Deep Neural Network Accelerator using Verilog

Name: Sounder Rajendran
Unityid: srajend4
StudentID: 200475996

Delay (ns to run provided example):
28400 ns
Clock period: 5
# cycles": 5680 cycles

Logic Area:
(um^2)
12070.2820
um^2

1/(delay x area)
($ns^{-1}.um^{-2}$)

2.917×10^{-9} $ns^{-1}.um^{-2}$

Delay (TA provided example. TA to complete)

1/(delay x area) (TA)

Abstract:
A deep neural network accelerator circuit is designed using Verilog, keeping in mind the inherent parallel nature of silicon circuits and parallel algorithms that can be utilized in design of multi-stage and multi-state hardware. Convolution layer, max-pooling layer and ReLu activation layers are implemented, that take in an image matrix containing pixel values and a kernel matrix of desired values, to apply a filter, extract useful information (edge pixels, feature point extraction, etc.) from the image and decrease the overall size of the image while keeping the useful information intact. The approach taken to design, interfaces used, implementation, and results achieved are discussed. A combination of testbench using Verilog, and python based golden output creation and verification using modelsim timing diagrams were used. A performance per area factor of $2.917e^{-9}$ is achieved, which is calculated using synopsis design_vision synthesis tool.

Deep Neural Network Accelerator using Verilog

Sounder Rajendran

1. Introduction:

An efficient multi-stage artificial neural network is designed using Verilog.

Input images are transformed into input arrays and are convoluted against a kernel acting like a filter in order to extract useful information from the image. The hardware accelerator that is designed, processes these image arrays inside a hidden layer. The hidden layer consists of a convolution layer, ReLu, and a max-pooling layer. In the convolution layer (whose outputs are fitted into a range using a linear activation function known as ReLu; ReLu is perfect for activating image bits as the range of pixels does not fall beneath 0), the output of the convolution is then aggregated, which helps to reduce the overall image size, while still preserving the valuable information obtained from convolution. The aggregation algorithm used is max-pooling, which keeps the most intense bit within a chosen group of bits.

The neural network accelerator was designed in a manner such that two convolution elements are being processed every six clock cycles (after all inputs were available for fetching) and thereby we get one-half of the max pooling output every six clock cycles on average.

Usually, convolution and max pooling layers require multiple rows of input from the input and kernel matrices before they can arrive at a single output element. Not waiting for the complete availability of inputs, finding the temporary values of convolution and max pooling, and iteratively arriving at the complete elements of convolution and max-pooling helps transform the general serial algorithm of the convolution layer and max-pooling layer into a highly parallel implementation without spending too many cycles waiting for a complete set of input values.

The algorithms utilized, parallel and pipelined data fetching and processing, the microarchitecture of design, the design choices made, and techniques used to overcome the constraints that allowed arrival at the microarchitecture are discussed in the forthcoming paragraphs.

2. Micro-architecture:

The design of any hardware, apart from its own execution algorithms, is restricted by other hardware it depends on. The neural network accelerator that is to be designed depends on input SRAM and weights SRAM to fetch its input and kernel matrices upon which convolution and max pooling are to be done, and it also depends on output SRAM to store its output. The SRAMs are configured in such a way that they contain two blocks of data on each address line which is arranged in a linear fashion. Therefore, in order to obtain all necessary inputs to compute one element of the convolution matrix, we have to have 3 rows and 3 columns of input elements. Since the 9 elements of the kernel remain unchanged, we fetch them once, store them in a register array and use them whenever required. As for the input elements, we fetch data from one address at a time. The number of data elements available is found by using the size value given as the first element. Therefore, we fetch the size value of an input matrix first and use it to calculate the number of convolution elements. To calculate a particular convolution element, convolution_matrix_element (p, q), the input elements received (3x3) are paired with corresponding kernel elements (3x3) for one-on-one multiplication-accumulation. The MAC function is done till we arrive at the complete convolution output. Since we get two data at the same time, we multiply-accumulate the two inputs at the same time. For every second input received, we have a stray input value that isn't useful at any stage of convolution for the convolution element being calculated. To make use of this value and not discard it, we calculate the value of convolution_matrix_element (p, q+1). Therefore, we calculate 4 multiplication and 2 accumulation values in parallel with two sets of data lines containing 4 data elements. The parallel MAC hardware is shown below. The temporary conv elements calculated are considered complete when we process all 9 elements required to complete convolution calculation. Then the values are transferred from temp registers to the max-pooling stage.

