ECE 284 Final Project

Tensor Group

Part 1

This initial part of the project focused on training the VGG16 model using quantization-aware training (QAT) with 4-bit input activations and 4-bit weights. We were able to achieve an accuracy of 91% as well as a quantization error of 3.9279e-7. The 27th layer of the VGG16 model was specifically changed to have an 8x8 convolutional layer and after the 27th layer batch normalization was removed ensuring the layer went through the “conv->relu” sequence.

The notebook generates multiple .txt files to store binary representations of activations, weights, partial sums (psum), memory addresses, and output data for use in systolic array simulations.The activation.txt file contains binary representations of input activations. The activations are reshaped into a row-time-step format and quantized to 4 bits. Each quantized activation is written in row-wise and time-step-wise order, ensuring compatibility with the 8x8 systolic array. Similarly, the weight.txt file stores the quantized weights of the convolutional layers in binary format. These weights are reshaped and written for each column-row combination in the kernel, with each value represented as a 4-bit binary number. The psum.txt file records partial sums (psum) computed during matrix multiplications within the systolic array. Each psum value is quantized to 16 bits and written for every kernel element across time steps, representing the intermediate results of the computations. The address.txt file stores memory addresses calculated for each computational step. These addresses, written in 11-bit binary format, ensure proper memory alignment and retrieval during operations. Finally, the output.txt file contains the final output activations from the systolic array. The outputs are reshaped and quantized to 16 bits, then written sequentially in binary format.

In conclusion, the results from our training demonstrated the effectiveness of QAT in maintaining high accuracy despite layer modifications. The model achieved 91% accuracy, exceeding the 90% requirement, while the quantization error was measured at 3.9279e-7, staying under the threshold of 10-3.

Part 2 & 3

Part 2 of the project creates building blocks for the base implementation of our systolic array AI accelerated using a weight stationary structure. At the heart of the project is an 8x8 MAC array capable of handling 4-bit quantized inputs and 16-bit partial sums. This is supported by a comprehensive memory architecture including dual SRAMs, an L0 buffer for weight staging, and output FIFOs for results collection.

The design is modularized into two key components, the core module and the corelet module. The core module manages memory interfaces and top-level control. The 34 bit control signal in the core is mapped to various control signals. Bits [34:33] handle ReLU and accumulation control, bits [32:20] manage PMEM operations, bits [19:7] control XMEM, and bits [6:0] handle FIFO and execution control.

The data input is managed through two memory interfaces, XMEM for activations and wights, and PMEM for partial sums. The XMEM interface receives 32 bit data words, accommodating eight 4-bit values to match the row width of our systolic array. The interface also includes standard memory control signals such as chip enable and write enable alongside an 11-bit address bus. This provides us access to 2048 memory locations. On the other hand, the PMEM interface is designed as a 128-bit data path to handle the partial sums from all eight columns simultaneously, with each partial sum being 16 bits.

The corelet module contains the processing elements and data movement components. Additional input interfaces are implemented in the corelet for better control. These include dedicated signals for L0 buffer operations (read/write enables, data input), execution control (load and execute signals), and special function unit control (acc and ReLU enables). The L0 bugger serves a staging area for the weights and activations and is used to manage the loading process. Weights and activations are first loaded into memory then into the L0 buffer before being loaded into the MAC array.

The SFP and the output FIFO are the main components in our accelerator’s output processing pipeline. The SFP is instantiated for each column of the systolic array. Each SFP accumulates partial sums across multiple kernel operations retrieved from PMEM and handles ReLU activations. The output FIFO serves a buffer between our MAC array and the output memory

subsystem. It captures the 16-bit partial sums from each and manages their transfer to the PMEM. The FIFO implements a ready/valid handshaking protocol to ensure correct data transfer, using status signals to provide necessary flow control. This buffering mechanism allows the MAC array to continue processing while results are being written to memory.

Part 3 of our project is the implementation of our test bench. Our test bench provides a comprehensive verification of our systolic array accelerator's functionality. We implemented the test bench using a modular approach as well, with distinct phases for initialization, data loading, execution, and verification.

The data loading phase begins by reading activation and weight values from external text files before writing the data into appropriate memory locations. This is done using control signals such as CEN and WEN and addressing using A_xmem. For kernel weights, we specifically write the data to memory at a designated offset to avoid overwriting activation data. Weights are then transferred through the L0 buffer before loading into the PE array.

