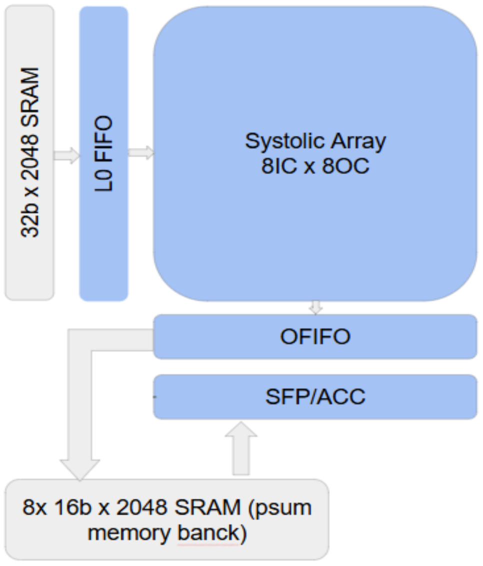
Part 1(Baseline Vanilla Version)

Hardware Design Components

In this hardware model we implement the 2-D Systolic Array with 8 input and 8 output channel in Weight Stationary mode. The Core consists of L0 FIFO, OFIFO, MAC_Array (Systolic 8*8 Array),Accumulator, and 9 SRAMs. The L0 FIFO is used for loading the input activations and weights. The OFIFO is used for loading the psum to the SRAM. MAC_Array computes the partial sum using the weight stationary method. Accumulator is used to finally add the correct Nij and Kij combination to get the output for each output channel. We have a one 32*2048 SRAMs (Activation SRAM) to store the input activations, and eight (one for each output channel) 16*2048 SRAMs (PSUM Memory bank) to store the psum for all Kij and Nij combinations. We will have a total of 324 (36 Input activations * 9 Kij) Address locations occupied in each of the 8 output SRAMs.



Testbench Verification Flow

The Testbench initially stores all the input activations in the Activation SRAM (starting address 0) via Input txt file generated through the python scripts for VGG-16 4bit Quantized Model. After that for one Kij we will store the weights to the Activation SRAM (starting address 400 in hex) via the weight_kij.txt files. These weights will then be read from SRAM to the L0 FIFO which will write it to the MAC Array.

Now we use parallel threads (fork/join) to read the Activations from the memory, send it to the MAC Array via the L0 FIFO for computation, and store the computed value in the PSUM memory bank for each output channel. All the computations for Kij=0 and Nij=0 to 35 happen for all the output channels and we get some clk cycle reduction in simulation because of the usage of fork-join and parallel execution. Similarly for the remaining Kijs we will calculate the PSUMs.

After that we will perform the accumulation via the SFU by selecting the correct address location from 0 to 323. This is done through acc_final.txt file which will provide the correct address location for each of the 16 output nijs for all the 8 channels. We finally apply the Relu_en signal and compare the computed output with the expected output generated by the python script. The testbench and design was successfully verified.

FPGA Synthesis for Vanilla Model — was done using Quartus Prime Cyclone IV GX FPGA family and device EP4CGX150DF31I7AD. We got Max frequency of 121.07MHz and dynamic power dissipation of 35.96 mW. Detailed summary and description is provided in our submission along with the RTL Netlist diagram. The Synthesis of the corelet was completed successfully.

Quantization aware training was performed for the model with 4bit input activations and 4bit weight after modifying the 29th layer of the original VGG-16 model to reduce the input and output channels. This was done so that we can use those input and output channel values for verifying our hardware. We achieved a 92.02% accuracy and Quantization error of 2.73e-06 with the updated model.

