Lei Liu
[redacted]

Assignment 2

ECE 1756
October 31, 2023

1. FPGA Implementation

1.1 Block diagram and description

Figure 1 denotes an overview of an image with padding(blue denotes image pixel, green denotes padding). We will have 512 pixels + 2 paddings in a row. The yellow highlighted square denotes the first convolution window that is available for a valid output.

$$\text{buffer size} = 514 \times 2 + 3 = 1031 \tag{1}$$

According to Equation 1, we need to buffer 1031 input pixels to compute the first output.

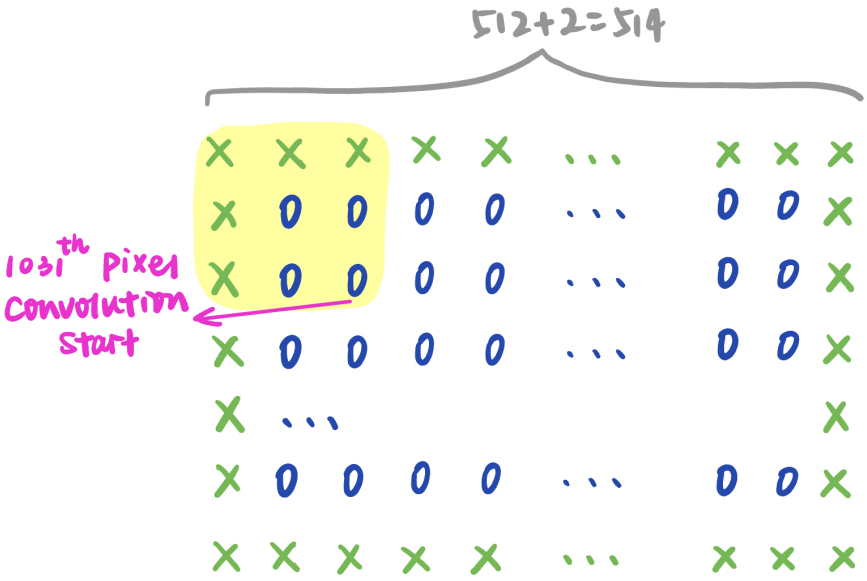

Figure 1: Overview demo of the design.

A 2D-register structure is created to store those input pixels. As shown in Figure 2, there will be 1031 "rows" of 8-bit register. A valid input pixel will be stored in input_buffer[0](the first register), and then shift to the next register in the next clock. In such way, we can store all inputs before the convolution start, and read in new input by shifting all the previous inputs.
Then all items(pixels or paddings) within the convolution window will be a fixed index position in the buffer, and they will be computed with the corresponding fitter coefficients(Figure 3). For example, input_buffer[0] will be associated with fitter[2][2], input_buffer[515] will be associated with fitter[1][1], and so on.
However, when convolution window is transiting to the next rows(Figure 4 shows two scenarios), the fixed-index of the input_buffer will not provide the corresponding input value.

Therefore, we need to lower the output valid signal (for two cycles for each row) to deal with the offset caused by switching rows.

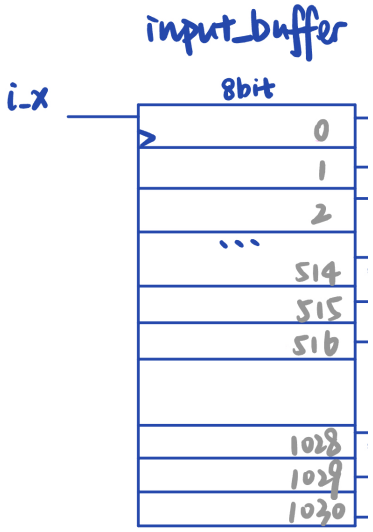

Figure 2: Block diagram of input buffer.

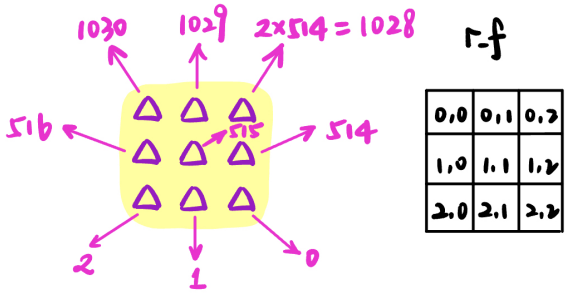

Figure 3: Index for convolution window.

Figure 5 shows a completed block diagram of the design. 9 values from fixed-index of input_buffer will be multiplied by its corresponding fitter coefficients. 8 adders are implemented to compute the total additions. Pipeline registers are inserted between operators to shorten the clk-to-clk path.

