Lab 2 - Image Convolution on FPGA, CPUs, and GPUs

ECE 1756 (Fall 2025)

Kuan-Yu Chang (1007359760)

November 04, 2025

1.0 Introduction

This document serves as the lab report for ECE1756 Fall 2025 Section. It is authored by Kuan-Yu Chang (1007359760).

Section 2.0 describes the FPGA implementation of the convolution function, its operation and performance while Section 3.0 describes the same metrics of the CPU and GPU implementation. A comparison of the different implementations will be done in Section 4.0.

2.0 FPGA Implementation

To enable a comparison between FPGA, CPU and GPU, a convolution engine was implemented on an Intel Arria 10 GX device (10AX115N2F45I1SG). The following section describes the details of this implementation as well as performance metrics.

2.1 Convolution and Input Data Details

The FPGA convolution engine processes a 512-pixel-wide image of arbitrary height using a provided 3×3 filter. Pixels arrive in row-major order (top-left first, one pixel per cycle), and a convolution is computed for every 3×3 neighborhood. Because neighborhoods span adjacent rows, incoming pixels must be retained across lines. To do this, we use a serial-in/parallel-out (SIPO) shift register. With a fixed line length of 514 pixels (512 image pixels plus zero padding), the SIPO must be at least 1031 entries long. Pixels shift in from the left and advance one register per clock cycle.

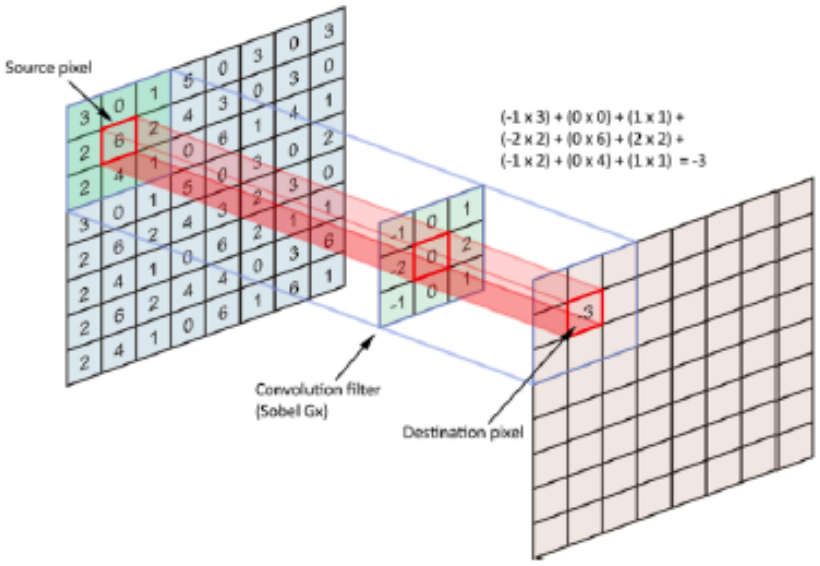

Figure 1: Convolution example of the design

2.2 FPGA Datapath

The pipelined circuit improves performance by dividing the computation into multiple stages, each separated by registers. Each stage performs a smaller portion of the overall calculation, allowing several input values to be processed simultaneously in different stages.

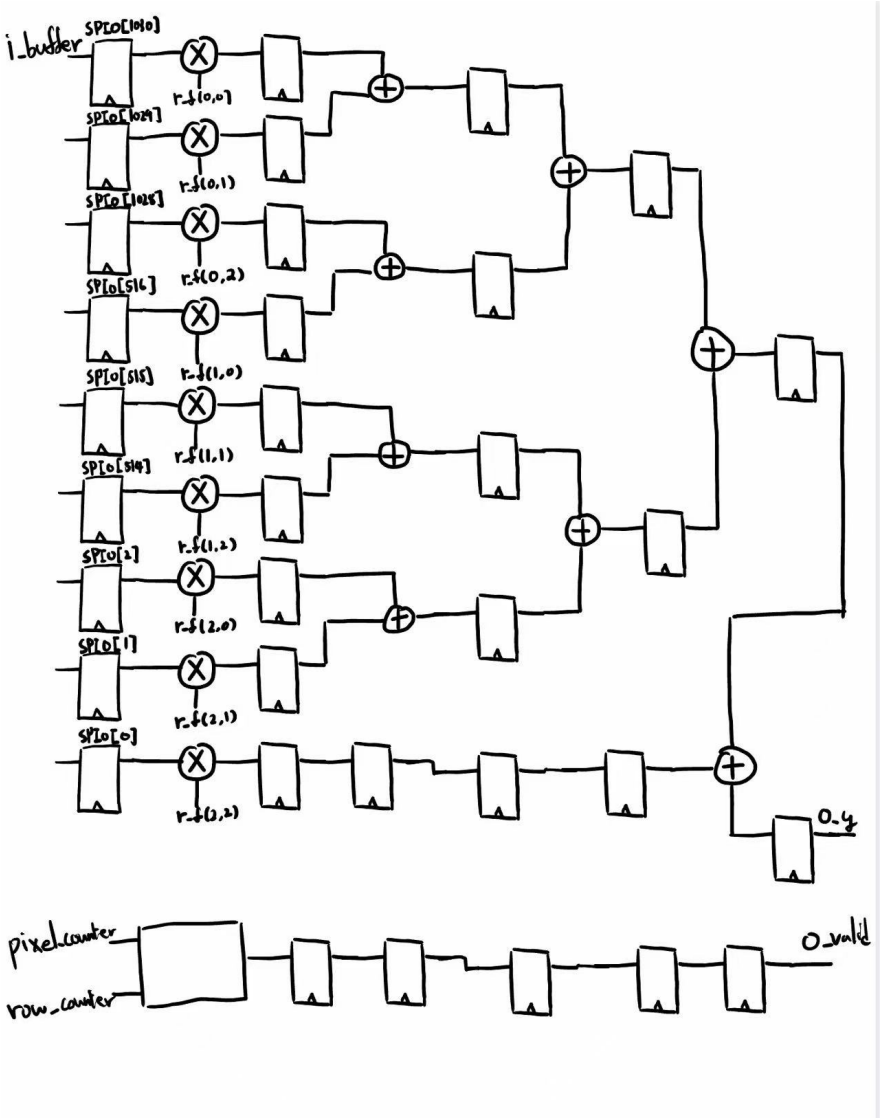

Figure 2: Pipelined convolution datapath

The datapath for this computation of convolution is presented above. Which is compsoed by 9 multipliers performing the 9 multiplcations in parallel than summed in a 3 level adder tree. Data are pipelined at every stage to ensure the data correctness of the output.

2.3 Operation

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. For example, i buffer[0] will be associated with fitter[2][2], i buffer[515] will be associated with fitter[1][1],

and so on. The input data will first be multiplied with the filter window, than added together to preform a pixel. Different filters would give different result to teh convolving an image. As for the o valid signal, pixel counter will determine how many input pixels have already flowed into the buffer(maxium of 1031), and row counter will deal with the offset during row transition. 5 additional pipeline registers are added to match the number of registers in o y path to secure the correctness.

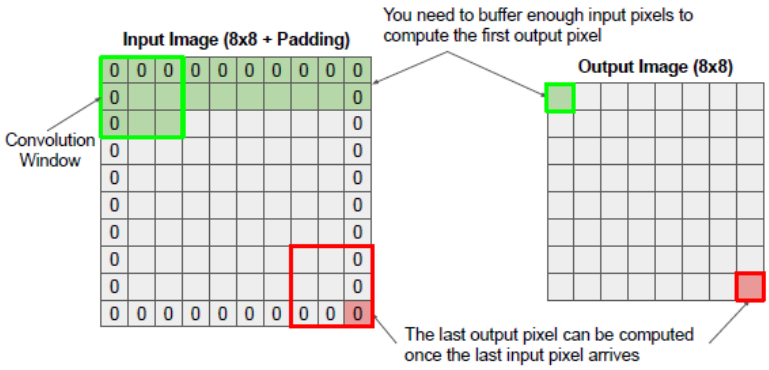

Figure 3: Convolution operation example

2.4 Correctness

The FPGA implementation successfully completes all test cases (Appendix A).

Test	Completed?
1	Yes
2	Yes
3a	Yes
3b	Yes
4a	Yes
4b	Yes
5a	Yes
5b	Yes
6a	Yes
6b	Yes
7a	Yes
7b	Yes

Table 1: FPGA implementation correctness

2.5 Performance Metrics

	Result (Appendix B)
ALM Utilization	2220
DSP Utilization	9
BRAM (M20K) Utilization	0
Maximum Operating Frequency (MHz)	442.28
Cycles for Test 7a (Hinton)	264243
Dynamic Power for one module @ maximum frequency (W)	108.09×10^{-3}
Throughput of one module (GOPS)	7.466
Throughput of a full device (GOPS)	1.251×10^{3}
Total Power for a full device (W)	12.66

Table 2: FPGA implementation results

Details calculations are included in Appendix B.

3.0 CPU and GPU Implementations

3.1 Operations

Both the CPU and GPU versions compute the image–filter convolution, but they differ in how the work is carried out on their respective hardware. On the CPU, without dedicated convolution hardware, the code relies on nested loops over filters, rows, and columns and, if vectorized, over the filter window to perform multiply-accumulate operations. The GPU version emphasizes memory hierarchy and parallel execution—using CUDA cores, thread blocks, and shared memory—to reduce explicit nesting and improve throughput through better data reuse and concurrency.

