Final Project Report – <u>Convolutional Neural Networks: An ASIC Approach</u>

Name: Soumyadeep Chatterjee
Unityid: schatte5
StudentID:. 200423906

Delay (ns to run provided provided example). 263445ns Clock period: 5.0ns # cycles”: (19014 + 7222) = 26,236	Logic Area: (um^2) 7893.5500 Memory: N/A	1/(delay.area) ($ns^{-1}.um^{-2}$) = 1/(263445 * 7893.5500) = $4.8088 * 10^{-10}$ $ns^{-1}.um^{-2}$
Delay (TA provided example. TA to complete)		1/(delay.area) (TA)

Abstract

The art of ASIC design specializing in custom-designed Machine Learning (ML) modules is becoming ever-important as we continue on the course toward independent automation. This report outlines the design and delineates the details of a Convolutional Neural Network (CNN) module programmed in Verilog, and serves as the final project of ECE 564 (ASIC and FPGA design with Verilog). The implemented module interfaces with the following: an input SRAM that contains input data and the size of the specific input set, a weights SRAM that containing the kernel as well), and an output SRAM. The implemented design leverages hardware parallelism to build up a 4*4 input matrix, as well as a 3*3 kernel matrix, performs parallel multiplication, and accumulates to complete convolution. This step is following by a ReLu activation function, and a stage for Max Pooling, to produce one output for storage – therefore, we are able to obtain an output with every read once the pipeline is full by striding the input matrix subset until the entire dataset has been processed. Finally, the control logic is implemented using a number of small Finite State Machines (FSMs), further leveraging hardware parallelism and reducing clock cycles.

Convolutional Neural Networks: An ASIC Approach

ECE 564 Final Project
Soumyadeep Chatterjee

1. Introduction

The central aim of this project is to design and verify the first 3 stages of a CNN-specialized Verilog module, namely the convolution layer, followed by a ReLu activation function that saturates outputs to between 0 and 127, and lastly, a MaxPooling function that reduces the matrix size by selecting the largest value in striding 2*2 windows.
The input values, obtained from an input SRAM, are a 16*16 matrix, while the kernel values are obtained from a kernel SRAM, and comprise a 3*3 matrix. Implementation was driven by the desire to find a balance between area and speed, with the overall goal of leveraging hardware-level parallelism to enhance the design. This report details the design process, provides a high-level view of the choices made during the design process, an explanation of the control path, and a UML overview of the finished design. Further, details on the verification methodology, as well as the results of performance metrics and area achieved by this design. Finally, we take an objective approach towards discussing further optimization strategies that can be exercised to further improve this design.

