FPGA Implementation

Block Diagram

Figure 1 below shows the block diagram of image convolution on FPGA. In the beginning, filter coefficients (i_f) are saved to registers and input pixels (i_x) are saved to the memory block in raster order. The memory block is designed to save 3 rows of input pixels. After 3 rows are saved, the oldest row is replaced by the new row. As soon as the memory block reaches the third row, control signals will be sent through the pipelined registers, DSPs will receive pipelined signals and start valid computations. Inside each DSP block, the multiplication and summation of three pixels and three coefficients are reduced to:

(p0 + p2) x c0 + p1 x c1, where p0,p1,p2 are pixels and c0,c1 are filter coefficients.

This reduction step is achieved by taking the advantage of symmetric filter coefficients. It significantly reduces the DSP usage, because each row-wise convolution can be implemented using the primitive 18x19 sum of 2 DSP mode from Arria 10. After three rows compute their partial convolution results, we just need to sum the three values and send to the output. In the meanwhile, pipelined control signals will also indicate the output value is valid.

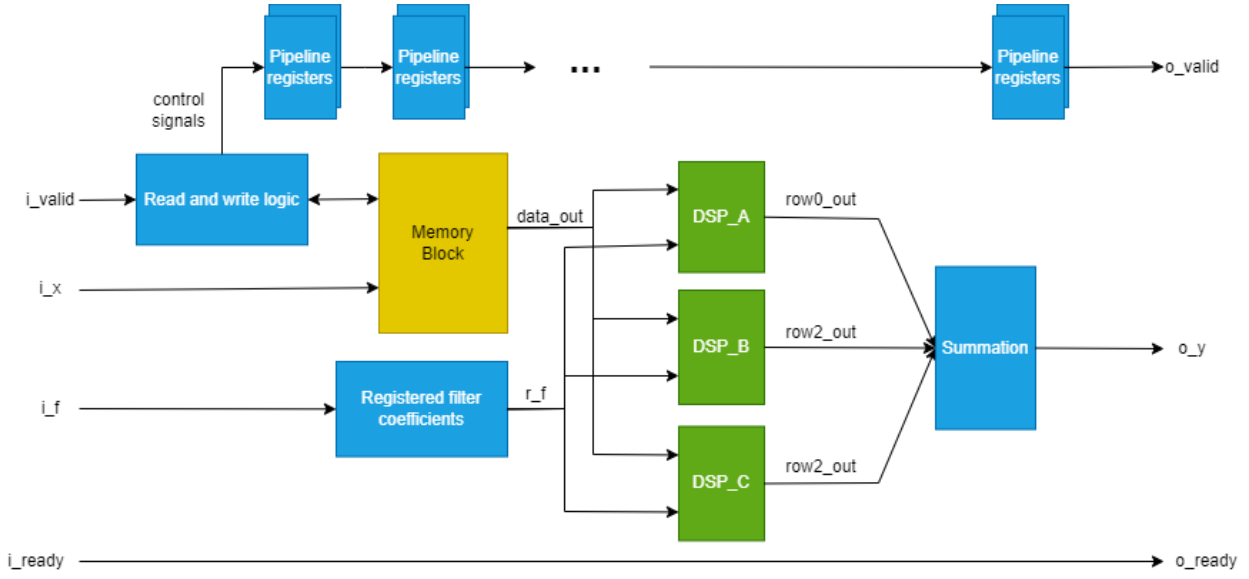

Figure 1: FPGA block diagram.

Table Results

Table 1: FPGA implementation results

	Result
ALM Utilization	148
DSP Utilization	3
BRAM (M20K) Utilization	3
Max Frequency (MHz)	772.20 MHz
Cycles for Test 7a	264222
Dynamic Power for one module @ max Frequency	26.87 mW
Throughput of one module (GOPS)	13.024 GOPS
Throughput of full device (GOPS)	6590.144 GOPS
Total power for full device (W)	15.3 W

ALM, **DSP** and **BRAM utilizations** are collected from the Resource Usage Summary under the Resource Section inside Fitter.