The execution then begins after all weights have been loaded into corresponding tiles in our PE array. Activations are then passed through the L0 buffer before loading into the PE array column by column. After the execution phase has been completed, results are retrieved from the PMEM and sent to SFP for accumulation calculations. After accumulations are completed, these results are returned to PMEM storage and results are compared to output.txt for verification.

Part 4

	Baseline (VGG16)	Alpha1 (Resnet20)	Alpha2 (Pruning)	Alpha3 (Dual Core)	Alpha4 (Zero Gating)
Frequency (MHz)	131.21	131.21	131.21	131.21	131.21
Dynamic Power (mW)	32.42	31.50	30.03	64.84	**25.01**
GOPs/s	1.003	1.003	1.003	**2.006**	1.003
GOPs/W	3.054	3.182	3.346	3.00	**4.012**
Logic Elements	**16,595**	16,700	**16,595**	33,200	17,200

The baseline configuration, utilizing a VGG16 model, serves as a reference point for frequency, power consumption, performance, and hardware complexity. At 131.21 MHz, the baseline system achieves 1.003 GOPs/s with a dynamic power draw of 32.42 mW. The resulting power

efficiency stands at approximately 3.054 GOPs/W, and the implementation requires 16,595 logic elements. This benchmark allows us to gauge the impact of subsequent modifications.
For Alpha1 (ResNet20), the main changes involved adjusting the network architecture to fit the 2D systolic array constraints and applying quantization-aware training. These adjustments slightly reduced dynamic power to 31.50 mW by possibly streamlining data flow and handling smaller convolutional dimensions. Although the maximum achievable GOPs/s remains similar to the baseline, the modest decrease in power leads to a slightly improved GOPs/W of 3.182. The logic elements increase marginally to 16,700, reflecting the minor architectural tweaks needed for ResNet20's structure.

Alpha2 (Pruning) demonstrates the effects of structured and unstructured weight removal from the VGG16 model. By pruning 50% of the weights, the design reduces the number of effective operations. While the frequency and peak GOPs/s remain unchanged at the hardware level (since the clock and compute capabilities are constant), the reduced computational intensity slightly lowers dynamic power to 30.03 mW. This reduction in power consumption leads to an improved efficiency of 3.346 GOPs/W. The logic element count remains essentially the same as the baseline, indicating that pruning does not inherently alter the hardware requirements, but it increases efficiency by eliminating unnecessary operations.

Alpha3 (Dual Core) focuses on scaling performance through parallelization rather than changing the model or pruning. By instantiating two identical cores, the design effectively doubles the computational throughput to 2.006 GOPs/s at the same clock frequency. However, doubling the number of cores nearly doubles the dynamic power consumption to 64.84 mW and significantly increases the logic elements required to 33,200. While the parallelization enhances raw throughput, the efficiency in terms of GOPs/W stabilizes around 3.00, slightly lower than the baseline when normalized, due to overheads associated with managing two cores simultaneously.

Finally, Alpha4 (Zero Gating) targets efficiency by eliminating operations involving zero-valued inputs or weights. This approach reduces unnecessary switching activity and dynamic power consumption to 25.01 mW. Although the peak GOPs/s does not increase—since the hardware's fundamental computation rate is unchanged—the ability to skip redundant operations enables the system to achieve a substantially higher GOPs/W metric of about 4.012. This improvement represents a significant leap in energy efficiency and stems directly from lowering the number of performed computations. The logic elements increase to 17,200, reflecting the added zero-detection and gating logic, but this hardware investment yields considerable power and efficiency benefits.

In summary, each alpha variant offers a distinct trade-off between performance, power efficiency, and hardware complexity. Alpha1 shows that modifying the model architecture and using QAT can slightly improve efficiency. Alpha2's pruning approach demonstrates better overall power

efficiency without altering throughput capacity. Alpha3's dual-core approach clearly boosts throughput but at the cost of higher power and logic utilization. Lastly, Alpha4's zero gating strategy provides a substantial improvement in efficiency, enabling higher GOPs/W by selectively bypassing unnecessary computations. Together, these results illustrate how architectural changes, model adaptations, and pruning techniques interact to shape the power, performance, and efficiency profiles of a 2D systolic array-based computing system.

