Lichen Liu
1001721498

Assignment 2

ECE 1756H
November 1, 2022

1 FPGA Implementation

1.1 Design Description

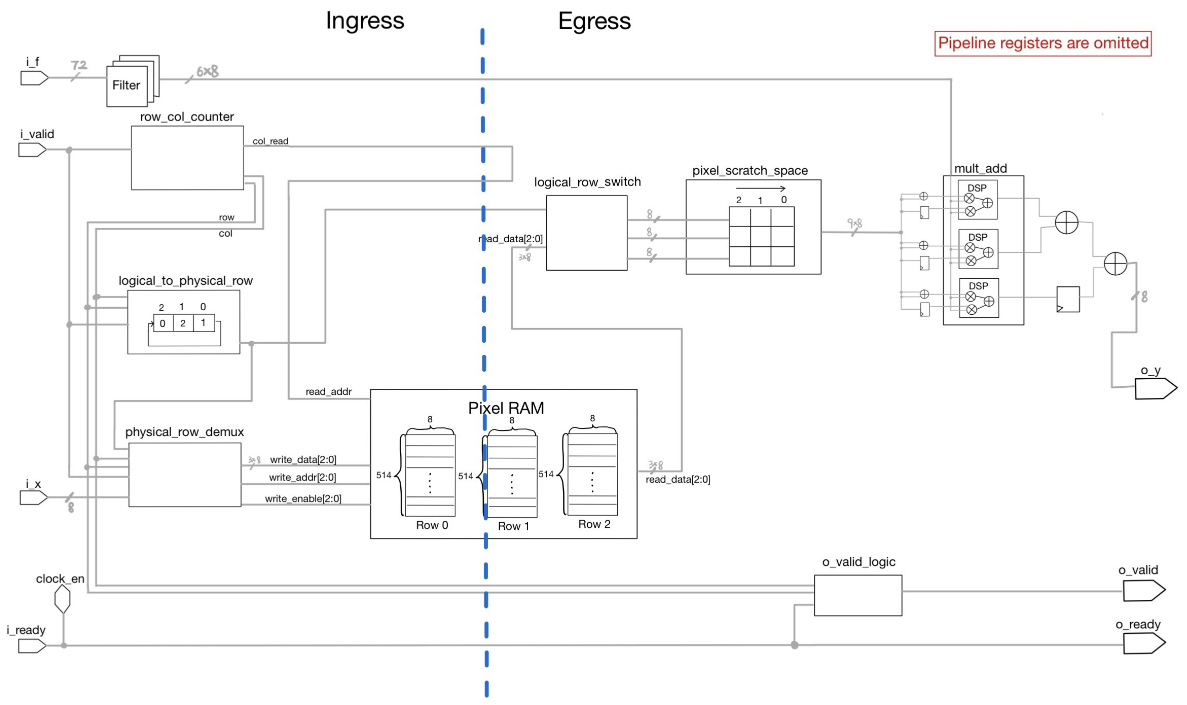
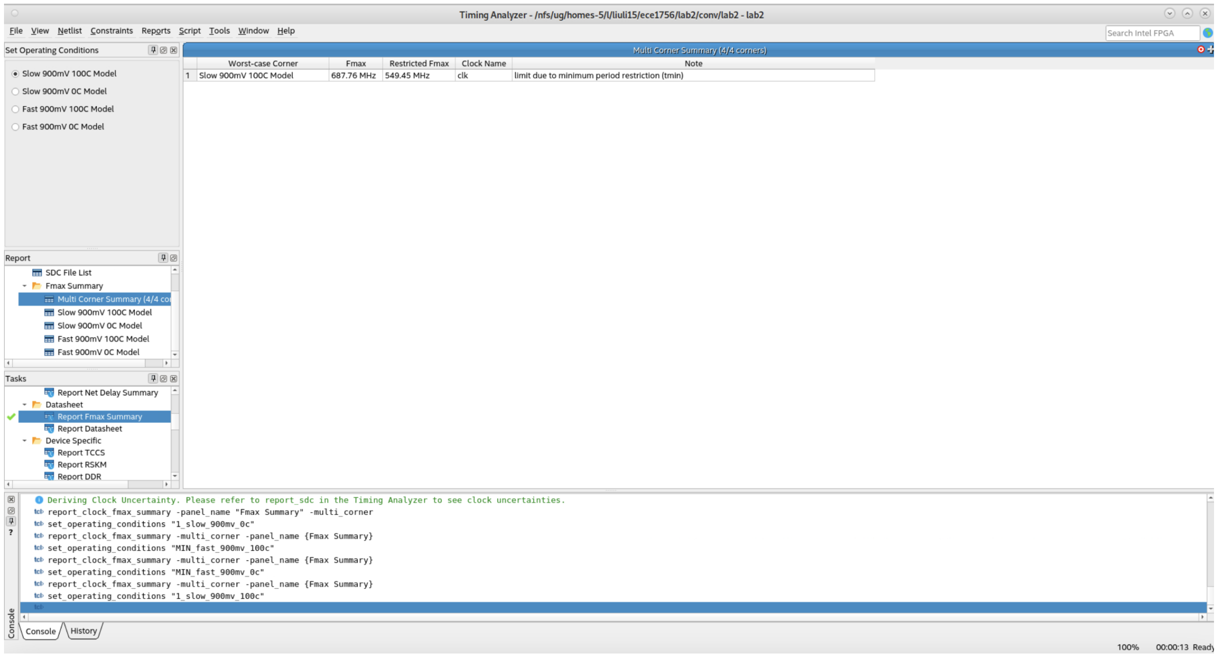


Figure 1: A simplified block diagram of the design, which runs at Restricted Fmax 549.45 MHz, Fmax 687.76 MHz, with a throughput of 1 output pixel per cycle

The design can be logically separated into two main stages, ingress and egress, with respect to the RAM that stores the input image pixels (pixel RAM).

Ingress path is responsible for keeping track of the column and row value of the input pixel, which is used as both the write address to the pixel RAM, and for determining the current convolution window (i.e., pointing to the bottom right corner of the convolution window). Another important responsibility for ingress path is to control the row shifting logic for the pixel RAM.