To generate o_valid signal, pixel_counter will determine how many input pixels have already flowed into the buffer(1031 is our target number), and row_counter will deal with the offset during row transition. 5 additional pipeline registers are added in this path to match the number of registers in o_y path. Thus, they maintain the correctness of the circuit.

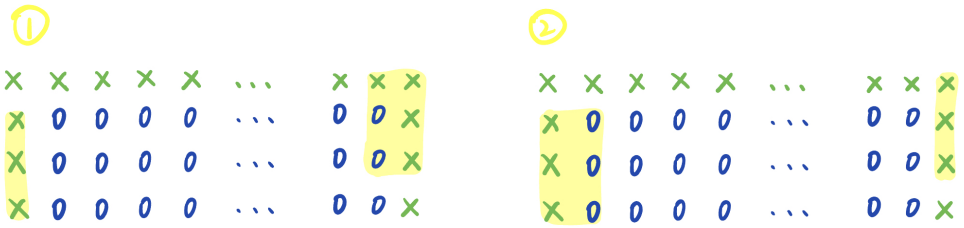

Figure 4: Invalid output scenarios.

The control signals are labeled properly in blue and match the signal naming in HDL(see appendix I).

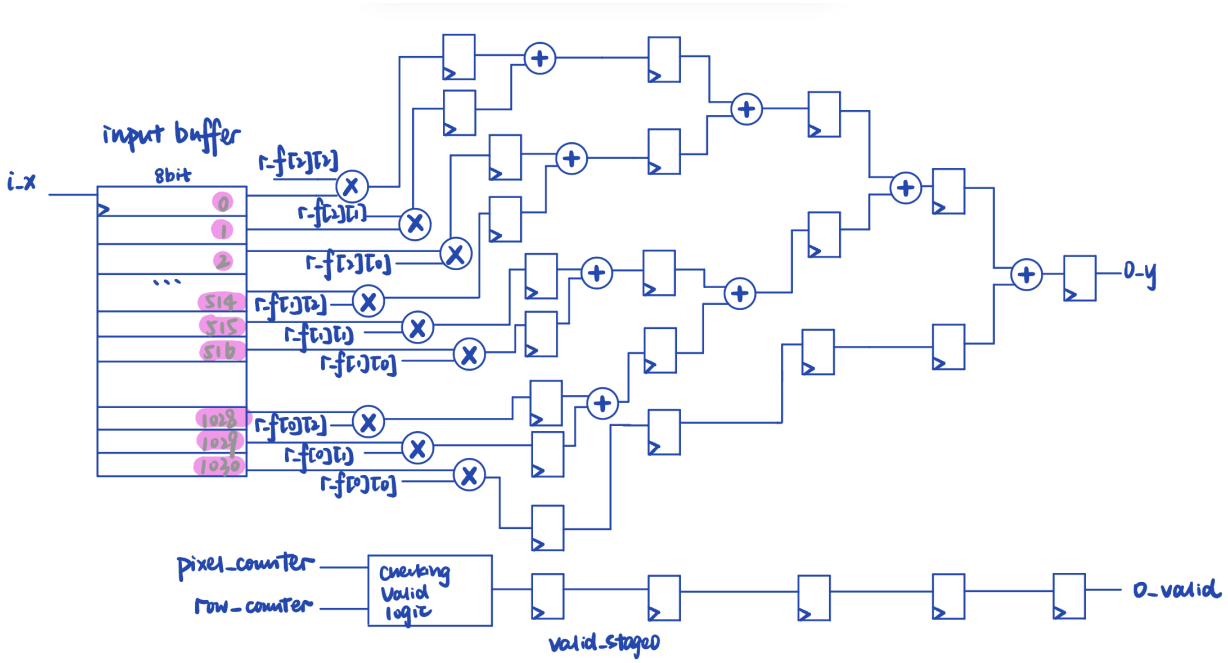

Figure 5: Block diagram of FPGA design.

1.2 Simulation and testbench output

1.2.1 Simulation waveform

We will show the correctness of the design in following approach:

1. Input pixels are stored in the input_buffer, shift to the next position of the input_buffer

in the next clock(Figure 6).

2. The fixed-index of the input_buffer provides the corresponding input value. Let's take first two input index in the testbench 6a as an example. index0 = 89, index1 = 92, as shown in Figure 6. When pixel_counter = 1031, its indicates our first convolution window start to take effect. At that time, inputs are stored in input_buffer[515],[514] respectively(Figure 7).

3. As mentioned before, pixel_counter = 1031 indicates we have enough buffer to generate a valid output. Figure 8 shows valid_stage0 will update on the next cycle, and later propagate a o_valid signal output.

4. Row_counter is used to track if convolution window is in the phase of transition between rows. A value of 1 or 2 will indicate the situation shown in Figure 4. Valid signal will be lowered(Figure 9).