Part 5

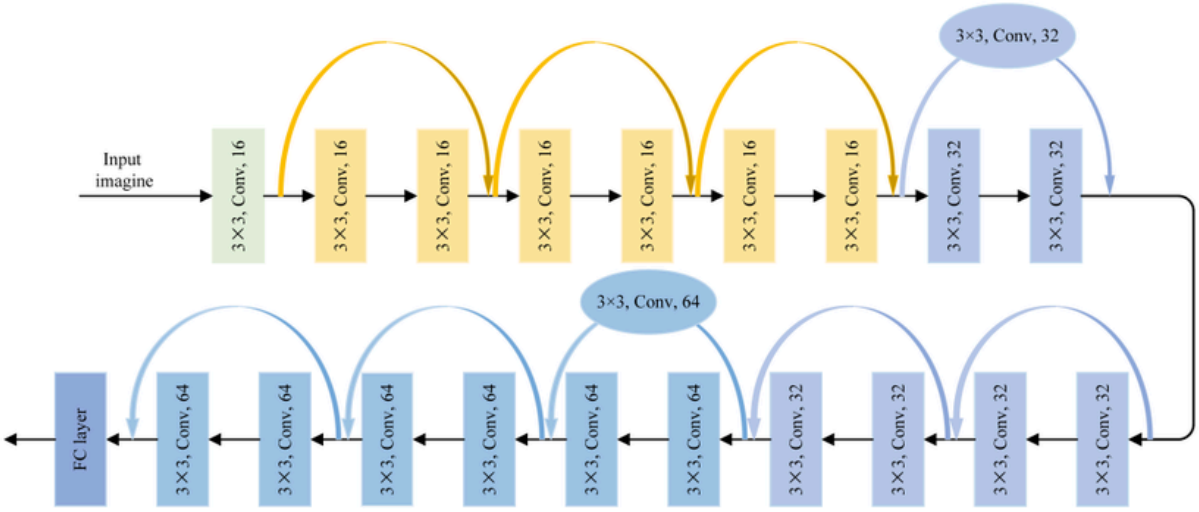
In part 5 we modify our design in part 2 & 3 in order to create a configurable dual-mode systolic array design that supports both Weight-Stationary and Output-Stationary dataflow. In part 2 & 3 we implemented the design for WS so we modified the design to include the architecture for OS. Compared to WS, OS receives weights from in_n and activations from in_w, and psums are accumulated inside each tile. To implement this modification we introduced several key changes. First, an IFIFO was added to handle weight distribution across the array in OS mode since weights and activations are passed to the array simultaneously in OS mode. The key memory interfaces are the same with the addition of an additional memory allocated for weights that is used for IFIFO loading. The data routing infrastructure was also enhanced to support both OS while also maintaining compatibility with WS mode. MAC tiles were also enhanced with operation and result validation logic to track when tile computations are complete. Each tile now maintains the partial sum locally until all required MAC operations are finished. The output collection mechanism was modified to handle full row-wise data movement, with a state machine controlling the extraction of completed data from the array. The control logic was also expanded to handle both modes through a united instruction set, with the mode bit determining how the control signals are interpreted (1: OS, 0: WS). The corelet instantiation is modified to support OS mode on top of WS, including the tile_psum_array output that provides direct access to the MAC array results in OS mode.

Part 6

Alpha1 (ResNet20):

For the Alpha 1 of our project, we chose to map the ResNet20 model onto our 2D systolic array as a proof of concept. This decision allowed us to explore the adaptability of our quantization-aware training (QAT) workflow and verify its effectiveness on a different neural network architecture beyond VGG16. To align the ResNet20 model with the constraints of our systolic array, we made targeted modifications to its structure. Specifically, we edited the first basic block of the ResNet20 model, focusing on the second convolutional layer within this block.

This layer was modified to be an 8x8 convolutional layer to match the array dimensions, ensuring optimal compatibility with our hardware setup.



In addition to modifying the second convolutional layer, we adjusted the surrounding layers to maintain the overall flow of the network. The layer preceding the modified convolution was adjusted to have dimensions of 3x8, while the subsequent layer was updated to have dimensions of 8x16. Furthermore, to simplify computations and reduce dependencies on floating-point operations, we removed the batch normalization layers from this segment of the model. These changes streamlined the data flow through the systolic array, making the design both practical and efficient for hardware deployment.

