clk), .SN(rst_n), .Q(n93), .QN(cnt1[4])
       );
  DFFSXL \cnt1_reg[5]  ( .D(n45), .CK(clk), .SN(rst_n), .QN(cnt1[5]) );
  DFFSXL \current_state_reg[1]  ( .D(n44), .CK(clk), .SN(rst_n), .QN(
       current_state[1]) );
  DFFSXL \cnt2_reg[0]  ( .D(n42), .CK(clk), .SN(rst_n), .Q(n100), .QN(cnt2[0])
       );
```

Figure 7: Netlist Result.

```
     optDesign Final Summary
---------------------------------------------------------

+--------------------+---------+---------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | in2reg  | reg2out | in2out  | default |
+--------------------+---------+---------+---------+---------+---------+---------+
|           WNS (ns):|  0.002  |  0.002  |  5.350  |  2.672  |   N/A   |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |   N/A   |    0    |
|          All Paths:|  11913  |  11861  |   100   |   52    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+
|slow                |  0.002  |  0.002  |  5.350  |  2.672  |   N/A   |  0.000  |
|                    |  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|                    |    0    |    0    |    0    |    0    |   N/A   |    0    |
|                    |  11913  |  11861  |   100   |   52    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +-----------------+-------------+------------------|
|                |  Nr nets(terms) | Worst Vio   |  Nr nets(terms)  |
+----------------+-----------------+-------------+------------------+
|   max_cap      |      0 (0)      |   0.000     |      1 (1)       |
|   max_tran     |      0 (0)      |   0.000     |      0 (0)       |
|   max_fanout   |    695 (695)    |    -55      |    696 (696)     |
|   max_length   |      0 (0)      |     0       |      0 (0)       |
+----------------+-----------------+-------------+------------------+

