Part 1: FPGA Implementation

<u>Design Overview</u>

Overall, the design circuit is implemented in a streaming fashion. Figure 1 shows an example where the circuit processes a stream of 4 input pixels in 4 clock cycles. When the circuit receives an input pixel, it will be multiplied with each value in the 3x3 filter shown in blue. Then the products are added to their corresponding output pixels circled in the 3x3 red rectangles. An output pixel value is not completed until one clock cycle after its bottom right pixel streams into the circuit, multiplied with the bottom right filter value and added to itself. This process is pipelined so that after a warm-up period of 2*514(padded image width)+1(padding)+3(filter width), the circuit is able to output one valid output pixel every clock cycle, except when it's receiving padding zeros at the end of each input image row.

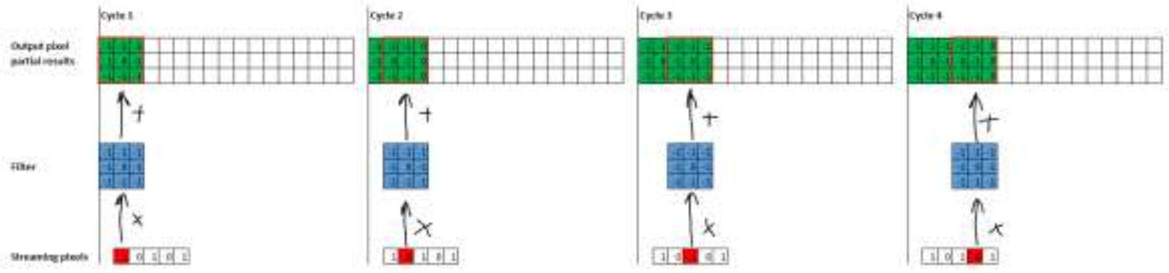


Figure 1. Streaming input pixels

<u>Schematic Overview</u>

The block diagram of the FPGA circuit is shown in Figure 2. The design contains two main blocks, the **Image Convolution Datapath** and the **Padded input image row, col index tracker**, which are outlined with two light yellow boxes respectively. The input and output signals of the module are shown in green.

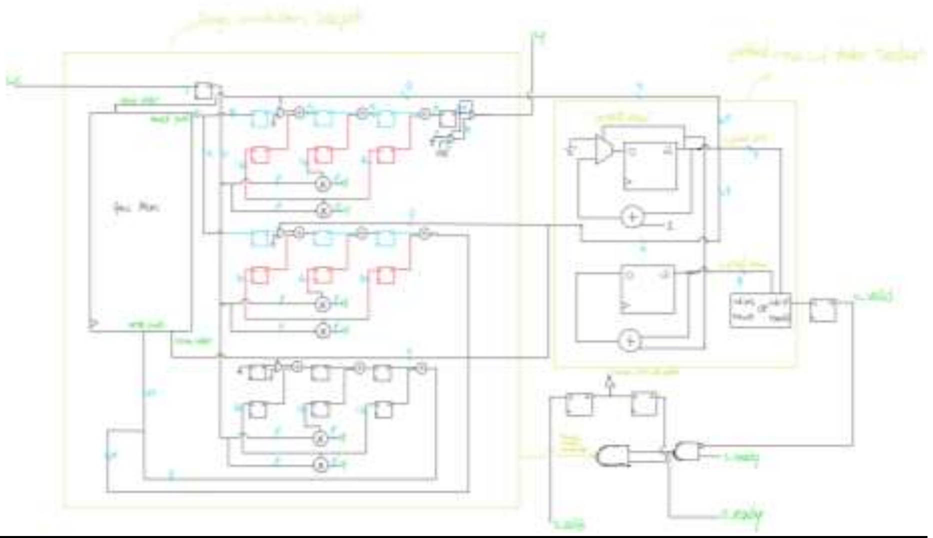


Figure 2. Block diagram

Peripherals
The entire circuit is operating on the same clock and receives the same reset signal (not shown in Figure 1). Input signals i_x, i_valid are buffered to ensure no valid input is missed, and output signals o_y, o_valid are also buffered to hold valid results. When the downstream module is not ready (I_ready==1'b0) but there is a valid (o_valid == 1'b1) output pixel in the output buffer, the Datapath and the index tracker will be stalled and the circuit will not be ready to accept new inputs (o_ready==1'b0) until the downstream module is ready again.