Pixel RAM is a 2D array of pixels, with a dimension of $3 \times (512 + 2) \times 8$. Only three rows are stored because that is the size of the convolution filter. Write is always done at the bottom row, and the row shifts upward when column counter reaches the bound. To save area, this shifting register file is implemented in BRAM, where the shifting is realized by shifting the mapping from logical row index to physical row index circularly, so that after the shifting, new incoming pixels are essentially overwriting the oldest row. Pixel RAM is implemented with three independent BRAM (M20K) blocks, each representing a row, with a shape of $(512 + 2) \times 8$. For ease of testing and maximal compatibility, the BRAM block

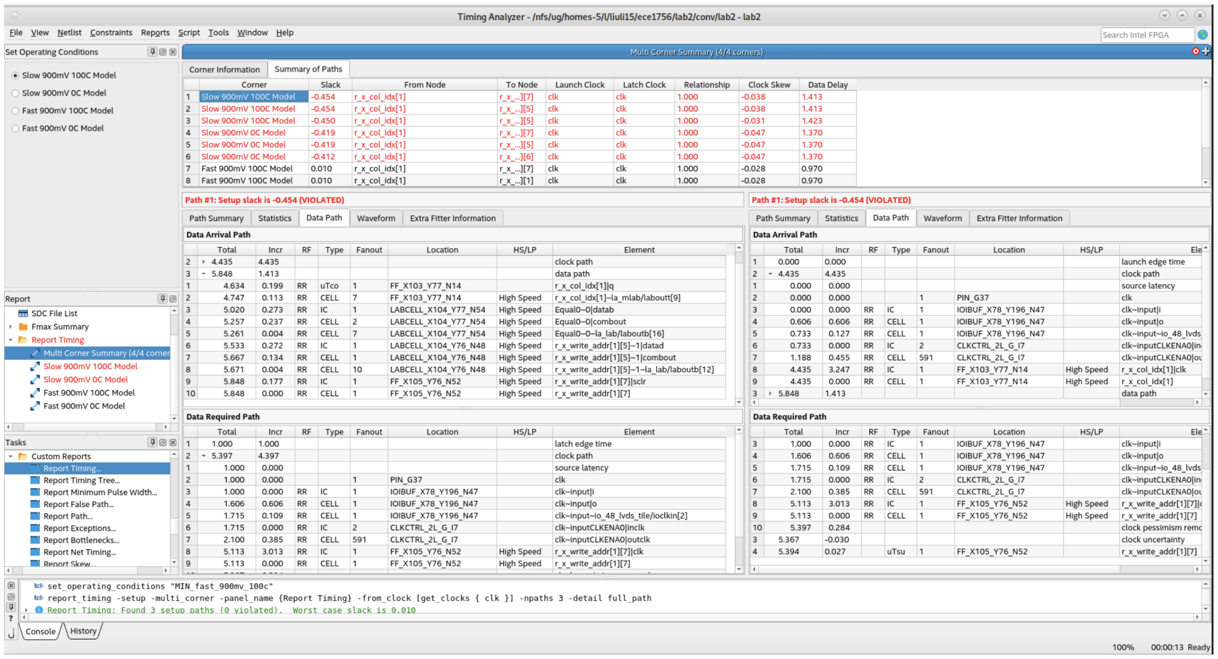


Multi Corner Summary (4/4 corners)

	Worst-case Corner	Fmax	Restricted Fmax	Clock Name	Note
1	Slow 900mV 100C Model	687.76 MHz	549.45 MHz	clk	limit due to minimum period restriction (tmin)

```
Deriving Clock Uncertainty. Please refer to report_sdc in the Timing Analyzer to see clock uncertainties.
tcl report_clock_fmax_summary -panel_name "Fmax Summary" -multi_corner
tcl set_operating_conditions "1_slow_900mv_0c"
tcl report_clock_fmax_summary -multi_corner -panel_name {Fmax Summary}
tcl set_operating_conditions "MIN_fast_900mv_100c"
tcl report_clock_fmax_summary -multi_corner -panel_name {Fmax Summary}
tcl set_operating_conditions "MIN_fast_900mv_0c"
tcl report_clock_fmax_summary -multi_corner -panel_name {Fmax Summary}
tcl set_operating_conditions "1_slow_900mv_100c"
```

Figure 2: Fmax Summary



Multi Corner Summary (4/4 corners)

	Corner	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	Slow 900mV 100C Model	-0.454	r_x_col_idx[1]	r_x_...][7]	clk	clk	1.000	-0.038	1.413
2	Slow 900mV 100C Model	-0.454	r_x_col_idx[1]	r_x_...][5]	clk	clk	1.000	-0.038	1.413
3	Slow 900mV 100C Model	-0.450	r_x_col_idx[1]	r_x_...][5]	clk	clk	1.000	-0.031	1.423
4	Slow 900mV 0C Model	-0.419	r_x_col_idx[1]	r_x_...][7]	clk	clk	1.000	-0.047	1.370
5	Slow 900mV 0C Model	-0.419	r_x_col_idx[1]	r_x_...][5]	clk	clk	1.000	-0.047	1.370
6	Slow 900mV 0C Model	-0.412	r_x_col_idx[1]	r_x_...][6]	clk	clk	1.000	-0.047	1.370
7	Fast 900mV 100C Model	0.010	r_x_col_idx[1]	r_x_...][7]	clk	clk	1.000	-0.028	0.970
8	Fast 900mV 100C Model	0.010	r_x_col_idx[1]	r_x_...][1]	clk	clk	1.000	-0.028	0.970

Path #1: Setup slack is -0.454 (VIOLATED)

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	HS/LP	Element
2	4.435	4.435						clock path
3	5.848	1.413						data path
1	4.634	0.199	RR	uTco	1	FF_X103_Y77_N14		r_x_col_idx[1]|q
2	4.747	0.113	RR	CELL	7	FF_X103_Y77_N14	High Speed	r_x_col_idx[1]~la_mlab/laboutt[9]
3	5.020	0.273	RR	IC	1	LABCELL_X104_Y77_N54	High Speed	Equal0~0|datab
4	5.257	0.237	RR	CELL	2	LABCELL_X104_Y77_N54	High Speed	Equal0~0|combout
5	5.261	0.004	RR	CELL	7	LABCELL_X104_Y77_N54	High Speed	Equal0~0~la_lab/laboutb[16]
6	5.533	0.272	RR	IC	1	LABCELL_X104_Y76_N48	High Speed	r_x_write_addr[1][5]~1|datad
7	5.667	0.134	RR	CELL	1	LABCELL_X104_Y76_N48	High Speed	r_x_write_addr[1][5]~1|combout
8	5.671	0.004	RR	CELL	10	LABCELL_X104_Y76_N48	High Speed	r_x_write_addr[1][5]~1~la_lab/laboutb[12]
9	5.848	0.177	RR	IC	1	FF_X105_Y76_N52	High Speed	r_x_write_addr[1][7]|sclr
10	5.848	0.000	RR	CELL	1	FF_X105_Y76_N52	High Speed	r_x_write_addr[1][7]