3.2 Preformance Metrics

From the table above, we can see that for all the test case given, runtime increases with the number of filters linearly. At every optimization level, the vectorized implementation finishes faster than the basic one—except with O3. The speedup comes from doing fewer explicit nested loops and more efficient arithmetic. With higher compiler optimizations (O2, O3), the compiler applies more aggressive transforms—e.g., deeper loop unrolling and late-stage instruction optimizations—reducing runtime further. The basic version, having more loops, tends to benefit disproportionately from O3, which narrows the gap with the vectorized code. Enabling multithreading yields substantial gains when the number of filters is large, cutting

	Runtime (ms) (Appendix C)			
No. of filters	1	4	16	64
GPU	0.0146194	0.0382884	0.129539	0.494356
CPU (basic - no opt - 1 thread)	6.06443	24.4438	97.0654	389.269
CPU (vectorized - no opt - 1 thread)	3.20216	13.6765	51.7357	208.745
CPU (basic - O2 - 1 thread)	1.44004	7.59237	28.6857	86.6324
CPU (vectorized - O2 - 1 thread)	0.844237	3.37243	13.7614	54.3744
CPU (basic - O3 - 1 thread)	0.528666	2.11186	8.45428	33.8729
CPU (vectorized - O3 - 1 thread)	0.850539	3.3954	13.8491	55.3561
CPU (basic - O3 - 4 threads)	0.529679	2.12084	8.43355	33.806
CPU (vectorized - O3 - 4 threads)	0.846091	3.37604	13.4985	54.034

Table 3: Runtimes of CPU and GPU convolution implementations

runtimes by up to 4× as work is split across CPU cores. Compared with the CPU, the GPU is faster overall due to its massive parallelism and memory hierarchy tuned for convolution-style workloads—effectively like running on a CPU with far more threads.

4.0 Evaluations

Device	Model	Process Tech.	Die Size (mm^2)	TDP (W)
CPU	Intel core i7-11700 (8 cores)	14nm	276	65
GPU	Nvidia GeForce RTX3070	8nm	393	220
FPGA	Arria 10 GX 1150	20nm	∼400	70

Table 4: Specification of the three compute platforms

	Throughput (GOPS)	Power (W)	Energy Efficiency (GOPS/W)	Area Efficiency (GOPS/mm^2)
FPGA (20nm)	1.2510×10^3	11.98	104.42	3.5743
GPU (28nm)	15.786	165	0.095673	0.39864
GPU (scaled 20nm)	22.1	247.5	0.089293	0.055808
CPU (22nm)	163.72	84	1.9490	0.92497

Table 5: Device evaluations

Base on the results, FPGA performs way better than CPU and GPU. This is mostly caused by FPGA is only performing the calclation that directly contribute to teh convolutions. CPU and GPU involves with lots of other operations that handles with the control flow and other setting which are required for the convolution.

Additionally, the CPU and GPU power figures reflect theoretical upper limits and are unlikely to match the actual draw during our runs. In contrast, the FPGA power numbers are more representative, coming from a power analysis that uses realistic toggle rates derived from a ModelSim simulation.

Furthermore, the area metrics also amplify the FPGA's advantage: compared with the CPU and GPU, a far larger fraction of the FPGA's counted area is actively dedicated to the convolution itself, rather than to general-purpose overhead needed by CPUs/GPUs.

In conclusion, the FPGA throughput and power figures are likely optimistic, since they are extrapolated from a single module. Scaling to near-full device utilization would not be linear, routing congestion and suboptimal placement can raise power and push down achievable clock frequency.

