ECE 284 Project: VGGNet on 2D systolic array and mapping on Cyclone IV GX

Suraj Sathya Prakash, Surya Lakshmi Subba Rao Pilla, Yash Jain
Electrical and Computer Engineering
Course: Low power VLSI for Machine Learning

I. MOTIVATION

This project optimizes $VGG16$ for hardware by leveraging quantization and channel reduction. Focused on RTL core design, FPGA mapping, and multi-channel implementation, we aim for efficient 8×8 array mapping, precise testing, and enhanced throughput. This project highlights our innovations for high hardware efficiency and accuracy.

II. OBJECTIVE

A. Part 1: Train VGG16 with Quantization-Aware Training

- Train $VGG16$ for 4-bit input activation and weight, targeting accuracy of over 90%.
- Optimize a convolution layer by reducing its input and output channel numbers to 8 and remove the batch normalization layer following it. Map the optimized layer onto an 8×8 2D systolic array.

B. Part 2: Complete RTL Core Design

- Design and connect various components, including a 2D array with MAC units, scratchpad memories, and a special function processor. Achieve a complete core design without compilation errors.

C. Part 3: Test Bench Generation

- Develop a testbench to run various stages of the system, ensuring all stages are correctly implemented and verified.
- Generate stimulus and expected output files, ensuring zero verification error.

D. Part 4: Mapping on FPGA

- Map the 'corelet.v' on the FPGA using Quartus Prime.
- Complete synthesis, placement, and routing while measuring frequency and power.
- Report the final frequency, power numbers, and efficiency metrics.

E. Part 5: Multi-channel Implementation in Each PE

- Implement multi-channel execution in each Processing Element (PE).
- Ensure zero verification error in RTL results compared to PyTorch simulation estimates.

F. Part 6: +Alpha Enhancements

- Add custom enhancements or techniques to improve the system.
- Conduct thorough verification of the enhancements and present the results.
- Report improvements in TOPS/watt, area, and per second metrics when mapped to FPGA.

III. IMPLEMENTATION

A. *Train VGG16 with Quantization-Aware Training*

As per the instructions in part 1, the VGG16 is modified as shown in Fig. 1 to replace a conv layer with 8 input and output channels. To achieve this 27^{th} layer is modified to have 8 input channels and 8 output channels and the batch normalization layer is removed. The accuracy for $VGG16$ with and without pruning is tabulated in Table I:

TABLE I
VGG16 RESULTS

Model	Accuracy	output difference
VGG16	91.38 %	5.9093e-8
VGG16 - parameterized multichannel	89 %	7.0089e-7

B. *2D systolic array design*

The systolic array is designed for the 8×8 conv layer. The input image size is 4×4, and the output size is 4×4. It comprises two SRAM modules, one for storing weights and activations and the other for storing output. One SRAM has 108 rows containing 8 elements, each consisting of four bits, resulting in a size of 108×32. This is shown in Fig. 2.

C. *Test Bench Generation*

From a functional standpoint, it is noticed that the systolic array operates independently with minimal external control, so we designed it to be as standalone as possible. The testbench mainly manages data loading into the core, triggers computation initiation, and validates the outcomes. The FSM takes care of the remaining complexities, as previously described. While this setup adequately serves a network that aligns perfectly with the array, such as the VGG16 layer in our scenario, the testbench handles data rearrangement.