Figure 6: Input pixels in input_buffer.

Figure 7: Fixed-index of the input_buffer.

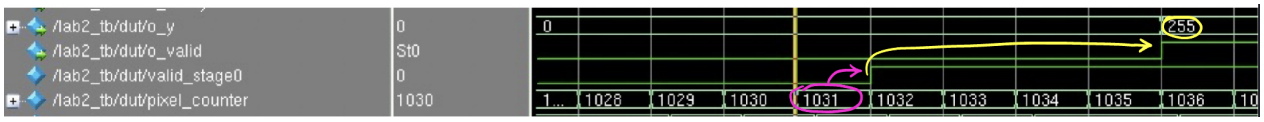

Figure 8: Valid signal propagation.

1.2.2 Testbench output

All tests are passed as shown in Figure 10.

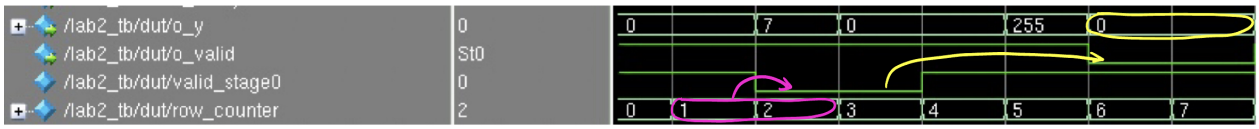

Figure 9: Invalid signal propagation.

```
# Pixel       25594 matching!  Expected: 255  Got: 255
# Pixel       25595 matching!  Expected: 255  Got: 255
# Pixel       25596 matching!  Expected: 255  Got: 255
# Pixel       25597 matching!  Expected: 255  Got: 255
# Pixel       25598 matching!  Expected: 255  Got: 255
# Pixel       25599 matching!  Expected: 255  Got: 255
# TEST PASSED! ALL PIXELS ARE MATCHING GOLDEN RESULT!
```

Figure 10: testbench of FPGA design.

1.3 Table and description

	Result
ALM Utilization	2171
DSP Utilization	9
BRAM (M20k) Utilizaition	0
Maximum Operating Frequency (MHz)	442.28
Cycles for Test 7a (Hinton)	264215
Dynamic Power for one module @ maximum frequency (W)	5.93×10^{-3}
Throughput of one module (GOPS)	7.46
Throughput of full device (GOPS)	1.25×10^{3}
Total Power for full device (W)	1.36

Table 1: FPGA implementation results

1.3.1 ALM, DSP, and BRAM Utilization

$$\#\ \text{ALMs} = \text{total ALMs} - \text{unavailable \# ALMs due to virtual I/Os} = 2218 - 47 = 2171 \quad (2)$$

Resources used are shown in Figure 11-12.

1.3.2 Maximum Operating frequency

$$\text{operating frequency} = \frac{1}{1 + \text{worst-case setup slack(900mV, 0C)}} \quad (3)$$

	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	2,218 / 427,200	< 1 %
2	ALMs needed [=A-B+C]	2,218	
1	[A] ALMs used in final placement [=a+b+c+d]	2,410 / 427,200	< 1 %
1	[a] ALMs used for LUT logic and registers	113	
2	[b] ALMs used for LUT logic	62	
3	[c] ALMs used for registers	2,235	
4	[d] ALMs used for memory (up to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	239 / 427,200	< 1 %
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	47 / 427,200	< 1 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	0	
4	[d] Due to virtual I/Os	47	
3			

Figure 11: ALM used in FPGA design.

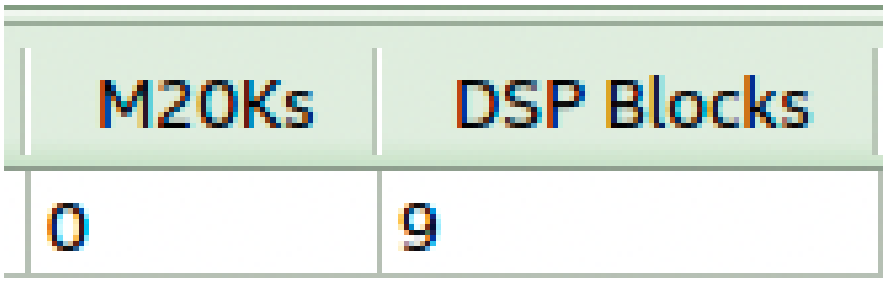

Figure 12: DSP and BRAM used in FPGA design.

Worst-case setup slack is shown in Figure 13.

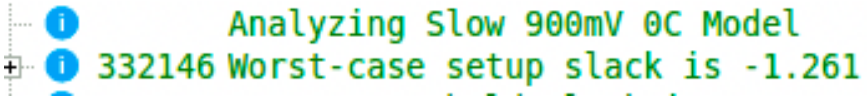

Figure 13: Worst-case setup slack(900mV, 0C) in FPGA design.