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	HS/LP	Element
1	1.000	1.000						latch edge time
2	5.397	4.397						clock path
1	1.000	0.000						source latency
2	1.000	0.000			1	PIN_G37		clk
3	1.000	0.000	RR	IC	1	IOIBUF_X78_Y196_N47		clk~input|i
4	1.606	0.606	RR	CELL	1	IOIBUF_X78_Y196_N47		clk~input|o
5	1.715	0.109	RR	CELL	1	IOIBUF_X78_Y196_N47		clk~input~io_48_lvds_tile/ioclkin[2]
6	1.715	0.000	RR	IC	2	CLKCTRL_2L_G_I7		clk~inputCLKENA0|inclk
7	2.100	0.385	RR	CELL	591	CLKCTRL_2L_G_I7		clk~inputCLKENA0|outclk
8	5.113	3.013	RR	IC	1	FF_X105_Y76_N52	High Speed	r_x_write_addr[1][7]|clk
9	5.113	0.000	RR	CELL	1	FF_X105_Y76_N52	High Speed	r_x_write_addr[1][7]

Path #1: Setup slack is -0.454 (VIOLATED)

Data Arrival Path

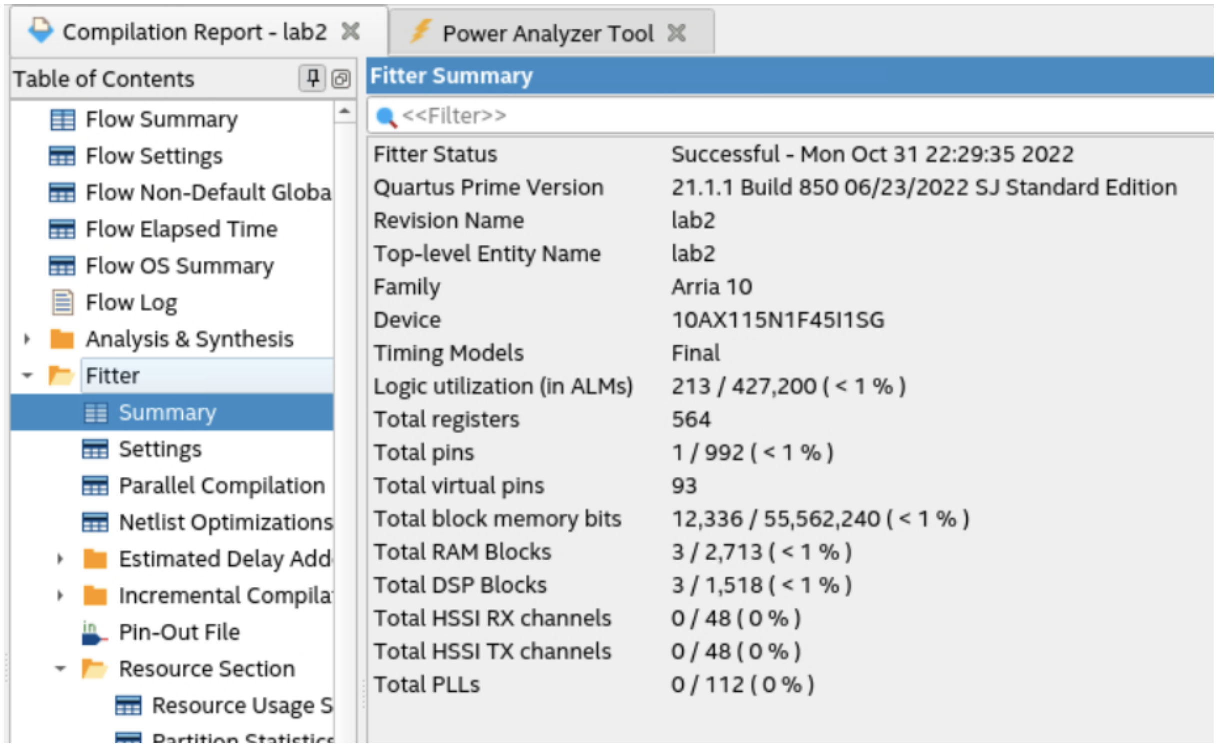
	Total	Incr	RF	Type	Fanout	Location	HS/LP	Ele
1	0.000	0.000						launch edge time
2	4.435	4.435						clock path
1	0.000	0.000						source latency
2	0.000	0.000			1	PIN_G37		clk
3	0.000	0.000	RR	IC	1	IOIBUF_X78_Y196_N47		clk~input|i
4	0.606	0.606	RR	CELL	1	IOIBUF_X78_Y196_N47		clk~input|o
5	0.733	0.127	RR	CELL	1	IOIBUF_X78_Y196_N47		clk~input~io_48_lvds
6	0.733	0.000	RR	IC	2	CLKCTRL_2L_G_I7		clk~inputCLKENA0|in
7	1.188	0.455	RR	CELL	591	CLKCTRL_2L_G_I7		clk~inputCLKENA0|ou
8	4.435	3.247	RR	IC	1	FF_X103_Y77_N14	High Speed	r_x_col_idx[1]|clk
9	4.435	0.000	RR	CELL	1	FF_X103_Y77_N14	High Speed	r_x_col_idx[1]
3	5.848	1.413						data path

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	HS/LP	Ele
3	1.000	0.000	RR	IC	1	IOIBUF_X78_Y196_N47		clk~input|i
4	1.606	0.606	RR	CELL	1	IOIBUF_X78_Y196_N47		clk~input|o
5	1.715	0.109	RR	CELL	1	IOIBUF_X78_Y196_N47		clk~input~io_48_lvds
6	1.715	0.000	RR	IC	2	CLKCTRL_2L_G_I7		clk~inputCLKENA0|in
7	2.100	0.385	RR	CELL	591	CLKCTRL_2L_G_I7		clk~inputCLKENA0|ou
8	5.113	3.013	RR	IC	1	FF_X105_Y76_N52	High Speed	r_x_write_addr[1][7]|
9	5.113	0.000	RR	CELL	1	FF_X105_Y76_N52	High Speed	r_x_write_addr[1][7]
10	5.397	0.284						clock pessimism remo
3	5.367	-0.030						clock uncertainty
4	5.394	0.027		uTsu	1	FF_X105_Y76_N52		r_x_write_addr[1][7]

```
tcl set_operating_conditions "MIN_fast_900mv_100c"
tcl report_timing -setup -multi_corner -panel_name {Report Timing} -from_clock [get_clocks { clk }] -npaths 3 -detail full_path
Report Timing: Found 3 setup paths (0 violated).  Worst case slack is 0.010
```