Image Convolution Datapath
The datapath includes a single 32-bit read-write port synchronous RAM of depth 512 (output image width) words implemented using an M20K block. Each RAM cell stores the partial results of two output pixels that are signed values of 16 bits. The datapath also includes 9 16-bit adders and 3 DSP blocks. Each DSP has 2 18x18 independent multipliers that accept 2 8-bit operands and output a 16-bit product. When valid input pixels stream into the circuit, the datapath multiply each buffered input pixel with the values in the first two columns of the 3x3 filter using the 3 DSPs. The 3x3 product buffers (red flip-flops in Figure 2) will map the second column products into its own second column, and the first column products into its own first and third columns as the filter is x-symmetric. In the next clock cycle, the buffered products will be added to the partial output pixel values stored in the 3x3 shift registers shown as blue flip-flops.
At each clock cycle, the 3x3 shift registers shift the partial results from column to column so that the output pixels can get accumulated with all the products between itself or their neighboring pixels and the filter values. The first column of the 3x3 shift registers also loads partial results from the RAM while the last column sends the value of the first row to the output buffer and stores the values of the second and third rows back to the RAM. This essentially resembles how the red rectangle 3x3 box moves to the right at every clock cycle when there is a valid input pixel as shown in Figure 1.

Padded input image row, col index tracker
The column index of the padded input image is incremented by 1 when the circuit sees a valid buffered input pixel. It will be reset to 0 when it's at the end of a row and the row index will increase by one. The row and column indices are used to decide whether a valid output pixel can be provided in the next clock cycle. The column index is also used as read or write addresses to the RAM.

FPGA implementation results

	Calculations	**Result**
ALM Utilization	**ALM needed = 199** **ALM unavailable = 47** **199 – 47 = 152**	**152 / 427200 (<1%)**
DSP Utilization	**N.A**	**3 / 1518 (<1%)**
M20K Utilization	**N.A**	**1 / 2713 (<1%)**
Maximum Operating Frequency	**Worst Slack = -1.642ns** **10^9 / (1+1.642) ~= 378.5M**	**378.5MHz**
Cycles for Test 7a	**Simulation time ~= 5284258ns** **Clock cycle length = 20ns**	**264213 cycles**

	5284258 / 20 ~= 264213	
Dynamic Power for one module @ maximum frequency (W)	**DSP Power = 0.34mW** **M20K Power = 0.63mW** **Combinational cell = 0.27mW** **Register cell = 0.44mW** **Fmax = 378.5MHz** **Simulation Fmax = 50MHz** **(0.34+0.63+0.27+0.44) *378.5/50/1000 = 0.01272W**	**0.01272W**
Throughput of one module (GOPS)	**Fmax = 378.5MHz** **Cycles for Test 7a = 264213** **Ops = 512 x 512 x (9+9)** **= 4718592** **4718592 / 264213 * 378.5M = 6.75965 GOPS**	**6.75965 GOPS**
Throughput of full device (GOPS)	**#Max. copy = # DSP util / #DSP = 1518 / 3 = 506** **Throughput of one module = 6.75965 GOPS** **506 * 6.75965** **= 3420.3822 GOPS**	**3420.3822 GOPS**
Total Power for full device (W)	**Static power = 1704.48mW** **I/O power = 0.19mW** **Clock power = 2.72mW** **#Max. copy = 506** **#Dynamic Power for one module @ Fmax = 0.01272W** **(1704.48+0.19+2.72)/1000 + 506*0.01272 = 8.14371W**	**8.14371W**

Table 1: FPGA implementation results

Simulation and Testbench Outputs

The following simulation waveforms and results are from test 7a. The waveforms shown in Figures 3,4 capture the partial results before storing into the RAM and after loading from the RAM of the first valid output pixel and its final correct value. As it's a streaming pipelined design where all the output pixels are computed in the same way, only showing the final testbench pass result shown in Figure 6. and how the first output pixel is computed should be sufficient.