5.0 Appendix

Appendix A: FPGA Correctness

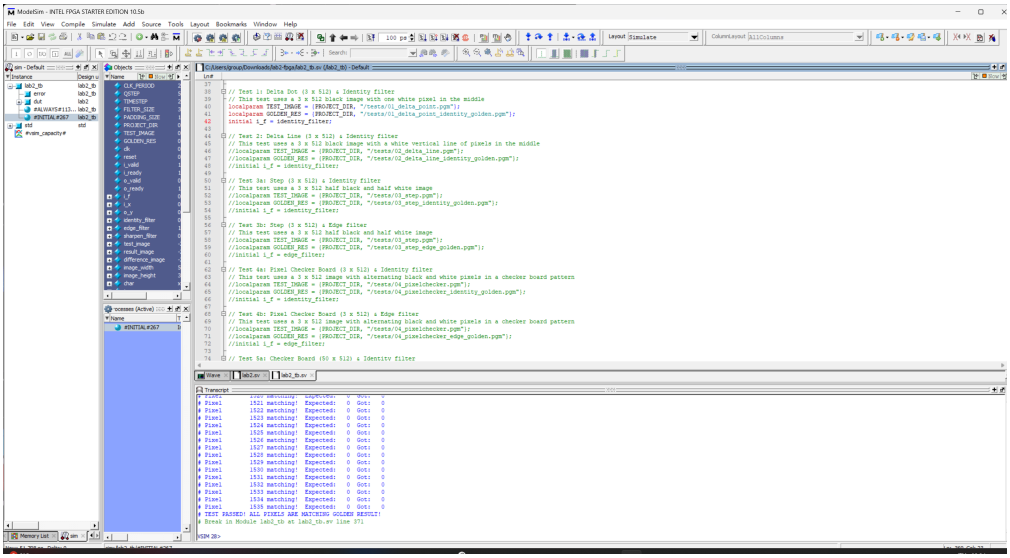

Figure 4: FPGA test 1 test bench

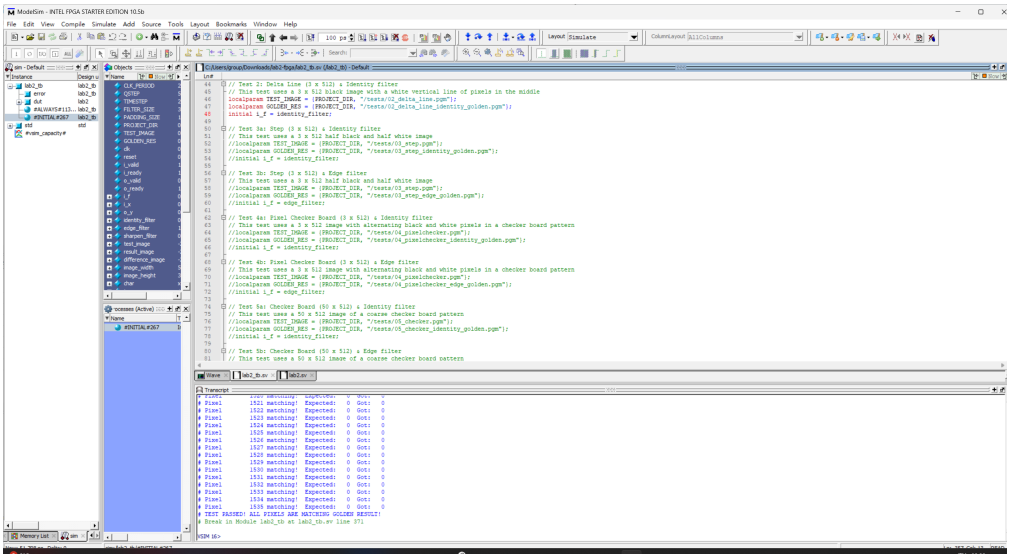

Figure 5: FPGA test 2 test bench

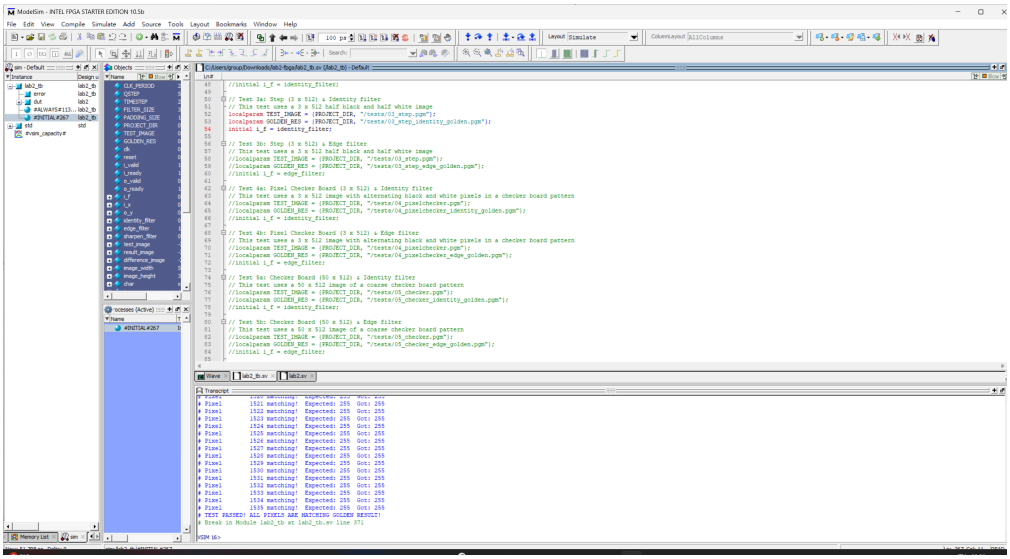

Figure 6: FPGA test 3a test bench

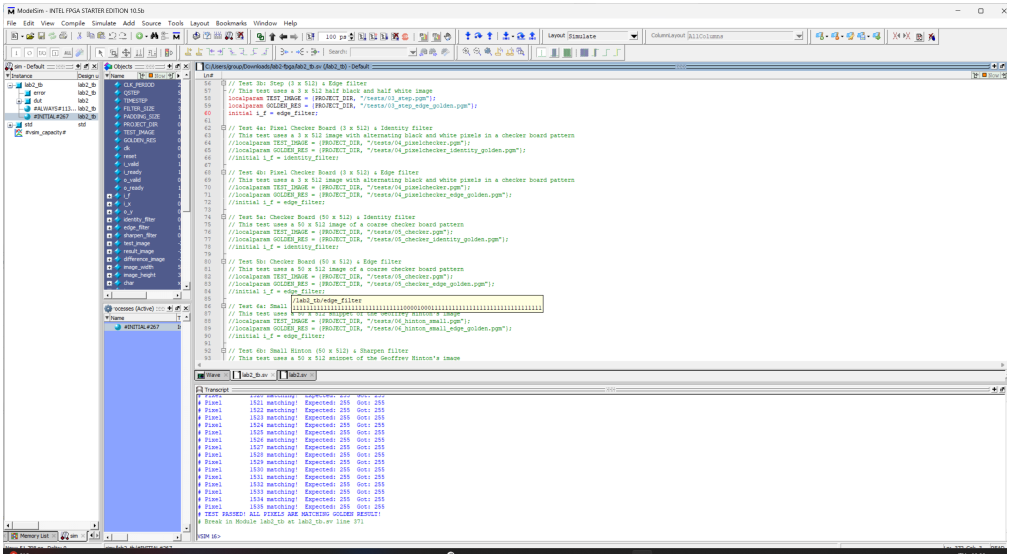

Figure 7: FPGA test 3b test bench

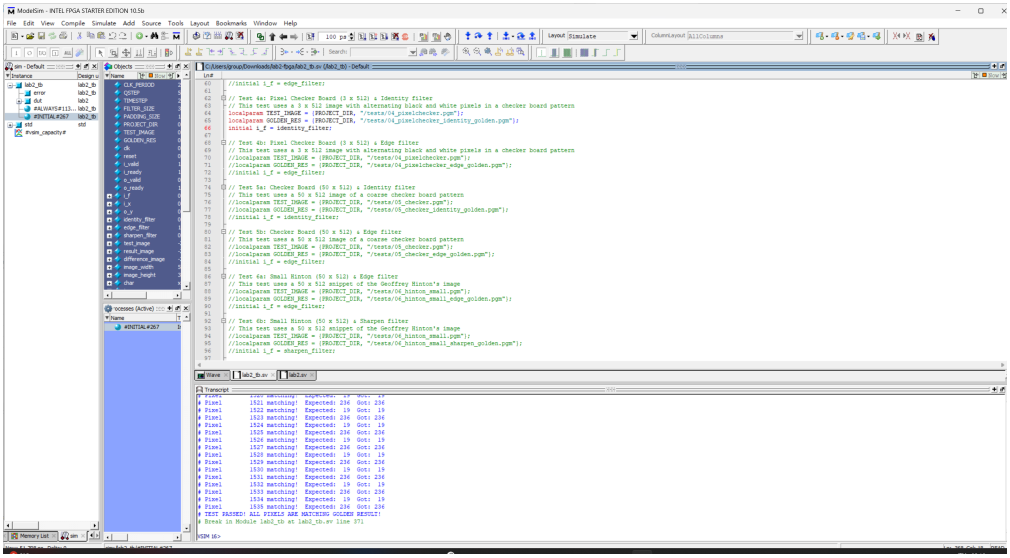

Figure 8: FPGA test 4a test bench

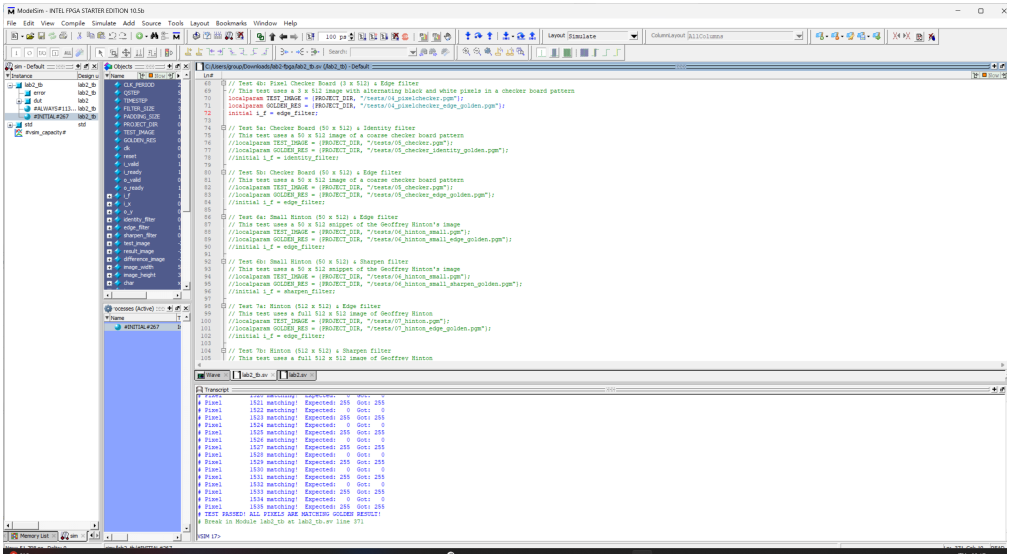

Figure 9: FPGA test 4b test bench

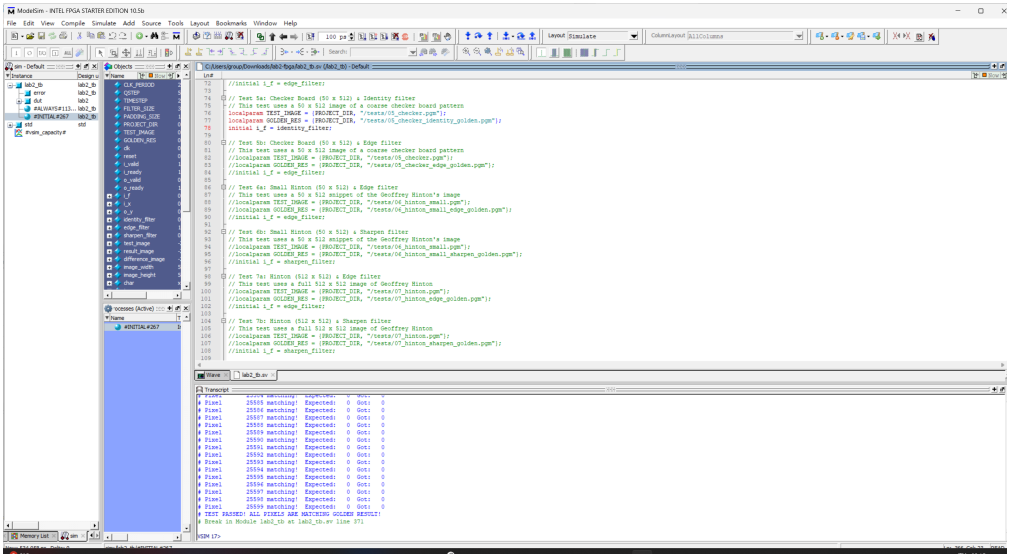

Figure 10: FPGA test 5a test bench

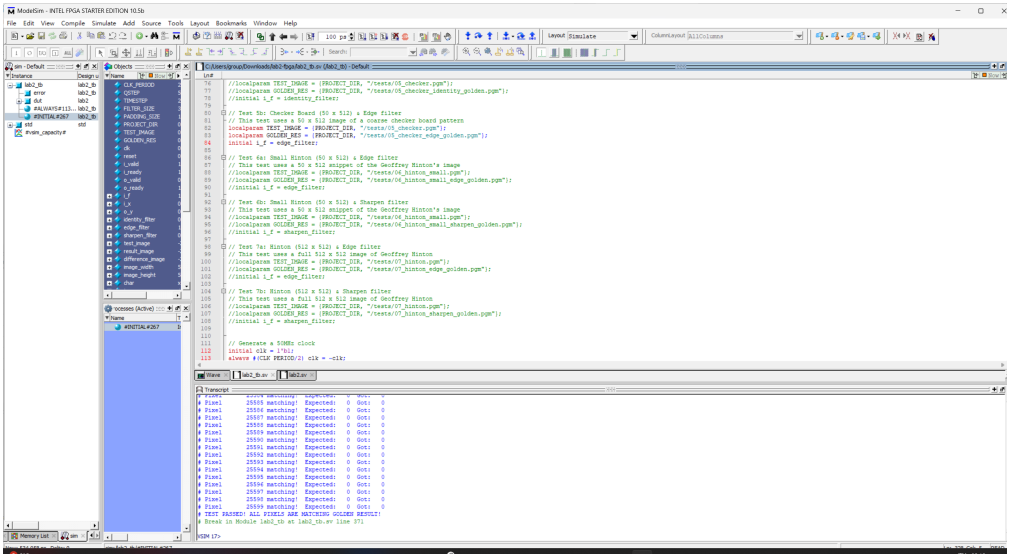

Figure 11: FPGA test 5b test bench

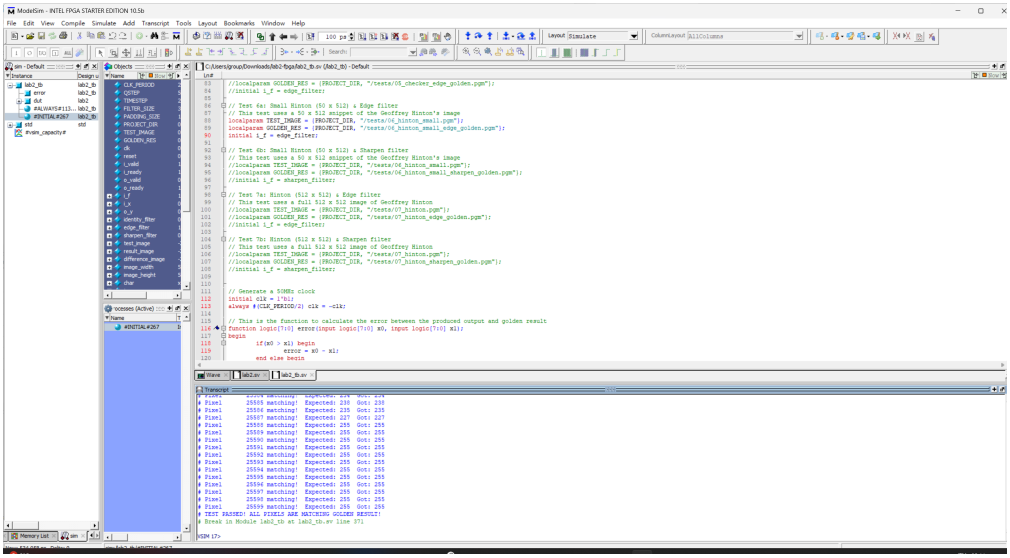

Figure 12: FPGA test 6a test bench

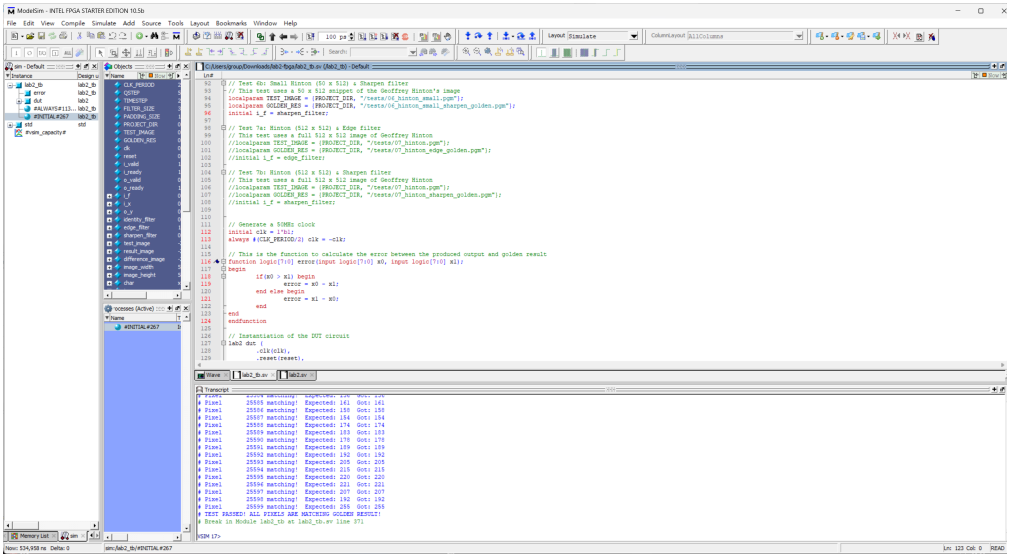

Figure 13: FPGA test 6b test bench

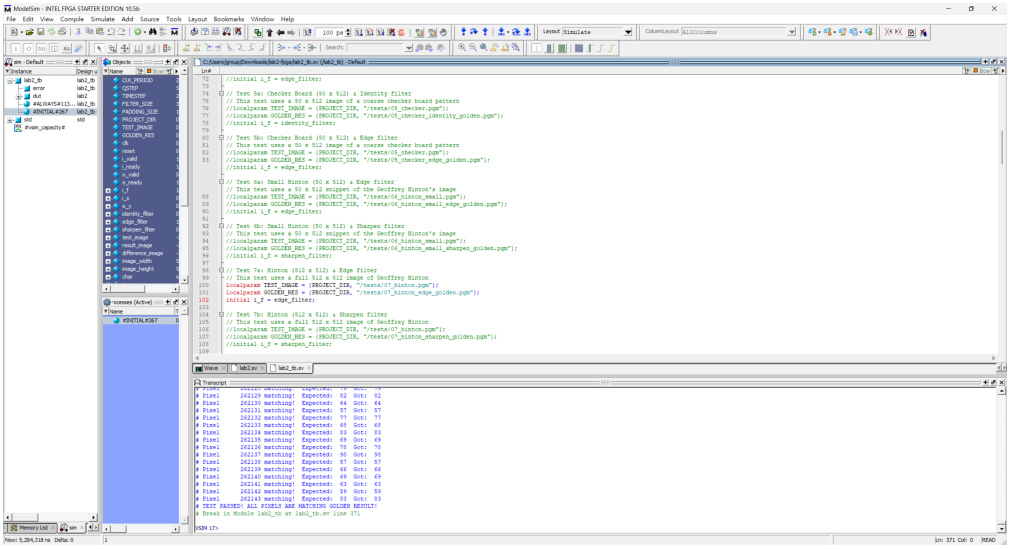

Figure 14: FPGA test 7a test bench

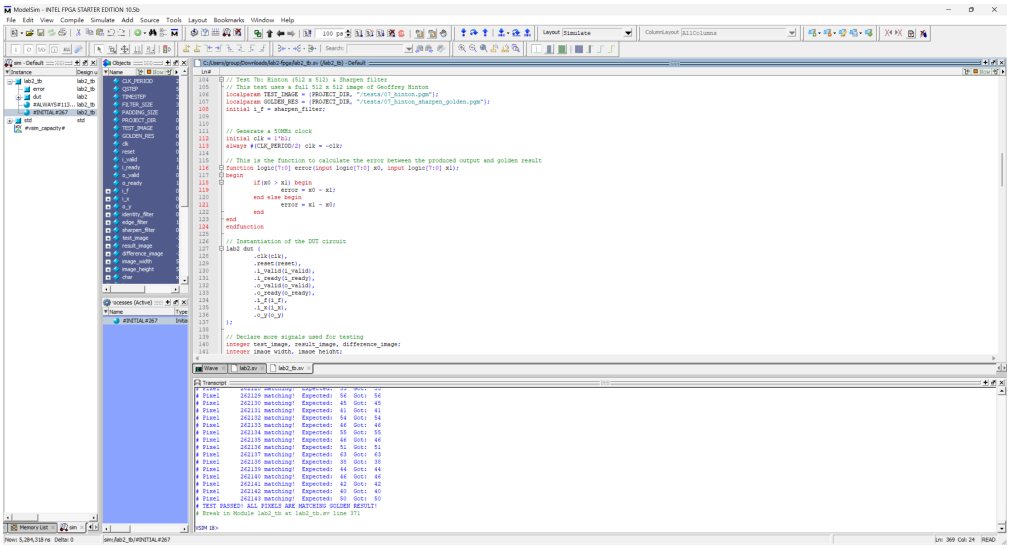

Figure 15: FPGA test 7b test bench

Appendix B: FPGA Implementation Results Calculations

ALM, DSP, BRAM Utilization

Flow Status	Successful - Tue Nov 04 17:40:45 2025