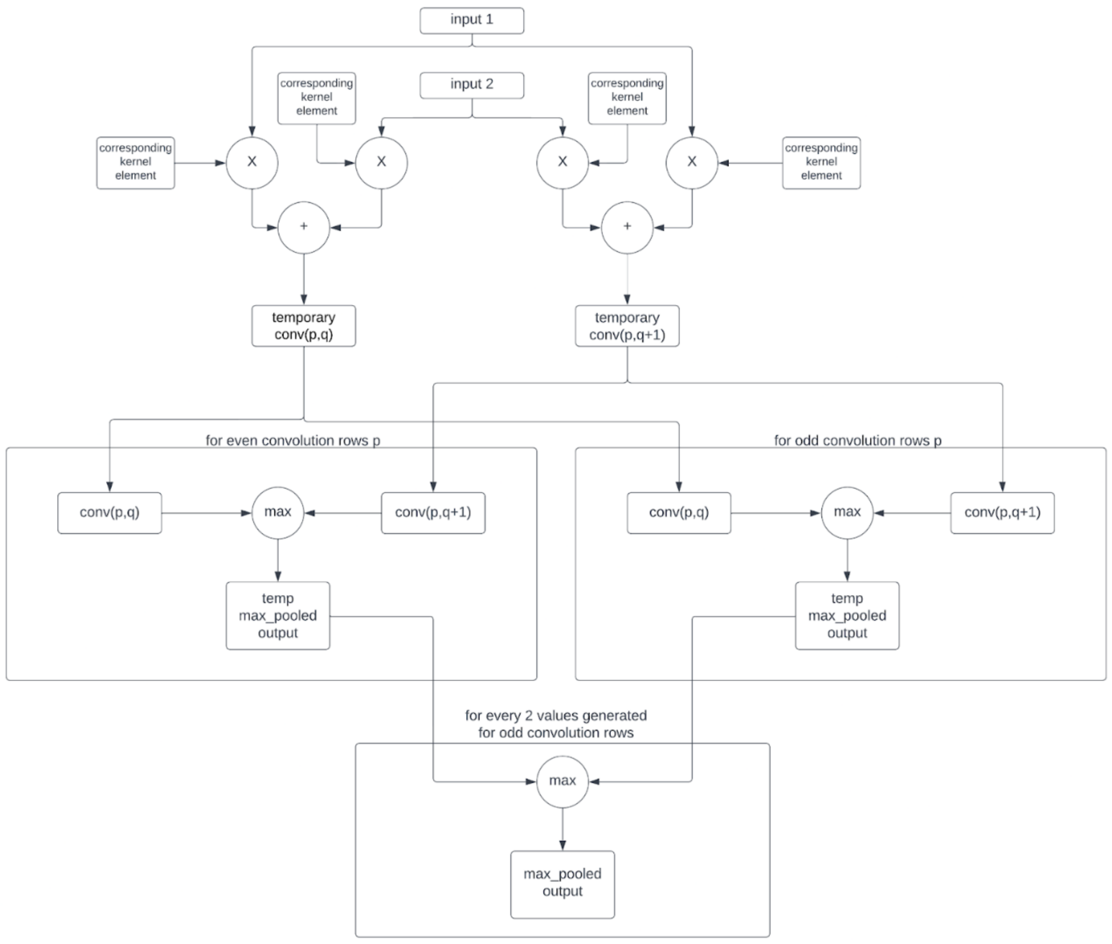

input 1
input 2
corresponding kernel element
corresponding kernel element
corresponding kernel element
corresponding kernel element
X
+
temporary conv(p,q)
temporary conv(p,q+1)
for even convolution rows p
for odd convolution rows p
conv(p,q)
max
conv(p,q+1)
temp max_pooled output
for every 2 values generated for odd convolution rows
max_pooled output

Fig 1: The parallel algorithm

The two values that are received at the max pooling stage, every cycle is compared with each other and the max is stored (for an even row) in a temporary max_pool register with 31 flipflops. For every odd row, the calculated temp_max_pooled elements are additionally compared with the temp_max pooled values stored in the 31 temp_max_pooled registers, and finally, we arrive at the final max_pooled output. In total, this takes 6 cycles to fetch every required element from input SRAM to achieve two convolution elements, and thereby one-half of the max pooled value. Therefore, for every complete max pooled output, we spend 12 cycles on average. This figure varies when we access edge values and because of state transitions during different input fetches.

3. Interface specification

The neural network accelerator is designed to receive data from input SRAM, which gives one data block containing two data values per address, for every cycle. At the end of computations, the hardware is designed to write to output SRAM one data line at a time, every clock cycle.

Optionally given scratchpad SRAM isn’t utilized to save area. Weights SRAM is accessed sequentially at the beginning of input fetch state to get kernel values. The kernel values are then stored in a register array to be accessed in parallel over multiple convolution stages.