Figure 3: Critical Paths



Fitter Status	Successful - Mon Oct 31 22:29:35 2022
Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Standard Edition
Revision Name	lab2
Top-level Entity Name	lab2
Family	Arria 10
Device	10AX115N1F45I1SG
Timing Models	Final
Logic utilization (in ALMs)	213 / 427,200 (< 1 %)
Total registers	564
Total pins	1 / 992 (< 1 %)
Total virtual pins	93
Total block memory bits	12,336 / 55,562,240 (< 1 %)
Total RAM Blocks	3 / 2,713 (< 1 %)
Total DSP Blocks	3 / 1,518 (< 1 %)
Total HSSI RX channels	0 / 48 (0 %)
Total HSSI TX channels	0 / 48 (0 %)
Total PLLs	0 / 112 (0 %)

Figure 4: Fitter Summary

is not instantiated from IP, but rather using an HDL template that can be inferred into BRAM (M20K) in simple dual-port mode by Quartus Synthesis, with read and write width of 8. With three independent BRAM (M20K) blocks, the pixel RAM is capable of doing a simultaneous read and write on each row in any clock cycle.

Egress path is responsible for performing the actual convolution computation for the convolution window that is pointed by the row and column counter aforementioned. For every window, it reads a single pixel for each row from the pixel RAM, and stores it into a scratch space that is 3×3 in size. The scratch space would shift horizontally to represent the fact that the convolution windows is also moving horizontally. So at any convolution window, only the right-most column is freshly read from the pixel RAM, while the other columns are cached from previous convolution windows. This effectively reduces the number of simultaneous read demand from 9 in a naive implementation, down to 3. To better exploit the fact that the filter is symmetric in x-direction, pixels at column 0 and column 2 are added before multiplying the filter, which effectively saves 3 multipliers in total. The total of 6 multipliers are divided into 3 sum of 2 18×18 multipliers groups, where each group can nicely fit into a DSP block on chip. Again, an HDL template that can be inferred to a fully registered DSP is used rather than instantiating from IP. Afterwards, a 2-level reduction (sum) tree is used to accumulate from the 3 DSP outputs. In total, 8 adders and 6 multipliers are used saving 3 multipliers from a naive implementation of 8 adders and 9 multipliers. A lot of pipeline stages are inserted to this arithmetic chain to boost Fmax. The entire arithmetic logic runs continuously, but the output is valid only if the convolution window pointer is at valid location and has not been computed before.

1.2 FPGA Implementation Results

	Result
ALM Utilization	166 out of 427200
DSP Utilization	3 out of 1518
BRAM (M20K) Utilization	3 out of 2713
Maximum Operating Frequency (MHz)	Restricted Fmax 549.45 MHz, Fmax 687.76 MHz
Cycles for Test 7a (Hinton)	264222
Dynamic Power for one module @ maximum frequency (W)	0.048 W
Throughput of one module (GOPS)	9.267 GOPS
Throughput of full device (GOPS)	4689.198 GOPS
Total Power for full device (W)	10.188 W

Calculations

Dynamic Power for one module @ maximum frequency (W): $(DSP+M20K+LUT+FF+ClockNetwork)\div 50MHz\times RestrictedFmax = (0.08mW+0.45mW+0.20mW+0.79mW+2.82mW)\div 50MHz\times 549.45MHz$

Throughput of one module (GOPS): $512\times 512\times(9+8)\div(264222\div 549.45MHz)$

Throughput of full device (GOPS): $506\times 9.267GOPS$

Total Power for full device (W): $(DSP+M20K+LUT+FF)\div 50MHz\times RestrictedFmax\times 506+ClockNetwork\div 50MHz\times RestrictedFmax+DeviceStatic(1704.56mW)+IO(0.19mW)$

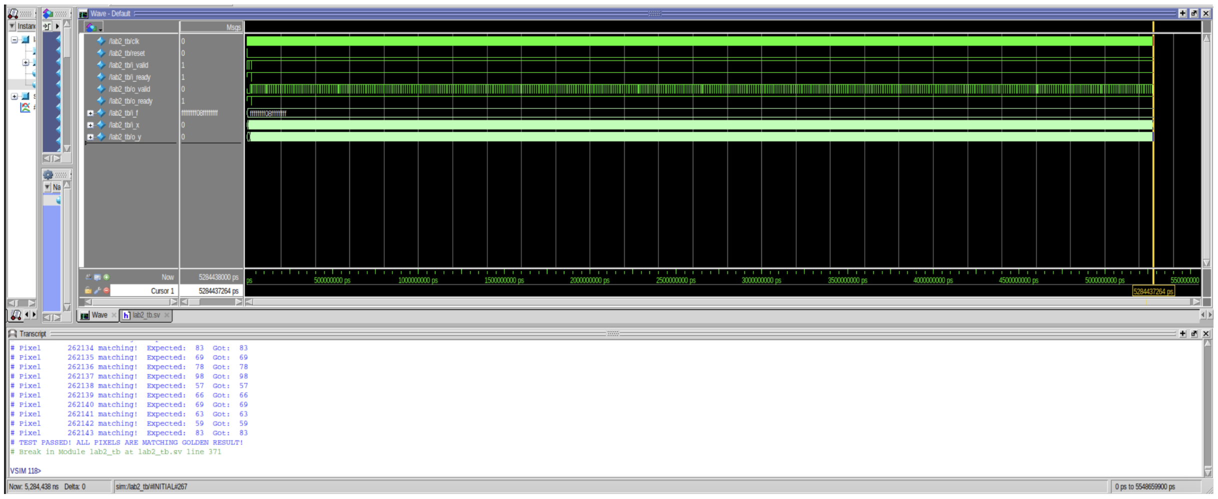
1.3 Simulation Waveforms and Testbench Output

Refer to Figure 5 and Figure 6.