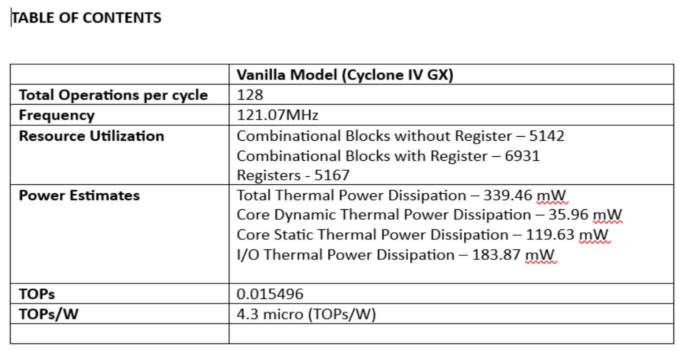
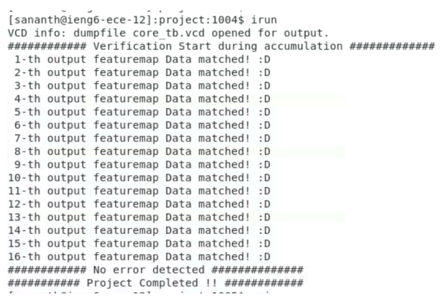


TABLE OF CONTENTS

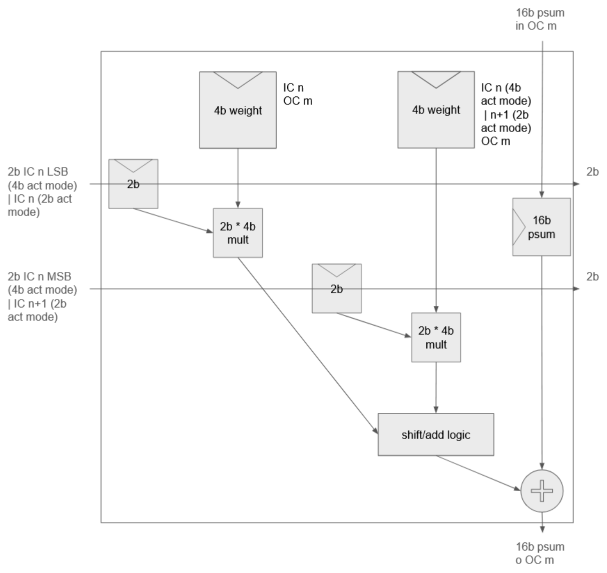
	Vanilla Model (Cyclone IV GX)
Total Operations per cycle	128
Frequency	121.07MHz
Resource Utilization	Combinational Blocks without Register – 5142 Combinational Blocks with Register – 6931 Registers - 5167
Power Estimates	Total Thermal Power Dissipation – 339.46 mW Core Dynamic Thermal Power Dissipation – 35.96 mW Core Static Thermal Power Dissipation – 119.63 mW I/O Thermal Power Dissipation – 183.87 mW
TOPs	0.015496
TOPs/W	4.3 micro (TOPs/W)

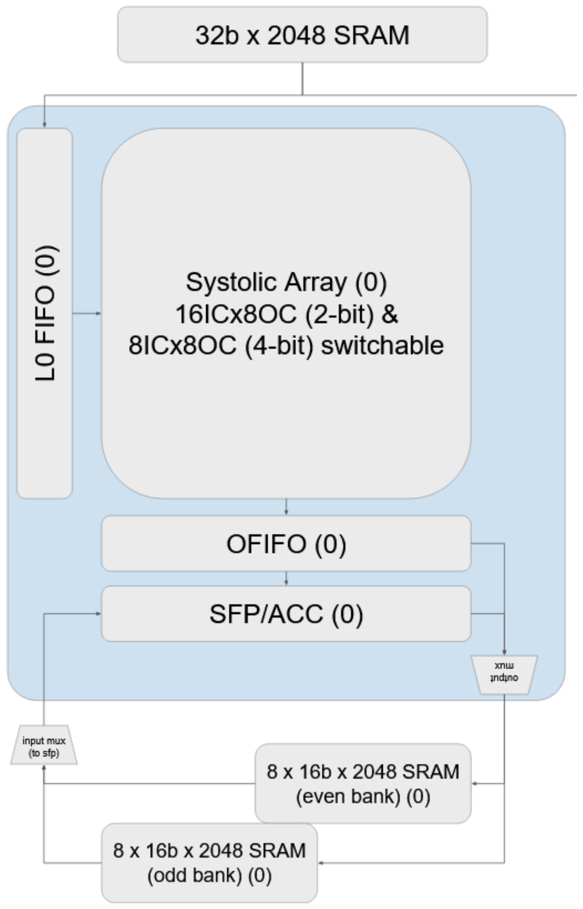


```
[sananth@ieng6-ece-12]:project:1004$ irun
VCD info: dumpfile core_tb.vcd opened for output.
############ Verification Start during accumulation #############
 1-th output featuremap Data matched! :D
 2-th output featuremap Data matched! :D
 3-th output featuremap Data matched! :D
 4-th output featuremap Data matched! :D
 5-th output featuremap Data matched! :D
 6-th output featuremap Data matched! :D
 7-th output featuremap Data matched! :D
 8-th output featuremap Data matched! :D
 9-th output featuremap Data matched! :D
10-th output featuremap Data matched! :D
11-th output featuremap Data matched! :D
12-th output featuremap Data matched! :D
13-th output featuremap Data matched! :D
14-th output featuremap Data matched! :D
15-th output featuremap Data matched! :D
16-th output featuremap Data matched! :D
############ No error detected ##############
########### Project Completed !! ############
```