Density: 75.823%
Routing Overflow: 0.00% H and 0.00% V
---------------------------------------------------------------
```

Figure 5: The Prects report.

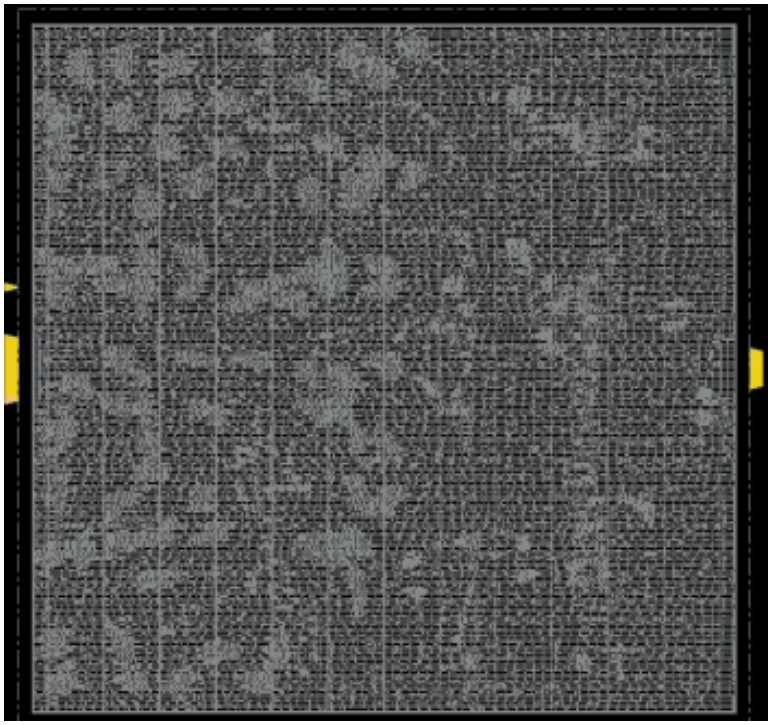


Figure 6: The filler.

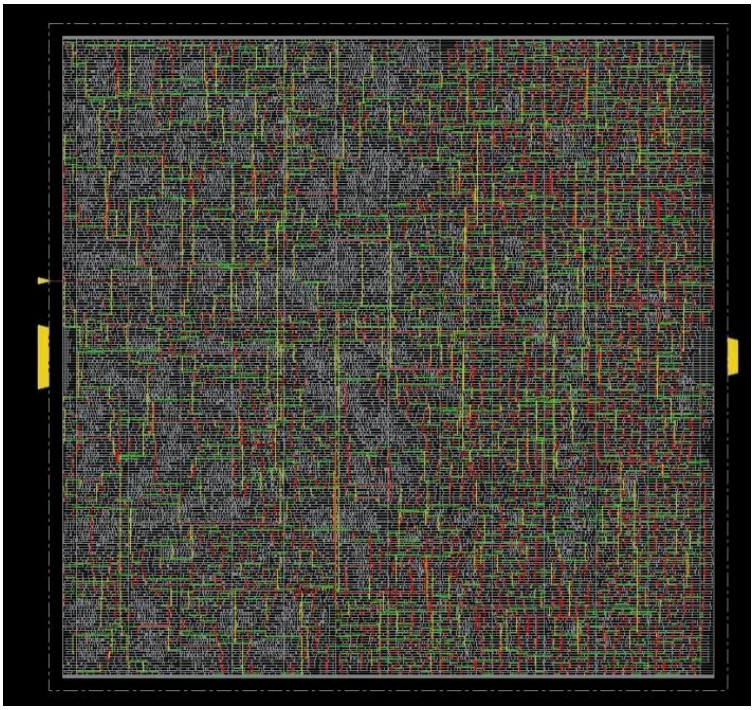


Figure 7: Postcts.

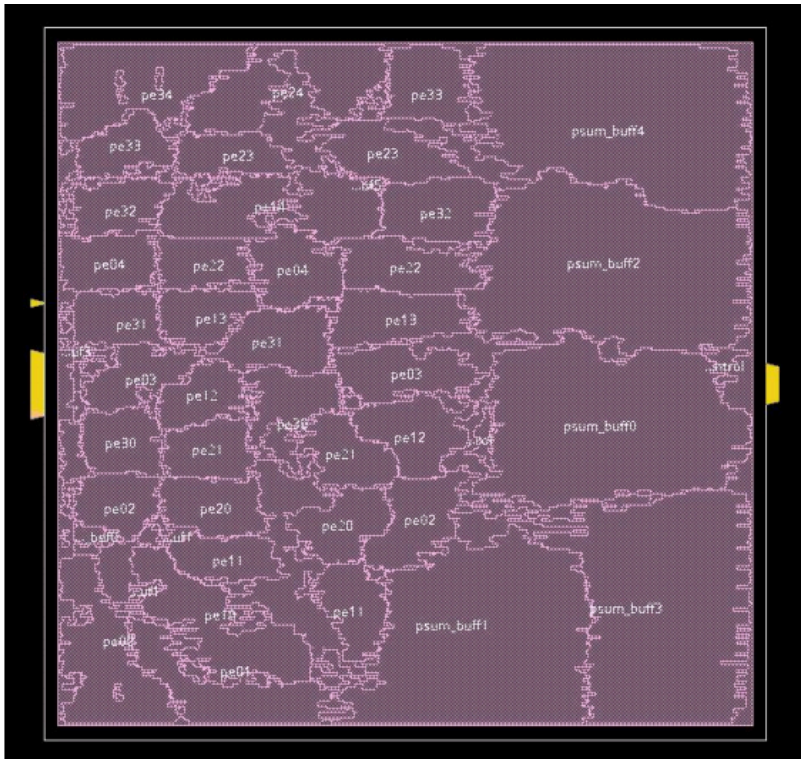


Figure 8: Afterplace.

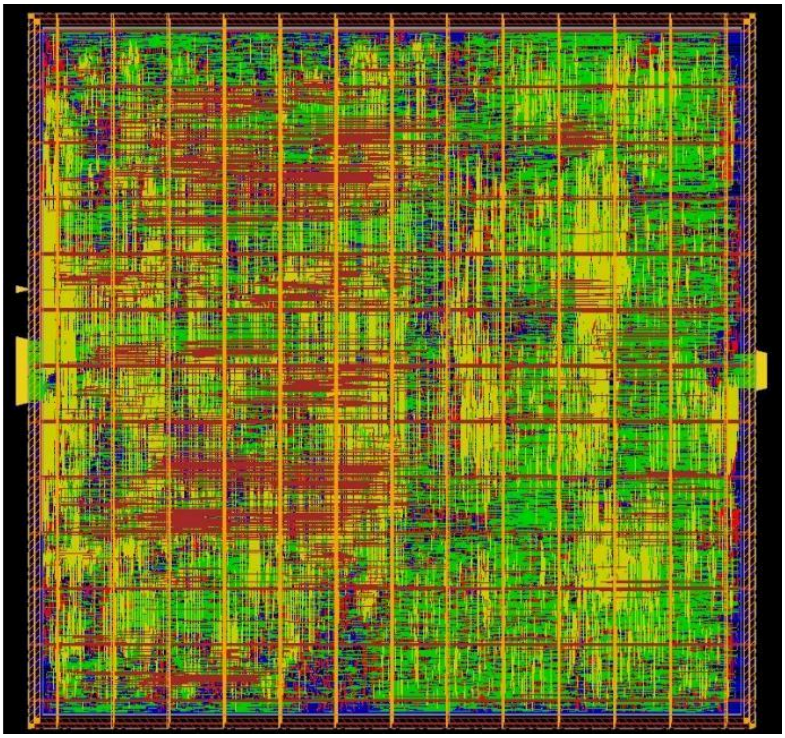


Figure 9: Post Route