2 Efficiency Comparison

2.1 CPU Implementation

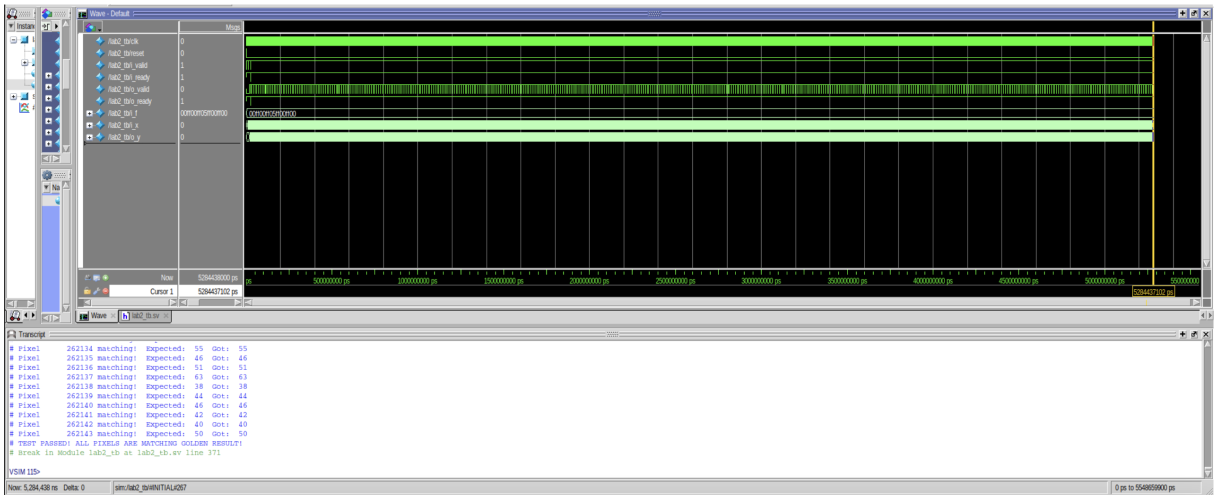
CPU convolution implementations are quite straight forward. Both single-threaded and multithreaded, and both basic and hand-vectorized versions loop for filter_id, output pixel



```
# 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 118>
```

Figure 5: ModelSim waveform and simulation log for Test7a as expected



```
# Pixel      262134 matching!  Expected:  55  Got:  55
# Pixel      262135 matching!  Expected:  46  Got:  46
# Pixel      262136 matching!  Expected:  51  Got:  51
# Pixel      262137 matching!  Expected:  63  Got:  63
# Pixel      262138 matching!  Expected:  38  Got:  38
# Pixel      262139 matching!  Expected:  44  Got:  44
# Pixel      262140 matching!  Expected:  46  Got:  46
# Pixel      262141 matching!  Expected:  42  Got:  42
# Pixel      262142 matching!  Expected:  40  Got:  40
# Pixel      262143 matching!  Expected:  50  Got:  50
# TEST PASSED! ALL PIXELS ARE MATCHING GOLDEN RESULT!
# Break in Module lab2_tb at lab2_tb.sv line 371

VSIM 115>
```

Figure 6: ModelSim waveform and simulation log for Test7b as expected

row and column respectively. Then a dot product is computed for the 3×3 convolution window involved. The basic implementation traverses through all 9 elements and accumulates the products, whereas the hand-vectorized version computes the dot product of the entire row in one shot (load, dot product and store) and then accumulate the the results of three rows together. Multi-threading is done using OpenMP directives, and is applied in the outermost filter_id dimension.

2.2 GPU Implementation

GPU convolution kernel executes in two stages. Firstly, each thread loads a few pixels of the current tile into shared memory. Then after all threads within the current block are done, each thread starts to calculate the dot product for the output pixel and filter_id assigned to it. The filters are stored in the GPU constant memory.

2.3 Runtime Results of CPU and GPU Implementations

	Runtime (ms)			
No. of filters	**1**	**4**	**16**	**64**
GPU	0.0151	0.0401	0.1374	0.5060
CPU (basic - no opt - 1 thread)	6.1022	24.306	97.235	389.02
CPU (vectorized - no opt - 1 thread)	3.1988	13.027	51.896	207.30
CPU (basic - O2 - 1 thread)	1.2414	4.9722	19.807	79.261
CPU (vectorized - O2 - 1 thread)	0.8441	3.3751	13.493	54.013
CPU (basic - O3 - 1 thread)	0.5334	2.1221	8.4699	33.843
CPU (vectorized - O3 - 1 thread)	0.8823	3.4746	13.898	55.689
CPU (basic - O3 - 4 threads)	0.5770	0.6108	1.2119	4.8529
CPU (vectorized - O3 - 4 threads)	0.9082	1.0459	1.8682	7.7334

When there is no compiler optimization, the vectorized version has a 2x performance than the basic version. When the compiler uses O2 optimization, the gap becomes smaller but still in favor of the vectorized version. Both versions have significant improvements comparing to the no opt version, but basic version gains more out of it suggesting that O2 optimizes both the dot product and iteration overheads for the basic version whereas the vectorized version only gains improvement from iteration overheads. This is even more obvious under O3 optimization, where the basic version has much better performance than the vectorized version and the performance of the vectorized version is the same as under O2 optimization.

After turning on multi-threading, comparing to the single-threaded result (with O3 optimization), the relative performance gap between the basic version and vectorized version stays the same. It is worth pointing out the speed up comparing to single-threaded version is dependent on the number of filters, there is no speed up when there is only 1 filter, and about 3.5x speed up for 4 filters. The speed up caps at around 8x for both 16 and 64 filters primarily due to the fact that the CPU under test has 8 cores.

2.4 Comparison of Results Between the 3 Compute Platforms

	Throughput (GOPS)	**Power (W)**	**Energy Efficiency (GOPS/W)**	**Area Efficiency (GOPS/mm^2)**
FPGA (20 nm)	4689.2 GOPS	10.188 W	460.27 GOPS/W	11.723 GOPS/mm^2
CPU (14 nm)	58.771 GOPS	65 W	0.904 GOPS/W	0.213 GOPS/mm^2
GPU (8 nm)	563.63 GOPS	220 W	2.562 GOPS/W	1.434 GOPS/mm^2
FPGA (scaled to 8nm)	7502.7 GOPS	6.875 W	1091.3 GOPS/W	75.027 GOPS/mm^2
CPU (scaled to 8nm)	73.464 GOPS	56.875 W	1.292 GOPS/W	0.532 GOPS/mm^2

Calculations

Scaled Throughput (GOPS): $OriginalThroughput \times ClockSpeedScaling$

Scaled Power: $OriginalPower \times ClockSpeedScaling \times Power@sameclockrateScaling$

Scaled Area: $OriginalArea \times AreaScaling$

When scaling to the same process technology, throughput wise, FPGA is one order ahead of GPU due primarily to that FPGA is programmed and dedicated to compute the convolution. One advantage with the FPGA is that it outputs a pixel at each clock cycle, where as GPU takes several. The GPU version is also a magnitude ahead of CPU, primarily due to the massive parallelism that convolution operation exhibits and that GPU can better exploit it.