The first output pixel of test 7a is dependent on the input pixels of the first three columns and rows of the padded input image. These 9 input pixels are [[0,0,0],[0,44,45],[0,44,44]]. As shown in Figure 3, the circuit is receiving input pixels at row=1. Given the edge filter in test 7a, the expected partial result should be 44*8+45*(-1)=307, which is exactly the value written into the RAM through the 'wr_upper' signal. As shown in Figure 4, the circuit is receiving input pixels

at row=2 and the partial result 307 is successfully loaded from the RAM through the ‘rd_upper’ signal. The expected final result should 307+44*(-1)+44*(-1)=219, which is the exact value of o_y at the clock cycle when o_valid is high as shown in Figure 5.
Figure 6 shows all the output pixel values match the expected values.

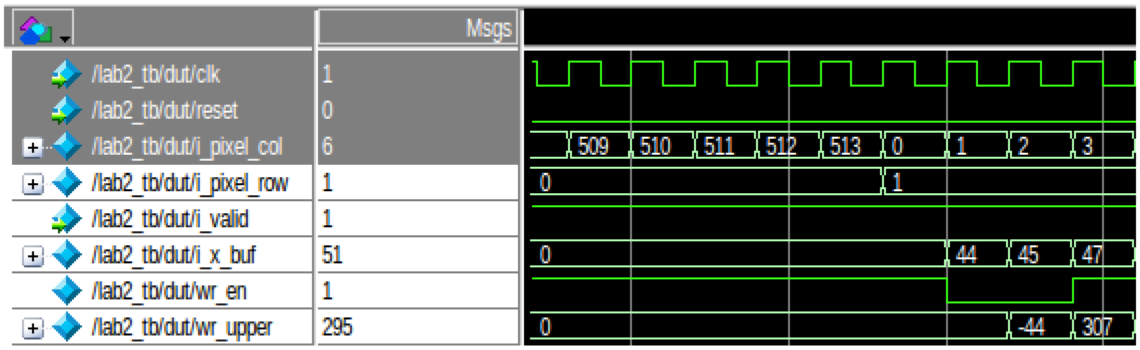


Figure 3. Writing 307 into RAM.