The control signals:

All control signals are 1 bit wide.

output reg dut_busy is set to true when the hardware accelerator accepts dut_run signal and starts data fetching.

input wire dut_run is received as input trigger to start the accelerator.

input wire reset_b active low reset. When signal is low, we set all controls to initial state.

input wire clk gives the clock signal for sequential logic.

SRAM interface

Input, output and weights SRAM have similar interfaces.

sram_write_enable is set to high when there is enough data to be written to output SRAM. This is a status signal and so it is one bit wide.

sram_write_addresss is set to write address when there is data to be written to output SRAM. This is 12 bits wide.

sram_write_data is set to 16-bit data packet containing two data values that is to be written to address mentioned in sram_write_address. It is 16 bits wide.

sram_read_address is used to set 12-bit address when trying to fetch data from SRAM.

input_sram_read_data is a 16-bit wide reg used to read a line of data addressed using sram_read_address. Each data line read contains 2 data blocks.

4. Technical Implementation:

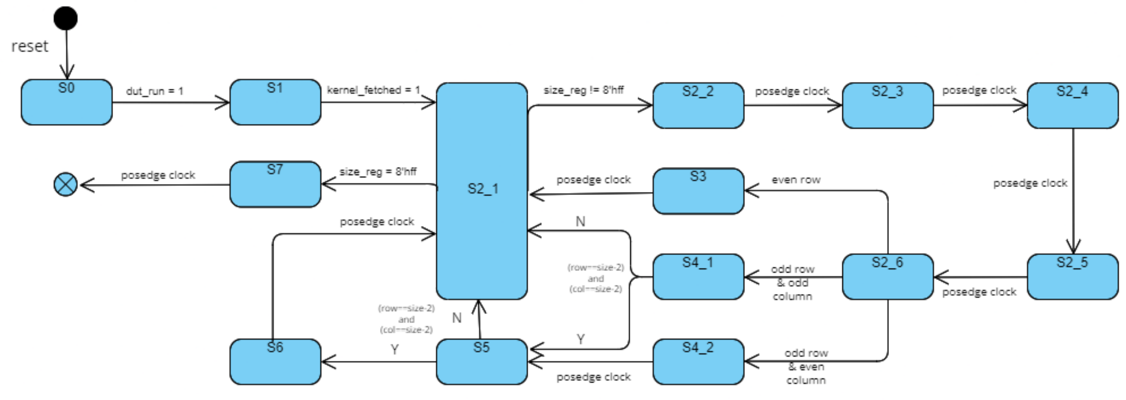
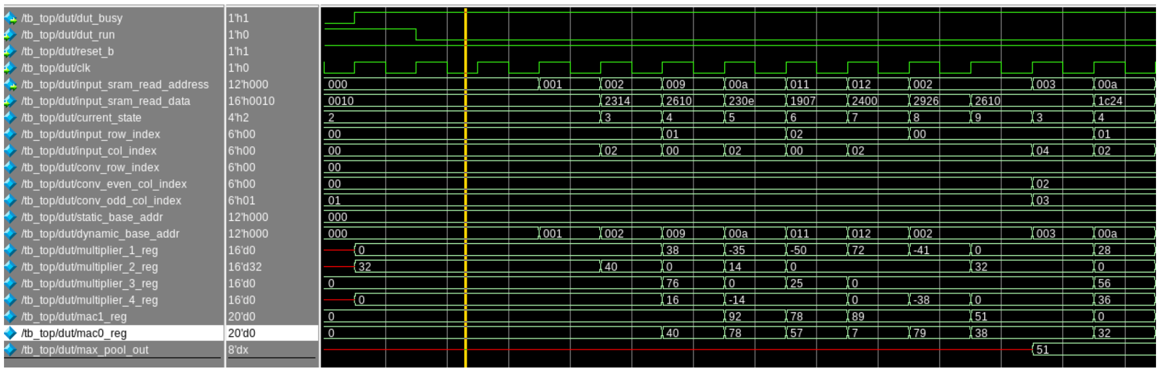


Fig 2: Finite State Machine Diagram

An FSM with 15 states is designed keeping in mind each state must last for minimum number of clock cycle (to make control easier). The hardware design is split into controller and datapath.

