Part 1: FPGA Implementation

1. Description and Block diagram

   As the module receives each new input pixel, it is placed into a buffer Register so that the module can access these in future calculations. When the program reaches the end of the row, each row propagates upward (3 to 2, 2 to 1). This allows each pixel to be calculated as the module receives the pixels in the correct order. When the module receives a pixel, instead of placing the pixel into the buffer register and then calculating, in order to improve efficiency, this calculation and storing is performed in the same clock cycle.

2. Results Table

Table 1: **FPGA implementation results**

	Result
ALM Utilization	10,317/427,200
DSP Utilization	8/1518
BRAM (M20K) Utilization	0/2713
Maximum Operating Frequency	273.149 MHz
Cycles for Test 7a (Hinton)	264721
Dynamic Power for one module @ maximum frequency mW	458.97
Throughput of one module (GOPS)	5.736
Throughput of full device (GOPS)	217.973
Total Power for full device (mW)	17440.86

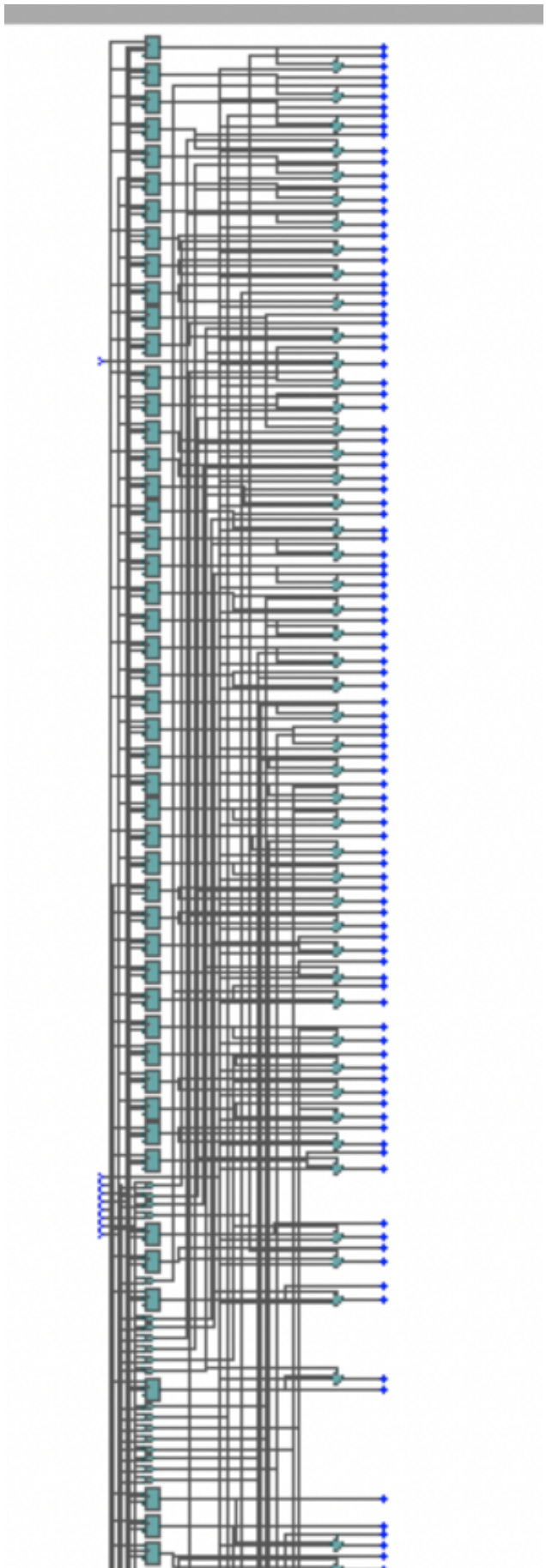
3. WaveForms and Test Bench outputs

4. Implementation

Algorithm 1: Lab2.sv

```

// 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
```