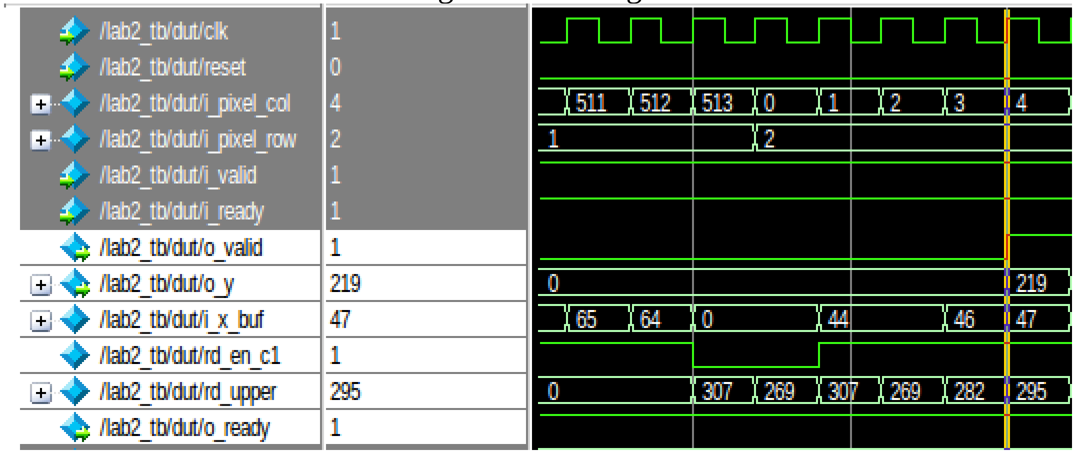
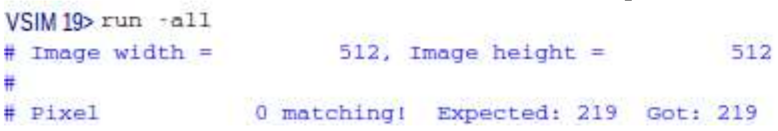
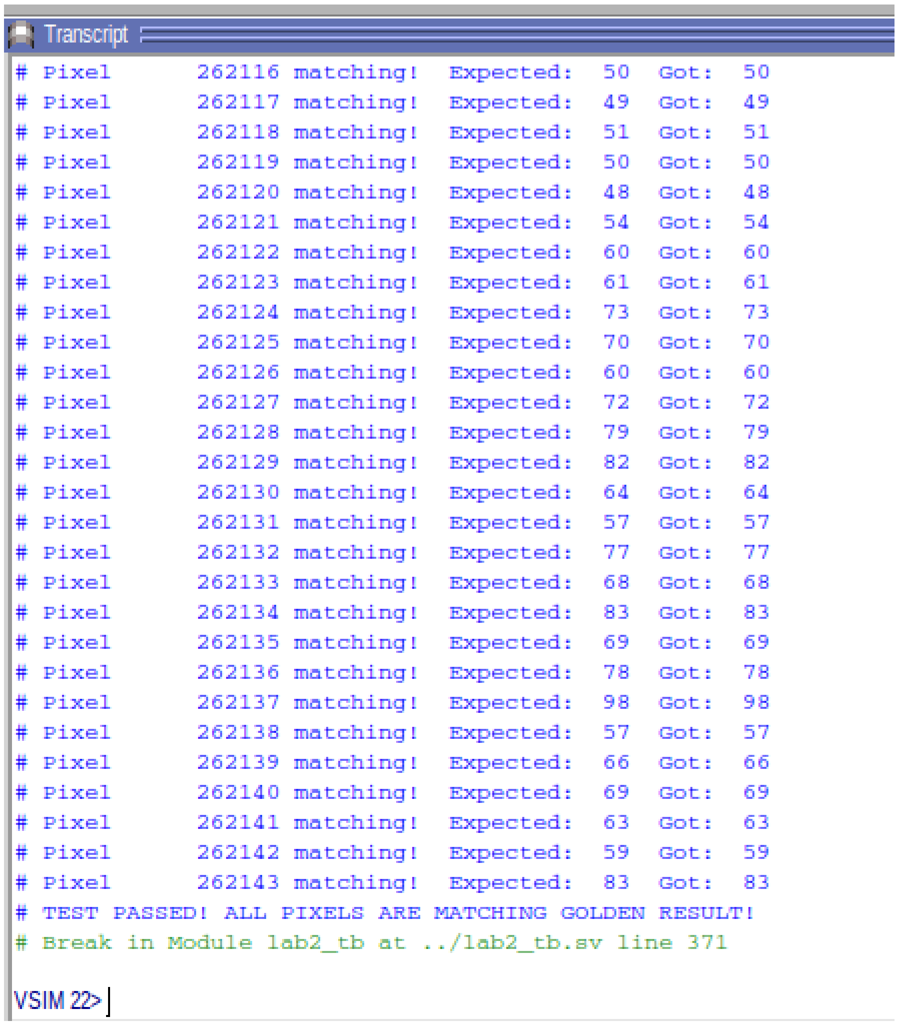


Figure 4. Reading 307 from the RAM and accumulating new products on it to get 219, the final output.



```
VSIM 19> run -all
# Image width =          512, Image height =          512
#
# Pixel          0 matching!  Expected: 219  Got: 219
```