Part 2

Part 2 is a “2-bit and 4-bit lane reconfigureable SIMD Systolic Array”. In this case, the hardware is designed to perform matrix multiplication/2d convolution with either 2-bit input activations and 4-bit weights for 16 input channels and 8 output channels, or 4-bit input activations and 4-bit weights for 8 input channels and 8 output channels.


16b psum in OC m
4b weight
IC n OC m
4b weight
IC n (4b act mode) \| n+1 (2b act mode) OC m
2b IC n LSB (4b act mode) \| IC n (2b act mode)
2b
2b
2b * 4b mult
16b psum
2b IC n MSB (4b act mode) \| IC n+1 (2b act mode)
2b
2b
2b * 4b mult
shift/add logic
16b psum o OC m


32b x 2048 SRAM
L0 FIFO (0)
Systolic Array (0) 16ICx8OC (2-bit) & 8ICx8OC (4-bit) switchable
OFIFO (0)
SFP/ACC (0)
output mux
input mux (to sfp)
8 x 16b x 2048 SRAM (even bank) (0)
8 x 16b x 2048 SRAM (odd bank) (0)

file	ACT	WGT	OUT
format	#time0ic15[msb-lsb],time0ic6[msb-lst],....,time0ic0[msb-lst]# #time1ic15[msb-lsb],time1ic6[msb-lst],....,time1ic0[msb-lst]#	#oc0ic14[msb-lsb],oc0ic12[msb-lst],....,oc0ic0[msb-lst]# #oc0ic15[msb-lsb],oc0ic13[msb-lst],....,oc0ic1[msb-lst]# #oc1ic14[msb-lsb],oc1ic12[msb-lst],....,oc1ic0[msb-lst]#	#time0oc7[msb-lsb],time0oc6[msb-lst],....,time0oc0[msb-lst]# #time1oc7[msb-lsb],time1oc6[msb-lst],....,time1oc0[msb-lst]#
explanation	ACT file should be IC in reverse order, time in forward order.	WGT should be IC in reverse order, splitting alternating EVEN and ODD ICs in next row. OC in forward order.	OUT should be in OC reverse order , time in forward order.

The table above describes the exact file format needed to use the systolic array correctly.

The PE tile is based off of the vanilla PE tile with a few crucial modifications to ensure it works with 2-bit and 4-bit activations.

In “weight loading” mode, the weights are loaded in through the two 2b west buses. The LSB and MSB are rejoined and cast into the 4b weight registers. In 2b mode we require 2x different 4b weights, accomplished over 2 cycles: in the first cycle, the first (IC n) weight is filled and in the second cycle the second (IC n+1) weight is filled. In 4b mode, this only requires 1x 4b weight, which can be accomplished in one cycle for the PE tile: the weight registers are filled simultaneously with the same weight.