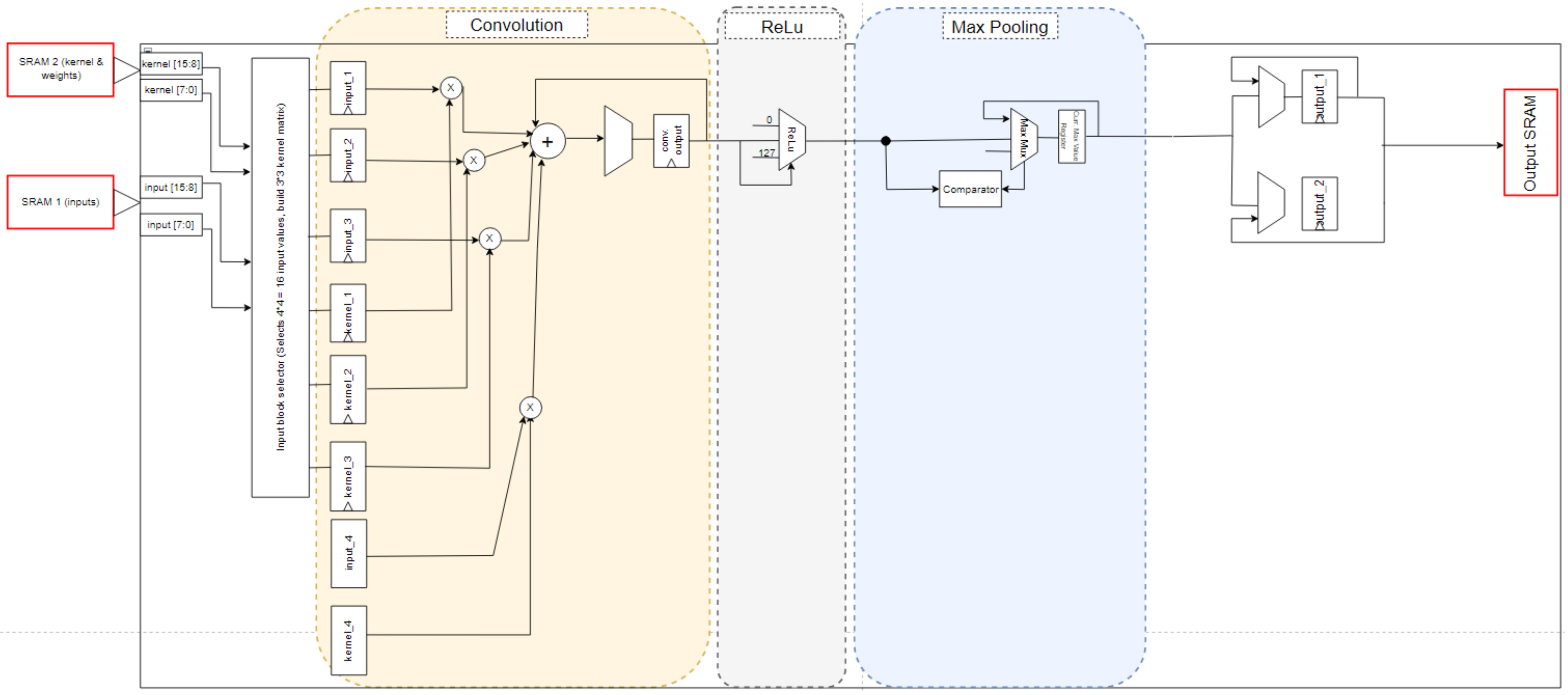
2. Micro-Architecture

The design builds a 4*4 subset matrix of the input values via a system of smart-selection that optimally selects input data, as well as the 3*3 kernel (in parallel), processes them, and then repeats the process until all input sets are fully processed. This optimization in input logic leverages ensures that all steps (convolution, ReLu and MaxPooling) can be completed with every input subset that is read in, producing one output to be written to the output SRAM.

Convolution is split into two states, and is optimized for area – there are 4 parallel multipliers, followed by 4 adders that complete convolution one set of outputs, and repeats 9 times, creating a MAC that enables the reuse of inter-stage memory modules. ReLu is combined with convolution, and is implemented as a simple mux that sets all negative convolution outputs to 0, while setting 127 as the upper value limit – this step is incorporated in the convolution dataflow, as is the MaxPooling function. The values are then written to the output SRAM via a system that shifts the values to accommodate new inputs into the same address, and increments output address every other time, thereby enabling the next set of inputs to be processed.

Control logic is split into 3 controllers for reading inputs, reading kernel values, and final FSM that handles computational operations, namely Convolution, ReLu, and MaxPooling. Additionally, writing to the output SRAM is also accomplished in this controller. A top-level controller controls the overall state of the machine, controlling movement to the different logical stages in the data path.