Energy efficiency wise, FPGA is almost 1000x ahead of CPU and 400x ahead of GPU. This time, CPU and GPU are within the same magnitude due to them both being a general purpose computing platform, whereas the programmed FPGA is more application specific thus avoiding a lot of the overhead cost associated with generality. GPU is still ahead of CPU because of its wider SIMD, reducing overhead cost of instruction fetching and decoding.

Area efficiency wise, FPGA is almost 150x ahead of CPU and 50x ahead of GPU. The reason is again related to the overhead cost with generality, and overhead cost of instruction fetching and decoding between CPU and GPU.

3 Appendix

3.1 HDL

```
// 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)

integer col, row, i; // variables to use in the for loop
```

```

// 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
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
logic enable;
assign enable = i_ready;

// **********************
// INGRESS
// **********************
// If pipelined, need to pipeline i_valid

// Logics for buffer of x
localparam IMAGE_WIDTH = 512;
localparam R_X_ROWS = FILTER_SIZE; // Always store 3 rows of i_x
localparam R_X_COL_WIDTH = IMAGE_WIDTH + 2;
localparam R_X_COL_ADDR_WIDTH = 10;


// Pixel RAM
// RAM input, need to be registered except for r_x_read_addr
logic [R_X_COL_ADDR_WIDTH-1:0] r_x_write_addr [R_X_ROWS-1:0]; // 0..511+2
↪  (10 bit)
logic r_x_write_enable [R_X_ROWS-1:0];
logic unsigned [PIXEL_DATAW-1:0] r_x_write_data [R_X_ROWS-1:0];
// Registered inside module, read a pixel from 3 rows
logic [R_X_COL_ADDR_WIDTH-1:0] r_x_read_addr;
// RAM output
logic unsigned [PIXEL_DATAW-1:0] r_x_read_data [R_X_ROWS-1:0]; // is
↪  registered inside module, 3 rows of 1 pixel
pixelram pixel_ram
(
        .clk(clk),
        .reset(reset),
        .enable(enable),
        // RAM input, unregistered inside module
        .i_write_addr(r_x_write_addr),
        .i_write_enable(r_x_write_enable),
        .i_write_data(r_x_write_data),
        .i_read_addr(r_x_read_addr),
        // RAM output
        .o_read_data(r_x_read_data)
);
```

```


// Registers
// 0:  []  []  []  []  []  []  []  []                              512 + 2
// 1:  []  []  []  []  []  []  []  []                              512 + 2
// 2:  []  []  []  []  []  []  []  []                              512 + 2
logic unsigned [1:0] r_x_row_logical_idx; // Count from 0 to R_X_ROWS - 1
↪   (incl), logical order, not necessarily physical
logic unsigned [R_X_COL_ADDR_WIDTH-1:0] r_x_col_idx; // Count from 0 to
↪   R_X_COL_WIDTH (incl)
logic unsigned [R_X_COL_ADDR_WIDTH-1:0] r_x_col_idx_read_addr_adjusted; //
↪   Count from 0 to R_X_COL_WIDTH (incl)
// Count from 0 to R_X_ROWS - 1 (incl), physical order
logic unsigned [R_X_ROWS-1:0][1:0] r_x_row_logical_to_physical_index;

// INGRESS: Stage -1
always_ff @ (posedge clk) begin
        if(reset) begin
                r_x_row_logical_idx <= 0;
                r_x_col_idx <= 0;
                r_x_col_idx_read_addr_adjusted <= 0;

                for(i = 0; i < R_X_ROWS; i = i + 1) begin
                        r_x_row_logical_to_physical_index[i] <= i;
                end

                for(row = 0; row < R_X_ROWS; row = row + 1) begin
                        r_x_write_addr[row] <= 0;
                        r_x_write_data[row] <= 0;
                        r_x_write_enable[row] <= 0;
                end
        end else if (enable) begin
                // Do not write by default
                for(row = 0; row < R_X_ROWS; row = row + 1) begin
                        r_x_write_addr[row] <= 0;
                        r_x_write_data[row] <= 0;
                        r_x_write_enable[row] <= 0;
                end
                r_x_col_idx_read_addr_adjusted <= 0;
                 if(i_valid) begin
                        if(r_x_col_idx == R_X_COL_WIDTH) begin
```

```verilog
                            // Load input pixel to a new row at the current logical idx 0 (R_X_COL_WIDTH implies 0),
                            // which would be discarded, then reused/overwritten as the new logical idx 2 in the next cycle
                            r_x_write_addr[r_x_row_logical_to_physical_index[0]] <= 0;
                            r_x_write_data[r_x_row_logical_to_physical_index[0]] <= i_x;
                            r_x_write_enable[r_x_row_logical_to_physical_index[0]] <= 1;

                            // Do the row shifting logic at the first input of the new row,
                            // rather than at the last input of the old row (will have conflict
                            // in writing old row and shifting old row)

                            // Instead of shifting the actual data, shift the mapping from logical index to physical index
                            // Shift the mapping, upward (idx[0]->idx[2])
                            r_x_row_logical_to_physical_index <= {r_x_row_logical_to_physical_index[0], r_x_row_logical_to_physical_index[R_X_ROWS-1:1]};

                            // Reset r_x_col_idx if necessary, continuing at idx 1.
                            // Skipping idx 0 because we are at idx 0 currently
                            r_x_col_idx <= 1;
                            r_x_col_idx_read_addr_adjusted <= 0;

                            // Increment r_x_row_logical_idx_ipipelined only when r_x_row_logical_idx_ipipelined is 0 or 1,
                            // so that r_x_row_logical_idx_ipipelined will reach to 2 in steady state
                            if(r_x_row_logical_idx < R_X_ROWS - 1) begin
                                    r_x_row_logical_idx <= r_x_row_logical_idx + 1;
                            end
                    end else begin
                            // Load data at logical idx 2
```

```
                                r_x_write_addr[r_x_row_logical_to_physical_index[R_X_ROW
↪   <= r_x_col_idx;
                                r_x_write_data[r_x_row_logical_to_physical_index[R_X_ROW
↪   <= i_x;
                                r_x_write_enable[r_x_row_logical_to_physical_index[R_X_R
↪   <= 1;

                                // Increment r_x_col_idx
                                r_x_col_idx <= r_x_col_idx + 1;

                                // Increment r_x_col_idx_read_addr_adjusted
                                r_x_col_idx_read_addr_adjusted <=
↪   r_x_col_idx;
                        end
                end
        end
end

// Pipeline registers for ingress
logic unsigned [1:0] r_x_row_logical_idx_ipipelined;
logic unsigned [R_X_COL_ADDR_WIDTH-1:0] r_x_col_idx_ipipelined;
logic unsigned [R_X_ROWS-1:0][1:0]
↪   r_x_row_logical_to_physical_index_ipipelined;
// INGRESS: Stage 0, to match registered RAM write
always_ff @ (posedge clk) begin
        if (enable) begin
                r_x_row_logical_idx_ipipelined <= r_x_row_logical_idx;
                r_x_col_idx_ipipelined <= r_x_col_idx;
                r_x_row_logical_to_physical_index_ipipelined <=
↪   r_x_row_logical_to_physical_index;
        end
end

// **********************
// EGRESS
// **********************

// Pipeline registers for egress
localparam NUM_EGRESS_STAGE = 9;
logic unsigned [NUM_EGRESS_STAGE-1:0] [R_X_COL_ADDR_WIDTH-1:0]
↪   r_x_col_idx_epipelined;
logic unsigned [NUM_EGRESS_STAGE-1:0] [1:0] r_x_row_logical_idx_epipelined;
logic unsigned [1:0] [R_X_ROWS-1:0][1:0] r_x_row_logical_to_physical_index_epipelined; // Not needed for full pipeline stage
always_ff @ (posedge clk) begin
        if(enable) begin
                r_x_col_idx_epipelined <= {r_x_col_idx_epipelined[NUM_EGRESS_STAGE-2:0], r_x_col_idx_ipipelined};
                r_x_row_logical_idx_epipelined <= {r_x_row_logical_idx_epipelined[NUM_EGRESS_STAGE-2:0], r_x_row_logical_idx_ipipelined};
                r_x_row_logical_to_physical_index_epipelined <= {r_x_row_logical_to_physical_index_epipelined[0:0], r_x_row_logical_to_physical_index_ipipelined};
        end
end

// Logics for convolution core
// Computation

// EGRESS: Stage -1
always_comb begin
        r_x_read_addr = r_x_col_idx_read_addr_adjusted;
end

// EGRESS: Stage 0, 1
// Signed x unsigned gets unsigned, which is not what we intend.
// So convert unsigned to signed by treating unsigned number as positive (by adding a 0 to msb)
logic signed [FILTER_SIZE-1:0] [PIXEL_DATAW:0] r_mult_i_pixel [R_X_ROWS-1:0];
logic [PIXEL_DATAW:0] r_x_read_data_reg [R_X_ROWS-1:0];
always_ff @ (posedge clk) begin
        if(enable) begin
                for(row=0; row<R_X_ROWS; row=row+1) begin
                        r_x_read_data_reg[row] <= r_x_read_data[row];
                        r_mult_i_pixel[row] <= {{1'b0, r_x_read_data_reg[r_x_row_logical_to_physical_index_epipelined[1][row]]}, r_mult_i_pixel[row][FILTER_SIZE-1:1]};
                end
        end
end