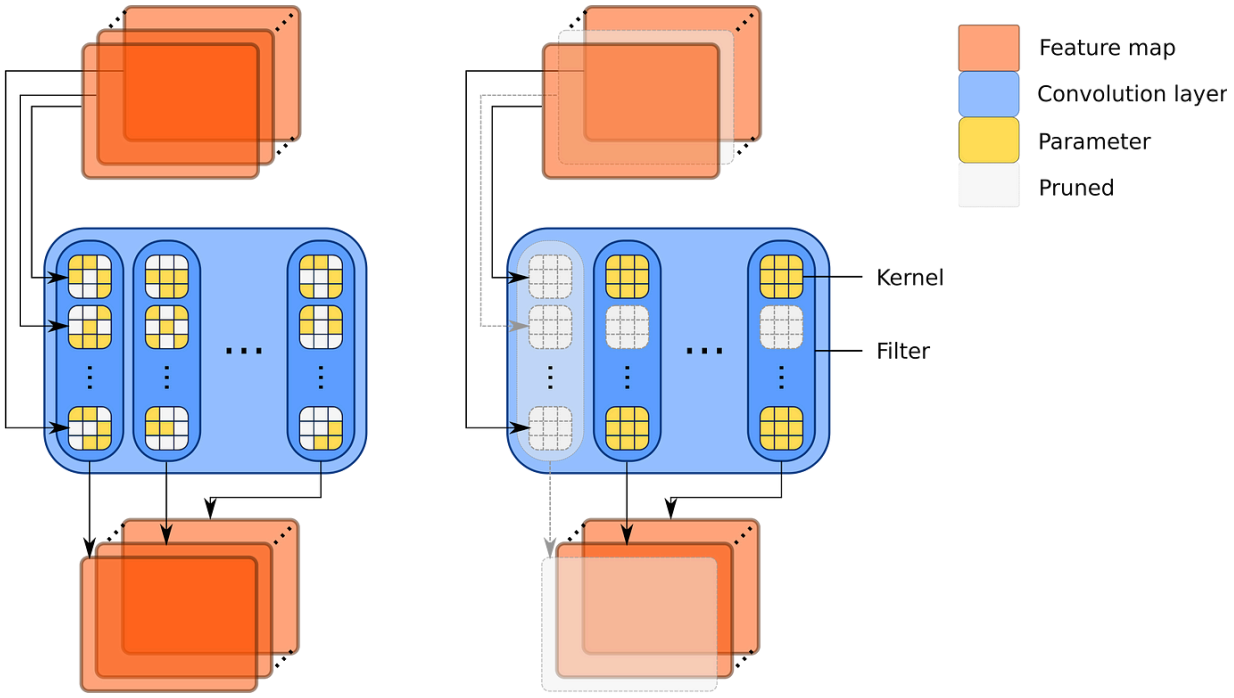
Following these modifications, we trained the ResNet20 model with QAT to evaluate its performance under quantized conditions. The model achieved an accuracy of 80 %, demonstrating robust performance despite the structural changes and quantization constraints. The quantization error for this setup was measured at 5.0165, which is an acceptable range given the trade-offs between precision and hardware efficiency. By successfully implementing and training the ResNet20 model, this experiment serves as a valuable proof of concept for the flexibility and scalability of our approach.

Alpha2 (Pruning):

The pruning of the VGG16 model was conducted using both structured and unstructured pruning on 50% of our weights, and the results reveal differences in their impact on the model's accuracy. Structured pruning reduced the model's initial accuracy to 10% after pruning, but fine-tuning brought it back to 91%, demonstrating the effectiveness of this approach in recovering performance. On the other hand, unstructured pruning showed better retention of initial accuracy, dropping to 37.81% after pruning. However, the fine-tuning process brought the accuracy to 88%, slightly below that of structured pruning.

Structured pruning involves removing entire structures, such as channels or filters, which significantly impacts the model's architecture. This explains the severe drop in initial accuracy. However, this method simplifies the model, making it more hardware-friendly, and the subsequent fine-tuning effectively restores performance. Conversely, unstructured pruning, which removes individual weights without considering the overall structure, retained a higher initial accuracy due to the more granular nature of the pruning. Despite this, its recovery during fine-tuning was less robust compared to structured pruning.

These results highlight a trade-off between the initial impact of pruning and the final accuracy after fine-tuning. While structured pruning caused a more significant drop in initial accuracy, it ultimately yielded higher fine-tuned performance, making it a preferable choice for scenarios where hardware efficiency and long-term performance are critical. Unstructured pruning, with its less invasive initial impact, may be suitable for applications where maintaining some immediate functionality is more important than achieving maximum fine-tuned performance.