Max frequency is calculated by knowing the worst-case setup slack of Slow 900mV 0C model is -0.295 ns. Then, the frequency is $\frac{1}{1.295e-9}$ = 775.20 MHz.

Cycles for Test 7a are obtained by knowing the 7a testbench takes 5284438 ns to finish and the testbench clock period is 20 ns. $\frac{5284438}{20}$ = 264222 cycles.

Dynamic Power at max Frequency is calculated by linearly scaling the dynamic power of 1.74 mW at 50 MHz. 1.74 x $\frac{772.20}{50}$ = 26.87 mW

Throughput of one module is calculated by assuming the total computation is 512 x 512 x 17 = 4456448 operations, where 512 x 512 is the number of pixels and 17 is the number of operations needed for each pixel. The number 17 comes from the 3x3 convolution filter. This requires 9 multiplication and 8 addition to generate one output. The time is calculated from max frequency and number of cycles, $\frac{264222}{772.20M}$ = 0.00034216783 secs. Thus, the throughput is given by $\frac{4456448}{0.00034216783}$ = 13.024 GOPS.

Throughput of full device is simply multiplying the throughput of one module by the number of copies. Our design is limited by the number of DSPs, only 506 copies are allowed. 13.024 x 506 = 6590.14 GOPS.

Total power for full device is measured by considering the power consumption of each copy, the number of copies, static power dissipation, and IO power dissipation. 26.87 x 506 + 1704.41 + 0.06 + 0.13 = 15300 mW = 15.3 W

Waveform and Testbench Screen Shots

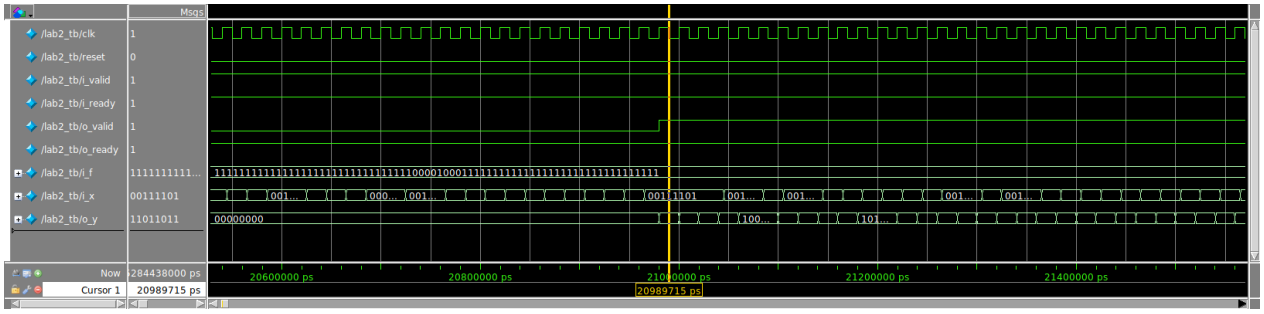

Figure 2: Testbench waveform for test 7a (when o_valid is true).