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Standard Edition
Revision Name	lab2
Top-level Entity Name	lab2
Family	Arria 10
Device	10AX115N1F45I1SG
Timing Models	Final
Logic utilization (in ALMs)	2,220 / 427,200 (< 1 %)
Total r	Logic utilization (in ALMs)
Total pins	1 / 992 (< 1 %)
Total virtual pins	93
Total block memory bits	0 / 55,562,240 (0 %)
Total DSP Blocks	9 / 1,518 (< 1 %)
Total HSSI RX channels	0 / 48 (0 %)
Total HSSI TX channels	0 / 48 (0 %)
Total PLLs	0 / 112 (0 %)

Figure 16: FPGA ALM, DSP and BRAM utilization

Maximum Operating Frequency

```
Analyzing Slow 900mV 100C Model
332148 Timing requirements not met
332146 Worst-case setup slack is -1.078
332146 Worst-case hold slack is 0.045
332140 No Recovery paths to report
332140 No Removal paths to report
332146 Worst-case minimum pulse width slack is -1.150
Analyzing Slow 900mV 0C Model
332146 Worst-case setup slack is -1.261
332146 Worst-case hold slack is 0.045
332140 No Recovery paths to report
```

Figure 17: FPGA setup and hold times

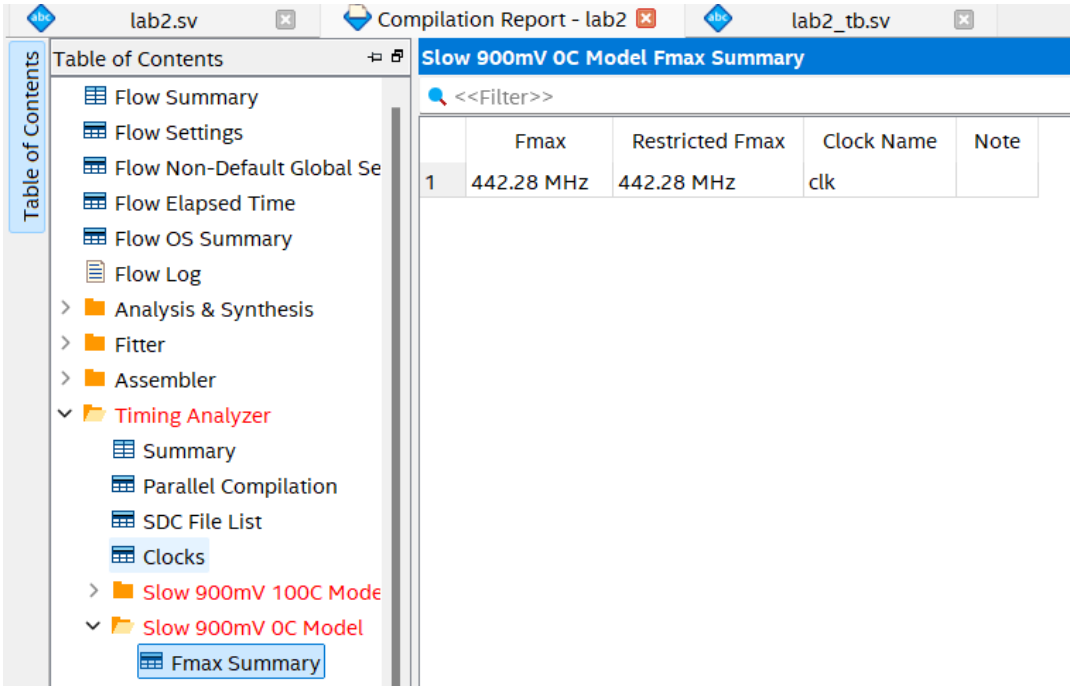

Figure 18: FPGA maximum operating frequency

Cycles for Test 7a (Hinton)