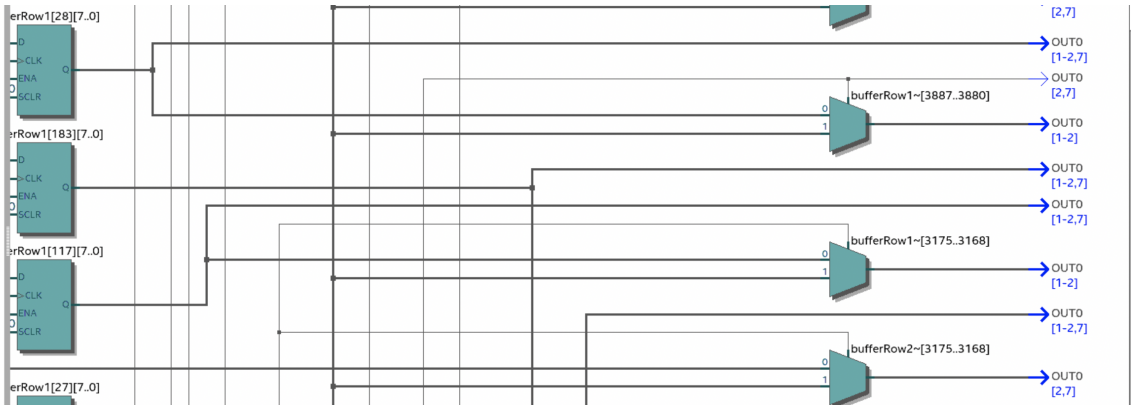


Figure 2: Block Diagram Partially Zoomed in

```
// from 0 (black) to 255 (white).
module lab2 (
    input  logic        clk,// Operating clock
    input  logic        reset,// Active-high reset signal
    (reset when set to 1)
    input  logic        signed [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  logic        i_valid,// Set to 1 if input pixel is
    valid
    input  logic        i_ready,// Set to 1 if consumer block is
    ready to receive a new pixel
    input  logic        [7:0] i_x,// Input pixel value (8-bit unsigned
    value between 0 and 255)
    output logic        o_valid,// Set to 1 if output pixel is valid
    output logic        o_ready,// Set to 1 if this block is ready
    to receive a new pixel
    output logic        [7:0] o_y// Output pixel value (8-bit unsigned
    value between 0 and 255)
) /* synthesis multstyle = "dsp" */;

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
```