In “execution” mode, we assume that the weights are already loaded in (though in reality it may start slightly earlier). In 2-bit mode, we can see that it works as simply two PE tiles vertically tiled together to work as 2 input channels to the same output channel with 2-bit activations. In 4-bit mode, the LSB and MSB are split across the 2x 2-bit input buses. The MSB is multiplied by the 4-bit weight to form a 6-bit partial, which should be shifted left by 2 (and sign extended) before being summed with the 6-bit LSB partial (also sign extended). This partial sum is then added to the input northern partial sum.

Partial sums are finally accumulated in the output FIFO. Integrating two alphas into the base part 2, we use a scheme with 2x8 16-bit SRAM banks. Alpha (1) is to allow for continuous accumulation (reading from 1 bank into the next with continuous OFIFO summation). Alpha (2) reduces these KIJ=9 banks into 2 banks. One bank is intended to accumulate for odd KIJ and the other bank is intended to accumulate for even KIJ (this is signified by inputting a bank selector). In other words, we read out of one SRAM bank, sum with output from OFIFO, and write into the other SRAM bank. This double buffered PSUM sram methodology allows us to complete accumulation in a single cycle – we can write and read at the same cycle which is prevented by only having one SRAM bank. If RELU instruction is passed on (should be passed on the last KIJ accumulation), it will apply RELU before writing into the SRAM.

For the software portion of Part 2, we trained a quantization-aware VGG16 model on CIFAR-10 using 2-bit input activations and 4-bit weights, while modifying a single convolution layer to have 16 input channels and 16 output channels. This squeezed layer was placed near the end of the feature extractor (around the 27th layer) to reduce the spatial dimension nij and keep verification tractable. The batch-normalization layer following this convolution was removed, replacing the original conv → BN → ReLU sequence with conv → ReLU, to match the intended hardware datapath.

Quantization-aware training was performed end-to-end, and the final model achieved 90.27% top-1 accuracy, exceeding the required threshold. To verify hardware correctness, we reconstructed the integer partial sums (psum_recovered) for the squeezed 16×16 convolution layer using integer convolution with quantized activations and weights, followed by rescaling, bias addition, and ReLU. This recovered output was compared against a reference computed by applying a floating-point convolution on quantized tensors and then ReLU, corresponding to the pre-hooked input of the next layer.

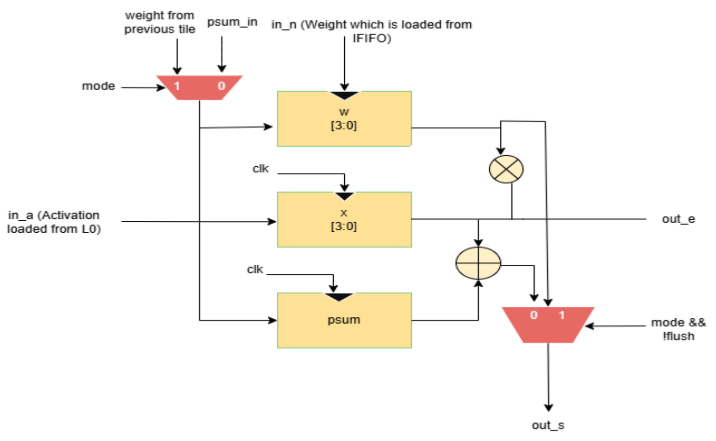
The resulting mean-squared error was 7.33×10^{-13} with a maximum absolute error of 3.05×10^{-5}, well below the required 10^{-3} threshold. This confirms that the integer arithmetic, scaling, and ReLU behavior of the software model are consistent with the expected hardware behavior for the 2-bit activation / 4-bit weight configuration. Exported activation, weight, and output tensors from this layer were used as golden references for RTL verification.