```
#####################################################################
#  Generated by:      Cadence Innovus 20.10-p004_1
#  OS:                Linux x86_64(Host ID wyx)
#  Generated on:      Sat Dec  2 17:17:40 2023
#  Design:            CONV_ACC
#  Command:           timeDesign -hold -numPaths 500 -outDir ./out/OptDesignPostR -prefix postR_hold -expandedViews -postRoute
#####################################################################

------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | in2reg  | reg2out | in2out  | default |
+--------------------+---------+---------+---------+---------+---------+---------+
|           WNS (ns):|  0.001  |  0.001  |  0.174  |  1.324  |   N/A   |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |   N/A   |    0    |
|          All Paths:|  11913  |  11861  |   100   |   52    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+
|fast                |  0.001  |  0.001  |  0.174  |  1.324  |   N/A   |  0.000  |
|                    |  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|                    |    0    |    0    |    0    |    0    |   N/A   |    0    |
|                    |  11913  |  11861  |   100   |   52    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+

Density: 77.173%
------------------------------------------------------------
```

Figure 10: Post R Timing Hold

```
------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | in2reg  | reg2out | in2out  | default |
+--------------------+---------+---------+---------+---------+---------+---------+
|           WNS (ns):|  0.017  |  0.017  |  6.497  |  3.176  |   N/A   |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |   N/A   |    0    |
|          All Paths:|  11913  |  11861  |   100   |   52    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+
|slow                |  0.017  |  0.017  |  6.497  |  3.176  |   N/A   |  0.000  |
|                    |  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|                    |    0    |    0    |    0    |    0    |   N/A   |    0    |
|                    |  11913  |  11861  |   100   |   52    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+

+----------------+-------------------------------+-------------------+
|                |             Real              |       Total       |
|    DRVs        +------------------+------------+-------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)   |
+----------------+------------------+------------+-------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)        |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)        |
|   max_fanout   |    695 (695)     |    -55     |    958 (958)      |
|   max_length   |      0 (0)       |     0      |      0 (0)        |
+----------------+------------------+------------+-------------------+