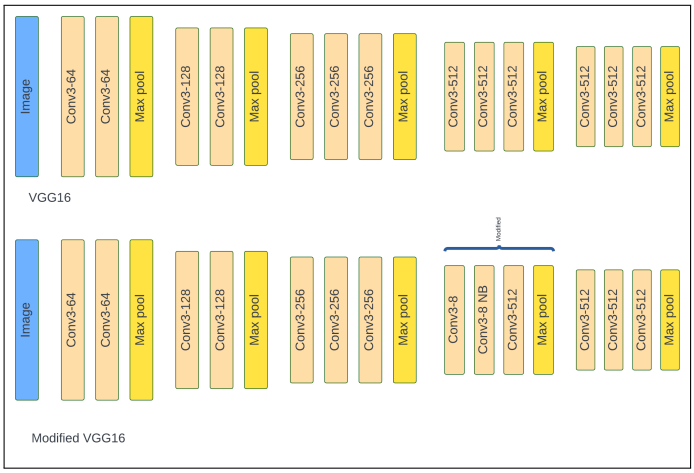

Fig. 1. Modified VGG 16

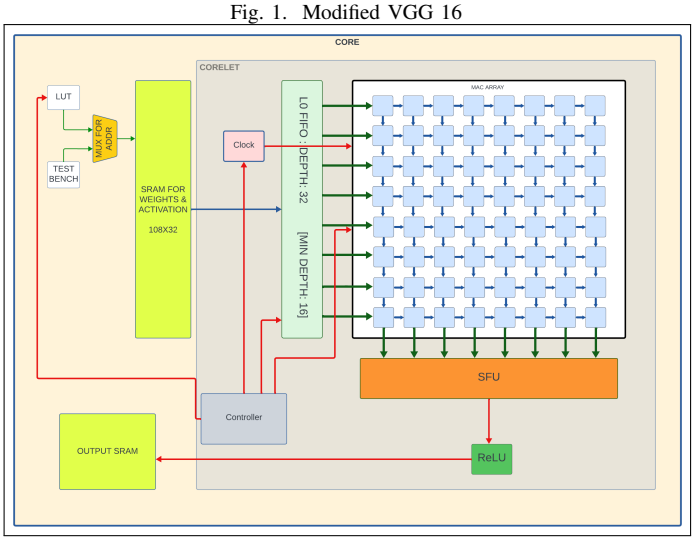

Fig. 2. Hardware design

D. *Mapping on FPGA*

Quartus Prime is used to simulate mapping onto FPGA. The results of quartus prime simulation are given in Table II:

TABLE II
QUARTUS PRIME SIMULATION TABLE

Design	Fmax	Dynamic power	TOPS/s	# Gates
Vanilla	$119.4Mhz$	$26.87mW$	0.059	15324
Alpha - Single channel	$103.35Mhz$	$14.47mW$	0.063	10404
Alpha - Multi channel (2)	$96.98Mhz$	$18.74mW$	0.056	12157

E. *Multi-channel Implementation in each PE*

The input channel for the multichannel implementation was 16×8, i.e., we have taken 2 channels to implement the multichannel case. The MAC tile was modified to hold 2 weights, and a mechanism for multiplying two different weights and activations was implemented. So, for every cycle, the MAC tile would take in 2 different weights and activations and output the psum.

F. *Alpha*

Three different alphas are implemented to optimize the mapping of the module to a 2D systolic array:

1. *Alpha 1 - Filtered convolution*
   In Fig. 3, the convolution of 6×6 input feature map with 3×3 kernel has 324 ($9K_{i,j} \times 36N_{i,j}$) operations per channel to be performed. Let's consider $K_{0,0}$. It gets multiplied with only the values highlighted in the $N_{i,j}$ matrix with the same color. This implies that we only need the appropriate $16N_{i,j}$ values for each index on the kernel to be multiplied. To implement the $N_{i,j}$ filtering, we employ a LUT to index to the first $N_{i,j}$ and subsequently use the state counter from our state machine to select the rest of $N_{i,j}$ for the appropriate $K_{i,j}$. From the above method, we have the following benefits:
   - We now have only 144 ($9K_{i,j} \times 16N_{i,j}$) operations to compute per channel instead of 324 computations earlier. This saves us 180 computations per channel.
   - Only 16 appropriate $N_{i,j}$ values are loaded in the L0, instead of 36 compared to vanilla, as shown in 4
   - In the vanilla version, we need 468 cycles , $9(K_{i,j}) \times (36N_{i,j} + 8(rows) + 8(columns))$. Whereas, in the optimized version, we need only 288 cycles $9K_{i,j} \times (16N_{i,j} + 8 + 8)$ in the systolic array, saving us 180 cycles.
2. *Alpha 2 - Pipelined load of weights and activations*
   In Fig. 6, the state machine illustrates the sequential flow of data during neural network operations: starting idle, transferring the kernel weights and input map activations from SRAM to local storage, processing them through a MAC array for multiplication and accumulation, applying a ReLU activation function, and finally storing the partial sums back to SRAM. The improvements to the vanilla version are listed below:
   - The Read-Writes to the L0 memory have been pipelined, such that reading from the L0 (and into the MAC array) begins a cycle after L0 writes (from SRAM) commence. We optimize the FIFO depth requirement, bringing it down to 9 elements (since we do not have to store all the values in the L0).
     Improvement: Significant decrease in loading times and FIFO depth.
   - Kernel weights are loaded in a parallel manner instead of the staggered manner used for activation values. Since the kernel values do not have to flow from North to South, the rows operate independently and, therefore, can support this parallel loading.
     Improvement: Reduced Kernel loading time by 8 clock cycles.
   - Kernel & Input Map loading has been pipelined. We wait for 8 clock cycles after loading the last kernel value into the MAC Array. The last PE of Row 0 receives the first activation value 1 clock cycle after