Figure 5. The first output pixel value is 219



Transcript

```
# Pixel      262116 matching! Expected:  50  Got:  50
# Pixel      262117 matching! Expected:  49  Got:  49
# Pixel      262118 matching! Expected:  51  Got:  51
# Pixel      262119 matching! Expected:  50  Got:  50
# Pixel      262120 matching! Expected:  48  Got:  48
# Pixel      262121 matching! Expected:  54  Got:  54
# Pixel      262122 matching! Expected:  60  Got:  60
# Pixel      262123 matching! Expected:  61  Got:  61
# Pixel      262124 matching! Expected:  73  Got:  73
# Pixel      262125 matching! Expected:  70  Got:  70
# Pixel      262126 matching! Expected:  60  Got:  60
# Pixel      262127 matching! Expected:  72  Got:  72
# Pixel      262128 matching! Expected:  79  Got:  79
# Pixel      262129 matching! Expected:  82  Got:  82
# Pixel      262130 matching! Expected:  64  Got:  64
# Pixel      262131 matching! Expected:  57  Got:  57
# Pixel      262132 matching! Expected:  77  Got:  77
# Pixel      262133 matching! Expected:  68  Got:  68
# Pixel      262134 matching! Expected:  83  Got:  83
# Pixel      262135 matching! Expected:  69  Got:  69
# Pixel      262136 matching! Expected:  78  Got:  78
# Pixel      262137 matching! Expected:  98  Got:  98
# Pixel      262138 matching! Expected:  57  Got:  57
# Pixel      262139 matching! Expected:  66  Got:  66
# Pixel      262140 matching! Expected:  69  Got:  69
# Pixel      262141 matching! Expected:  63  Got:  63
# Pixel      262142 matching! Expected:  59  Got:  59
# Pixel      262143 matching! Expected:  83  Got:  83
# TEST PASSED! ALL PIXELS ARE MATCHING GOLDEN RESULT!
# Break in Module lab2_tb at ../lab2_tb.sv line 371

VSIM 22>
```

Figure 6. Passed the testbench

Part 2: Efficiency Comparisons

CPU convolution functions:

The simple CPU convolution function (cpu_conv2d) iterates through all the filters stored in memory one by one. For each filter, it iterates through all the pixels of the zero-padded input image in a row-major order. For each pixel, the filter is mapped to the padded input image in such a way that the filter's top left corner is always matching the current pixel. All the filtering values are then multiplied with their matched pixel values. The sum of all the FILTER_SIZE*FILTER_SIZE number of products is the output image pixel value, which is stored into the corresponding output image position.

The hand-vectorized function (cpu_conv2d_vectorized) is similar to the simple convolution function. The difference is that it uses a dot product engine of size 3 to do the multiply-accumulate operations for each filter row. The sum of all the row accumulations is the output image pixel value.

When the convolution functions are multi-threaded (cpu_conv2d_multithreaded, cpu_conv2d_vectorized_multithreaded), the filters are working in parallel, each taking a thread, to compute output image pixels.

GPU convolution CUDA kernel:

The kernel splits the zero-padded input image into multiple tiles of size 34x34, which are shared memories for the threads in the same tile to access and work in parallel. The filters are stored in a constant memory for fast access. After loading all the split padded input image data into their corresponding tiles, where the data of the same tile are loaded synchronously, all the threads in the same tile will be multiplying the filter values with the corresponding pixel values and store the product accumulations into the correct output image positions synchronously. There is no dependency across the tiles, so that the padded input image data are loaded and processed (multiply-accumulated with different filter values) asynchronously.

CPU differences

As shown in table 2, hand-vectorized implementation is around 1.6x faster than the basic implementation when there is no optimization. This is because hand-vectorized implementation uses dot product engines (_mm_dp_ps) to do multiply-accumulate accumulations, which is more efficient than straight-forward '*', '+' operations used in the basic implementation. When the -O2 flag is high, the hand-vectorized implementation is around 1.1-1.3x faster than the basic implementation. When the -O3 flag is high, we hardly see any improvements from the hand-vectorized implementation, which is around 2x slower than the basic implementation. This is because when compiler optimizations are enabled, some vectorizing optimizations will be enabled (e.g. -ftree-vectorize) [1] so that the hand-vectorized implementation loses its advantages.
When -O3 is enabled, 1-thread and 4-thread implementation performances are compared for both basic and vectorized versions. When the number of threads increase from 1 to 4, the runtime of both basic and vectorized versions are reduced by around 3 to 4 times for filter numbers 4, 8 and 16.

