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

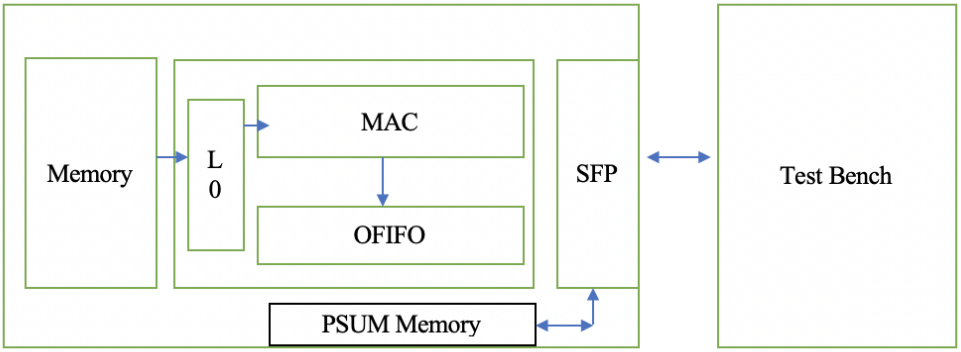
Monil Shah, Purvi Agrawal
Supervisor: Prof. Mingu Kang, Assistant Professor ECE, UCSD

Abstract: In this project, we have trained the VGGNet model in pytorch with quantization aware training to achieve greater than 90% accuracy and then mapped the network on a 2D systolic array weight stationary architecture. The design is later mapped on the Cyclone IV GX FPGA board to calculate operating frequency and power. The training part is implemented on Pytorch and mapping of hardware is done in Verilog. The Verilog designing for MAC and FIFO has been done. For verification, we have implemented the testbench, analyzed the waveform, and compared the result from software with hardware so that they are matching with one another. The power improvement achieved via clock gating is 15%. The frequency observed is 129 MHz.

Introduction: Machine Learning inference and training require special purpose cores to accelerate computations. The 2D systolic array architecture is widely used in AI applications because of the parallel architecture and specialized MAC units. Training and back propagation of loss cause weights and activations to be floating point numbers. Floating point operations have higher power consumption and are expensive in terms of hardware, therefore to reduce the power value we want the computation of weight and activation in integer values. For this purpose we use quantization. Quantization is generally used to bring a wide range of values into the user defined range. Basically, there are two types of quantization : post training quantization and quantization-aware training. For post training quantization the accuracy is lower, therefore to achieve higher frequency we have implemented quantization aware training. To achieve a higher computation rate, first, we train the VGG model with quantization aware training. Following which we implemented the 2D systolic array weight stationary architecture. The weights and activations are fed on the 2D systolic array and each processing element behaves as a multiplier and accumulator unit. All the multiplied values are added accordingly and partial sums are generated, which are fed to Output FIFO. We have done the verification to match software and hardware generated results. And then mapped the architecture on the FPGA board to measure the power, frequency, and other parameters.

Train VGG16 with quantization-aware training: In this module, we have trained the VGGNet model for 4 bit input activation and 4 bit weight quantization. For the purpose of hardware mapping without tiling we squeezed the 27th layer of the model to 8*8 (input channel* output channel). This modification was done in the VGGNet quant layer. Generally the layer convention used in VGGNet is Convolution, Batch Normalisation and then ReLU. For the purpose of simplicity in mapping to hardware we have removed the batch norm layer after the squeezed convolution layer. In this project, we have used pytorch to train the network. Firstly we trained this modified version of the model to achieve greater than 90% accuracy. Then we pre-hooked the input of the squeezed convolution layer. We applied ReLU to the first pre-hooked layer and then subtracted it with the second pre-hooked layer, to get layer parameters close to zero accuracy degradation. Following which we computed the quantized weight and activation values to be fed to the hardware. We wrote kernel and activation values in memory.

Complete RTL core design: In this module, we have used a bottom up approach. First we created a MAC unit and then from MAC we created a MAC_tile which includes the instruction flow mechanism. Following which we designed the mac_array from mac_tile which is basically the combination of 8*8 PEs, this is our 2D systolic array architecture. 'corelet.v' is the crux of the systolic array which instances MAC Array, 2 FIFOs to load/store data to/from Mac Array and a SFP row. L0 FIFO is designed for synchronising the upload of the weight and activation from memory. OFIFO is designed to store the partial sums and has valid signals to indicate when it's ready for reading the outputs. Special function processor (SFP) unit is responsible for accumulating the partial sums and then applying ReLU to the output. For hardware implementation we have mapped the 'corelet.v' file to FPGA. The top module 'core.v' instances corelet and memories. The completion of design was done with successful compilation of all files.