Transcript

```
# 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
```

Figure 3: Test Bench output of test 7A

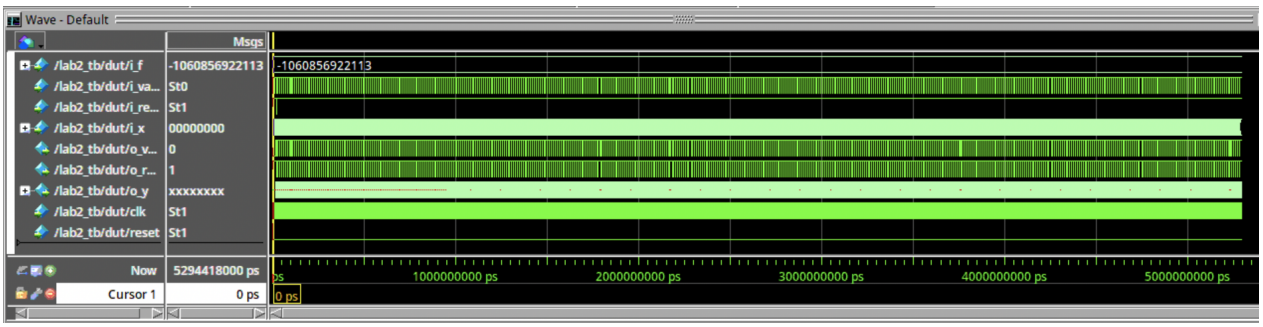


Figure 4: Wave Form of test 7A

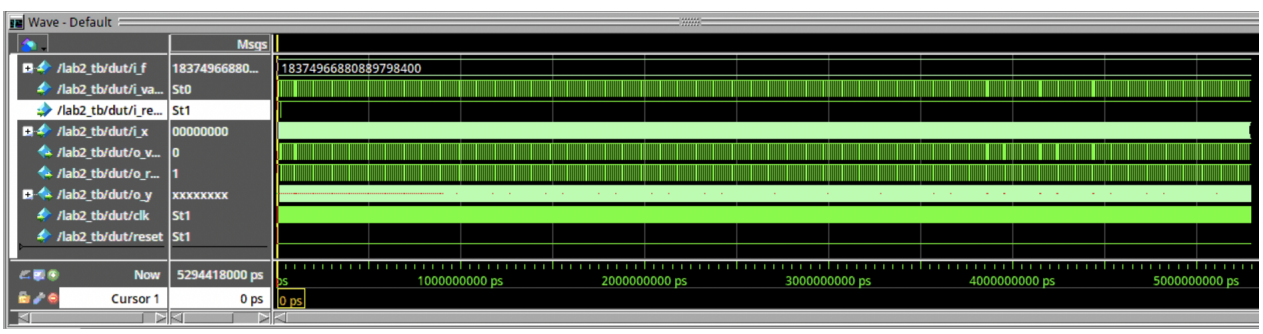


Figure 5: Wave Form of test 7B

```
Transcript
# Pixel      262115 matching!  Expected:  31  Got:  31
# Pixel      262116 matching!  Expected:  34  Got:  34
# Pixel      262117 matching!  Expected:  35  Got:  35
# Pixel      262118 matching!  Expected:  35  Got:  35
# Pixel      262119 matching!  Expected:  35  Got:  35
# Pixel      262120 matching!  Expected:  33  Got:  33
# Pixel      262121 matching!  Expected:  37  Got:  37
# Pixel      262122 matching!  Expected:  40  Got:  40
# Pixel      262123 matching!  Expected:  42  Got:  42
# Pixel      262124 matching!  Expected:  47  Got:  47
# Pixel      262125 matching!  Expected:  47  Got:  47
# Pixel      262126 matching!  Expected:  41  Got:  41
# Pixel      262127 matching!  Expected:  48  Got:  48
# Pixel      262128 matching!  Expected:  53  Got:  53
# Pixel      262129 matching!  Expected:  56  Got:  56
# Pixel      262130 matching!  Expected:  45  Got:  45
# Pixel      262131 matching!  Expected:  41  Got:  41
# Pixel      262132 matching!  Expected:  54  Got:  54
# Pixel      262133 matching!  Expected:  46  Got:  46
# 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
```

Figure 6: Test Bench output of test 7B

```
// 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
```

```

//Input data needs to be buffered before processing
/*
    - Every image is 512 pixels in width(511:0)
    - At initial startup, need to wait for 1024 inputs (2 rows)
    -  + 2 additional inputs for first 2 pixels on third row
    - Once the 3rd pixel of 3rd row is in, can start applying the filter
    - One buffer register per buffer row
    - once third row is fully in: 2 -> 1, 3->2
*/
logic [PIXEL_DATAW-1:0] bufferRow1 [513:0];
logic [PIXEL_DATAW-1:0] bufferRow2 [513:0];
logic [PIXEL_DATAW-1:0] bufferRow3 [513:0];
int currPixelptr = 0;//Pointer to decide where to store each pixel
int startupTimer = 0;//Counter to wait till the registers are filled up
correctly before starting process
int loops = 0;//wait till this reaches 2 before propogating.

logic propogatingReady;
logic calculated_0;


logic signed [15:0] temp0, temp1, temp2, temp3, temp4 ,temp5, temp6, temp7,
temp8;
logic signed [15:0] temp15out;
logic [7:0] tempout;


function [15:0] clamp(input signed [15:0] value);
    if(value > 255) begin
        clamp = 255;
    end else if (value <0) begin
        clamp = 0;
    end else begin
        clamp = value[7:0];
    end

endfunction

always_comb begin
    temp0 = $signed({{8{i_f[7]}},i_f[7:0]}) *
        $signed({8'b0, bufferRow1[currPixelptr-2]});
    temp1 = $signed({{8{i_f[15]}},i_f[15:8]}) *
        $signed({8'b0, bufferRow1[currPixelptr-1]});
```

```
    temp2 = $signed({{8{i_f[23]}},i_f[23:16]}) *
        $signed({8'b0, bufferRow1[currPixelptr]});
    temp3 = $signed({{8{i_f[31]}},i_f[31:24]}) *
        $signed({8'b0, bufferRow2[currPixelptr-2]});
    temp4 = $signed({{8{i_f[39]}},i_f[39:32]}) *
        $signed({8'b0, bufferRow2[currPixelptr-1]});
    temp5 = $signed({{8{i_f[47]}},i_f[47:40]}) *
        $signed({8'b0, bufferRow2[currPixelptr]});
    temp6 = $signed({{8{i_f[55]}},i_f[55:48]}) *
        $signed({8'b0, bufferRow3[currPixelptr-2]});
    temp7 = $signed({{8{i_f[63]}},i_f[63:56]}) *
        $signed({8'b0, bufferRow3[currPixelptr-1]});
    temp8 = $signed({{8{i_f[71]}},i_f[71:64]}) *
        $signed({8'b0, i_x});

    temp15out = temp0 + temp1 + temp2 + temp3 + temp4 + temp5 +
        temp6 + temp7 + temp8;
    tempout = clamp(temp15out);
end
always_ff @(posedge clk)begin
    calculated_0 <= 1'b0;
    propogatingReady <= 1'b1;
    if(i_valid && i_ready)begin
        //Waiting 1028 cycles to start convolution
        if(startupTimer <=1027)begin
            if(startupTimer<513)begin
                bufferRow1[currPixelptr] <= i_x;
            end else if(startupTimer==513)begin
                bufferRow1[currPixelptr] <= i_x;
                bufferRow2[0] <= i_x;
            end else if(startupTimer>513)begin
                bufferRow2[currPixelptr] <= i_x;
            end
            startupTimer = startupTimer +1;
        end else begin//Startup has completed
            bufferRow3[currPixelptr] <= i_x;
        end
        //Updating pointer to decide where to store each pixel -
            Needs to loop once it gets to the end
        if(loops>2) begin
            if(currPixelptr==0)begin
                propogatingReady <= 1'b0;
                bufferRow1<=bufferRow2;
                bufferRow2<=bufferRow3;
```

```
                    currPixelptr = currPixelptr +1;
                end else if(currPixelptr<513)begin
                    currPixelptr = currPixelptr +1;
                end else begin
                    currPixelptr = 0;
                    loops = loops+1;
                end
            end else begin
                if(currPixelptr<513)begin
                    currPixelptr = currPixelptr +1;
                end else begin
                    currPixelptr = 0;
                    loops = loops+1;
                end
            end
        end
    end

assign o_y = tempout;
assign o_ready = i_ready & propogatingReady;
//assign o_valid = calculated_O & i_ready & (loops>1);
assign o_valid = i_ready & (currPixelptr>1) & (loops>1);
// End of your code

endmodule
```