Table 2: CPU and GPU runtime

	Runtime (ms)			
Filter #	**1**	**4**	**16**	**64**
GPU	0.029995	0.0992019	0.373319	1.4955
CPU (basic – no opt – 1 thread)	8.87822	35.477	142.034	567.643
CPU (vectorized – no opt – 1 thread)	5.39382	21.4862	90.2207	348.405
CPU (basic – O2 – 1 thread)	1.60637	6.36839	25.61	101.829
CPU (vectorized – O2 – 1 thread)	1.26328	5.04155	23.0477	83.4966
CPU (basic – O3 – 1 thread)	0.682404	2.71905	10.89	43.2622
CPU (vectorized – O3 – 1 thread)	1.26105	5.04697	22.9792	83.4497
CPU (basic – O3 – 4 threads)	0.739527	0.761724	2.9678	11.7565
CPU (vectorized – O3 – 4 threads)	1.34084	1.19818	8.05229	22.1228

FPGA vs. GPU vs. CPU

I decided to derive GPU (20nm) performances by MOSFET scaling, which assumes power and area would be quadratically scaled with the size of transistors given a constant transistor power density. Assume the number of transistors remain the same, the capability doing parallel tasks

will remain the same so that runtime will not change. The estimated throughput, throughput/W and throughput/mm^2 are listed in Table 3 below.
As shown in Table 3, FPGA has the highest throughput which is more than 2x larger than GPU and more than 16x larger than CPU. FPGA achieves the highest throughput because it has enough resources to do the image convolutions for all the 64 filters completely in parallel, where the output pixel computations for the same filter are well pipelined. In contrast, the other two processors are not structured to work on different filters completely in parallel. In GPU, the threads of each tile can only access the same filter synchronously (3rd dimension of a dim3 block is 1). In CPU, the highest number of filters it can work on in parallel is the same as the number of threads, which is only 4.
The second reason is FPGA doesn't have data loading overheads. Each module that does image convolution for one filter is implemented as a streaming interface, where the inputs are processed as soon as they reach the module. But both CPU and GPU need to load the data from memory before they can process them to compute output pixels.
FPGA also has the highest Energy Efficiency, which is more than 100x and 50x larger than the two GPUs and more than 500x larger than CPU. This is mainly because FPGA Power consumption is much lower compared to GPU and CPU. As the input filters to the FPGA implementation are x-symmetric, the number of multipliers used by FPGA is 6 instead of 9, which saves considerable amount of dynamic power. FPGA Area Efficiency is the highest as well. It's more than 2x larger than GPU (28nm) but only around 1.2x larger than GPU (scaled 20nm), whose estimated area is only around 51% of GPU (28nm). Although CPU Die size is around 2x smaller than FPGA and GPU, its Area Efficiency is still much smaller than FPGA and GPU due to its much lower throughput.

Table 3: FPGA vs. GPU vs. CPU (64 filters, 512x512)

	#Operations	Runtime (ms)	**Throughput (GOPS)**	**Power (W)**	**Energy Efficiency (GOPS / W)**	Die size (mm^2)	**Area Efficiency (GOPS / mm^2)**
FPGA (20nm)	64x512x512x(9+9)= 301989888	Cycles = 264213 Fmax = 378.5MHz 264213 * 378.5M/1000 = 0.6981	**432.6176**	**64 * 0.01272 + (1704.48+ 0.19+ 2.72)/1000 = 2.52147**	**171.5736**	~350	**1.23605**
GPU (28nm)	64x512x512x(9+9)= 301989888	1.49	**202.6778**	**165**	**1.22835**	396	**0.51181**
GPU (scaled 20nm)	64x512x512x(9+9)= 301989888	1.49	**202.6778**	**165 * (20/28)^2 = 84.184**	**2.40757**	396 * (20/28)^2 = 202.041	**1.00315**

CPU (22nm)	64x512x512x(9+9)= 301989888	11.7565	**25.6871**	**84**	**0.30580**	177	**0.14512**