Part 3

Part 3 is the version where we develop a configurable PE which supports both weight and output stationary. In this we are using a signal named “mode” to either configure the PE to be Weight stationary or Output Stationary. When the mode = 1 , the PE works based on output stationary and when the mode = 0 , the PE works based on input stationary. Unlike , in “Vanilla Model” where the weight and activation are both loaded from L0 , here the weights are loaded from IFIFO . Thus the weights are propagated from north to south from IFIFO during the load phase and the activations are propagated from left to right from L0 . In WS mode , the weights are stationary during the execute phase and the psum of the current tile is propagated to the tile below (north to south). In OS mode , the weights of the previous tile gets propagated to the next tile while the psums are stationary .

In order to propagate data from one tile to the tile below (ie from north to south) , instead of using multiple separate signals , we have used a single in_n signal . This signal is designed in such a way that it will carry the weight values from north to south during load phase irrespective of the mode . On the other hand , during the execution phase , if it is mode = 1 (ie OS mode) , it will propagate the weight from one tile to the next and during mode = 0 (ie WS mode) , it will propagate the psum from one tile to the next.

In OS mode, the psums are propagated only when the entire execution is done and the flush signal is enabled . The flush signal will be enabled at the end of execution and once the flush signal is enabled the out_s signal will carry the psum value of the tile . This forwarding logic is enabled based on nested conditions where if the mode = 0 , out_s which is the output of the current tile will be assigned to the psum value of this tile whereas if the mode =1 then the value which gets propagated depends on the flush value . If the mode is 1 and the flush is not enabled then the weight from this tile is propagated to the next from north to south and when the flush is enabled then the psum will be propagated.


weight from previous tile
psum_in
in_n (Weight which is loaded from IFIFO)
mode
1
0
w [3:0]
clk
x [3:0]
in_a (Activation loaded from L0)
out_e
clk
psum
0
1
mode && !flush
out_s

Alphas

1) Dual SRAM Buffering & Continuous Psum Accumulation

Explained in Part 2 as it was developed in conjunction with the rest of it.

- **Output buffer:** Two single-port SRAMs, each with **8 banks for 8 output channels (OC)**, organized in a **ping-pong** fashion.
- **Parallel read–write:**
  - In each cycle, **one SRAM is read** while the **other is written**, enabling **continuous accumulation** without stalls.
- **Throughput benefit:**
  - Achieves up to **1.8× speedup** vs. a baseline that waits for *all* kernels' partial sums before accumulating.
- **Area efficiency:**
  - **78% memory area reduction** by **eliminating per-kernel psum storage** and directly keeping only accumulated outputs. (Occupies only the output matrix size)

Example (K2 → K3 flow):