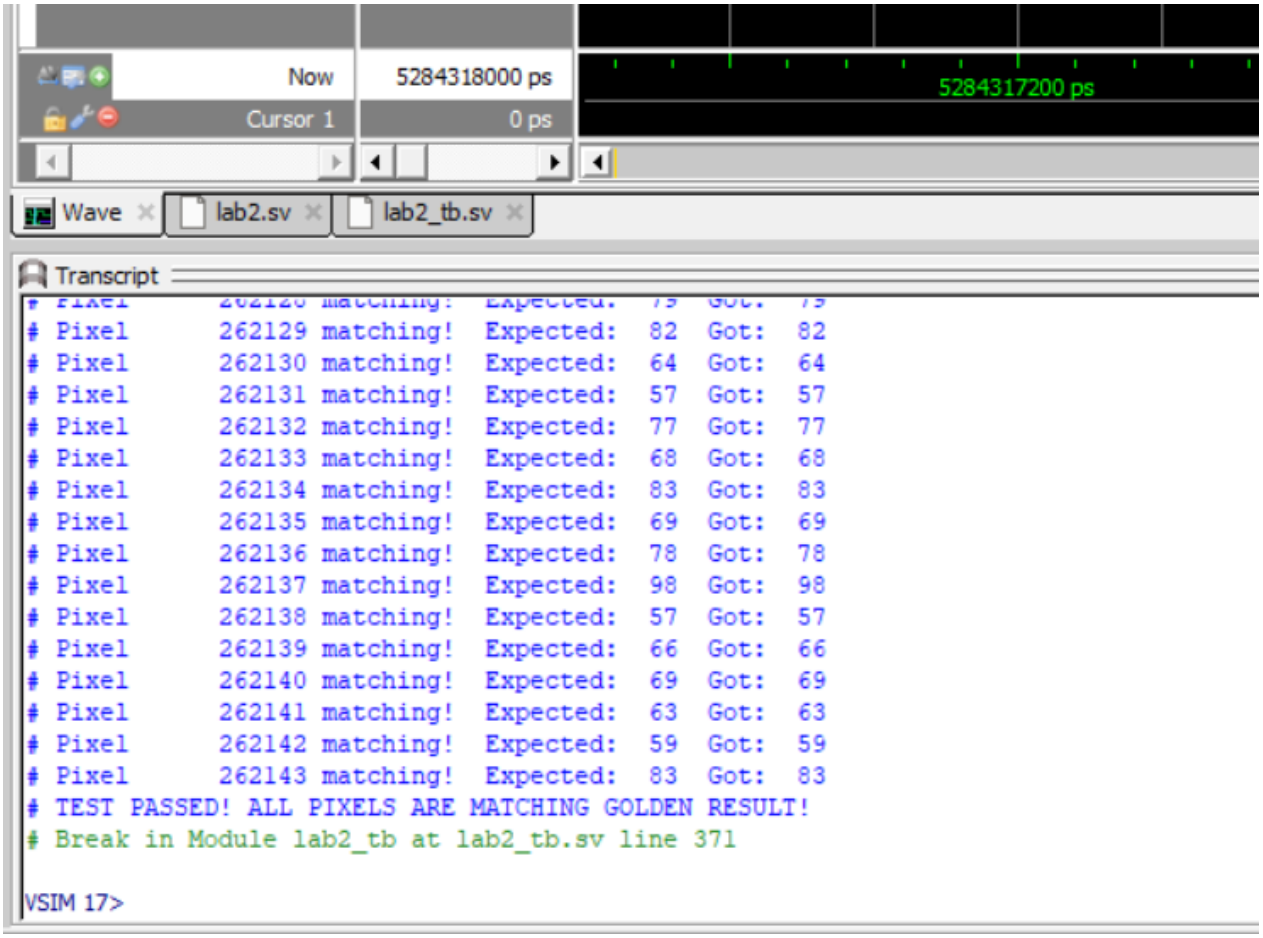

Figure 19: Test 7a total elapsed time

$$No.\ of\ cycles = 5284318000/20000 = 264216\ cycles$$

Dynamic Power for one module @ 442.28MHz

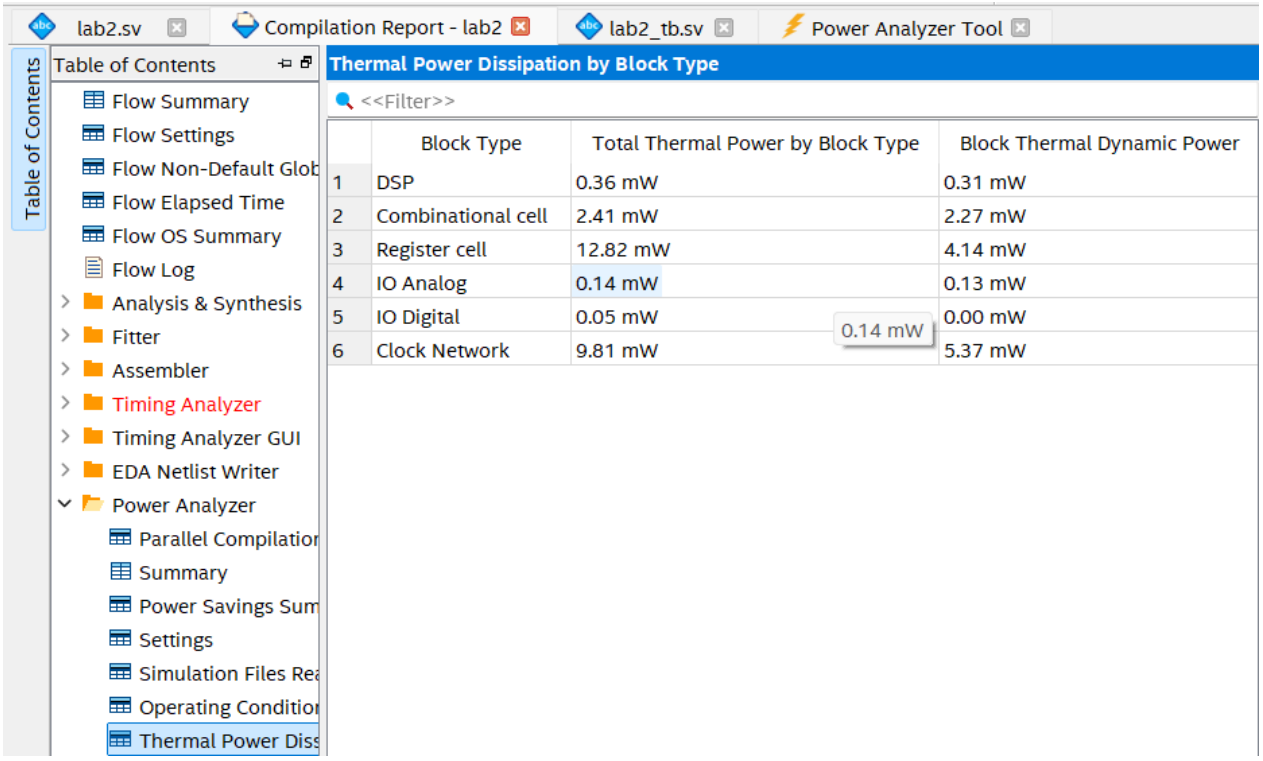

Figure 20: Power dissipation by block type

$$Total\ Dynamic\ Power\ @50MHz = 0.31 + 2.27 + 4.14 + 0.13 + 5.37 = 12.22\ mW$$

$$Total\ Dynamic\ Power\ @441.5MHz = 12.22 * (442.28/50) = 108.09\ mW = 108.09 \times 10^{-3}\ W$$

Throughput of one module (GOPS)

To compute the convolution of one pixel, 9 multiplications and a total of 8 additions needs to be performed over a period of 5 cycles (output ready by the 6th cycle). For a 512x512 picture as is given in test 7a, that would take a total of:

$$(512 * 512) * (9 + 8) = 4456448\ operations$$

With a frequency of 442.28 MHz, 264216 cycles is equal to:

$$(1/441.5 \times 10^{6}) * 264216 \approx 5.9845 \times 10^{-4}\ seconds$$

Which give us a total of:

$$4456448/5.9845 \times 10^{-4} \approx 7.4466 \times 10^{9} = 7.4466\ GOPs$$

Throughput of a full device (GOPS)

$$Max.\ copies\ (based\ on\ ALM\ count) = 427200/2220 \approx 192$$
$$Max.\ copies\ (based\ on\ DSP\ count) = 1518/9 \approx 168$$

Since DSP count is the limiting factor, approximate maximum number of copies per device is 168. As a result, the throughput of a full device:

$$7.4466 \times 10^{9} * 168 \approx 1.2510 \times 10^{12} = 1.2510 \times 10^{3}\ GOPs$$

Total Power for a full device (W)