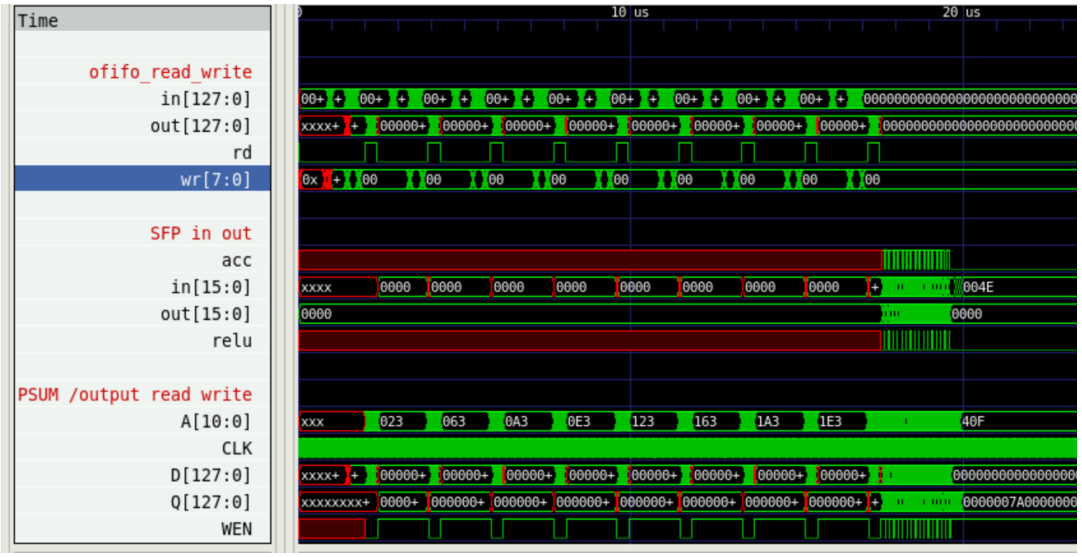


Test bench generation: This part of the project acts as a controller for all the Verilog coding part. In this module of the project, we loaded the memory with activation and weight which we got in different files after implementing the VGGnet network in pytorch. We loaded the kernel weights to Mac Array via L0 and then provided instruction for its execution in the processing element to compute partial sum. For this we first wrote the activation and weight files generated from software to memory. As the design which we are implementing is weight stationary we first load the kernel weights to L0. Then send this to the mac_array architecture. After that we loaded the activations to memory and then to L0. Following which the code computes the partial sum and uploads it to OFIFO. From OFIFO the values are sent to the memory. We have created some partitions to visualise different psum's. Then we reset the mac_array for loading the second kernel weights and compute psum store it in another location. Repeat this part as many times as the kernel size (in our case 9 times). After which all accumulation and ReLU is done in the SFP unit. Then store the final outputs to SRAM and write the address locations to a text file for verification. Then write all these values back to memory.

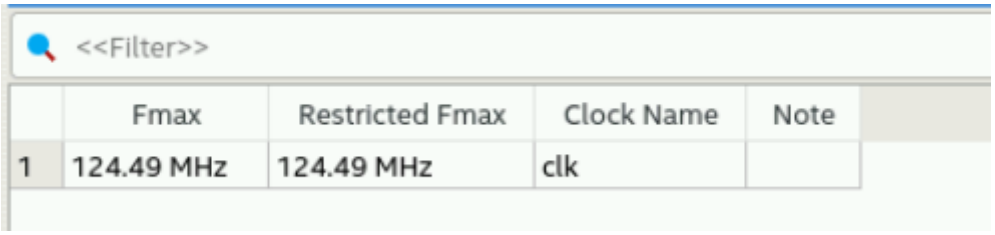
Verification: For verification purposes, we compared the output file from software with the hardware generated results which are stored in memory. We need to verify the order of the values which we are expecting and which are generated in the desired way.

Mapping on FPGA (Cyclone IV GX EP4CGX150DF31I7AD): For hardware realisation we have implemented the above design in FPGA and measured power, performance and frequency values and tried to optimise it. We mapped the corelet.v file to FPGA. Analysed the complete synthesis and placement/ route. We measured frequency at slow corners (worst case scenario) at 100C and power for 20% input activity factor (20% of the frequency).