Appendix A: HDL code

```
// This module implements 2D covolution between a 3x3 filter and a 512-pixel-wide
image of any height.
// It is assumed that the input image is padded with zeros such that the input
and output images have
// the same size. The filter coefficients are symmetric in the x-direction (i.e.
f[0][0] = f[0][2],
// f[1][0] = f[1][2], f[2][0] = f[2][2] for any filter f) and their values are
limited to integers
// (but can still be positive of negative). The input image is grayscale with 8-
bit pixel values ranging
// from 0 (black) to 255 (white).
module lab2 (
    input  clk,         // Operating clock
    input  reset,           // Active-high reset signal (reset when set to 1)
    input  [71:0] i_f,      // Nine 8-bit signed convolution filter coefficients
in row-major format (i.e. i_f[7:0] is f[0][0], i_f[15:8] is f[0][1], etc.)
    input  i_valid,         // Set to 1 if input pixel is valid
    input  i_ready,         // Set to 1 if consumer block is ready to receive a
new pixel
    input  [7:0] i_x,       // Input pixel value (8-bit unsigned value between 0
and 255)
    output o_valid,         // Set to 1 if output pixel is valid
    output o_ready,         // Set to 1 if this block is ready to receive a new
pixel
    output [7:0] o_y        // Output pixel value (8-bit unsigned value between 0
and 255)
);

localparam FILTER_SIZE = 3; // Convolution filter dimension (i.e. 3x3)
localparam PIXEL_DATAW = 8; // Bit width of image pixels and filter coefficients
(i.e. 8 bits)

// The following code is intended to show you an example of how to use paramaters
and
// for loops in SytemVerilog. It also arrages the input filter coefficients for
you
// into a nicely-arranged and easy-to-use 2D array of registers. However, you can
ignore
// this code and not use it if you wish to.
```

```
logic signed [PIXEL_DATAW-1:0] r_f [FILTER_SIZE-1:0][FILTER_SIZE-1:0]; // 2D
array of registers for filter coefficients
integer signed col, row; // variables to use in the for loop
always_ff @ (posedge clk) begin
    // If reset signal is high, set all the filter coefficient registers to zeros
    // We're using a synchronous reset, which is recommended style for recent
FPGA architectures
    if(reset)begin
        for(row = 0; row < FILTER_SIZE; row = row + 1) begin
            for(col = 0; col < FILTER_SIZE; col = col + 1) begin
                r_f[row][col] <= 0;
            end
        end
    // Otherwise, register the input filter coefficients into the 2D array signal
    end else begin
        for(row = 0; row < FILTER_SIZE; row = row + 1) begin
            for(col = 0; col < FILTER_SIZE; col = col + 1) begin
                // Rearrange the 72-bit input into a 3x3 array of 8-bit filter
coefficients.
                // signal[a +: b] is equivalent to signal[a+b-1 : a]. You can try
to plug in
                // values for col and row from 0 to 2, to understand how it
operates.
                // For example at row=0 and col=0: r_f[0][0] = i_f[0+:8] =
i_f[7:0]
                //         at row=0 and col=1: r_f[0][1] = i_f[8+:8] = i_f[15:8]
                r_f[row][col] <= i_f[(row * FILTER_SIZE * PIXEL_DATAW)+(col *
PIXEL_DATAW) +: PIXEL_DATAW];
            end
        end
    end
end

// Start of your code
localparam IMAGE_WIDTH = 512;
localparam BOARD_WIDTH = IMAGE_WIDTH+2;
localparam OFFSET = (FILTER_SIZE-1)>>1;
localparam RESULT_WIDTH = 2*PIXEL_DATAW;
logic signed [RESULT_WIDTH-1:0] dep_window [FILTER_SIZE-1:0][FILTER_SIZE-1:0];
logic signed [RESULT_WIDTH-1:0] products [FILTER_SIZE-1:0][FILTER_SIZE-1:0];
logic signed [RESULT_WIDTH-1:0] products_buf [FILTER_SIZE-1:0][FILTER_SIZE-1:0];
logic [2*RESULT_WIDTH-1:0] mem[0:IMAGE_WIDTH-1];
logic signed [RESULT_WIDTH-1:0] wr_upper, rd_upper;
logic signed [RESULT_WIDTH-1:0] wr_lower, rd_lower;
logic unsigned [8:0] wr_addr, rd_addr;