Assuming a fully packed device with 168 modules as calculated before (note that IO and clock dynamic power are not multiplied:

$$Total\ Dynamic\ Power\ @50MHz = (0.31 + 2.27 + 4.14) * 168 + 0.13 + 5.37 = 1134.46\ mW$$

$$Total\ Dynamic\ Power\ @442.28MHz = 1134.46 * (442.28/50) = 10034.98\ mW$$
$$Total\ Power\ @442.28MHz = 10034.98 + (0.36 + 2.41 + 12.82) * 168 + 0.14 + 0.05 + 9.81$$
$$= 12664.1\ mW \approx 12.66\ W$$

Appendix C: Runtimes for CPU and GPU implementations

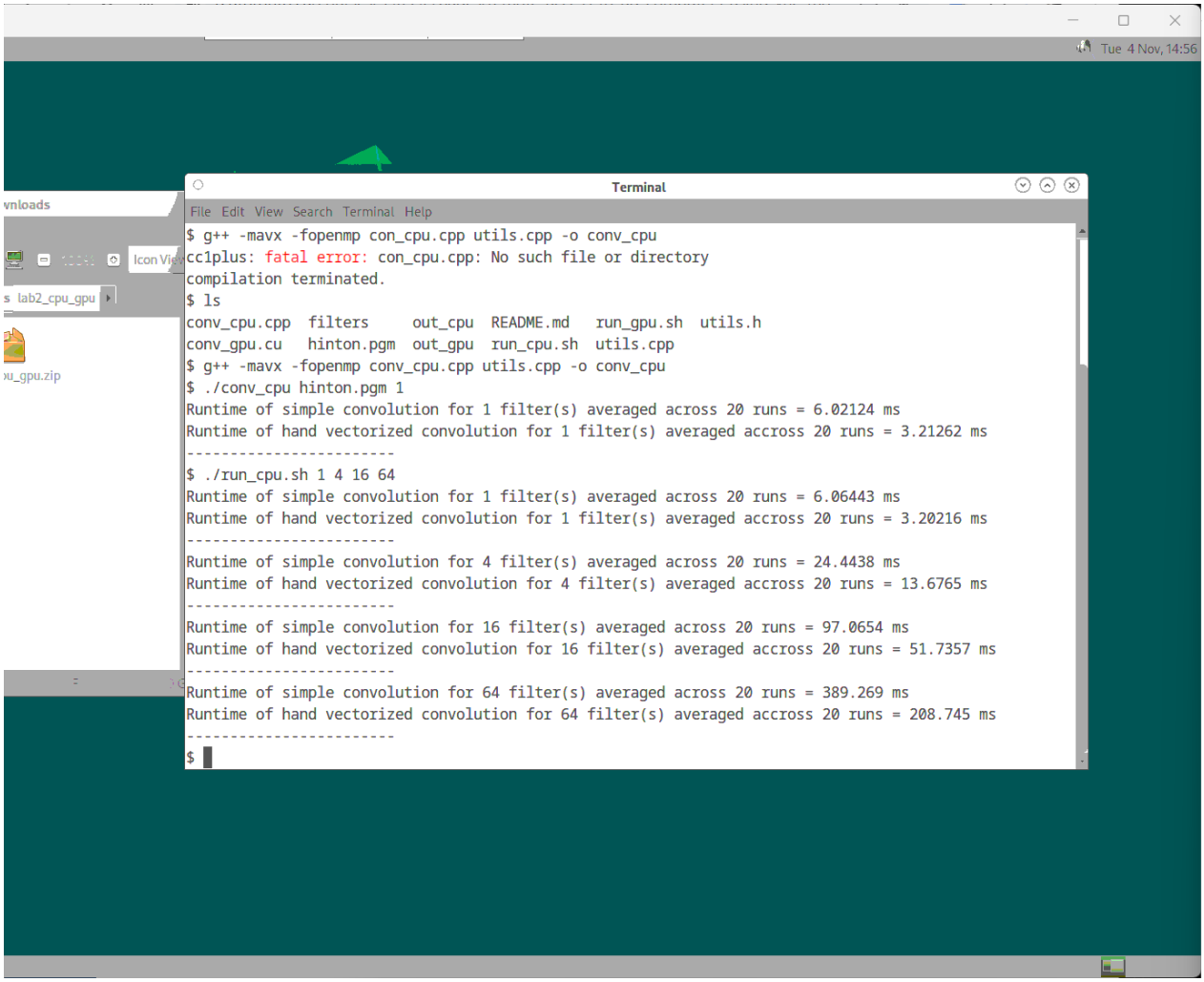

Figure 21: CPU no optimization runtimes