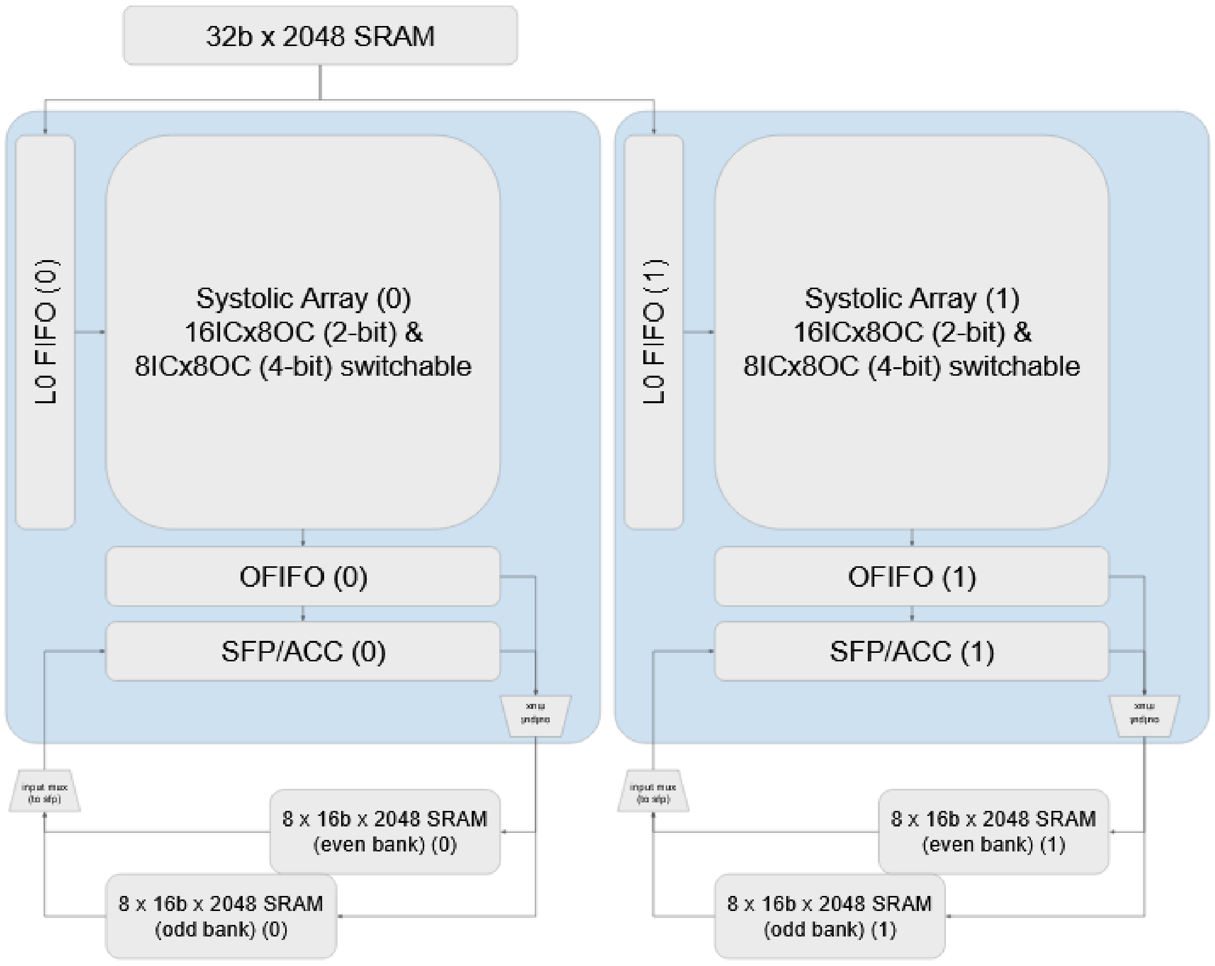
- For kernel **K2**: read psum from **Mem1**, accumulate, and write the updated value to **Mem2**.
- For kernel **K3**: read from **Mem2**, accumulate, and write back to **Mem1**.
- This alternating pattern continues across kernels, maintaining **high utilization** of both SRAMs.

2) 2x Horizontal Tiling of 2/4 bit SIMD for squeezed 2bit VGG16 16x16 convolution layer

Parallel horizontal tiling to instantiate 2 individual corelets fed by the same SRAM allows us to execute the convolution simultaneously.

Weights must be loaded in sequentially: the selector should be used to feed either weights to corelet 0 or corelet 1. During execution, activations should be simultaneously fed to both corelets as they use the same activation tile.

Upon completing the full convolution, the user may only pass in one SRAM read address at a time, and there is only one SRAM output bus, so the user may only read one of the SRAMs at a time, thus taking 16 cycles (plus overhead) to read all 16 outputs. It's suggested to use Core 0 for “even” ICs and Core 1 for “odd” ICs as that’s how our testbench is written, however it may also make sense to use Core 0 for the lower 7:0 ICs and Core 1 for the upper 15:8 ICs, whichever way the user prefers to have the output formatted/read in SRAM.