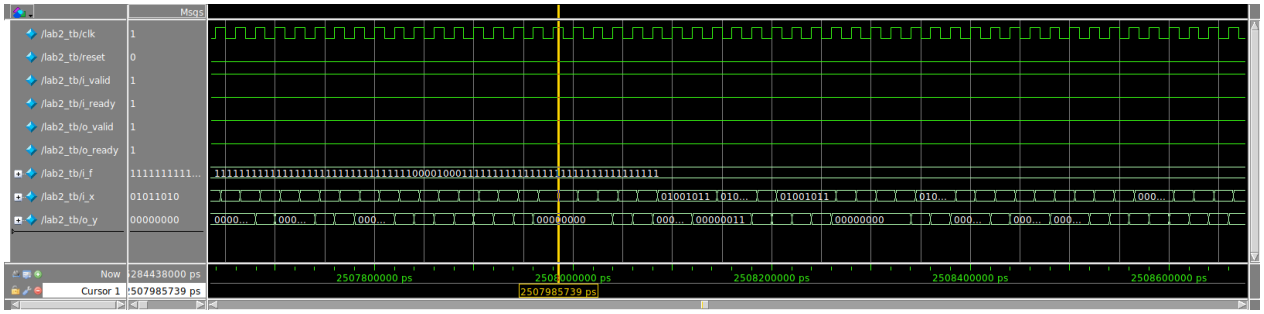

Figure 3: Testbench waveform for test 7a (pipelined outputs).

```
...
# Pixel      262106 matching!   Expected:  58   Got:  58
# Pixel      262107 matching!   Expected:  71   Got:  71
# Pixel      262108 matching!   Expected:  54   Got:  54
# Pixel      262109 matching!   Expected:  51   Got:  51
# Pixel      262110 matching!   Expected:  65   Got:  65
# Pixel      262111 matching!   Expected:  58   Got:  58
# Pixel      262112 matching!   Expected:  56   Got:  56
# Pixel      262113 matching!   Expected:  53   Got:  53
# Pixel      262114 matching!   Expected:  46   Got:  46
# Pixel      262115 matching!   Expected:  43   Got:  43
# 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
```

Testbench 1: Testbench outputs for test 7a.

CPU and GPU

CPU code high-level explanation

CPU code consists of four functions: cpu_conv2d, cpu_conv2d_multithreaded, cpu_conv2d_vectorized, and cpu_conv2d_vectorized_multithreaded. As we can see from the names, each function corresponds to basic, basic + threaded, vectorized, and vectorized + threaded implementations. The basic one simply computes convolutions in nested loops until it finishes. The vectorized versions replace the multiply and add process in the basic version with a vectorized dot-product operation of size 3 to speed up the process. The multithreaded implementations use openMP pragmas to unroll the outer loops and assign unrolled loops to different threads in order to reduce the time.

GPU code high-level explanation

First, CPU needs to take the jobs of reading the PGM file, allocating memory, and selecting proper block sizes. After all the pre-steps are done, GPU takes over the computation task. In the GPU, images are divided into blocks of size 34x34, where 32 of them are the pixels not at the edge and 2 additional apron pixels at the edge to take care of convolution around edges. Blocks of the images are loaded into a shared memory for fast access. After the memory is loaded, blocks are assigned to different threads according to their thread indexes. Similarly, threads can read corresponding filter coefficients based on their thread indexes. Then, threads perform the convolution on the assigned block and output results to the allocated memory. Since each thread has its own block and all memory accesses are read-only, all the threads can compute in parallel. When a thread finishes, it will be assigned to a new block and new filter coefficients until the blocks run out. After the whole computation finishes, the allocated memory with convolution results is transferred back to CPU and saved to the hard drive.

Table 2: Runtime of different versions of the CPU and GPU implementations of 2D convolution with different number of filters.

	Runtime (ms)			
No. of filters	1	4	16	64
GPU	0.0145	0.0379	0.129	0.482
CPU (basic - no opt - 1 thread)	6.101	24.278	97.311	389.324
CPU (vectorized - no opt - 1 thread)	3.232	12.900	52.557	209.432
CPU (basic - O2 - 1 thread)	1.270	4.963	19.858	79.264
CPU (vectorized - O2 - 1 thread)	0.853	3.383	13.530	53.959
CPU (basic - O3 - 1 thread)	0.535	2.125	8.471	33.832
CPU (vectorized - O3 - 1 thread)	0.953	3.406	13.856	55.490
CPU (basic - O3 - 4 threads)	0.966	1.011	1.459	5.928
CPU (vectorized - O3 - 4 threads)	1.268	1.191	1.641	7.177

Table 2 above shows the runtime differences in milliseconds between CPU and GPU, also between the number of filters and optimization levels. One can easily notice the higher the optimization level the faster the runtimes. This is because g++ automatically applies different optimization methods based on the the compilation command. With O2 command enabled, g++ will apply techniques such as redundant instruction elimination, assign variables to registers, loop unrolling, software-level pipelining, control flow simplification and many more. This is also the level where the majority of optimization techniques are used, which means the speedup should be the best at this level. We can verify this point by checking the speedup from no-op to O2 and speedup from O2 to O3. At O3 level, g++ will apply more aggressive techniques to gain performance, but most of them come with trade-offs. The major one is trading floating-point accuracy for better computation speed. Although aggressive methods are used, the speedup is decreasing compare to O2 optimization.