A UML diagram of the implemented design is shown below:



3. Interface Specification

This project interfaces with and handles data from the Input and Weight SRAMs, processes them, and writes to the Output SRAM. The following table delineates the control signals and logical memories used in the implementation of this design:

Declarations	**Purpose**
Address	
reg [11:0] input_req_offset;	base counter of input_sram_read_address
reg [11:0] Read_Offset;	counter to shift within a row to increment input_sram_read_address
reg [11:0] input_req_row_strider;	stride counter to shift to correct location based on N
reg [11:0] kernel_req_base;	base counter of weights_sram_read_address
reg [11:0] kernel_req_offset;	counter to shift within a row to increment weights_sram_read_address
reg [11:0] output_req_base;	base counter of output_sram_write_address
reg [11:0] output_req_offset;	counter to shift within a row to increment output_sram_write_address
reg [11:0] output_req_row_strider	stride counter to shift to correct location based on N
Memories	
reg signed [7:0] Kernel_Matrix [0:8];	stores nine 8-bit values
reg signed [7:0] Input_Matrix_Section [0:15];	stores the 4*4 input subset

reg [7:0] N_input;	stores the input matrix dimension
reg signed [19:0] Conv_Output [0:3];	stores the 4 outputs of the MAC
reg [15:0] output_maxpooling;	stores packed outputs to be written to the output SRAM
STATES	
System Control	
reg [3:0] Top_Level_Current_State;	System control states
reg [3:0] Top_Level_Next_State;	
Kernel	
reg [3:0] Kernel_Current_State;	Kernel reading states
reg [3:0] Kernel_Next_State;	
Reading Inputs	
reg [3:0] Reading_Current_State;	Reading Input FSM states
reg [7:0] Input_next_State;	
Convolution, ReLu, MaxPooling and Output	
reg [3:0] Conv_Current_State;	Operational States
reg [3:0] Conv_Next_State;	
Control Flags & Counters	
reg input_begin;	controls the reading of inputs
reg N_received;	flag that signals if input dimensions have been stored
reg Final_Input_Found;	end of matrix flag
reg [7:0] count_reading_row_switch;	Counters, based on N, to determine when end of row and end of current image are reached
reg [7:0] count_reading_column_switch;	
reg kernel_begin;	controls reading of kernel values from weights SRAM
reg kernel_ready;	controls when to begin convolution
reg conv_begin;	flag to begin convolution when we have sufficient values
reg [10:0] relu_counter;	counts number of ReLus done according to N to determine movements to next stage
reg maxpooling_done;	controls reading in a new set of data when current operations are done
reg maxpooling_flag;	inverting flag to alternate between sliding output values or incrementing write address

4. Technical Implementation

The system design process involved designing the data path incrementally, a process that was augmented by performing mathematical simulations on paper and Excel. This was followed by identifying the required states for the machine, and building controller logic for each section. Frequent verification and analysis in ModelSim provided an opportunity for constant verification, and enabled the identification of errors early in the development phase.

Finally, the design was synthesized on Synopsys, with a focus on eliminating warnings, errors, and undesirable design outcomes such as latches. This process was repeated throughout the development phase to provide insights on performance improvements and algorithmic optimizations.

5. Verification

A testbench was provided, written in SystemVerilog, which compares the outputs of this design with those from an un-synthesizable golden model. Serving as the primary verification tool used in this project, this was used to facilitate iterative development and debugging, as well as to ensure that optimization efforts continued to produce correct outputs.

6. Results Achieved

- All verification tests in the testbench were successfully asserted, and Synopsys indicates a lack of warnings and errors of an undesirable nature.
- The following details were captured from Synopsys reports:
- Total Clock Cycles for Execution: 26,236 (ModelSim)
- Clock Period Achieved = 5.0 ns (Synopsys)
- Chip area of the design, obtained after synthesis: 7893.5500 um^2

7. Conclusions

The design provided in this report passes verification and synthesis without error, and therefore successfully accomplishes the objectives of this project. The desire to balance size and performance is shown in the metrics achieved here; a first step at future improvements would be to reduce the overall size of the synthesized module via a concerted effort to minimize the number of finite state machines used in the final design. A further step would be to pipeline the data path, thereby enabling more outputs to be written in a shorter time, in order to enhance the critical path and provide significant performance improvements.