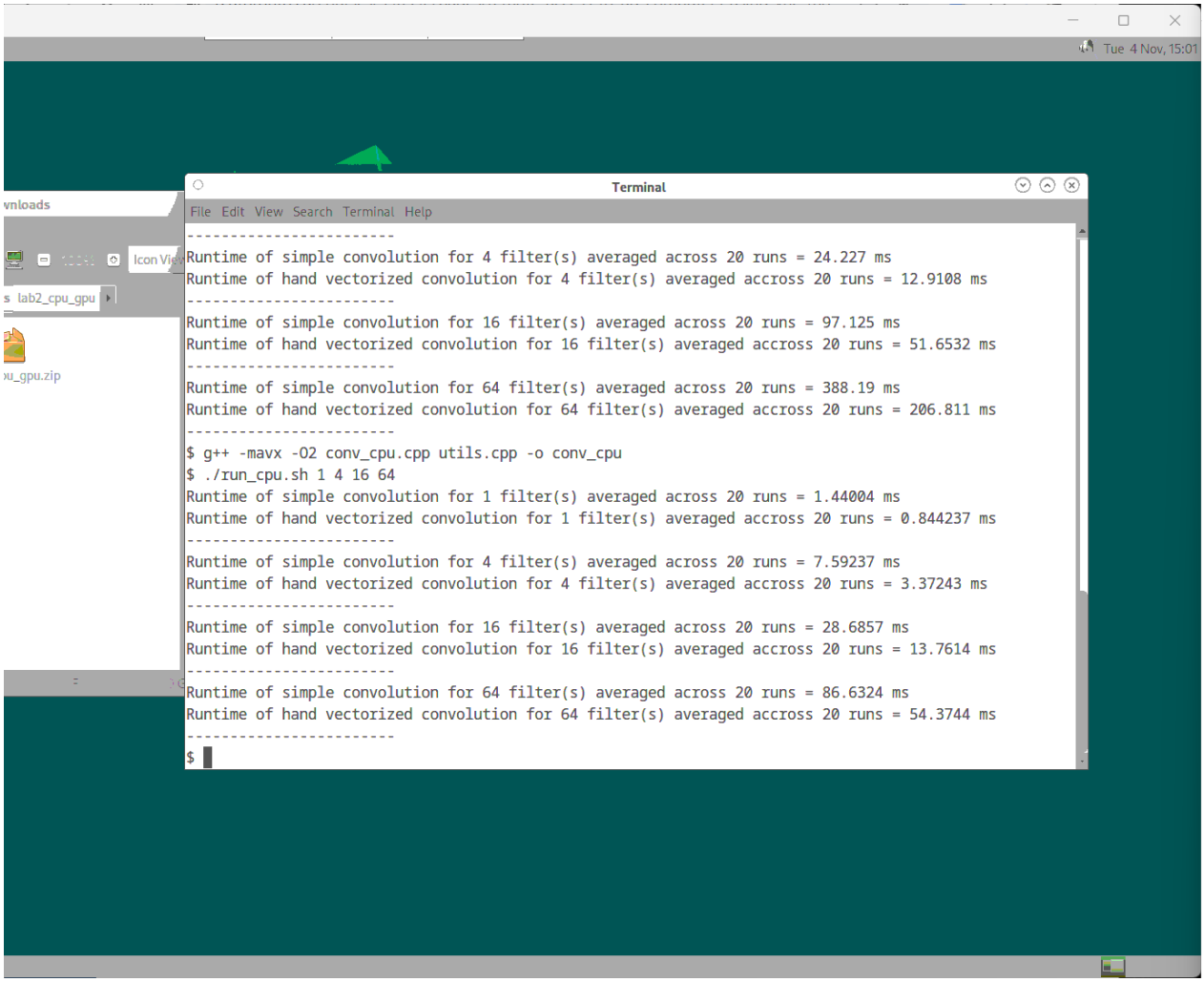

Figure 22: CPU runtimes with O2 optimization

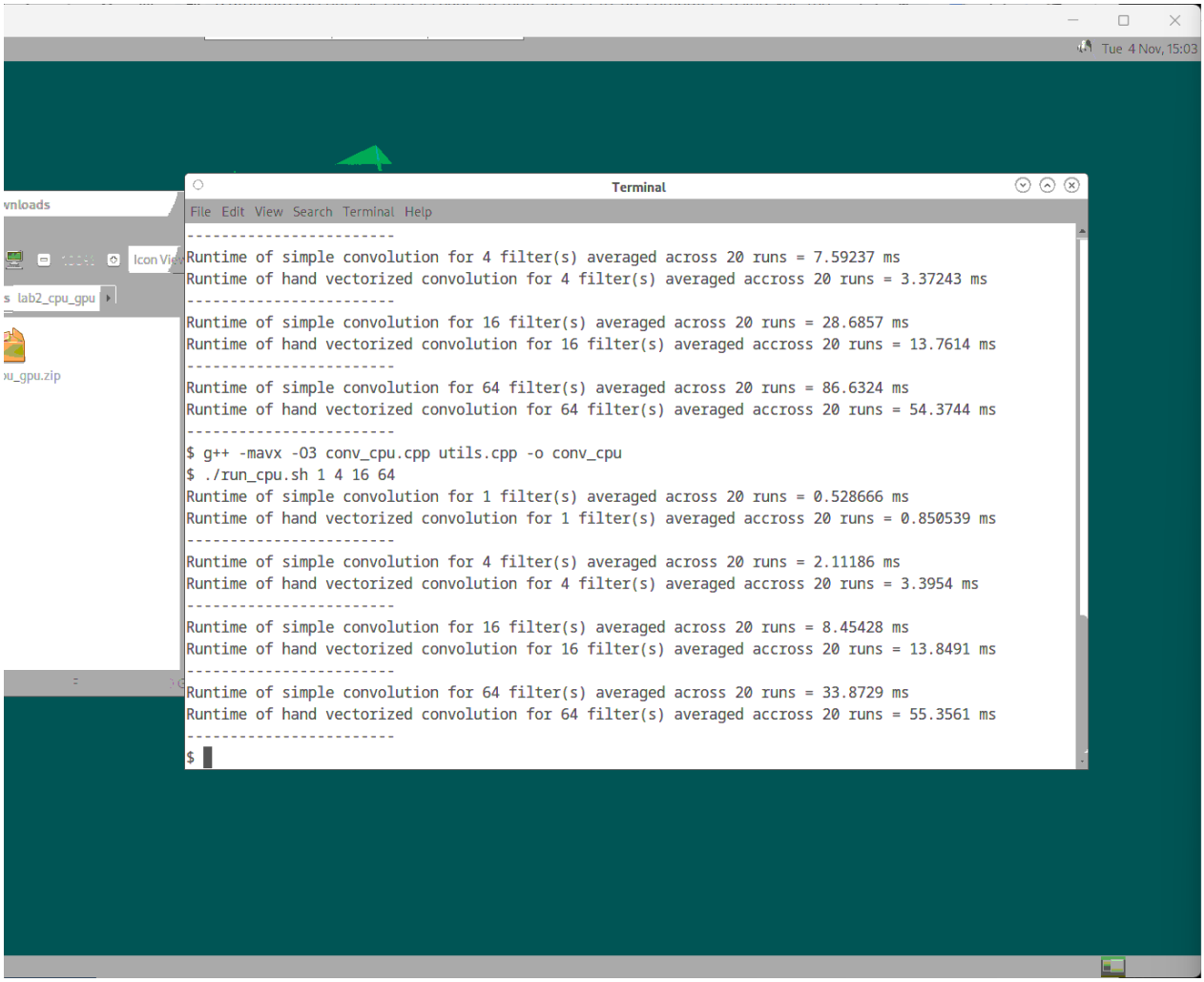

Figure 23: CPU runtimes with O3 optimization

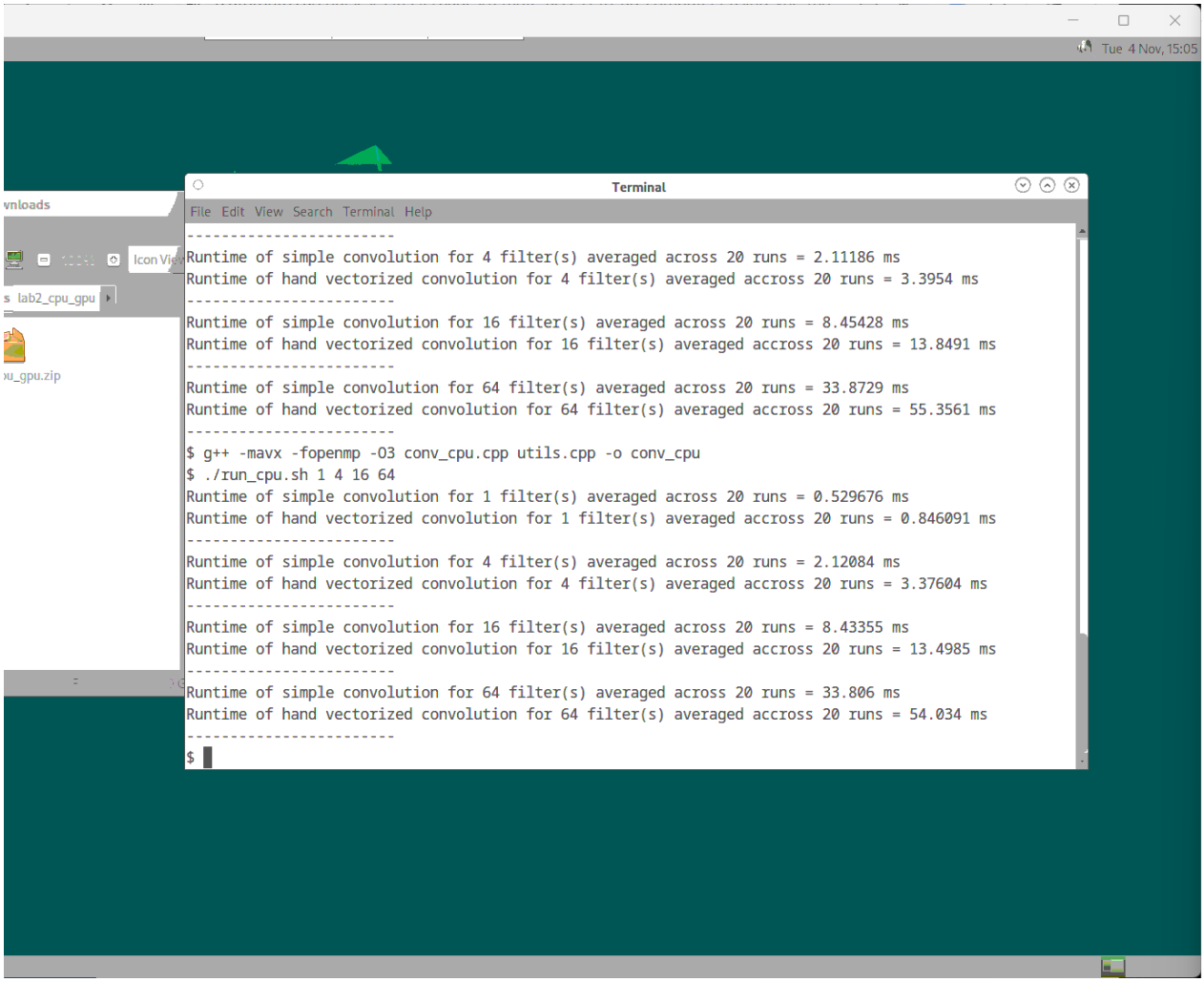

Figure 24: CPU runtimes with O3 optimization, multithreaded

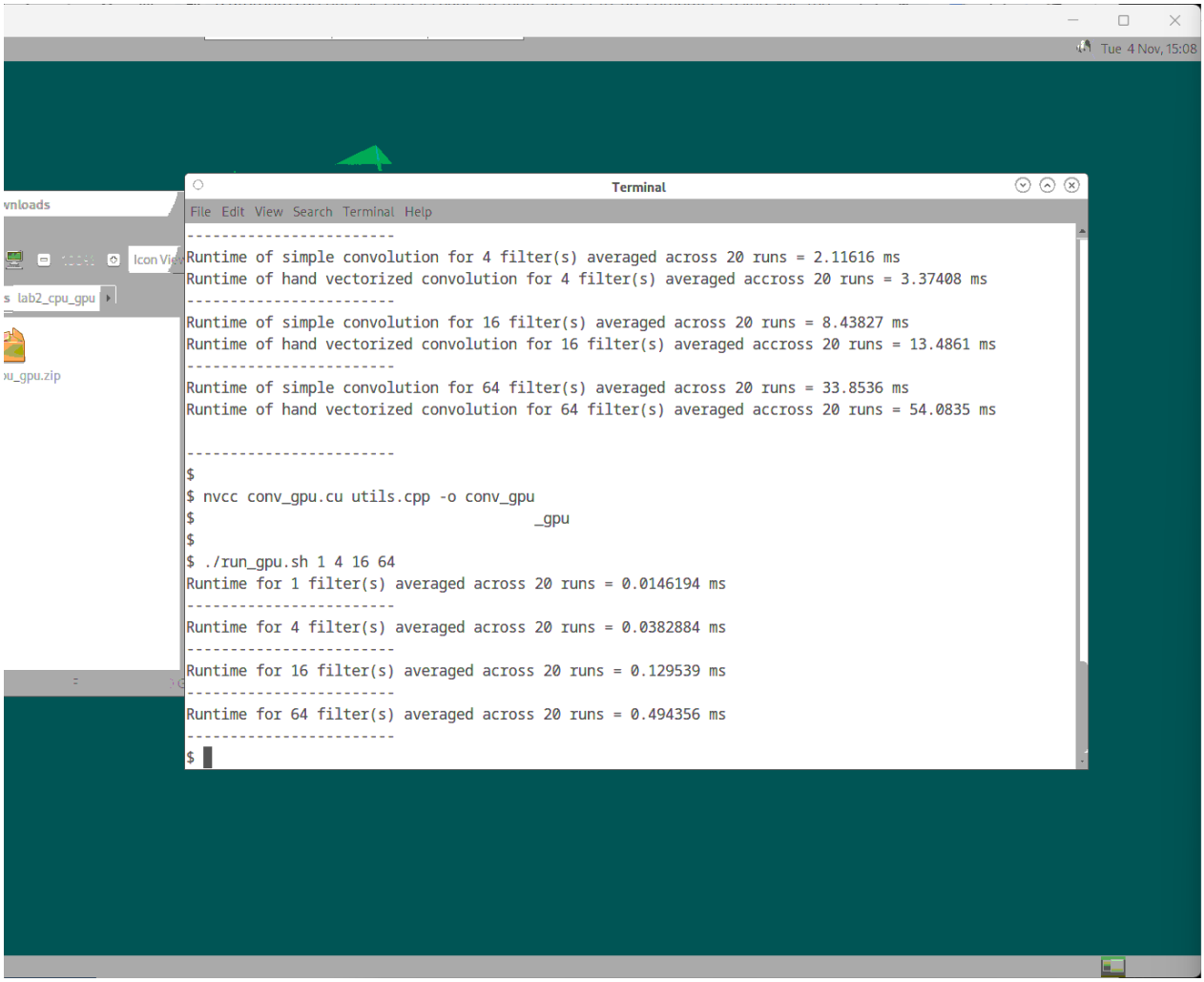

Figure 25: GPU runtimes

Appendix D: Evaluation Calculations

FPGA related

$$Energy\ Efficiency = 1.2510 \times 10^3 / 11.98 \approx 104.42\ GOPS/W$$
$$Area\ Efficiency = 1.2510 \times 10^3 / 400 \approx 3.1275\ GOPS/mm2$$