32b x 2048 SRAM
L0 FIFO (0)
Systolic Array (0)
16ICx8OC (2-bit) &
8ICx8OC (4-bit) switchable
OFIFO (0)
SFP/ACC (0)
input mux (to sfp)
8 x 16b x 2048 SRAM (even bank) (0)
8 x 16b x 2048 SRAM (odd bank) (0)
L0 FIFO (1)
Systolic Array (1)
16ICx8OC (2-bit) &
8ICx8OC (4-bit) switchable
OFIFO (1)
SFP/ACC (1)
input mux (to sfp)
8 x 16b x 2048 SRAM (even bank) (1)
8 x 16b x 2048 SRAM (odd bank) (1)

3) Receptive Field Optimized Computation

For an *MxM* matrix, each kernel kij needs to be multiplied only by *(M-K+1)*(M-K+1)* activations. So we feed only the required activations for a kij efficiently. This gives a speedup of *(MxM) / ((M-K+1) * (M-K+1))*.
For the M value of 6 and K value of 3, we get **2.25x speedup** and output memory **area savings of *(K-1)*OC***.

4) Automated Control Logic done for Weight Stationary

We give the start signal after loading the activation and weight into the memory (IDLE), and then the core will do the kernel loading from memory to L0 (LOAD_WGT_L0), and then load the weight from L0 to the systolic array (LOAD_WGT_PE), and then Load Activations from Memory to L0 to PE (LOAD_ACT_L0), and finally Execute and Write the final output matrix in output memory and send back a signal done to core_tb (EXECUTE).

5 STATE FSM

```
IDLE        = 3'd0;
LOAD_WGT_L0  = 3'd1;
LOAD_WGT_PE  = 3'd2;
LOAD_ACT_L0  = 3'd3;
EXECUTE      = 3'd4;
```

Software Alphas

Activation Sparsity Analysis and Zero-Skipping Opportunity in Quantized VGG16

We measured activation sparsity in the trained 2-bit activation / 4-bit weight VGG16 model to quantify the potential benefit of zero-skipping in hardware. During inference, we instrumented all ReLU layers and recorded the fraction of activations that were exactly zero across 10 validation batches. In total, 130 ReLU layer executions were evaluated. The average zero-activation ratio was 68.32%, with a minimum of 48.48% and a maximum of 98.79%, reflecting both layer-wise and input-dependent variation in sparsity.

This result indicates that a large fraction of multiply-accumulate (MAC) operations are mathematically redundant after ReLU. In the ideal case, zero-activation gating could skip approximately 68% of MAC operations, directly translating to reduced dynamic power and memory activity. This motivates unstructured sparsity-aware gating, where MACs and memory accesses are skipped for zero activations without affecting correctness.

Structured Pruning

The **unpruned quantized model achieved 92.0% accuracy** on the test dataset. After applying L1-norm–based structured filter pruning and subsequent fine-tuning, the **model achieved 90.27%** training **accuracy** and 89.7% test accuracy. The observed ~2.3% accuracy degradation is expected due to the permanent removal of convolutional filters, which reduces model capacity. Importantly, the small train–test gap indicates good generalization. This accuracy trade-off is justified by the substantial reductions in MAC operations, model size, and memory bandwidth, making the pruned model significantly more suitable for systolic-array-based hardware deployment.

Deep CNNs contain a large number of redundant parameters, especially in convolutional layers. While unstructured pruning creates sparse weight matrices, it does **not translate efficiently to hardware acceleration**, particularly on **systolic array architectures**, which are optimized for dense, regular computations.

To address this, we applied **structured pruning**, which removes entire computational structures (filters/channels) instead of individual weights. This enables **actual reductions in computation, memory access, and execution latency on hardware**.

Sparsity : 0.4998

Estimated Memory Bandwidth Reduction: 49.98%

We used **L1-norm–based structured pruning** applied to convolutional layers.

- **Granularity**: Filter-level (output-channel level)
- **Criterion**: L1 norm of each convolutional filter
- **Target layers**: Quantized convolution layers (`QuantConv2d`)
- **Weight precision**: 4-bit (INT4)

The pruning process followed these steps:

1. **Train quantized model (INT4 weights & activations)**
2. **Apply structured L1 pruning to convolutional layers**
3. **Fine-tune the pruned model** to recover accuracy
4. **Evaluate accuracy, sparsity, MACs, and hardware behavior**

This ensured that pruning-induced accuracy loss was minimized.