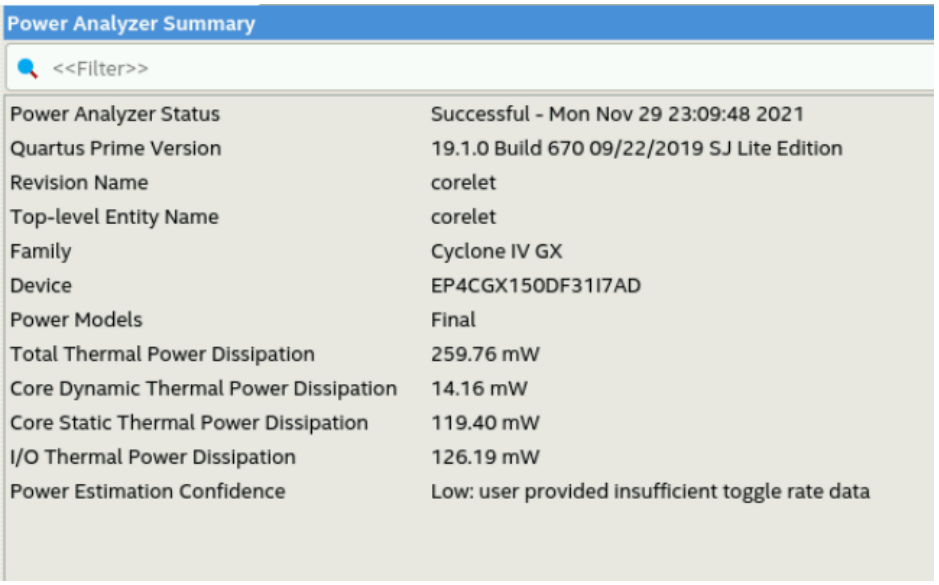
Simulation Results:



Output Waveform with Input Clock Gating cell



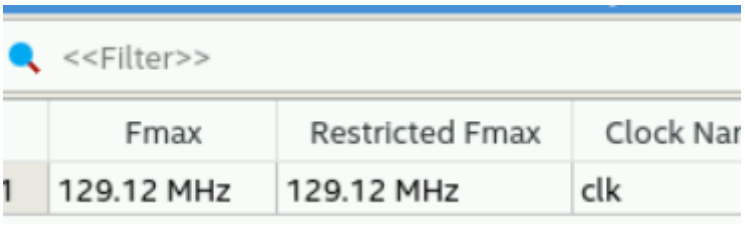
Frequency with Input Clock Gating cell



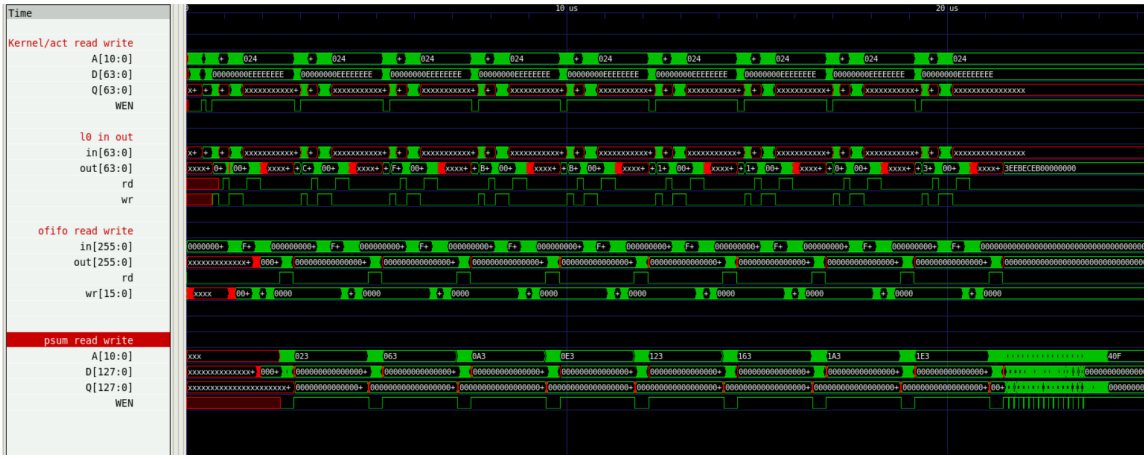
Power Consumption with Input Clock Gating cell

Total Thermal Power Dissipation	265.06 mW
Core Dynamic Thermal Power Dissipation	33.47 mW
Core Static Thermal Power Dissipation	119.42 mW
I/O Thermal Power Dissipation	112.16 mW