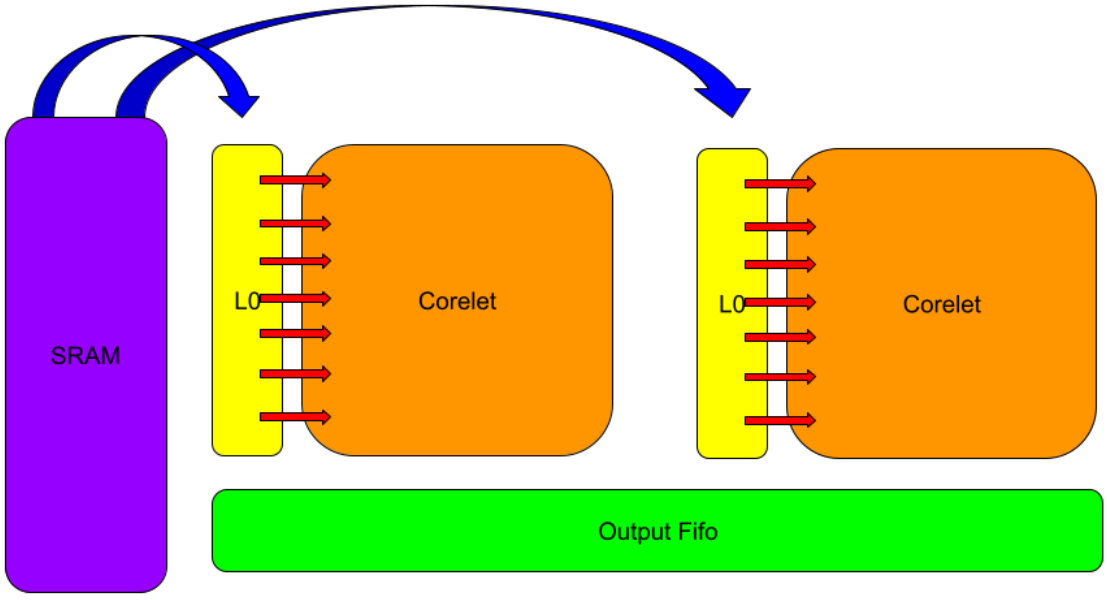
Alpha3 (Dual Core):

This alpha builds on the original single-core, 2D systolic array architecture by adding a second core. The new top-level module instantiates two identical cores side by side. Each core receives the same input instructions and data signals, and the outputs from both cores are combined at the end. This straightforward replication effectively doubles the system’s computational capacity while preserving the structure and parameters of the original design.

The primary advantage of using two cores in parallel is the improvement in throughput. With both cores operating simultaneously on their own sets of data, the design can perform roughly twice as many operations per cycle compared to the single-core version. This increase is particularly useful for computations like matrix multiplication in machine learning or signal processing tasks, where parallel operations can significantly reduce total run time.

The modular structure of this dual-core setup also makes it easier to scale the design further. Adding more cores, if needed, follows the same pattern: replicate the core module, feed it the same or separate instructions, and then combine the outputs. Although the current design uses a single instruction stream for both cores, future improvements could include giving each core its own set of instructions. This would allow more specialized workloads or better load balancing. Another potential area of enhancement is the memory subsystem. For example, assigning separate memory interfaces to each core or using more advanced caching techniques could improve data delivery and reduce delays.

In summary, the dual-core integration doubles the available computational resources and can lead to higher overall performance. It maintains the original design's modular nature and leaves room for future refinements, such as separate instruction streams, better scheduling, and more efficient memory management.



Alpha4 (Zero Gating):

The zero gating feature in the 2D systolic array architecture significantly enhances computational efficiency by eliminating unnecessary operations when inputs are zero. This optimization is particularly beneficial for handling sparse data, which is common in applications like neural networks. By incorporating zero gating, the system can detect zero values in both activations and weights, allowing it to skip the multiplication and accumulation processes when either input is zero. This approach effectively reduces the number of operations, leading to lower power consumption and improved performance.

The implementation involves updating the mac module to include logic for zero detection. The module checks if either the activation or weight is zero before proceeding with computations. If a zero is detected, the multiplication result is set to zero, and the accumulation step is bypassed. This logic is implemented using registers to store intermediate results, ensuring synchronization with the clock signal. The mac_tile module integrates this updated mac module, ensuring consistent application of zero gating across the systolic array.

The impact of zero gating is substantial. By reducing unnecessary computations, the array operates more efficiently, with lower power consumption and reduced heat generation. This is crucial for energy-constrained environments, such as battery-powered devices. Additionally, the reduction in computational load allows the array to achieve higher throughput and lower latency, as processing elements can focus on non-zero data more effectively. This feature also enhances the scalability of the systolic array, enabling larger implementations without a proportional increase in power usage.