```

```
// Multiplication
// EGRESS: Stage 2, 3, 4, 5
logic signed [FILTER_SIZE-1:0] [2*PIXEL_DATAW-1:0] sums_stage_0;
genvar gen_row;
generate
        for(gen_row=0; gen_row<FILTER_SIZE; gen_row=gen_row+1) begin: mult
                mult8x8p8x8 m0(
                        .clk(clk),
                        .enable(enable),
                        .i_filtera(r_f[gen_row][0]),
                        .i_pixela0(r_mult_i_pixel[gen_row][0]),
                        .i_pixela1(r_mult_i_pixel[gen_row][2]),
                        .i_filterb(r_f[gen_row][1]),
                        .i_pixelb(r_mult_i_pixel[gen_row][1]),
                        .o_res(sums_stage_0[gen_row])
                );
        end
endgenerate

// Reduction tree
// EGRESS: Stage 6
logic signed [FILTER_SIZE-1:0] [2*PIXEL_DATAW-1:0] sums_stage_0_reg;
always_ff @ (posedge clk) begin
        if(enable) begin
                sums_stage_0_reg <= sums_stage_0;
        end
end

logic signed [2*PIXEL_DATAW-1:0] sums_stage_1 [1:0];
always_comb begin
        sums_stage_1[0] = sums_stage_0_reg[0] + sums_stage_0_reg[1];
        sums_stage_1[1] = sums_stage_0_reg[2];
end

logic signed [2*PIXEL_DATAW-1:0] sums_stage_2;
always_comb begin
        sums_stage_2 = sums_stage_1[0] + sums_stage_1[1];
end
// EGRESS: Stage 7
logic signed [2*PIXEL_DATAW-1:0] sums_stage_2_reg;
always_ff @ (posedge clk) begin
        if(enable)begin
```

```
                sums_stage_2_reg <= sums_stage_2;
        end
end

logic unsigned [PIXEL_DATAW-1:0] y;
always_comb begin
        if(sums_stage_2_reg>255) begin
                y = 255;
        end else if (sums_stage_2_reg<0) begin
                y = 0;
        end else begin
                y = sums_stage_2_reg[PIXEL_DATAW-1:0];
        end
end

// Output interface logics
// EGRESS: Stage 8
logic unsigned [PIXEL_DATAW-1:0] r_y;
logic r_y_valid;
logic unsigned [R_X_COL_ADDR_WIDTH-1:0] r_x_col_idx_prev;
always_ff @ (posedge clk) begin
        if(reset) begin
                r_y <= 0;
                r_y_valid <= 0;
                r_x_col_idx_prev <= 0;
        end else if(enable) begin
                r_x_col_idx_prev <=
 ↪  r_x_col_idx_epipelined[NUM_EGRESS_STAGE-1];
                // By the time r_x_col_idx is 3, pixel at idx 2 is already
 ↪  written with i_x
                if(r_x_col_idx_prev !=
 ↪  r_x_col_idx_epipelined[NUM_EGRESS_STAGE-1] &&
                        r_x_col_idx_epipelined[NUM_EGRESS_STAGE-1] >=
 ↪  FILTER_SIZE &&
                        r_x_row_logical_idx_epipelined[NUM_EGRESS_STAGE-1]
 ↪  == R_X_ROWS - 1) begin
                        r_y <= y;
                        r_y_valid <= 1;
                end else begin
                        r_y <= 0;
                        r_y_valid <= 0;
                end
        end
end

assign o_y = r_y;
// Ready for inputs as long as receiver is ready for outputs
assign o_ready = i_ready;
assign o_valid = r_y_valid & i_ready;

// End of your code

endmodule

/*****************************************************************************

// Multiplier module for 8x8 multiplications + 8x8 multiplications
module mult8x8p8x8 (
        input clk,
        input enable,
        input signed [7:0] i_filtera,
        input signed [8:0] i_pixela0,
        input signed [8:0] i_pixela1,
        input signed [7:0] i_filterb,
        input signed [8:0] i_pixelb,
        output logic signed [15:0] o_res
);

// Pipeline 0
logic signed [8:0] i_pixela0_reg, i_pixela1_reg, i_pixelb_reg;
logic signed [7:0] i_filtera_reg, i_filterb_reg;
always_ff @(posedge clk) begin
        if(enable)begin
                i_pixela0_reg <= i_pixela0;
                i_pixela1_reg <= i_pixela1;
                i_pixelb_reg <= i_pixelb;
                i_filtera_reg <= i_filtera;
                i_filterb_reg <= i_filterb;
        end
end

// Pipeline 1
logic signed [9:0] i_pixela01_reg_reg;
logic signed [8:0] i_pixelb_reg_reg;
logic signed [7:0] i_filtera_reg_reg, i_filterb_reg_reg;
always_ff @ (posedge clk) begin
        if(enable) begin
                i_pixela01_reg_reg <= i_pixela0_reg + i_pixela1_reg;
                i_pixelb_reg_reg <= i_pixelb_reg;
                i_filtera_reg_reg <= i_filtera_reg;
                i_filterb_reg_reg <= i_filterb_reg;
        end
end

// Pipeline 2
logic signed [9:0] i_pixela01_reg_reg_reg;