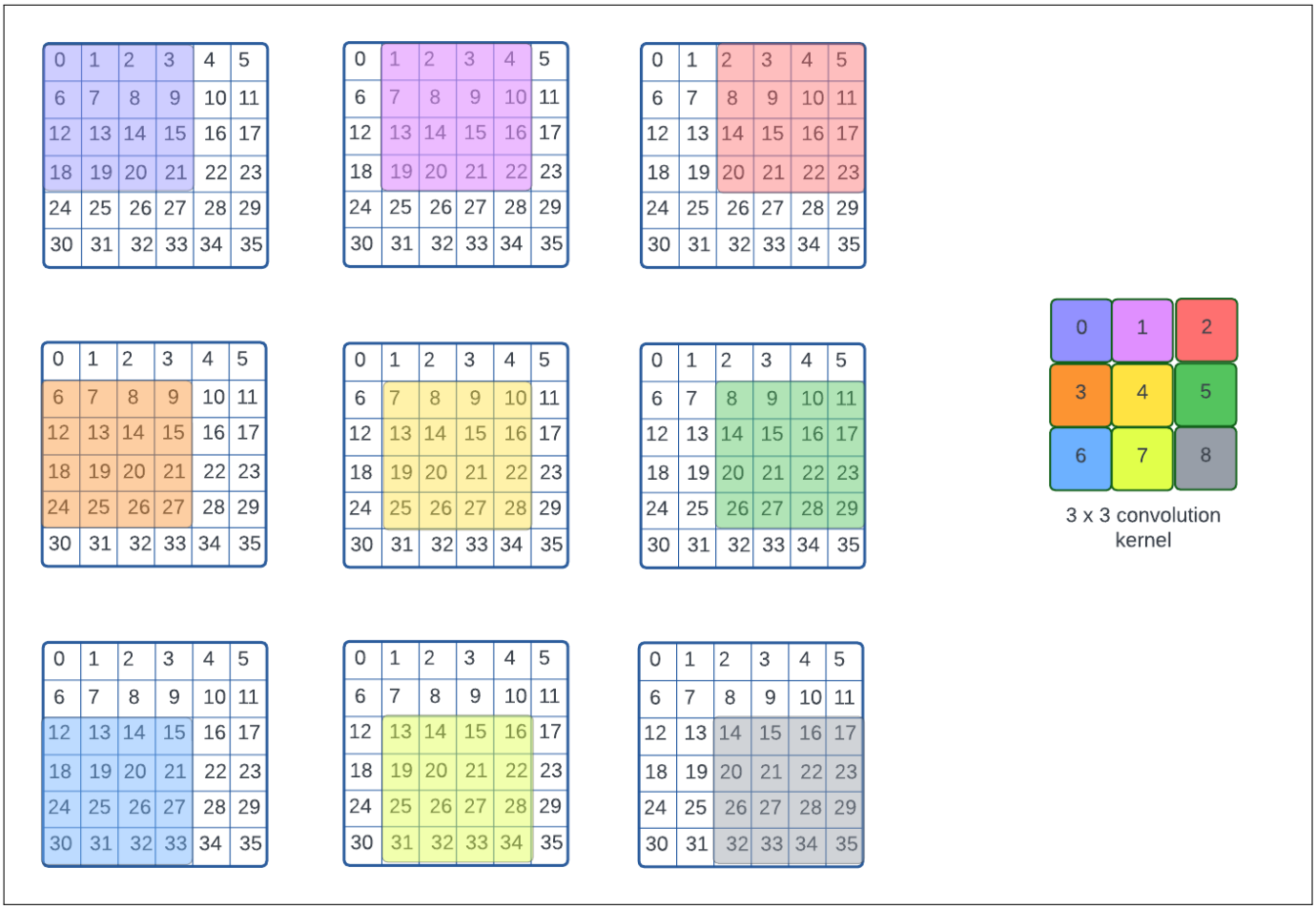

Fig. 3. Alpha 1: Filtered convolution

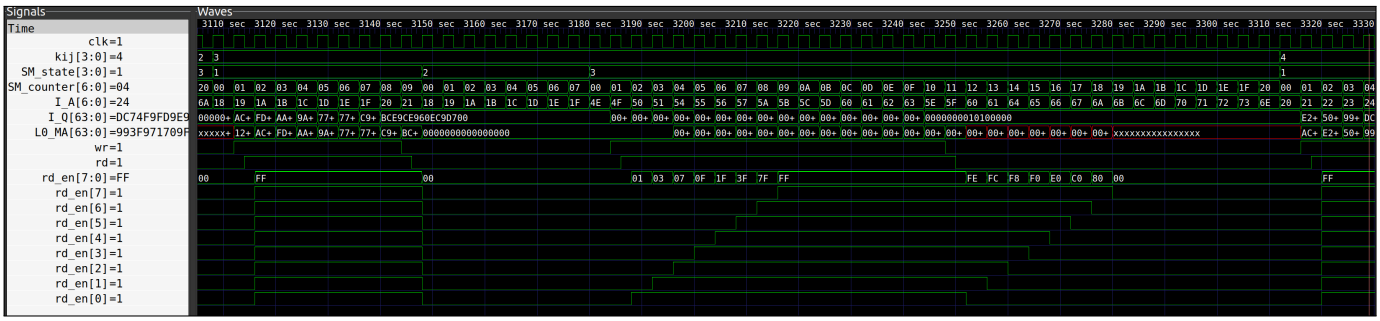

Fig. 4. Simulation: Controller Optimization

the designated weight value, minimizing the idle time between these two different loads.

Improvement: 8 clock cycles saved per input load.

The working of the implemented alpha can be observed in 4. We can observe that the weights are loaded parallelly, and the activations are cascaded as in the case of the vanilla implementation.

3. *Alpha 3 - In place SFU accumulation*
In the vanilla version, the OFIFO is used to temporarily hold the psum values from the MAC array and store the values to the SRAM for $\forall K_{i,j}$. The values are then retrieved from the SRAM and then accumulated in the SFU. However, this process has a high cost to pay in terms of area and latency, especially the reads and writes to the SRAM. Hence, we implemented SFU accumulation, which takes in values from the MAC array and stores them directly in the SFU register after summing them with the previous value. This process is repeated for every $K_{i,j}$. The secondary motivation for us to incorporate this method is due to the reduced number of psums