1.3.3 Cycles for Test 7a (Hinton)

$$\#\text{ of cycles} = \frac{\text{total time used}}{\text{time for one cycle}} = \frac{5284300000\text{ ps}}{20000\text{ ps}} = 264215 \tag{4}$$

Total time used is shown in Figure 14.

1.3.4 Dynamic Power for one module @ maximum frequency

$$\text{Dynamic power of one module @ 50 MHz} = \text{Thermal Power of (DSP + Combinational cell + Register cell)} \tag{5}$$

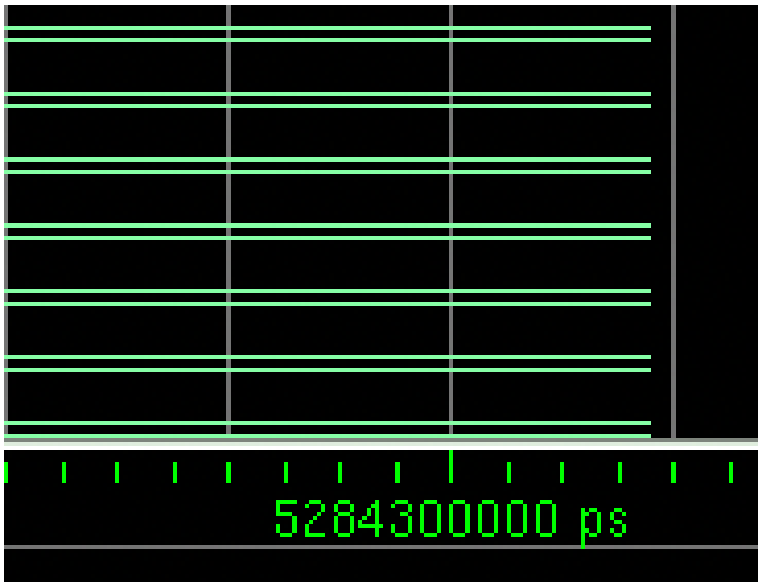

Figure 14: Total time used in 7a testbench.

$$\text{Dynamic power of one module @ 442.28 MHz} = \text{Dynamic power @ 50 MHz} \times \frac{442.28}{50} \quad (6)$$

Thermal power by block type of one module is denoted in Figure 15.

Thermal Power Dissipation by Block Type

<<Filter>>

	Block Type	Total Thermal Power by Block Type
1	DSP block	0.15 mW
2	Combinational cell	0.25 mW
3	Register cell	0.27 mW
4	I/O	8.32 mW

Figure 15: Thermal power of one module.

1.3.5 Throughput of one module (GOPS)

There will be 17 operations needed(9 multiplications and 8 additions) to compute one pixel out.

$$\begin{aligned}&\text{total operations} = \#\text{ of pixels} \times \text{operations per pixel}\\ &= 512 * 512 \times 17 = 4456448\end{aligned} \quad (7)$$

$$\begin{aligned}&\text{total time @ 442.28MHz} = \text{time per cycle} \times \text{total cycles}\\ &= \tfrac{1}{442.28\text{MHz}} \times 264215 = 5.97 \times 10^{-4}\text{seconds}\end{aligned} \quad (8)$$

$$\text{Throughput of one module} = \frac{(7)}{(8)} = 7.46\text{GOPS} \tag{9}$$

1.3.6 Throughput of full device (GOPS)

$$\text{Max. \# of copies / device} = 1518(\text{total \# of DSPs}) \div (\text{\# of DSPs used}) = 168 \tag{10}$$

$$\text{Throughput of full device (GOPS)} = (9) \times (10) = 1.25 \times 10^3\text{GOPS} \tag{11}$$

1.3.7 Total Power for full device

$$\begin{aligned}\text{total power} = \text{dynamic power in (6)} \ \times \ \text{copies in (10)} \ + \\ \text{IO thermal power} \ + \ \text{device static thermal power}\end{aligned} \tag{12}$$

IO thermal power and Static thermal power are denoted in Figure 16.

Power Analyzer Summary

<<Filter>>

Power Analyzer Status	Successful - Sun Oct 22 23:37:15 2023
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Standard Edition
Revision Name	lab2
Top-level Entity Name	lab2
Family	Cyclone V
Device	5CGXFC7C7F23C8
Power Models	Final
Total Thermal Power Dissipation	362.86 mW
Core Dynamic Thermal Power Dissipation	0.67 mW
Core Static Thermal Power Dissipation	349.51 mW
I/O Thermal Power Dissipation	12.68 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 16: Static thermal power for FPGA design.