By comparing between basic and vectorized versions, I found the vectorized runtimes are faster for no optimization and O1 optimization. However, this situation is reversed under high-level optimization. For O2 and O4 optimizations, vectorized runtimes are slower than basic ones. The reason could be g++ higher optimization level introduced smarter optimizations in the computation part (the inner two loops in the basic version) than the vectorized computation. But g++ doesn't know how to optimize the vector operation further. Therefore, the vectorized computation limits the g++ optimization and slows down the runtime.

By comparing a single thread and multiple threads, the single-threaded versions are actually faster when there is only one filter. But multi-threaded versions are faster given the number of filters is larger than 4. Based on the code, we can see the openMP pragma only unrolls the outer-most loop, which is looping based on the number of filters. When filter number is 1, there is nothing to unroll for the first loop. Therefore, the pragma only introduces more overhead instead multi-threading the code for filter number of 1. On the other hand, the outer-most loop can be unrolled for filters of 4, 16, and 64. As a result, we can see the speedup in runtimes.

Efficiency Comparison

Table 3: Comparison between the 3 compute platforms implementing 2D convolution with 64 filters.

	Throughput (GOPS)	Power (W)	Energy Efficiency (GOPS/W)	Area Efficiency (GOPS/mm^2)
FPGA (20nm)	833.536	3.424	243.556	2.084
CPU (14nm)	48.113	65	0.740	0.174
GPU (8nm)	591.728	220	2.690	1.506
FPGA (scaled to 8nm)	1333.658	2.462	541.697	13.337
CPU (scaled to 8nm)	60.141	54.6	1.101	0.436

Comparing GPU, FPGA and CPU with the same 8nm technology in table 3, FPGA wins CPU and GPU in all categories, especially in power, energy efficiency and area efficiency. FPGA's throughput is approximately 2X better than GPU and 20X better than CPU. FPGA's power consumption is about 90X and 20X better than GPU and CPU respectively. Due to the tremendous power differences, FPGA leads both GPU and CPU in energy efficiency about 200X and 500X. In terms of area efficiency, FPGA wins again due to the smallest die area of 100 mm^2 among all three chips. This gives FPGA 8X more area efficient than GPU and 30X more area efficient than CPU.

Since CPU is a general purpose processing unit with more focus on serial tasks and branching operations, it's reasonable CPU has much worse throughput and power comparing to both GPU and FPGA. Therefore, we should put more focus on GPU and FPGA comparison, since they both have advantages on parallel processing. One can observe the FPGA roughly has 2X throughput and 90X power differences comparing to GPU. Although the throughput difference seems acceptable, the power difference is way too unrealistic. This is due to the comparison is unfair and the assumption of TDP is unreasonable. First, the FPGA's design and code are some-what-optimized, whereas GPU's code is not optimized. It's unfair to make a comparison in throughput since both chips are not pushed to the limit. Second, we used Quartus to simulate a reasonable FPGA power consumption but assumed the GPU power consumption is 220 W without actual testing. The true GPU power consumption should be much lower than the TDP specifications for the 64-filter convolution task. So the actual power difference shouldn't be as large as the 90X difference. This factor hugely affect the both power and energy efficiency categories. Third, the comparison metric is more biased toward FPGA because only a maximum of 64 filters are tested. GPUs is designed for high throughput with large dataset, which means GPU should have better throughput with larger number of filters. A more detailed table extends to a higher number of filters should also be tested to show the true performance of GPU. With all that being said, FPGA should still have a higher throughput for 64 filters and much better power consumption than GPU, but the exact differences can not be shown in this experiment. The above terms also applicable to the CPU, but it should still be the worst one among the three chips.