Density: 76.121%
Total number of glitch violations: 0
------------------------------------------------------------
```

Figure 11: Post R Timing Summary

Hinst Name	Module Name	Inst Count	Total Area
CONV_ACC		32305	1065099.974
ifm_buf0	IFM_BUF_7	71	2651.141
ifm_buf1	IFM_BUF_6	82	2797.502
ifm_buf2	IFM_BUF_5	95	2987.107
ifm_buf3	IFM_BUF_4	96	2977.128
ifm_buf4	IFM_BUF_3	91	2907.274
ifm_buf5	IFM_BUF_2	97	3000.413
ifm_buf6	IFM_BUF_1	82	2790.850
ifm_buf7	IFM_BUF_0	76	2711.016
pe00	PE_19	896	25540.099
pe01	PE_18	903	25619.933
pe02	PE_17	843	26112.240
pe03	PE_16	853	26255.275
pe04	PE_15	848	26185.421
pe10	PE_14	899	25566.710
pe11	PE_13	842	26132.198
pe12	PE_12	839	26062.344
pe13	PE_11	852	26251.949
pe14	PE_10	848	26172.115
pe20	PE_9	857	26325.130
pe21	PE_8	854	26305.171
pe22	PE_7	843	26125.546
pe23	PE_6	849	26198.726
pe24	PE_5	898	25546.752
pe30	PE_4	834	25999.142
pe31	PE_3	845	26138.851
pe32	PE_2	837	26032.406
pe33	PE_1	894	25503.509
pe34	PE_0	906	25666.502

32	pe_fsm	PE_FSM	161	3868.603
33	psum_buff0	PSUM_BUFF_data_width25_addr_width5_depth16_4	2623	98078.904
34	psum_buff0/adder_tree	PSUM_ADD_data_width25_4	619	14183.770
35	psum_buff0/synch_fifo0	SYNCH_FIFO_data_width25_addr_width5_depth16_9	912	40715.136
36	psum_buff0/synch_fifo1	SYNCH_FIFO_data_width25_addr_width5_depth16_8	911	40661.914
37	psum_buff1	PSUM_BUFF_data_width25_addr_width5_depth16_3	2634	98172.043
38	psum_buff1/adder_tree	PSUM_ADD_data_width25_3	628	14290.214
39	psum_buff1/synch_fifo0	SYNCH_FIFO_data_width25_addr_width5_depth16_7	908	40615.344
40	psum_buff1/synch_fifo1	SYNCH_FIFO_data_width25_addr_width5_depth16_6	911	40668.566
41	psum_buff2	PSUM_BUFF_data_width25_addr_width5_depth16_2	2628	98108.842
42	psum_buff2/adder_tree	PSUM_ADD_data_width25_2	622	14233.666
43	psum_buff2/synch_fifo0	SYNCH_FIFO_data_width25_addr_width5_depth16_5	911	40635.302
44	psum_buff2/synch_fifo1	SYNCH_FIFO_data_width25_addr_width5_depth16_4	911	40668.566
45	psum_buff3	PSUM_BUFF_data_width25_addr_width5_depth16_1	2634	98231.918
46	psum_buff3/adder_tree	PSUM_ADD_data_width25_1	624	14273.582
47	psum_buff3/synch_fifo0	SYNCH_FIFO_data_width25_addr_width5_depth16_3	912	40715.136
48	psum_buff3/synch_fifo1	SYNCH_FIFO_data_width25_addr_width5_depth16_2	911	40595.386
49	psum_buff4	PSUM_BUFF_data_width25_addr_width5_depth16_0	2636	98048.966
50	psum_buff4/adder_tree	PSUM_ADD_data_width25_0	620	14190.422
51	psum_buff4/synch_fifo0	SYNCH_FIFO_data_width25_addr_width5_depth16_1	913	40602.038
52	psum_buff4/synch_fifo1	SYNCH_FIFO_data_width25_addr_width5_depth16_0	916	40661.914
53	wgt_buf0	WGT_BUF_3	99	3003.739
54	wgt_buf1	WGT_BUF_2	97	2990.434
55	wgt_buf2	WGT_BUF_1	97	2990.434
56	wgt_buf3	WGT_BUF_0	97	2990.434
57	writeback_control	WRITE_BACK_data_width25_depth16	249	6902.280

Figure 12: Report Area

