DNN Accelerator Technical Report

Li Beier, Li Kairui

January 23, 2026

1 High-Level Model Correlation (Software)

We provide a script `Test_Generator/simulator.py` to ensure this.

2 Hardware Correlation (Simulation)

Report simulation results showing that RTL outputs align with the provided reference vectors. Include stimulus sources (testbenches), quantization settings, and coverage of convolution, pooling, and post-processing kernels.
Specifically, each script under `Test_Generator/` folder generates test data for different modules. Then run the corresponding testbench under `DNN_Accel.srcs/sim_1/new/` to verify the RTL implementation of each module. The testbenches read the generated test data, feed them into the DUT, and compare the output with the golden data.

3 Architecture and Parallel Strategy

The overall architecture of the system is shown in the figure:

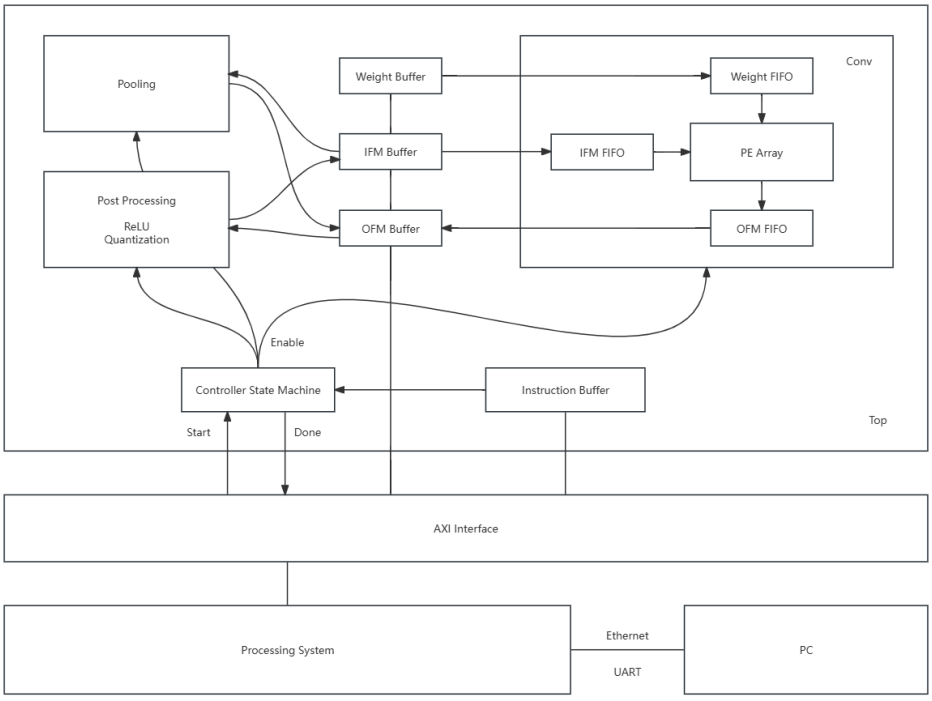

Figure 1: DNN Accelerator Architecture

3.1 Dataflow

Convolution unit reads input feature map data from IFM BRAM and convolution kernel weights from Weight BRAM. After processing by the convolution computation unit, the output feature map data is written to OFM BRAM. The pooling unit reads input feature map data from IFM BRAM, processes it through the pooling computation unit, and writes the output feature map data to OFM BRAM. The post-processing unit reads the data processed by convolution/pooling from OFM BRAM, performs quantization and ReLU processing, and writes the final results back to IFM BRAM for use in the next layer.
In convolution and pooling layers, data needs to be repeatedly accumulated/compared at the output. Therefore, the OFM BRAM is designed as a dual-port RAM to read-accumulate-write data in parallel.

3.2 PE Array

Each processing element (PE) in the array is responsible for performing MAC operations, that is, multiplying the stored weight with the value from left neighbor PE and accumulating the result with the value from the top neighbor PE. The PE array is organized in a 2D 16×16 grid. We adopt a weight-stationary dataflow. We load weights into the PE array, then stream input feature map (IFM) values from the left side (with each row input staggered by one cycle) and from the bottom side (with each column output staggered by one cycle) to complete a matrix-vector multiplication.
If we continuously stream vectors from the left side, the PE array can perform matrix multiplication.