2. Efficiency Comparison

2.1 High-level explanation of provided CPU and GPU implementations

CPU basic implementation
It uses nested loops to iterate through each filter coefficient and each pixel and compute the dot product between. Accumulate the result. For each filter, a pixel will be iterated FILTER_SIZE times to generate a value for accumulation.
CPU vectorized implementation
It uses nested loops to iterate through each pixel. It then calculates the dot-product of the 3 rows of one filter using the hand-vectorized function. For each filter, a pixel will be itertated 1 time to generate a value for accumulation.
CPU multithreads
It uses OpenMP pragma to perform multi-thread computation. In such way, multiple filters can compute with the image at the same time.
GPU implementation
It will first copy the input image and fitter into GPU memory. It then divide image into several tiles. After loading a tile in GPU shared memory, multiple threads can work on the same tile with different filters.

2.2 Table of different results

	Runtime (ms)			
No. of filters	1	4	16	64
GPU	0.0142553	0.0376308	0.129307	0.494115
CPU (basic - no opt - 1 thread)	6.12433	24.3261	97.4168	389.258
CPU (vectorized - no opt - 1 thread)	3.20562	12.9146	51.7344	209.513
CPU (basic - O2 - 1 thread)	1.27183	4.97758	19.8807	79.2996
CPU (vectorized - O2 - 1 thread)	0.843004	3.38834	13.5209	54.0469
CPU (basic - O3 - 1 thread)	0.532726	2.12236	8.45501	33.9703
CPU (vectorized - O3 - 1 thread)	0.845898	3.37679	13.8811	55.5519
CPU (basic - O3 - 4 threads)	0.942565	1.30897	1.42699	5.79596
CPU (vectorized - O3 - 4 threads)	1.18042	1.48684	1.58726	6.39623

Table 2: Runtime of different versions of CPU and GPU implementations

2.3 Comment on performance difference

basic vs. hand-vectorized
Hand-vectorized performs better than basic when the compiler optimization settings is from no optimization to O2. This is due to hand-vectorized eliminate one loop iteration in the nested loops. In O3 mode, compiler can further unroll all the nested loop for basic and achieve a better performance than hand-vectorized. Once we increase the threads, hand-vectorized will perform similar results due to their similar structures.
single vs. multiple threads
Multiple threads achieve better performance when filters = 4, 16, 64. Noticed that, we did not gain a better performance when filter = 1 because threads are performed parallel with different filters. A single filter only introduces the overhead when issuing threads. Moreover, the amount of the performance increased did not increase as number of filters go from 16 to 64. This is caused by limited number of cores used in CPU.

2.4 Comparison between 3 platforms

	Throughput (GOPS)	Power (W)	Energy Efficiency (GOPS/W)	Area Efficiency (GOPS/mm^2)
FPGA (20nm)	1.25×10^3	1.36	919.12	3.125
CPU (14nm)	4.52	65	0.0695	0.0164
GPU (8nm)	2.78×10^3	220	12.64	7.074
FPGA (scaled to 8nm)	2.0×10^3	0.612	3267.97	20
CPU (scaled to 8nm)	5.65	45.5	0.1242	0.0409

Table 3: Comparison between 3 compute platforms

2.4.1 Throughput(GOPS) for CPU

In Figure 17, we can conclude there are 11 operations within an iteration.

Figure 17: Operations within innerloop for CPU.

In Figure 18, we can conclude there are 5 additional operations for store purpose.

$$\begin{aligned}&\text{total operations} = \#\text{ of filters} \times \#\text{ of pixels} \times (\text{filter size} \times \text{inner operations} + \text{store operations}) \\ &= 64 \times 514 \times 514 \times (9 \times 11 + 5) = 1758488576\end{aligned} \tag{13}$$

```
output_image[(filter_id * image_width * image_height) + (image_y * image_width) + image_x] = accum;
```

Figure 18: Operations for store in CPU.

$$\begin{aligned}&\text{throughput} = \text{operations in (13)} \div \text{Runtime (CPU basic - no opt - 1 thread)}\\&= 1758488576 \div 0.389 = 4.52\text{GOPS}\end{aligned} \tag{14}$$

2.4.2 Throughput(GOPS) for GPU

In Figure 19, during load stage, we will perform 15 operations. Since TILE_SIZE = 32, TILE_LOAD_SIZE = 34, we will perform load iteration 2 times.

```
// Load tiles from Global memory to Shared memory for faster access
for (int itr = 0; itr <= (TILE_LOAD_SIZE * TILE_LOAD_SIZE) / (TILE_SIZE * TILE_SIZE); itr++){
    // Calculate destination x and y indecies
    thread_id = (threadIdx.y * TILE_SIZE) + threadIdx.x + (itr * TILE_SIZE * TILE_SIZE);
    tile_location_y = thread_id / TILE_LOAD_SIZE;
    tile_location_x = thread_id % TILE_LOAD_SIZE;

    // Calculate source pixel index
    image_location_y = blockIdx.y * TILE_SIZE + tile_location_y - FILTER_RADIUS;
    image_location_x = blockIdx.x * TILE_SIZE + tile_location_x - FILTER_RADIUS;
    pixel_id = (image_location_y * image_width) + image_location_x;
```

Figure 19: Operations within load for GPU.