```
    Power Net Detected:
            Voltage     Name
                 0V     VSS
              1.62V     VDD
    Using Power View: slow.
    Starting SI iteration 1 using Infinite Timing Windows
    ############################################################################
    # Design Stage: PostRoute
    # Design Name: CONV_ACC
    # Design Mode: 180nm
    # Analysis Mode: MMMC OCV
    # Parasitics Mode: SPEF/RCDB
    # Signoff Settings: SI On
    ############################################################################
    Start delay calculation (fullDC) (8 T). (MEM=3006.5)
    *** Calculating scaling factor for slow libraries using the default operating condition of each library.
    AAE_INFO: 8 threads acquired from CTE.
    Total number of fetched objects 43712
    AAE_INFO: Total number of nets for which stage creation was skipped for all views 0
    AAE_INFO-618: Total number of nets in the design is 36074,  100.0 percent of the nets selected for SI analysis
    End delay calculation. (MEM=3338.74 CPU=0:00:20.1 REAL=0:00:03.0)
    End delay calculation (fullDC). (MEM=3338.74 CPU=0:00:21.3 REAL=0:00:03.0)
    Loading CTE timing window with TwFlowType 0...(CPU = 0:00:00.0, REAL = 0:00:00.0, MEM = 3355.7M)
    Add other clocks and setupCteToAAEClockMapping during iter 1
    Loading CTE timing window is completed (CPU = 0:00:00.3, REAL = 0:00:00.0, MEM = 3338.7M)
    Starting SI iteration 2
    Start delay calculation (fullDC) (8 T). (MEM=3042.87)
    Glitch Analysis: View slow -- Total Number of Nets Skipped = 0.
    Glitch Analysis: View slow -- Total Number of Nets Analyzed = 43712.
    Total number of fetched objects 43712
    AAE_INFO: Total number of nets for which stage creation was skipped for all views 0
    AAE_INFO-618: Total number of nets in the design is 36074,  0.8 percent of the nets selected for SI analysis

  Starting Levelizing
    2023-Dec-02 17:18:14 (2023-Dec-02 09:18:14 GMT)
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 10%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 20%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 30%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 40%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 50%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 60%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 70%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 80%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 90%

    Finished Levelizing
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT)

    Starting Activity Propagation
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT)
    ** INFO:  (VOLTUS_POWR-1356): No default input activity has been set. Defaulting to 0.2.
    Use 'set_default_switching_activity -input_activity' command to change the default activity value.

    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 10%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 20%
    2023-Dec-02 17:18:15 (2023-Dec-02 09:18:15 GMT): 30%

    Finished Activity Propagation
    2023-Dec-02 17:18:16 (2023-Dec-02 09:18:16 GMT)
    Ended Processing Signal Activity: (cpu=0:00:01, real=0:00:01, mem(process/total/peak)=2139.26MB/4837.96MB/2351.93MB)
```

```
Begin Power Computation

      --------------------------------------------------------
      # of cell(s) missing both power/leakage table: 0
      # of cell(s) missing power table: 1
      # of cell(s) missing leakage table: 1
      # of MSMV cell(s) missing power_level: 0
      --------------------------------------------------------
CellName                                Missing Table(s)
TIELO                                   internal power, leakge power,



Starting Calculating power
2023-Dec-02 17:18:16 (2023-Dec-02 09:18:16 GMT)
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 10%
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 20%
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 30%
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 40%
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 50%
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 60%
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 70%
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 80%
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT): 90%

Finished Calculating power
2023-Dec-02 17:18:17 (2023-Dec-02 09:18:17 GMT)
Ended Power Computation: (cpu=0:00:02, real=0:00:00, mem(process/total/peak)=2437.28MB/4963.77MB/2437.28MB)

End delay calculation. (MEM=3386.36 CPU=0:00:01.3 REAL=0:00:00.0)
End delay calculation (fullDC). (MEM=3386.36 CPU=0:00:01.4 REAL=0:00:00.0)
Load RC corner of view slow

Begin Power Analysis

            0V     VSS
         1.62V     VDD

Begin Processing Timing Library for Power Calculation

Begin Processing Timing Library for Power Calculation



Begin Processing Power Net/Grid for Power Calculation

Ended Processing Power Net/Grid for Power Calculation: (cpu=0:00:00, real=0:00:00, mem(process/total/peak)=2125.24MB/4837.96MB/2351.93MB)

Begin Processing Timing Window Data for Power Calculation

clk(100MHz) CK: assigning clock clk to net clk
Ended Processing Timing Window Data for Power Calculation: (cpu=0:00:00, real=0:00:00, mem(process/total/peak)=2134.48MB/4837.96MB/2351.93MB)

Begin Processing User Attributes

Ended Processing User Attributes: (cpu=0:00:00, real=0:00:00, mem(process/total/peak)=2134.55MB/4837.96MB/2351.93MB)

Begin Processing Signal Activity
```

Figure 13: Report Power

6. Dataflow and Verification:

In this stage, we will compare our CNN kernel's output with python's built-in convolution method to guarantee our work. To begin with, we will generate the input feature map and the weights through python. We used python's pytorch to generate random values, the specific method is as follow:

```
# randomize input feature map
input_fm = torch.rand(1, input_channel, input_heights, input_weights)*255-128
# guarantee all input are integer
input_fm = torch.round(input_fm)
# randomize weight
weight = torch.rand(output_channel, input_channel, k_size, k_size)*255-128
# guarantee all input are integer
weight = torch.round(weight)
```

The reason for us to have a *255-128 following is because the kernel runs binary data and our design has a channel size of 8. As a result, we define data to have 8 bit length that represents at most ±128 in binary. This specification tells us that we need to generate random number in the range of [-128,128), which explains *255-128. After the rand() method, we use the round() method to make sure all inputs are integers as those have binary representations. At this stage, we are ready to generate the output feature map. Just simply pass in our weights into the convolutional network and then run python's built-in conv2d method to generate the result. This python-generated result will be used to verify the result generated by our own CNN kernel.

From the analysis of the algorithm, we also notice that the input feature map, weights, and output feature map are mutually independent and can be decoupled. This provides insights for the design of our memory access module, leading us to store these three types of data independently in different files, which not only facilitates data reuse but also avoids interference between data exchanges. For our own convenience, making the data more readable, we created a copy of each data file and translated them into decimal representations.

To save data into .txt files, there is no short path. We need to loop through our parameters and write them into the corresponding .txt file. The only thing we can do is to transform parameters into numpy arrays so the data manipulation process can be easier.

```
# transfer the feature map into numpy form for easier operation›
ifm_np = input_fm.data.numpy().astype(int)
weight_np = weight.data.numpy().astype(int)
ofm_np = ofm_relu.data.numpy().astype(int)
```

Following picture shows how we create .txt files for input feature map and save it as binary

```
# write data as a 2's complement binary representation type
with open("ifm_binary.txt"%(input_channel, input_heights, input_weights), "w") as f:
    for i in range(input_channel):
        for j in range(input_heights):
            for k in ifm_np[0, i, j, :]:
                # input feature map has the bit length of 8
                s = np.binary_repr(k, 8) + " "
                f.write(s)
            f.write("\n")
        f.write("\n")
```

Following picture shows how we create .txt files for input feature map and save it as decimal

```
 write out data as decimal type
ith open("ifm_decimal.txt" % (input_channel, input_heights, input_weights), "w") as f:
   for i in range(input_channel):
       for j in range(input_heights):
           for k in ifm_np[0, i, j, :]:
               s = str(k) + "\t "
               f.write(s)
           f.write("\n")
       f.write("\n")
```

Some of the other features we did related to data flow including data tiling. We group a large piece of data into smaller tiles based on our kernel size, channel, and parameter size. We believe data tiling is necessary since it can reduce our kernel's workload and make it become more efficient while processing. Below is a portion of code used for sampling how data was tiled and stored. Notice that we only do data tiling with the data/file we are going to process through our kernel. In the previous step, the parameter data we saved/processed was for display / read-only / testing purposes.

```
tile_length = 16
num_tile = 64//tile_length

with open("ifm.txt", "w") as f:
    for ii in range(13):
        for jj in range(num_tile):
            for c in range(input_channel):
                for j in range(tile_length + 3):
                    col = jj*tile_length + j
                    for i in range(8):
                        row = ii*5+i
                        k = ifm_np[0, c, row, col] if ((row < 64) and (col < 64))else 0
                        s = np.binary_repr(k, 8) + " "
                        f.write(s)
                    f.write("\n")
            f.write("\n")
        f.write("\n")
    f.write("\n")
```

Our CNN kernel will generate a output file called "conv_acc_out.txt" after it finishes processing. We will use this file to compare with data generated by python's built in method. To compare the two, we will use csv.reader to extract data from each .txt file, put them into multi-dimensional vectors/arrays, and compare each numeric value piece by piece inside the data cluster. In the end, we got the exact same result as python's output.
Apply csv to extract data:

```
import csv
standard_result = csv.reader(open("script/ofm.txt"), delimiter=',')
my_result     = csv.reader(open("src/conv_acc_out.txt"), delimiter=' ')
```

Transform the reader object into multi-dimensional vectors/arrays for data manipulation
The following sample shows the process to transfer python results into vectors/arrays:

```
standard_vec = []
for line in standard_result:
    if (line == []):
        continue
    single_vec = []
    for item in line:
        if item != "":
            single_vec.append(item)
    standard_vec.append(single_vec)
```

The following sample shows the comparison process. The logic is very straight forward:

```
result = 0;
for i in range(len(my_vec)):
    if (len(my_vec[i]) != len(standard_vec[i])):
        print( i, " ", len(my_vec[i]), " ",   len(standard_vec[i]))
    for j in range(len(my_vec[i])):
        if (my_vec[i][j] != standard_vec[i][j]):
            result = 1
            print ("\033[33mFail, wrong result", my_vec[i][j], ", ", standard_vec[i][j],"\033[0m")

if (not result):
    print ("\033[32mPass, Correct result !!!\033[0m")
else:
    print ("\033[31mFail, Wrong result !!!\033[0m")
```

Final result marked the success of our project:

```
(base) Yingqiaos-MacBook-Pro:ECE393FinalProject gyq888$ /usr/bin/python3 /Users/gyq888/Desktc
ct/script/compare.py
Testing 488 [ConvKernel results]  488 [standard results]  lines
Pass, Correct result !!!
```

Conclusion

1. What We Learned

Designing a Neural Network Processor as a project offers us a range of valuable insights and learning experiences. Knowledge wise, this project deepens our understanding of how neural networks work, including various architectures, activation functions, and optimization techniques. Those expertise are foundational for us to design chips that serve popular fields such as machine learning and artificial intelligence. Collaboration wise, this project emphasizes the importance of effective communication, planning, and conflict resolution skills. With effective communication through zoom, we eliminate the schedule uncertainties of face-to-face meetings. With scientific planning, we avoid working overload before the due date. Confliction is

something that couldn't be avoided in a group project. We did have different ideas about the designing approach and workload distribution throughout the quarter. But eventually, we negotiate and compromise with each other so we can unite to reach our common goal.

2. Future Improvements

Since we finished the convolution and the ReLU for our NPU we would like to add another layer of functionality. Convolution and ReLU are both two main features that are part of CNN but other essential components such as pooling and padding will be important to add. Besides, we believe that it is important for us to train the CNN model further. Although we proved that our CNN kernel produces the same result as the python build-in method, we only tested it with randomly generated numeric data. We would love to use more complicated input, such as images, to train and verify our model. We believe such a process will improve our model's ability and accuracy. Also, we would like to consider adding memory such as SRAM and DRAM instead of loading random weights and implementing real time classification would be interesting as well. At last, backend verification is an essential process for any chip designing project, and it is certainly crucial for us to add it in the future. Because it ensures the physical layout of the chip matches our intended design. Moreover, backend verification helps in achieving timing closure, ensuring that signals meet the required setup and hold times and preventing issues like clock skew. With these potential improvements done, we believe our project will be more successful and can certainly act as a strong supporting material when we are in the industry.

3. Work and Split

Work-split for the overall project is as following:
Yingqiao Gou: Dataflow, front-end verification, final report, presentation(Assistant)
Jingyang Cui: Front-end simulation, back-end design & verificaiton, final report, front-end design(Assistant)
Giovanni Michel: Front-end design & front-end verification, presentation, front-end simulation, front-end verification, and final report

4. References

1.Y. -H. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127-138, Jan. 2017, doi: 10.1109/JSSC.2016.2616357.

2. J. J. Tithi, N. C. Crago and J. S. Emer, "Exploiting spatial architectures for edit distance algorithms," 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, 2014, pp. 23-34, doi: 10.1109/ISPASS.2014.6844458.

3. Seo, Jae-sun, Bernard Brezzo, Yong Liu, Benjamin D. Parker, Steven K. Esser, Robert K. Montoye, Bipin Rajendran, et al. “A 45nm CMOS Neuromorphic Chip with a Scalable Architecture for Learning in Networks of Spiking Neurons.” *2011 IEEE Custom Integrated Circuits Conference (CICC)*, 2011. https://doi.org/10.1109/cicc.2011.6055293.

4.Akopyan, Filipp, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, et al. “TrueNorth: Design and Tool Flow of a 65 MW 1 Million Neuron Programmable Neurosynaptic Chip.” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, no. 10 (2015): 1537–57. https://doi.org/10.1109/tcad.2015.2474396.