Timing Diagram:

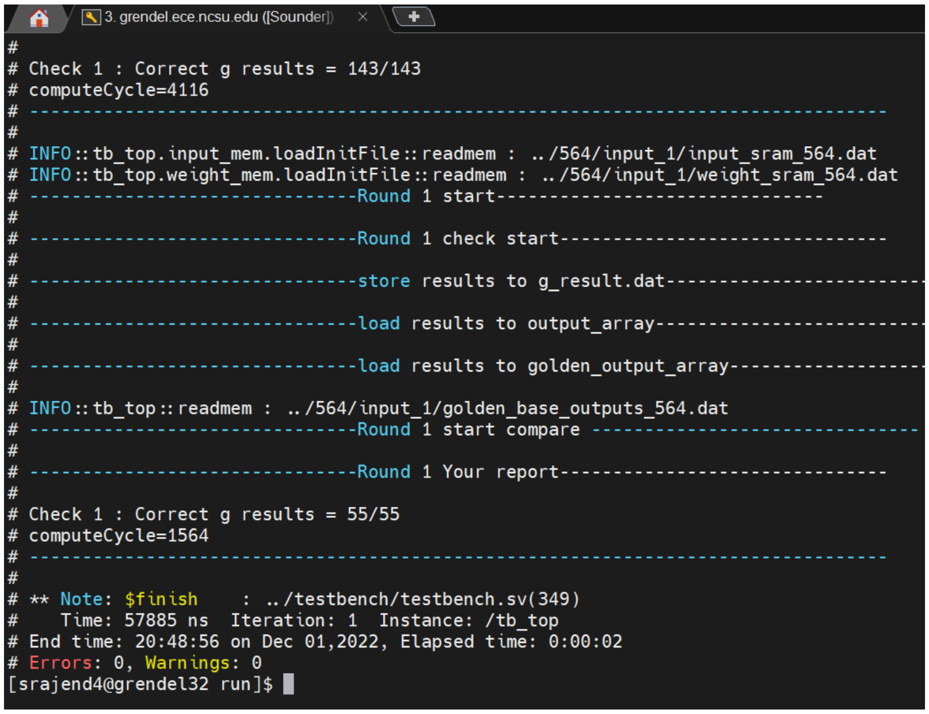


```
/tb_top/dut/dut_busy                 1'h1
/tb_top/dut/dut_run                  1'h0
/tb_top/dut/reset_b                  1'h1
/tb_top/dut/clk                      1'h0
/tb_top/dut/input_sram_read_address  12'h000
/tb_top/dut/input_sram_read_data     16'h0010
/tb_top/dut/current_state            4'h2
/tb_top/dut/input_row_index          6'h00
/tb_top/dut/input_col_index          6'h00
/tb_top/dut/conv_row_index           6'h00
/tb_top/dut/conv_even_col_index      6'h00
/tb_top/dut/conv_odd_col_index       6'h01
/tb_top/dut/static_base_addr         12'h000
/tb_top/dut/dynamic_base_addr        12'h000
/tb_top/dut/multiplier_1_reg         16'd0
/tb_top/dut/multiplier_2_reg         16'd32
/tb_top/dut/multiplier_3_reg         16'd0
/tb_top/dut/multiplier_4_reg         16'd0
/tb_top/dut/mac1_reg                 20'd0
/tb_top/dut/mac0_reg                 20'd0
/tb_top/dut/max_pool_out             8'dx
```