Part 2: Efficiency Comparison

1. A high-level explanation of the provided CPU convolution functions and GPU convolution CUDA kernel.

   CPU Basic Convolution - The basic CPU 2D convolution works using nested for loops to iterate over each pixel of the image, assuming that it is already padded. For each pixel it calculates the new convoluted image and then stores it in the corresponding pixel location in the output image.

   CPU Vectorized Convolution - The core function behind the vectorized version of convolution is the dot3 function. This works by calculating the dot product of two vectors that are size 3. The main vector convolution function then iterates over each pixel in a similar way to that of the basic convolution, however, it calculates the result using the dot product of each row and its respective filter row, sums up the result, then places it in the corresponding pixel location in the output image.

CPU Multithreaded Convolution - For both of the previously mentioned functions, there is a multithreaded version. This multithreaded version is the exact same functionality-wise, however, there is one key line "#pragma omp parallel for". This line first spawns a group of threads for the for loop to iterate on and then divides the total loop iterations over the newly spawned threads resulting in a quicker execution time.

GPU Convolution CUDA kernel - This function takes an input pointer to the input and output image in GPU memory, along with the image dimension. With these parameters, it is able to divide the image into smaller "Tiles" which are then loaded into shared memory so that the group of thread can work on it in a much quicker way than that of the CPU.

2. Comment on the difference in performance between the different versions of CPU code (basic vs. hand-vectorized, single vs. multiple threads) under different compiler optimization settings (no optimization, O2, and O3) and the reasons for these differences.

   As the level of optimization increased, the runtime decreased for all amount of filters. Similarly, hand-vectorizing the convolution process also decreased the runtime however, when optimizing from level 2 to level 3, the runtime only dropped 0.003ms, which is a relatively insignificant amount. This is most likely due to how the optimization is done, the vectorized function likely has a limit to how fast it can run. When parallelizing the function, it slowed down the runtime for one filter, most likely due to taking additional time to spawn and dispatch each thread. However, this runtime did not increase as proportionally as the single threaded version when increasing the number of filters due to the nature of multi-threading.

3. Comment on the throughput in Giga-operations per second (GOPS), throughput/W, and throughput/mm2 of the FPGA, CPU and GPU devices. Briefly explain what leads to the differences on these metrics between the devices.