Appendix A: HDL source codes

```
// 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 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
// ---------------- define parameters-------------------------------
localparam WIDTH = 512; // Input image width without padding
localparam WIDTH_PAD = 514; // Input image width with padding
localparam WIDTH_DATAW = 10; // 512 + padding, need 10 bits to store
localparam PIPELINE_STAGES = 10; // number of pipeline stages

integer i;

// ----------------initialize memory_3row module--------------------
// buffer the enough pixels to generate the first output
logic write_enable [2:0];
logic [WIDTH_DATAW-1:0] write_address [2:0];
logic [PIXEL_DATAW-1:0] data_in [2:0];
logic [WIDTH_DATAW-1:0] read_address;
logic [PIXEL_DATAW-1:0] data_out [2:0];

logic enable;
always_comb begin
    // signal for enable
    enable = i_ready;
```

```
end

memory_3row Memory3row (.CLK(clk), .ENA(enable), .write_ENA(write_enable), .
    write_address(write_address), .data_in(data_in), .read_address(read_address
    ), .data_out(data_out));

// ---------create logics to save pixel data into memory-------------------
// row_counter and col_counter indicate the current location in the memory
// data should be read from col_read
// idx is used to overwrite an old row with a new row
logic unsigned [1:0]row_counter;
logic unsigned [WIDTH_DATAW-1:0] col_counter;
logic unsigned [WIDTH_DATAW-1:0] col_read;
logic unsigned [2:0][1:0] idx;

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        for(i=0; i<3; i=i+1) begin
            write_enable[i] <= 0;
            write_address[i] <= 0;
            data_in[i] <= 0;
            idx[i] <= i;
        end
        row_counter <= 0;
        col_counter <= 0;
        col_read <= 0;
    end else if(enable) begin
        if(i_valid) begin
            if(col_counter != WIDTH_PAD) begin
                write_enable[idx[2]] <= 1;
                write_address[idx[2]] <= col_counter;
                data_in[idx[2]] <= i_x;

                col_read <= col_counter;
                col_counter <= col_counter + 1;
            end else begin
                write_enable[idx[0]] <= 1;
                write_address[idx[0]] <= 0;
                data_in[idx[0]] <= i_x;

                idx <= {idx[0], idx[2:1]};
                col_read <= 0;
                col_counter <= 1;

                // when row counter = 2 indicates three rows are filled
                if(row_counter < 2) begin
                    row_counter <= row_counter + 1;
                end
            end
        end
```

```
    end
end

// read from saved data
always_comb begin
    read_address = col_read;
end

// ------------pipeline the control signals-----------------
// need to match the memory and the dsp cycles, 10 pipeline stages are used
   here
logic unsigned [PIPELINE_STAGES-1:0][1:0] row_counter_p; // row counter
   pipelined
logic unsigned [PIPELINE_STAGES-1:0][WIDTH_DATAW-1:0] col_counter_p; // column
    counter pipelined
logic unsigned [2:0][2:0] [1:0] idx_p; // this parameter only needs to be
   pipelined to prepare data section

// pipelined the signals
always_ff @(posedge clk) begin
    if (enable) begin
        row_counter_p <= {row_counter_p[PIPELINE_STAGES-2:0], row_counter};
        col_counter_p <= {col_counter_p[PIPELINE_STAGES-2:0], col_counter};
        idx_p <= {idx_p[1:0], idx};
    end