In Figure 20, during convolution, we will perform 9 operations for each pixel and each filter coefficients(see yellow circle). 9 operations for each pixel to write the output(see pink circle, but notice that its 512*512 which excludes paddings).

$$\begin{aligned}&\text{total operations} = 15 \times 2 + 514 \times 514 \times 64 \times 9 \times 9 + 512 \times 512 \times 9\\&= 1371951390\end{aligned} \tag{15}$$

$$\begin{aligned}&\text{throughput} = \text{operations in (15)} \div \text{GPU Runtime in 64 filters}\\&= 1371951390 \div 0.494 \times 10^{-3} = 2.78 \times 10^{3}\text{GOPS}\end{aligned} \tag{16}$$

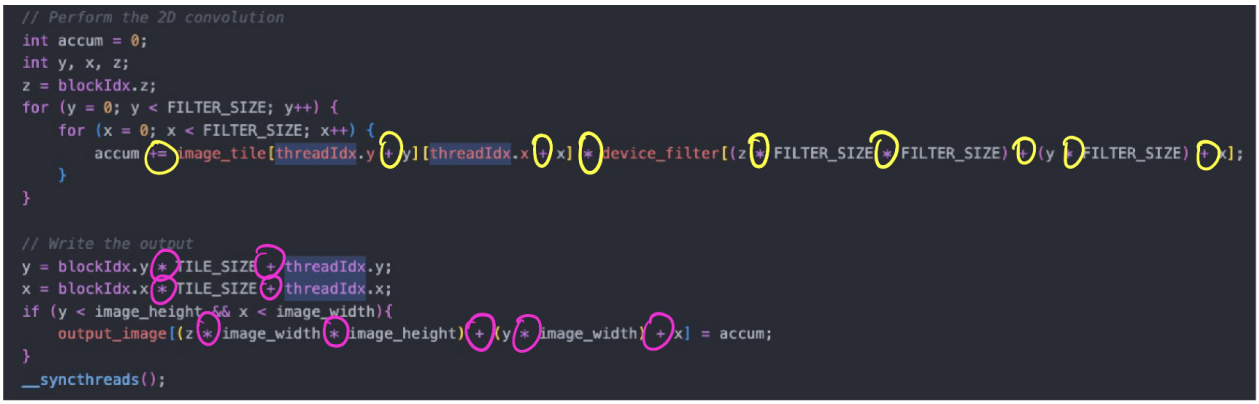

Figure 20: Operations within convolution for GPU.

2.4.3 Area Efficiency for 3 platforms

We will use Die Size denoted in Table 4 in the Handout to calculate Area Efficiency for each platform.

$$\text{Area Efficiency} = \text{throughput} \div \text{Die Size} \tag{17}$$

2.4.4 Scaling process

We will first calculate the scaled area, we will get 100mm^2 for FPGA and 138mm^2 for CPU.

$$\text{Scaled area} = \text{original Die Size} \times \text{Area scaling} \tag{18}$$

A 1.25x/1.6x increase in scaling clock speed implies a 1.25x/1.6x increase in frequency. It further denotes that total time used for calculation decrease 1.25x/1.6x, which results in a 1.25x/1.6x increase in throughput. Therefore, we can conclude:

$$\text{Scaled throughput} = \text{original throughput} \times \text{Clock speed scaling} \tag{19}$$

Finally, we can directly use power scaling factor to compute the scaled power.

$$\text{Scaled power} = \text{original power} \times \text{Power scaling} \tag{20}$$

We will use the elements in Table 4 to complete the entries left in Table 3.

	Scaled area	Scaled throughput	Scaled power
FPGA (scaled to 8nm)	100	2.0×10^3	0.612
CPU (scaled to 8nm)	138	5.65	45.5

Table 4: Scaling element for FPGA and CPU

2.5 Comment on the comparison between 3 platforms

Throughput comparison
GPU scores the highest in the throughput because it is more suitable to handle large sets of streaming data with memory interactions and multi-thread.(In our GPU example, multiple threads are working on the shared memory). FPGA scores the second with a tiny gap, this is due to the highly-pipelined design. "Massive" operations can now be "packed" into one second in a high frequency.
Throughput/W comparison
FPGA is "way more" ahead compared with GPU and CPU. Power estimated for FPGA is more design-specific which is directly measured from simulation through CAD tool. In comparison, GPU and CPU power are estimated as the thermal design power of the platform. They are the maximum value the platform could consume and not design-specific. Moreover, GPU and CPU consumes more power on overhead of threads, branch prediction when looping, etc...
Throughput/mm^2 comparison
FPGA is still ahead compared with GPU and CPU. Again, this is achieved by the reduced overhead. Compared to general-purposed platform(GPU and CPU), it eliminates the use of structure such as instruction cache, register file, and etc...