3.3 Convolution

The convolution operation can be expressed by the following equation:

$$OFM(c_{out}, x, y) = \sum_{c_{in}=0}^{C_{in}-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} IFM(c_{in}, x + si - p, y + sj - p) \cdot W(c_{out}, c_{in}, i, j) \quad (1)$$

where OFM denotes the output feature map, IFM denotes the input feature map, and W denotes the convolution kernel weights. c_{in} and c_{out} represent the number of input and output channels, respectively. K represents the kernel size, s represents the stride, and p represents the padding size.
Note that if we change the order of the indices, placing the loops over c_{in} and c_{out} innermost, and use a state machine to control the changes of indices x, y, i, j related to convolution, we can transform the convolution operation into multiple matrix-vector multiplications followed by accumulation, thus utilizing the PE array for computation.
We need to align the channel dimension to an integer multiple of 16 and perform blocked matrix multiplication between input channels and output channels. The extra weights and input feature maps are padded with zeros.
It is worth mentioning that we have added FIFO buffers to all inputs and outputs of the PE Array. On one hand, this can alleviate timing pressure, and on the other hand, it simplifies the design of control logic (read/write signals).

3.4 Pooling

Pooling layer also uses a state machine to generate addresses. The only difference from convolution is that the input and output channels are the same here. For average pooling, our approach is to first accumulate all the values to be pooled, and then merge the division operation into the subsequent quantization operation. Therefore, we only support pooling window sizes that are powers of 2.

3.5 Post-Processing

Post-processing mainly includes quantization and ReLU operations. The quantization operation is implemented using fixed-point arithmetic, where we right-shift the accumulated result by a specified number of bits to reduce precision. The ReLU operation is straightforward, setting any negative values to zero.

3.6 Linear

We regard the the linear layer as a special case of convolution layer with kernel size 1×1. Therefore, it shares the same hardware module with the convolution layer.

3.7 Instruction and Executor

Each instruction is 64 bits long:

From high to low bits:

- bits [63:60]: opcode (4 bits) - 0000: Convolution, 0001: Max Pooling, 0011: Average Pooling, FFFF: Halt.
- bits [59:56]: 1 if ReLU is enabled, 0 otherwise. (4 bits)
- bits [55:48]: Quantization (8 bits) - number of bits to right shift.
- bits [47:32]: Weight BRAM base address (16 bits).
- bits [31:24]: Input feature map size (8 bits).
- bits [23:20]: Kernel size (4 bits).
- bits [19:18]: Stride (2 bits).
- bits [17:16]: Padding (2 bits).
- bits [15:8]: Output feature map size (8 bits).
- bits [7:4]: Number of input channels divided by 16 (4 bits).
- bits [3:0]: Number of output channels divided by 16 (4 bits).

Although the output feature map size can be calculated from input feature map size, kernel size, stride, and padding, we still store it as a separate field in the instruction to avoid complex calculations in the control logic.
In the executor, we use a simple state machine to control the currently enabled module. According to the opcode field in the instruction, we first start the convolution or pooling module, with intermediate results written to OFM BRAM. Then we start the post-processing module, which writes the quantized and activated values back to IFM BRAM and clears the values in OFM BRAM for the next layer.
The Top module fetches instructions from Instruction BRAM one by one and feeds them to the executor until it encounters the Halt instruction.

3.8 Memory Layout

- IFM/OFM BRAM: [channels / 16][y][x][16]
- Weight BRAM: [out channels / 16][in channels / 16][ky][kx][16(in)][16(out)]

The advantages of this design are obvious: since we use weight stationary, when loading weights, the address increments by 1 each time, with a 128-bit width, 16 numbers per group (corresponding to a row of the PE Array). After loading 16 times, it stops, and kernel y and kernel x do not change, waiting for the state machine to scan through output y and output x before continuing to load the next group of weights. For output, when the BRAM width is set to 16*ACC_WIDTH, the address changes basically by +1 each time. This design can maximize the simplification of address generation logic and reduce the complexity of control logic.

4 Static Timing Analysis (STA)

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 21	0.534	18	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top..._reg[11][0]/CE	8.872	3.987	4.885	10.0
Path 22	0.534	18	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top_...reg[11][10]/CE	8.872	3.987	4.885	10.0
Path 23	0.534	18	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top_...reg[11][11]/CE	8.872	3.987	4.885	10.0
Path 24	0.534	18	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top_...reg[11][12]/CE	8.872	3.987	4.885	10.0
Path 25	0.534	18	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top_...reg[11][13]/CE	8.872	3.987	4.885	10.0
Path 26	0.534	18	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top_...reg[11][14]/CE	8.872	3.987	4.885	10.0
Path 27	0.534	18	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top..._reg[11][8]/CE	8.872	3.987	4.885	10.0
Path 28	0.534	18	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top..._reg[11][9]/CE	8.872	3.987	4.885	10.0
Path 29	0.578	17	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top_...reg[10][14]/CE	8.813	4.000	4.813	10.0
Path 30	0.578	17	115	design_1_i/T..._0/CLKBWRCLK	design_1_i/Top..._reg[10][1]/CE	8.813	4.000	4.813	10.0

Figure 2: Timing Report Summary

The system frequency can reach 100MHz. The main bottlenecks appear in two areas: one is the comparison logic in the AGU that determines whether the address is out of bounds, and the other appears in various variable multiplications in the control logic, especially in the AGU.