end

//---------------prepare data-------------------------------
// read data from memory each cycle
// each cycle fill MAC_input_row012[0], next cycle MAC_input_row012[1] ...
logic signed [2:0][PIXEL_DATAW:0] MAC_input_row0;
logic signed [2:0][PIXEL_DATAW:0] MAC_input_row1;
logic signed [2:0][PIXEL_DATAW:0] MAC_input_row2;
logic [PIXEL_DATAW:0] data_from_memory [2:0];

always_ff @(posedge clk) begin
    if (enable) begin
        data_from_memory[0] <= {1'b0, data_out[0]};
        data_from_memory[1] <= {1'b0, data_out[1]};
        data_from_memory[2] <= {1'b0, data_out[2]};

        MAC_input_row0 <= {data_from_memory[idx_p[2][0]], MAC_input_row0
            [2:1]};
        MAC_input_row1 <= {data_from_memory[idx_p[2][1]], MAC_input_row1
            [2:1]};
        MAC_input_row2 <= {data_from_memory[idx_p[2][2]], MAC_input_row2
            [2:1]};
    end
end
```

```
//----------------perform multiply and add operations----------------------
// each DSP corresponds to the convolution of size 3 in a row. Three rows in
    total
// p0*c0 + p1*c1 + p2*c2 are reduced to (p0+p2)*c0 + p1*c1 due to symmtric
    filter
logic signed [15:0] sum_row0;
logic signed [15:0] sum_row1;
logic signed [15:0] sum_row2;

// row 0, first DSP
DSP_sum2 DSP_A (.CLK(clk), .ENA(enable), .pixel0(MAC_input_row0[0]), .pixel1(
    MAC_input_row0[1]), .pixel2(MAC_input_row0[2]),
            .coeff0(r_f[0][0]), .coeff1(r_f[0][1]), .result(sum_row0));

// row 1, second DSP
DSP_sum2 DSP_B (.CLK(clk), .ENA(enable), .pixel0(MAC_input_row1[0]), .pixel1(
    MAC_input_row1[1]), .pixel2(MAC_input_row1[2]),
            .coeff0(r_f[1][0]), .coeff1(r_f[1][1]), .result(sum_row1));

// row 2, third DSP
DSP_sum2 DSP_C (.CLK(clk), .ENA(enable), .pixel0(MAC_input_row2[0]), .pixel1(
    MAC_input_row2[1]), .pixel2(MAC_input_row2[2]),
            .coeff0(r_f[2][0]), .coeff1(r_f[2][1]), .result(sum_row2));

//----------------sum three row sums----------------------------
// sum the partial sums of three rows to get the final result
// this summation is done by LEs, not DSPs
logic signed [16:0] sum_row0_p0;
logic signed [16:0] sum_row1_p0;
logic signed [16:0] sum_row2_p0;

logic signed [16:0] sum_row01_p1;
logic signed [16:0] sum_row2_p1;

logic signed [16:0] sum_row012_p2;

logic signed [16:0] total_sum;
logic unsigned [PIXEL_DATAW-1:0] final_sum;

always_ff @ (posedge clk) begin
    if(enable) begin
        sum_row0_p0 <= sum_row0;
        sum_row1_p0 <= sum_row1;
        sum_row2_p0 <= sum_row2;

        total_sum <= sum_row012_p2;
    end
end

always_comb begin
```

```
    sum_row01_p1 <= sum_row0_p0 + sum_row1_p0 + sum_row2_p0;
end

// since the output type should be 8 bit unsigned,
// take care of the overflow and underflow
always_comb begin
    sum_row012_p2 <= sum_row01_p1;

    if(total_sum < 0) begin
        final_sum = 0;
    end else if(total_sum > 255) begin
        final_sum = 255;
    end else begin
        final_sum = total_sum[PIXEL_DATAW-1:0];
    end
end

// ------------------assign outputs-------------------
logic unsigned [PIXEL_DATAW-1:0] y_Q;
logic y_valid;

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        y_Q <= 0;
        y_valid <= 0;
        // prev_col_counter <= 0;
    end else if(enable) begin
        // results are valid when more than than 3 columns
        // and 3 rows are saved in the memory
        if(col_counter_p[PIPELINE_STAGES-1] >= 3 &&