Appendix

Appendix I: HDL source code for FPGA design

```
// 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, buffer_i; // variables to use in the for loop


// Start of your code
localparam IMAGE_SIZE = 514;        //size of the image plus padding(512 + 2)
// We need a buffer to store all the input pixels before the first convolution start
// The first pixel can only be convoluted after two full rows + 3 pixels at the third row
localparam BUFFER_SIZE = IMAGE_SIZE*2 + FILTER_SIZE;
logic signed [PIXEL_DATAW-1:0] input_buffer [BUFFER_SIZE];
// We need a counter to record how many pixels have been added to the buffer
// We use this to tell: convolution begin valid
// The full size could be 514*514 so we need 20 bits for counter
logic [19:0]pixel_counter;
// We also need a row counter to deal with the offset of adding padding into pixel_counter
// Valid need to be stalled for two cycles for every row (due to additional two paddings)
logic [9:0]row_counter;
// Once the counter reach to the BUFFER_SIZE, buffer is enough to perform the first pixel
// Due to the pipelining during convolution, we need to pipeline the valid signal
// In order to match the output cycle
logic valid_stage0;
logic valid_stage1;
logic valid_stage2;
logic valid_stage3;
logic valid_stage4;
```

```
// Value to store the output from multiplier and adder
logic signed [15:0] mult_0, mult_1, mult_2, mult_3, mult_4, mult_5, mult_6,
↪  mult_7, mult_8;
logic signed [18:0] firstadd_0, firstadd_1, firstadd_2, firstadd_3;
logic signed [18:0] secondadd_0, secondadd_1;
logic signed [18:0] thirdadd_0;
// The maximum bits from the last adder will be 20 bits
logic signed [19:0] finaladd_out;
// Cap output to 8 bits
logic unsigned [7:0] finalcap_out;

// Pipeline registers to propogate the value
logic signed [18:0] mult_0_reg, mult_1_reg, mult_2_reg, mult_3_reg,
↪  mult_4_reg, mult_5_reg, mult_6_reg, mult_7_reg, mult_8_reg;
logic signed [18:0] firstadd_0_reg, firstadd_1_reg, firstadd_2_reg,
↪  firstadd_3_reg, firstadd_wait;
logic signed [18:0] secondadd_0_reg, secondadd_1_reg, secondadd_wait;
logic signed [18:0] thirdadd_0_reg, thirdadd_wait;

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
                // Clean out all the pixels stored in the buffer
                for(buffer_i =0; buffer_i < BUFFER_SIZE; buffer_i = buffer_i
                ↪  + 1)begin
                        input_buffer[buffer_i] <= '0;
                end

                // Reset the counters
                pixel_counter <= '0;
                row_counter <= '0;
                // Reset the valid signal
```

```
            valid_stage0 <= 0;

        // 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

            if(i_valid) begin
                // Prepare for the input buffer
                // The newest input will be put in the first index
                ↪  of buffer
                // All the rest will be shifted by 1
                input_buffer[0] <= i_x;
                pixel_counter <= pixel_counter + 1;
                for(buffer_i=0; buffer_i < BUFFER_SIZE - 1; buffer_i
                ↪  = buffer_i + 1)begin
                    // Shift all pixels by 1 in the buffer
                    input_buffer[buffer_i + 1] <=
                    ↪  input_buffer[buffer_i];
                end

                // Continously record row_counter
                // To know which row we are currently at
                row_counter <= row_counter + 1;
```

```
// Propogate the output from multiplier/adder
// Using the pipeline registers
mult_8_reg <= mult_8;
mult_7_reg <= mult_7;
mult_6_reg <= mult_6;
mult_5_reg <= mult_5;
mult_4_reg <= mult_4;
mult_3_reg <= mult_3;
mult_2_reg <= mult_2;
mult_1_reg <= mult_1;
mult_0_reg <= mult_0;

firstadd_0_reg <= firstadd_0;
firstadd_1_reg <= firstadd_1;
firstadd_2_reg <= firstadd_2;