logic signed [8:0] i_pixelb_reg_reg_reg;
logic signed [7:0] i_filtera_reg_reg_reg, i_filterb_reg_reg_reg;
always_ff @ (posedge clk) begin
        if(enable) begin
                i_pixela01_reg_reg_reg <= i_pixela01_reg_reg;
                i_pixelb_reg_reg_reg <= i_pixelb_reg_reg;
                i_filtera_reg_reg_reg <= i_filtera_reg_reg;
                i_filterb_reg_reg_reg <= i_filterb_reg_reg;
        end
end

// Pipeline 3
always_ff @ (posedge clk) begin
        if(enable) begin
                o_res <= i_pixela01_reg_reg_reg * i_filtera_reg_reg_reg +
 ↪  i_pixelb_reg_reg_reg * i_filterb_reg_reg_reg;
        end
end
endmodule

/*****************************************************************************

module pixelram #
(
        parameter FILTER_SIZE = 3,
        parameter PIXEL_DATAW = 8,
        parameter IMAGE_WIDTH = 512,
        parameter R_X_ROWS = FILTER_SIZE,
        parameter R_X_COL_ADDR_WIDTH = 10
)
```

```
(
        input clk,
        input reset,
        input enable,
        // RAM input, unregistered inside the module except for
    ↪   i_read_addr
        input [R_X_COL_ADDR_WIDTH-1:0] i_write_addr [R_X_ROWS-1:0],
        input i_write_enable [R_X_ROWS-1:0],
        input unsigned [PIXEL_DATAW-1:0] i_write_data [R_X_ROWS-1:0],
        input [R_X_COL_ADDR_WIDTH-1:0] i_read_addr, // registered inside
    ↪   the module
        // RAM output
        output unsigned [PIXEL_DATAW-1:0] o_read_data [R_X_ROWS-1:0] //
    ↪   registered, 3 rows of 1 pixel
);
        // Wrap as RAM
        // 0: [] [] [] [] [] [] [] []                          512 + 2
        // 1: [] [] [] [] [] [] [] []                          512 + 2
        // 2: [] [] [] [] [] [] [] []                          512 + 2
        genvar gen_row;
        generate
                for(gen_row=0; gen_row<R_X_ROWS; gen_row=gen_row+1) begin:
    ↪   pixel_ram_row
                        pixelrowram pixel_row_ram
                        (
                                .clk(clk),
                                .reset(reset),
                                .enable(enable),
                                // RAM input, unregistered
                                .i_write_addr(i_write_addr[gen_row]),
                                .i_write_enable(i_write_enable[gen_row]),
                                .i_write_data(i_write_data[gen_row]),
                                .i_read_addr(i_read_addr), // read 1 pixel
                                // RAM output
                                .o_read_data(o_read_data[gen_row]) //
    ↪   registered, 1 pixel
                        );
                end
        endgenerate
endmodule

/*****************************************************************************
```

```

module pixelrowram #
(
        parameter PIXEL_DATAW = 8,
        parameter IMAGE_WIDTH = 512,
        parameter R_X_COL_WIDTH = IMAGE_WIDTH + 2,
        parameter R_X_COL_ADDR_WIDTH = 10
)
(
        input clk,
        input reset,
        input enable,
        // RAM input, unregistered inside the module except for
    ↪   i_read_addr
        input [R_X_COL_ADDR_WIDTH-1:0] i_write_addr,
        input i_write_enable,
        input unsigned [PIXEL_DATAW-1:0] i_write_data,
        input [R_X_COL_ADDR_WIDTH-1:0] i_read_addr, // registered inside
    ↪   the module
        // RAM output
        output logic unsigned [PIXEL_DATAW-1:0] o_read_data // registered
);
        // Wrap as RAM
        // 0: [] [] [] [] [] [] [] []                          512 + 2
        // 2D array of registers for input pixels, row major
        // set_global_assignment -name
    ↪   ADD_PASS_THROUGH_LOGIC_TO_INFERRED_RAMS OFF
        logic unsigned [PIXEL_DATAW-1:0] mem [R_X_COL_WIDTH-1:0];

        logic [R_X_COL_ADDR_WIDTH-1:0] i_read_addr_reg;

        integer i;
        initial begin
                for (i=0; i<R_X_COL_WIDTH; i=i+1) begin
                        mem[i] = 0;
                end
        end

        always_ff @ (posedge clk) begin
                if(enable) begin
                        i_read_addr_reg <= i_read_addr;
                end
```

```
	end

	always_ff @ (posedge clk) begin
		if(enable) begin
			if(i_write_enable) begin
				mem[i_write_addr] <= i_write_data;
			end
			o_read_data <= mem[i_read_addr_reg];
		end
	end
endmodule
```