        row_counter_p[PIPELINE_STAGES-1] == 2) begin
            y_Q <= final_sum;
            y_valid <= 1;
        end else begin
            y_Q <= 0;
            y_valid <= 0;
        end
    end
end

assign o_y = y_Q;
assign o_ready = i_ready;
assign o_valid = y_valid & i_ready;

endmodule

//-------------------------------------------------------
// RAM that saves three rows of input pixels, 514 * 3 total pixels
// Re-usebility is achieved by index manipulation - <idx> parameter
// each cycle three pixels in the same column is read out
```

```
module memory_3row(
    input CLK,
    input ENA,
    input write_ENA[2:0],
    input [9:0] write_address [2:0],
    input unsigned [7:0] data_in [2:0],
    input [9:0] read_address,
    output logic unsigned [7:0] data_out [2:0]
);

logic unsigned [7:0] memory_row0 [513:0];
logic unsigned [7:0] memory_row1 [513:0];
logic unsigned [7:0] memory_row2 [513:0];
logic [9:0] read_address_reg;

integer i;

always_ff @(posedge CLK) begin
    if(ENA) begin
        read_address_reg <= read_address;

        if(write_ENA[0]) begin
            memory_row0[write_address[0]] <= data_in[0];
        end
        if(write_ENA[1]) begin
            memory_row1[write_address[1]] <= data_in[1];
        end
        if(write_ENA[2]) begin
            memory_row2[write_address[2]] <= data_in[2];
        end

        data_out[0] <= memory_row0[read_address_reg];
        data_out[1] <= memory_row1[read_address_reg];
        data_out[2] <= memory_row2[read_address_reg];
    end
end
endmodule

//-----------------------------------------------------------------------------
// multiply and accumulate 3 pixels with 3 filter coefficients (row-wise)
// since filter is symmetric in the x-direction,
// we can omit the third coefficient to skip a multiplication
// This allows the one-row computation implemented with one DSP block
// pixel0*coeff0 + pixel1*coeff1 + pixel2*coeff2 = (pixel0+pixel2)*coeff0 +
    pixel1*coeff1
// registers are added to match DSP structure
// Quartus will map this module to DSP 18x18 sum of 2 mode automatically
// each module consumes two 18x19 DSP block or one 27x27 DSP block
module DSP_sum2 (
    input CLK,
```

```
    input ENA,
    input signed [8:0] pixel0,
    input signed [8:0] pixel1,
    input signed [8:0] pixel2,
    input signed [7:0] coeff0,
    input signed [7:0] coeff1,
    output logic signed [15:0] result
) /* synthesis multstyle = "dsp" */;

logic signed [8:0] pixel0_p0, pixel1_p0, pixel2_p0;
// NOTE: use 10 bit here, in case overflow in addition!!!
logic signed [9:0] pixel02_p1, pixel02_p2;
logic signed [8:0] pixel1_p1, pixel1_p2;
logic signed [7:0] coeff0_p0, coeff1_p0;
logic signed [7:0] coeff0_p1, coeff1_p1;
logic signed [7:0] coeff0_p2, coeff1_p2;

always_ff @(posedge CLK) begin
    if(ENA) begin
        pixel0_p0 <= pixel0;
        pixel1_p0 <= pixel1;
        pixel2_p0 <= pixel2;
        coeff0_p0 <= coeff0;
        coeff1_p0 <= coeff1;

        pixel02_p1 <= pixel0_p0 + pixel2_p0;
        pixel1_p1 <= pixel1_p0;
        coeff0_p1 <= coeff0_p0;
        coeff1_p1 <= coeff1_p0;

        pixel02_p2 <= pixel02_p1;
        pixel1_p2 <= pixel1_p1;
        coeff0_p2 <= coeff0_p1;
        coeff1_p2 <= coeff1_p1;

        result <= pixel02_p2*coeff0_p2 + pixel1_p2*coeff1_p2;
    end
end
endmodule
//-----------------------------------------------------------------
```