   The FPGA circuit is dedicated towards performing the image convolution as fast as possible which results in it being the fastest. The GPU on the other hand, while not dedicated, is able to perform many calculations synchronously due to its multi-threading capabilities. The CPU finally, is the least dedicated and when not multi-threading the calculation, it is much slower. In terms of efficiency, the FPGA is able to utilize it's quick performance in combination with it's minimal power usage to achieve such a low score in terms of both power and area efficiency. The GPU on the other hand, while quite fast, is unable to achieve a low score due to its large power usage and relative size. While the CPU is much slower than the GPU, its relative size and power usage results in it achieving a higher score in terms of efficiency than its GPU counterpart.

   It was found that the GPU achieved a higher throughput when compared to that of

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	0.015	0.038	0.130	0.493
CPU (basic - no opt - 1 thread	6.073	24.457	97.180	388.772
CPU (vectorized - no opt - 1 thread	3.223	12.947	51.819	210.387
CPU (basic - O2 - 1 thread)	0.963	3.889	15.638	62.127
CPU (vectorized - O2 - 1 thread)	0.852	3.371	13.559	54.172
CPU (basic - O3 - 1 thread)	0.532	2.128	8.452	33.908
CPU (vectorized - O3 - 1 thread)	0.849	3.381	13.522	54.198
CPU (basic - O3 - 4 threads)	0.902	0.941	1.287	5.192
CPU (vectorized - O3 - 4 threads)	1.152	1.167	1.542	5.859

the FPGA and the CPU implementations, however, it came at a cost of a much lower power efficiency when compared to the FPGA. While it was still much more efficient than the CPU in both terms of energy and area efficiency, it was only more efficient than the FPGA in terms of area.

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/mmˆ2)
FPGA (20 nm)	217.973	17.44086	12.498	0.545
CPU (14 nm)	336.061	65	5.17	1.218
GPU (8nm)	2756.503	220	12.529	7.014
FPGA (scaled to 8 nm)	348.757	7.848	44.439	2.18
CPU (scaled to 8 nm)	420.076	45.5	9.232	3.044

4.

$$Throughput_{CPU14nm}(GOPS) = \frac{\#ofOperations}{Time(ms) * 10^9}$$

$$Throughput_{GPU8nm}(GOPS) = \frac{\#ofOperations}{Time(ms) * 10^9}$$

$$Throughput_{FPGA}(GOPS) = Frequency * \#Operations * \#Copies$$

The first main calculation that needed to be performed was the throughput calculation for each device. For the CPU and GPU, this number was easy to obtain as it involved

just counting the number of operations per image and dividing it by the time that was obtained in Table 2. For the CPU I had found the number of operations to be 512 (Width) * 512 (Height) * 104 (Calculations/pixel) * 64 (filters) = 1744830464. For the GPU it divided the image into multiple tiles so that it could perform the convolution in a multithreaded way. It divided the image into tiles that were 3x3. Therefore, I found the number of operations to be 512 (Width) * 512 (Height) * 81 (calculations/tile) +18 (setup in terms of loading the tile from memory) + 9 (initial setup) * 64 (filters) = 1358956224.

For the FPGA however, this was much more complex and required calculating the number of copies able to be created on the circuit, the # of operations performed per pixel and finally the maximum operating frequency. Finally, the product of these 3 values resulted in the throughput of the FPGA.

It is important to note that the CPU calculation was performed using the time obtained using the CPU (basic - O3 - 4 threads) version.

5.
$$Throughput_{CPU8nm}(GOPS) = Throughput_{CPU14nm} * 1.25$$

When converting the throughput of the CPU from 14nm to 8nm, I multiplied the original throughput by 1.25 which was the new clock speed of the smaller CPU.

6.
$$Power_{8nm} = Power_{14nm} * 0.7$$

When converting the power of the CPU from 14nm to 8nm, the original TPD value was multiplied by the given factor of 0.7.

7.
$$Area_{8nm} = Area_{14nm} * 0.5$$

When converting the area of the CPU from 14nm to 8nm, the original area value was multiplied by the given factor of 0.5.

8.
$$Throughput_{FPGA8nm}(GOPS) = Throughput_{FPGA20nm} * 1.6$$

When converting the throughput of the FPGA from 20nm to 8nm, the original throughput value was multiplied by the given factor of 1.6.

9.
$$Power_{8nm} = Power_{20nm} * 0.45$$

When converting the power of the FPGA from 20nm to 8nm, the original throughput value was multiplied by the given factor of 0.45.

10.

$$Area_{8nm} = Area_{20nm} * 0.25$$

When converting the area of the FPGA from 20nm to 8nm, the original throughput value was multiplied by the given factor of 0.25.