Fig 3: Timing diagram showing calculation of one element of max-pooling layer

5. Verification:

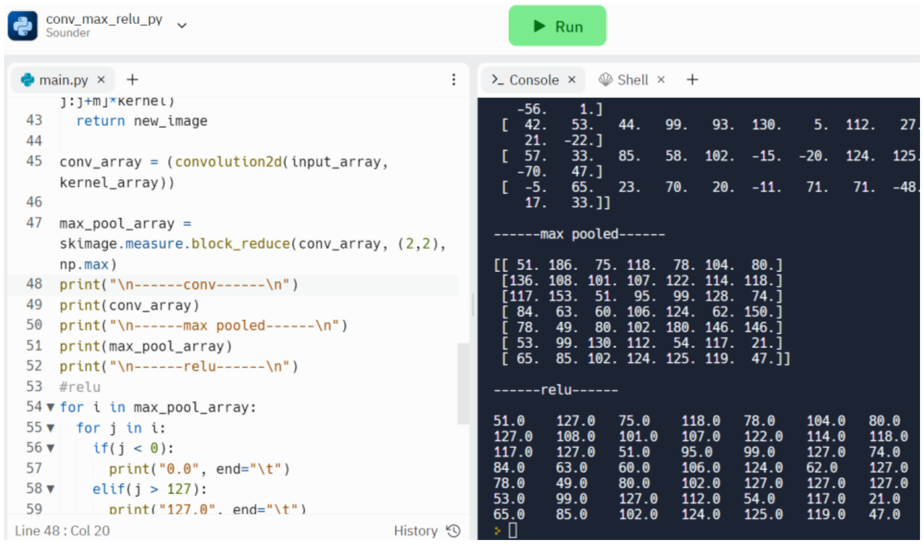
The make verify-564-base is run on Grendel to verify the correctness of the entire circuit.



```
#
# Check 1 : Correct g results = 143/143
# computeCycle=4116
# --------------------------------------------------------------------------------
#
# INFO::tb_top.input_mem.loadInitFile::readmem :  ../564/input_1/input_sram_564.dat
# INFO::tb_top.weight_mem.loadInitFile::readmem :  ../564/input_1/weight_sram_564.dat
# ------------------------------Round 1 start------------------------------
#
# ------------------------------Round 1 check start------------------------------
#
# ------------------------------store results to g_result.dat------------------------
#
# ------------------------------load results to output_array------------------------
#
# ------------------------------load results to golden_output_array-----------------
#
# INFO::tb_top::readmem :  ../564/input_1/golden_base_outputs_564.dat
# ------------------------------Round 1 start compare ------------------------------
#
# ------------------------------Round 1 Your report------------------------------
#
# Check 1 : Correct g results = 55/55
# computeCycle=1564
# --------------------------------------------------------------------------------
#
# ** Note: $finish    : ../testbench/testbench.sv(349)
#    Time: 57885 ns  Iteration: 1  Instance: /tb_top
# End time: 20:48:56 on Dec 01,2022, Elapsed time: 0:00:02
# Errors: 0, Warnings: 0
[srajend4@grendel32 run]$
```

Fig 4: make verify-564-base command results to verify output values using modelsim

To verify circuit output in parts, the intermediate timing diagram is analyzed and reg+wire outputs are compared against the intermediate values obtained by running.a python script that gives outputs after each stage of the hardware.



```
j:j+mj*kernel)
  return new_image

conv_array = (convolution2d(input_array,
kernel_array))

max_pool_array =
skimage.measure.block_reduce(conv_array, (2,2),
np.max)
print("\n------conv------\n")
print(conv_array)
print("\n------max pooled------\n")
print(max_pool_array)
print("\n------relu------\n")
#relu
for i in max_pool_array:
  for j in i:
    if(j < 0):
      print("0.0", end="\t")
    elif(j > 127):
      print("127.0", end="\t")
```

```
  -56.   1.]
 [  42.   53.   44.   99.   93.  130.    5.  112.   27.
   21.  -22.]
 [  57.   33.   85.   58.  102.  -15.  -20.  124.  125.
  -70.   47.]
 [  -5.   65.   23.   70.   20.  -11.   71.   71.  -48.
   17.   33.]]

------max pooled------

[[ 51. 186.  75. 118.  78. 104.  80.]
 [136. 108. 101. 107. 122. 114. 118.]
 [117. 153.  51.  95.  99. 128.  74.]
 [ 84.  63.  60. 106. 124.  62. 150.]
 [ 78.  49.  80. 102. 180. 146. 146.]
 [ 53.  99. 130. 112.  54. 117.  21.]
 [ 65.  85. 102. 124. 125. 119.  47.]]

------relu------

51.0    127.0   75.0    118.0   78.0    104.0   80.0
127.0   108.0   101.0   107.0   122.0   114.0   118.0
117.0   127.0   51.0    95.0    99.0    127.0   74.0
84.0    63.0    60.0    106.0   124.0   62.0    127.0
78.0    49.0    80.0    102.0   127.0   127.0   127.0
53.0    99.0    127.0   112.0   54.0    117.0   21.0
65.0    85.0    102.0   124.0   125.0   119.0   47.0
>
```

Fig 5: Python script generating outputs that should be present after each stage

6. Results Achieved and Conclusion:

A deep neural network accelerator with convolution layer and max-pooled layer with ReLu activation function has been developed using parallel utilization of data retrieved from SRAMs. The number of clock cycles to complete calculation of sample input of matrix sizes ranging from taken 8x8, 16x16, 32x32 and 64x64 elements is found to be 5680 cycles with a clock period of 5ns. This gives a delay of 28400 ns for the entire hardware accelerator to run the given input. This result was achieved on a silicon base with a cell area of 12070.3 um^2 that gives a performance/area parameter of 2.917 x 10-9 ns^{-1}.um^{-2}.

Detailed analysis of timing diagram revealed improvements can be made in reducing number of adders and multipliers that were idle in certain states, which could've been handled different using resource sharing. The resultant algorithm might have been more complex and thus increased the total clock cycles taken to achieve the results, but would've given an improvement in power and area.