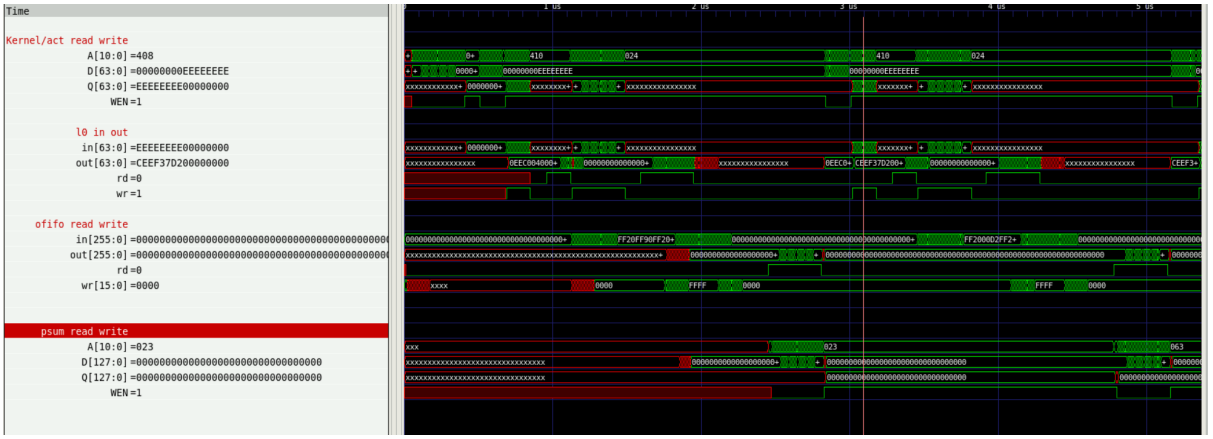
Power Consumption without Input Clock Gating cell



Frequency without Input Clock Gating cell



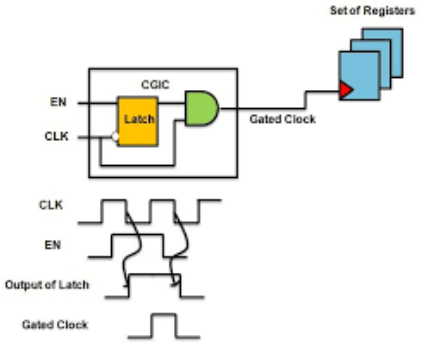
Multicore Output Simulation



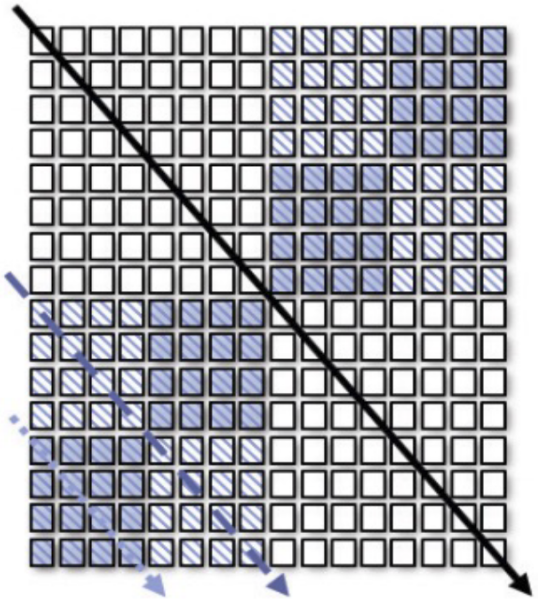
Multicore Output Simulation

Alpha Part: In this part of the project we tried different approaches which includes multicore (for tiled processing) architecture and clock gating.

Clock Gating: The clock gating is a powerful technique to reduce the dynamic power consumption. The whole circuit does not require the clock signal at all the time they may retain their previous output state. For this we have used a predefined input clock gating cell. It has a negative triggered latch for synchronising the clock and enabling signal which helps in removing glitches. First when we are loading the weights and activation in memory, we have clock gated for L0, OFIFO and mac_array. Whenever we are loading weights and activation in L0 we start it's clock and gate another clock. Then when we are loading in mac_array start it's clock. When psum is computing, start the ofifo clock. By this we reduce the switching activity factor henceforth reducing the switching power consumption also the dynamic power consumption.



Multicore tiled Processing: The multicore approach can be used when we have greater array size than the input, output channel number. In our case we have taken the array size as 16 and convolution layer size as 8*8 (input channel * output channel). Firstly, we trained this VGGNet to achieve greater than 90% accuracy. After that we calculated the quantized weight integer, activation integer and output integer values. Then we need to map these values to different tiles accordingly. In this case we have total 4 tiles of 8*8. Since we need to map only two of those therefore we assigned the tile 1 and tile 2 values to be zero. We created different text files for weight, activation and outputs. Following which we implemented the same architecture in Verilog. Test bench generation was done and weights and activation were fed to memory. For verification of the design, outputs were stored in a text file from memory and comparison was done with the software generated outputs.



Multicore 2D systolic Array Architecture

Conclusion: In this project, we trained the squeezed VGG neural network to attain greater than 90% accuracy. Implemented the design in verilog, generated test bench, verified the design and then mapped it to FPGA. After that we computed power and frequency of the design. For optimising the power consumption, we have implemented clock gating technique and reduced the power consumption by 15% power. We also implemented the multicore architecture in pytorch, implemented its design in Verilog and verified the same with software produced outputs.