logic unsigned [9:0] i_pixel_col, i_pixel_row;
logic i_valid_buf;
logic i_valid_dbuf;
logic unsigned [PIXEL_DATAW-1:0] i_x_buf;
logic o_valid_buf;
logic unsigned [RESULT_WIDTH-1:0] o_y_buf;
logic signed [RESULT_WIDTH-1:0] signed_x;
logic signed [RESULT_WIDTH-1:0] signed_f;
logic signed [RESULT_WIDTH-1:0] conv_result;
logic rd_en_c1;
logic wr_en;
integer signed r,c;
// generate 3 DSP blocks
genvar i;
generate
    for(i=0; i<FILTER_SIZE; i++) begin : gen_dsp_blocks
        mult multDSP (
            .ax      (i_x_buf),
            .bx      (i_x_buf),
            .ay      (r_f[FILTER_SIZE-1-i][FILTER_SIZE-1-0]),
            .by      (r_f[FILTER_SIZE-1-i][FILTER_SIZE-1-1]),
            .resulta (products[i][0]),
            .resultb (products[i][1])
        );
    end
endgenerate
always_comb begin
    conv_result = dep_window[0][0] + products_buf[0][0];
    wr_upper = dep_window[1][0] + products_buf[1][0];
    wr_lower = dep_window[2][0] + products_buf[2][0];
end
always_ff @ (posedge clk) begin
    if (reset) begin
        i_valid_buf <= 1'b0;
        i_valid_dbuf <= 1'b0;
        i_x_buf <= '0;
        o_valid_buf <= 1'b0;
        o_y_buf <= '0;
        i_pixel_col <= '0;
        i_pixel_row <= '0;
        wr_en <= 1'b0;
        rd_en_c1 <= 1'b0;
        wr_addr <= '0;
        rd_addr <= '0;
        for(row = 0; row < FILTER_SIZE; row = row + 1) begin
```

```
            for(col = 0; col < FILTER_SIZE; col = col + 1) begin
                dep_window[row][col] <= '0;
                products_buf[row][col] <= '0;
            end
        end
    end else begin
        i_valid_buf <= o_ready ? i_valid : i_valid_buf;
        i_x_buf <= o_ready ? i_x : i_x_buf;
        o_valid_buf <=
((i_pixel_row>1&&i_pixel_col>2)||(i_pixel_row>2&&i_pixel_col==0));
        rd_en_c1 <= i_pixel_row>0&&i_pixel_col<IMAGE_WIDTH;
        i_valid_dbuf <= i_valid_buf;
        if (o_ready) begin
            if (i_valid_buf) begin
                // buffer the products
                for(row = 0; row < FILTER_SIZE; row = row + 1) begin
                    for(col = 0; col <= FILTER_SIZE/2; col = col + 1) begin
                        products_buf[row][col] <= products[row][col];
                    end
                    products_buf[row][FILTER_SIZE-1] <= products[row][0];
                end
                // increment column counter, update write and read address
                if (i_pixel_col==BOARD_WIDTH-1) begin
                    i_pixel_col <= '0;
                    i_pixel_row <= i_pixel_row + 1;
                    rd_addr <= '0;
                end else begin
                    i_pixel_col <= i_pixel_col + 1;
                    rd_addr <= i_pixel_col + 1;
                end
                wr_en <= i_pixel_col!=0&&i_pixel_col!=1;
                wr_addr <= i_pixel_col - 2;
            end
            if (i_valid_dbuf) begin
                // first col
                o_y_buf <= conv_result;
                if (wr_en) begin
                    mem[wr_addr] <= {wr_upper,wr_lower};
                end
                // second col
                dep_window[0][0] <= dep_window[0][1] + products_buf[0][1];
                dep_window[1][0] <= dep_window[1][1] + products_buf[1][1];
                dep_window[2][0] <= dep_window[2][1] + products_buf[2][1];
                // third col
                dep_window[0][1] <= (rd_en_c1?rd_upper:'0) + products_buf[0][2];
                dep_window[1][1] <= (rd_en_c1?rd_lower:'0) + products_buf[1][2];
                dep_window[2][1] <= products_buf[2][2];
                // from memory
                {rd_upper, rd_lower} <= mem[rd_addr];
            end
        end
    end
end

assign o_valid = o_valid_buf;
assign o_ready = ~o_valid || i_ready;
assign o_y = (0<=o_y_buf&&o_y_buf<256) ? o_y_buf[PIXEL_DATAW-1:0] :
(o_y_buf[RESULT_WIDTH-1]) ? 0 : 255;
// End of your code

endmodule
```