5 Power and Resource Utilization

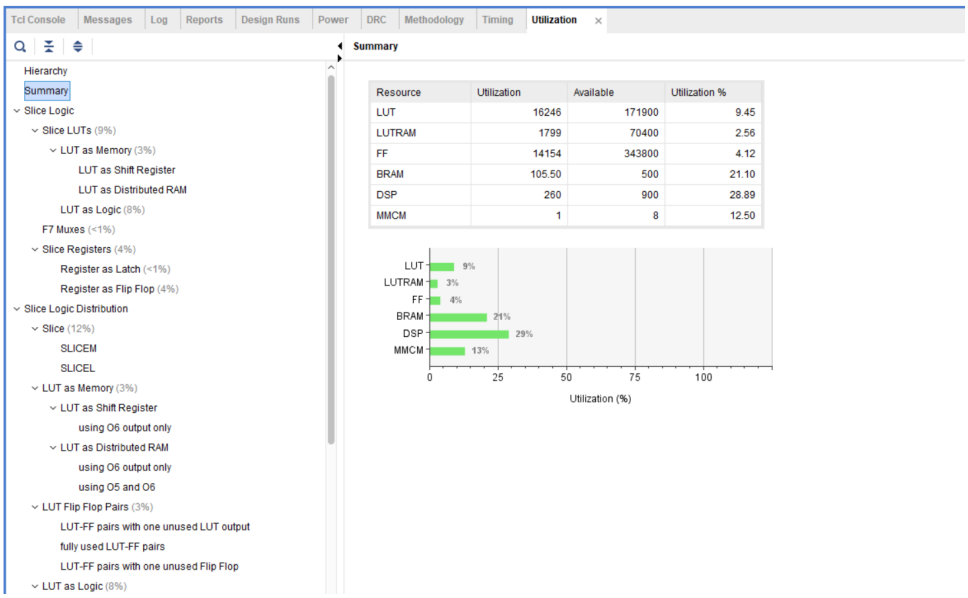

Figure 3: Resource Utilization Report

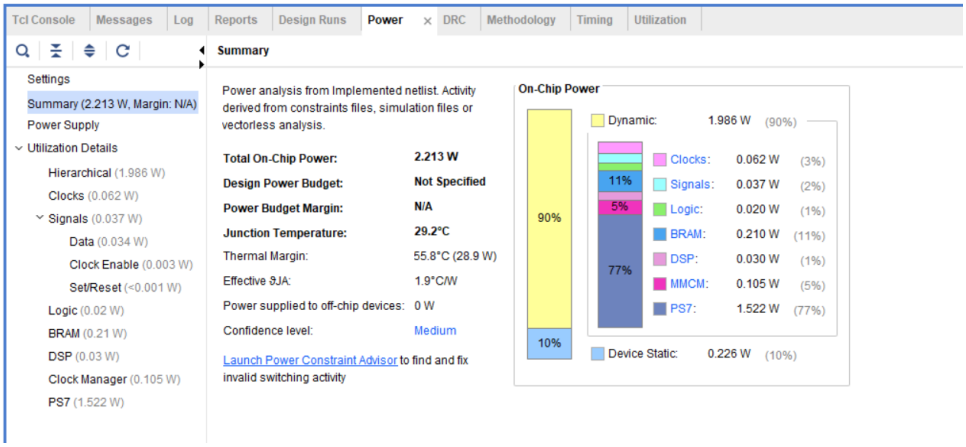

Figure 4: Power Report Summary

6 End-to-End Runtime

Layer	Type	IFM Size	Input Channel	OFM Size	Output Channel	Kernel Size	Stride	Padding (zero)	ReLU	Q_{IN}	Q_{OUT}	Q_W
Layer-1	Conv	32^2	1	32^2	32	5×5	1	2	√	7	5	7
Layer-2	Max Pooling	32^2	32	16^2	32	2×2	2	0	×	5	5	/
Layer-3	Conv	16^2	32	16^2	64	3×3	1	1	√	5	5	8
Layer-4	Conv	16^2	64	8^2	64	3×3	2	1	√	5	5	8
Layer-5	Conv	8^2	64	4^2	128	3×3	2	1	√	5	5	8
Layer-6	Average Pooling	4^2	128	1^2	128	4×4	1	0	×	5	5	/
Layer-7	FC	1	128	1	10	1×1	/	/	×	5	5	6

Figure 5: CNN Model Used in End-to-End Test

Refer to the project README.