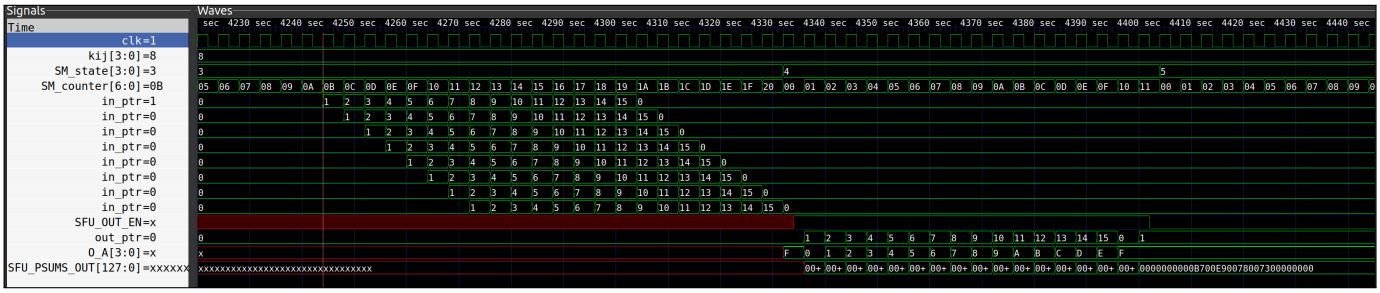

Fig. 5. Simulation: In place SFU accumulation

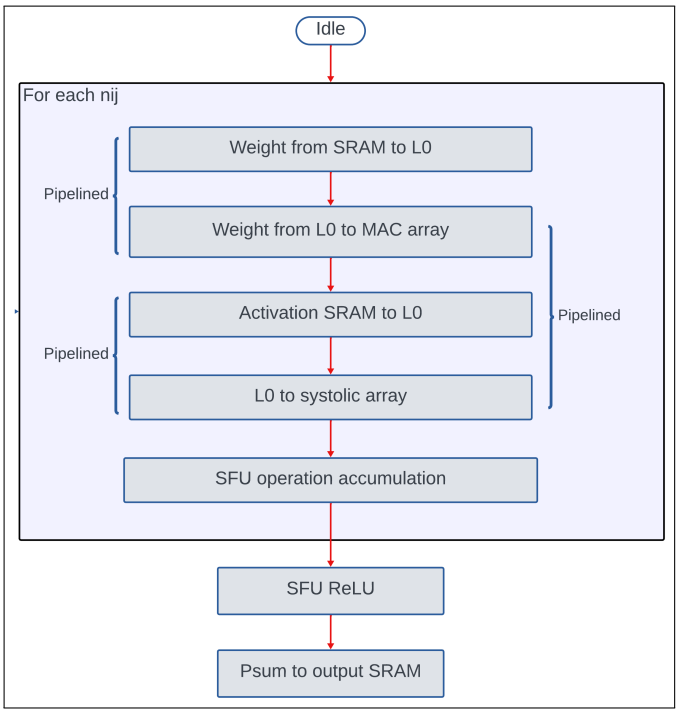

Fig. 6. Alpha 2: Pipelined loading of data

due to filtered convolutions. Even though registers are expensive, the area overhead is overcompensated by the latency overhead with the SRAM. We can observe in Fig. 5 that the input from the MAC array to the SFU is cascaded. The output is triggered to be flushed to the output SRAM when the signal SFU_OUT_EN goes high. When compared to the vanilla version, we save on writing and reading 128 (size of the one output channel) $36N_{i,j \times 9K_{i,j}}$, i.e., 324 times.

IV. Results and Summary

Our project attains high hardware efficiency and accuracy through meticulous testing and innovative strategies. We have successfully optimized FPGA mapping and implemented multi-channel processing, elevating throughput while maintaining precise functionality.

The advantages of the respective alphas are summarized below:

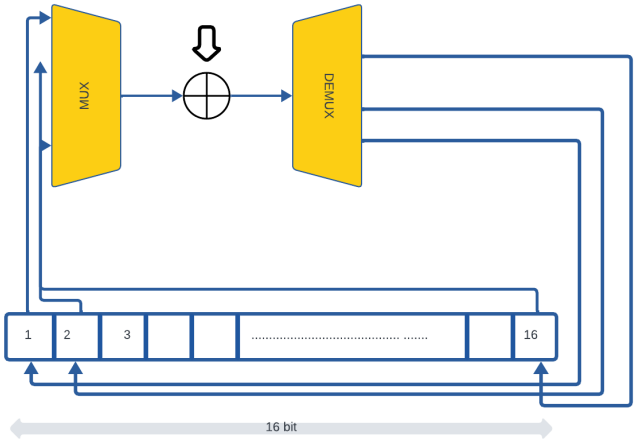

Fig. 7. Alpha 3: In place SFU accumulation

- Alpha 1: The results are summarized in Table III

TABLE III
SUMMARY OF ALPHA 1

Process	Baseline 36(nij)x9(kij)	Optimized 16(nij)x9(kij)
Load activation to L0	324 cycles	144 cycles
psum computation	468 cycles	288 cycles
Load weights to L0	72 cycles	72 cycles
Load weights to Systolic	216 cycles	216 cycles

- Alpha 2:
  - 128 bits of output can be loaded into SRAM in one clock cycle.
  - Cycles saved in computation 360 - 16 (additional cycles saved while loading weights to 2D Systolic array) cycles = 344 cycles.
  - Percentage of cycles saved = 31.8 %.
- Alpha 3:
  - In-place accumulation saves 324 transactions of reads and writes to the SRAM memory.

V. REFERENCES

[1] H. . -K. Kwan and T. S. Okullo-Oballa, "2-D systolic arrays for realization of 2-D convolution," in IEEE Transactions on Circuits and Systems, vol. 37, no. 2, pp. 267-233, Feb. 1990, doi: 10.1109/31.45721.

[2] Roman Holowinsky. *A Sieve Method for Shifted Convolution Sums*, 2008. https://arxiv.org/abs/0809.1669

[3] Eunhyeok Park and Sungjoo Yoo and Peter Vajda, ”Value-aware Quantization for Training and Inference of Neural Networks”