firstadd_3_reg <= firstadd_3;
firstadd_wait <= mult_0_reg;

secondadd_0_reg <= secondadd_0;
secondadd_1_reg <= secondadd_1;
secondadd_wait <= firstadd_wait;

thirdadd_0_reg <= thirdadd_0;
thirdadd_wait <= secondadd_wait;

// Cap the final output to 8 bits
if (finaladd_out > 20'sd255) begin
        finalcap_out <= 8'd255;
end
else if (finaladd_out < 20'sd0)begin
        finalcap_out <= 8'd0;
end
else begin
        finalcap_out <= finaladd_out[7:0];
end

// Propogate the valid signal
// Using the pipeline registers
valid_stage1 <= valid_stage0;
valid_stage2 <= valid_stage1;
valid_stage3 <= valid_stage2;
```

```
                        valid_stage4 <= valid_stage3;

                end

                // We have reached enough buffer to start a valid
                ↪  calculation
                if(pixel_counter >= BUFFER_SIZE) begin
                        // Additional output caused by two paddings when
                        ↪  transfering
                        // Rows, should stall valid signal for two cycles
                        if(row_counter == 'd1) begin
                                valid_stage0 = 1'b0;
                        end
                        else if(row_counter == 'd2)begin
                                valid_stage0 = 1'b0;
                        end
                        else begin
                                valid_stage0 <= 1'b1;
                        end
                end

                if(row_counter == IMAGE_SIZE - 1)begin
                        // Reset row counter for new rows
                        row_counter <= 'd0;
                end
        end
end


mult8x8 mult_layer0_8 (.i_dataa(input_buffer[0]), .i_datab(r_f[2][2]),
↪  .o_res(mult_8));
mult8x8 mult_layer0_7 (.i_dataa(input_buffer[1]), .i_datab(r_f[2][1]),
↪  .o_res(mult_7));
mult8x8 mult_layer0_6 (.i_dataa(input_buffer[2]), .i_datab(r_f[2][0]),
↪  .o_res(mult_6));
mult8x8 mult_layer0_5 (.i_dataa(input_buffer[IMAGE_SIZE]),
↪  .i_datab(r_f[1][2]), .o_res(mult_5));
mult8x8 mult_layer0_4 (.i_dataa(input_buffer[IMAGE_SIZE + 1]),
↪  .i_datab(r_f[1][1]), .o_res(mult_4));
mult8x8 mult_layer0_3 (.i_dataa(input_buffer[IMAGE_SIZE + 2]),
↪  .i_datab(r_f[1][0]), .o_res(mult_3));
```

```
mult8x8 mult_layer0_2 (.i_dataa(input_buffer[IMAGE_SIZE*2]),
↪  .i_datab(r_f[0][2]), .o_res(mult_2));
mult8x8 mult_layer0_1 (.i_dataa(input_buffer[IMAGE_SIZE*2 + 1]),
↪  .i_datab(r_f[0][1]), .o_res(mult_1));
mult8x8 mult_layer0_0 (.i_dataa(input_buffer[IMAGE_SIZE*2 + 2]),
↪  .i_datab(r_f[0][0]), .o_res(mult_0));

add19x19 add_layer1_3 (.i_dataa(mult_8_reg), .i_datab(mult_7_reg),
↪  .o_res(firstadd_3));
add19x19 add_layer1_2 (.i_dataa(mult_6_reg), .i_datab(mult_5_reg),
↪  .o_res(firstadd_2));
add19x19 add_layer1_1 (.i_dataa(mult_4_reg), .i_datab(mult_3_reg),
↪  .o_res(firstadd_1));
add19x19 add_layer1_0 (.i_dataa(mult_2_reg), .i_datab(mult_1_reg),
↪  .o_res(firstadd_0));

add19x19 add_layer2_1 (.i_dataa(firstadd_3_reg), .i_datab(firstadd_2_reg),
↪  .o_res(secondadd_1));
add19x19 add_layer2_0 (.i_dataa(firstadd_1_reg), .i_datab(firstadd_0_reg),
↪  .o_res(secondadd_0));

add19x19 add_layer3_0 (.i_dataa(secondadd_1_reg), .i_datab(secondadd_0_reg),
↪  .o_res(thirdadd_0));

add19x19 add_layer4_0 (.i_dataa(thirdadd_0_reg), .i_datab(thirdadd_wait),
↪  .o_res(finaladd_out));


assign o_y = finalcap_out;
assign o_valid = valid_stage4;
assign o_ready = i_ready;
// End of your code

endmodule

module mult8x8 (
        input unsigned [7:0] i_dataa,
        input signed [7:0] i_datab,
        output signed[15:0] o_res
);
        assign o_res = $signed({1'b0, i_dataa}) * i_datab;
```

```
endmodule

module add19x19(
        input  signed [18:0] i_dataa,
        input  signed [18:0] i_datab,
        output signed [19:0] o_res
);
        assign o_res = i_dataa + i_datab;
endmodule
```