7 PL Interface of the DPU

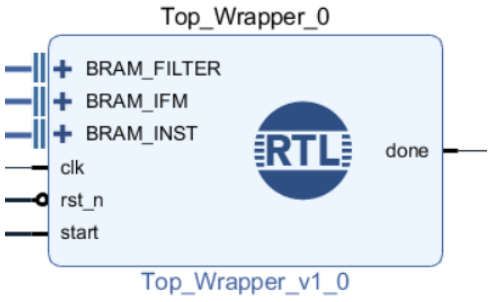

Figure 6: PL Interface of the DPU

The interfaces of the three BRAMs are directly connected to the AXI BRAM Controller, mapped to different address segments on the bus. During use, first write instructions to the Instruction BRAM and weights to the Filter BRAM, then give a rising edge to start. Once completed, done goes high, at which point the output results can be read from the IFM BRAM.

8 Limitations and Future Work

8.1 Small number of channels

When the number of channels in a layer is small (for example, the first convolutional layer has only one input channel), since the number of channels needs to be aligned to an integer multiple of 16, a large amount of computing resources are wasted on processing padded zeros, greatly reducing the utilization of computing units. In fact, the first layer consumes half of the computation time, while the utilization is less than 1/16!

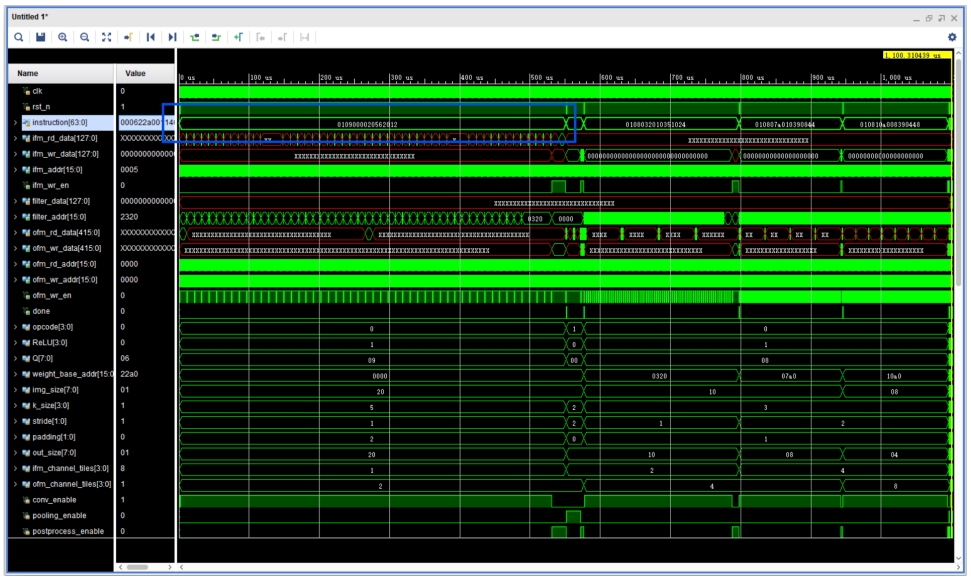

Figure 7: Inefficiency of Conv 1

One possible optimization method is to use the Img2Col approach to unfold the input feature map into a matrix. However, this method would introduce additional complexity in control logic and memory overhead, requiring a trade-off analysis.

8.2 Frequency Bottleneck

The current design can only reach a frequency of 100MHz, which is relatively low for FPGA designs. The main bottlenecks are in the address generation unit (AGU) and control logic, where complex combinational logic leads to long critical paths. Future work could focus on optimizing these modules, possibly by pipelining or pre-computing multiplications of layer parameters in PS side software.

9 Conclusion

This report presents the design and implementation of a DNN accelerator on FPGA, detailing its architecture, dataflow, instruction set, and memory layout. The accelerator supports convolution, pooling, and post-processing operations, utilizing a weight-stationary dataflow and a PE array for efficient computation. Simulation results confirm the correctness of the RTL implementation, and static timing analysis shows that the design meets the target frequency of 100MHz. Resource utilization and power consumption are also analyzed, providing insights into the design's efficiency. Limitations such as low channel utilization in early layers and frequency bottlenecks are discussed, along with potential future improvements.