CPU related

```
/*******************************************************************************************
 * This function implements the 2D convolution simply using nested for loops assuming that  *
 * the input image is already padded. It takes as an input pointers for input and output    *
 * images as well as convolution filter coefficients and number of filters used.           *
 *******************************************************************************************/
void cpu_conv2d(float* input_image, float* filter, float* output_image, int image_width, int image_height, int num_filters){
    // Declare variables used throughout the function
    float accum;
    int xx, yy;

    // Calculate the padded image size
    int padded_image_width = image_width + (2*FILTER_RADIUS);

    for(int filter_id = 0; filter_id < num_filters; filter_id++){ // For each filter
        for(int image_y = 0; image_y < image_height; image_y++){ // For each y location of the output image
            for(int image_x = 0; image_x < image_width; image_x++){ // For each x location of the output image
                // Start a new accumulation result
                accum = 0;

                // Compute the dot-product of the filter and the input image
                for(int filter_y = 0; filter_y < FILTER_SIZE; filter_y++){
                    for(int filter_x = 0; filter_x < FILTER_SIZE; filter_x++){
                        xx = image_x + filter_x;
                        yy = image_y + filter_y;
                        accum += input_image[(yy * padded_image_width) + xx] * filter[(filter_id * FILTER_SIZE * FILTER_SIZE) + (filter_y * FILTER_SIZE) + filter_x];
                    }
                }

                // Store the convolution result in the corresponding pixel location
                output_image[(filter_id * image_width * image_height) + (image_y * image_width) + image_x] = accum;
            }
        }
    }
}
```

Figure 26: CPU basic code

Based on the figure above, we can estimate the number of operations that are performed for the convolution in the basic implementation on the CPU. Number of operations within the most inner loop = 11. FILTER_SIZE = 3 image_height = image_width = 512 + 2 num_filters = 64 Margin of error = 10%

$$Total\ Operations = ((((11 * 3 * 3) + 5) * 514 * 514 * 64) * 1.10)/11.8149 \times 10^{-3}$$
$$\approx 163.72\ GOPS$$

$$Energy\ Efficiency = 163.72/65 \approx 2.518\ GOPS/W$$
$$Area\ Efficiency = 163.72/276 \approx 0.593\ GOPS/mm2$$

GPU related

```
global   void gpu conv2d(int* input image, int* output image, int image width, int image height) {

      // Allocate space in shared memory to load an image tile
        shared   int image tile[TILE LOAD SIZE][TILE LOAD SIZE];

      // Declare variables to use throughout the kernel
      int thread id, tile location x, tile location y;
      int image location x, image location y, pixel id;

      // 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;

              // Load pixels
              if (tile location y < TILE LOAD SIZE && tile location x < TILE LOAD SIZE){
                      if (image location y >= 0 && image location y < image height
                          && image location x >= 0 && image location x < image width){
                              image tile[tile location y][tile location x] = input image[pixel id];
                      } else {
                              image tile[tile location y][tile location x] = 0;
                      }
              }
      }
        syncthreads();
```

Figure 27: GPU code

Based on the figure above, we can estimate the number of operations that are performed for the convolution in the implementation on the GPU. Number of operations within loop = 15 TILE_SIZE = 32 TILE_LOAD_SIZE = 32 + 3 - 1 = 34

```
// Perform the 2D convolution
int accum = 0;
int y, x, z;
z = blockIdx.z;
for (y = 0; y < FILTER SIZE; y++) {
        for (x = 0; x < FILTER SIZE; x++) {
                accum += image tile[threadIdx.y + y][threadIdx.x + x] * device filter[(z * FILTER SIZE * FILTER SIZE) + (y * FILTER SIZE) + x];
        }
}

// Write the output
y = blockIdx.y * TILE SIZE + threadIdx.y;
x = blockIdx.x * TILE SIZE + threadIdx.x;
if (y < image height && x < image width){
        output image[(z * image width * image height) + (y * image width) + x] = accum;
}
  syncthreads();
```

Figure 28: GPU code

Number of operations within loop = 9 FILTER_SIZE = 3 Above repeated for each pixel. Number of operations outside loop = 9 Margin of error = 10%

$$Total\ Operations = (15*((34*34)/(32*32))+(9*3*3)*512*512+9)*1.10/1.479\,65\times10^{-3}$$

$$\approx 15.786\ GOPS$$

$$Energy\ Efficiency = 15.786/165 \approx 0.095673\ GOPS/W$$

$$Area\ Efficiency = 15.786/393 \approx 0.04016\ GOPS/mm2$$

GPU Estimation

$$Throughput = 15.786 * (8/20) \approx 6.31\ GOPS$$

$$Power = 165 * 1.5 \approx 247.5\ W$$

$$Energy\ Efficiency = 22.1/247.5 \approx 0.089293\ GOPS/W$$

$$Area\ Efficiency = 22.1/393 \approx 0.05623\ GOPS/mm2$$

Appendix E: .sv file for FPGA Implementation

```
// 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)localparam int FILTER_SIZE = 3;

// 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, buf_i, i; // variables to use in the for loop



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

```
                        end
                end
        end
end*/



// Start of your code

localparam IMG_SIZE = 512 + 2; // size of the image plus padding(512 + 2)
localparam BUFFER_SIZE = IMG_SIZE*2 + FILTER_SIZE;
logic signed [PIXEL_DATAW-1:0] i_buffer [BUFFER_SIZE];

logic[19:0] pixel_counter; // counter to record how many pixels have been added to the buffer
logic[9:0] row_counter; // row counter to deal with the offset of adding padding into pixel_counter

logic valid_stage [0:4];

// output woule be stored from multiplier and adder
logic signed [15:0] mult [0:8];
logic signed [18:0] add_1 [0:3];
logic signed [18:0] add_2 [0:1];
logic signed [18:0] add_3;

// The maxium bits from the last adder will be 20 bits
logic signed [19:0] add_final;

// capture output to 8 bits (0-255) grayscale
logic unsigned [7:0] cap_final;

// pipeline register to send and hold the value
logic signed [18:0] mult_reg [0:8];
logic signed [18:0] add_1_reg [0:3];
logic signed [18:0] add_2_reg [0:1];
logic signed [18:0] add_3_reg;
logic signed [18:0] add_wait [0:2]; // wait signal for add 1 to 3

always_ff @(posedge clk) begin
        // If reset signal is high, set all the filter coefficient registers to zeros
```

```
90          // We're using a synchronous reset, which is recommended style for
            ↪  recent FPGA architectures
91          if(reset)begin
92
93                  for(row = 0; row < FILTER_SIZE; row = row + 1) begin
94                          for(col = 0; col < FILTER_SIZE; col = col + 1) begin
95                                  r_f[row][col] <= 0;
96                          end
97                  end
98
99                  for(buf_i = 0; buf_i < BUFFER_SIZE; buf_i = buf_i + 1)begin
100                         i_buffer[buf_i] <= '0;
101                 end
102
103                 pixel_counter <= '0; // reset both counters
104                 row_counter <= '0;
105                 valid_stage[0] <= '0; // reset valid signal
106
107
108         // Otherwise, register the input filter coefficients into the 2D
            ↪  array signal
109         end else begin
110                 for(row = 0; row < FILTER_SIZE; row = row + 1) begin
111                         for(col = 0; col < FILTER_SIZE; col = col + 1) begin
112                                 // Rearrange the 72-bit input into a 3x3
                                    ↪  array of 8-bit filter coefficients.
113                                 // signal[a +: b] is equivalent to
                                    ↪  signal[a+b-1 : a]. You can try to plug
                                    ↪  in
114                                 // values for col and row from 0 to 2, to
                                    ↪  understand how it operates.
115                                 // For example at row=0 and col=0: r_f[0][0]
                                    ↪  = i_f[0+:8] = i_f[7:0]
116                                 //               at row=0 and col=1:
                                    ↪  r_f[0][1] = i_f[8+:8] = i_f[15:8]
117                                 r_f[row][col] <= i_f[(row * FILTER_SIZE *
                                    ↪  PIXEL_DATAW)+(col * PIXEL_DATAW) +:
                                    ↪  PIXEL_DATAW];
118                         end
119                 end
120
121                 // prepare for input buffer
```

```
122                 // the newest input will be put input_buffer[0]
123                 // rest will be shifted by 1
124                 if(i_valid) begin
125                         i_buffer[0] <= i_x;
126                         pixel_counter <= pixel_counter +1;
127 
128                         for(buf_i = 0; buf_i < BUFFER_SIZE - 1; buf_i =
                            ↪  buf_i + 1) begin
129                                 i_buffer[buf_i+1] <= i_buffer[buf_i]; //
                                    ↪  shift all pixels by 1 in the
                                    ↪  buffer
130                         end
131 
132                         row_counter <= row_counter + 1; // to know which row
                            ↪  we are at
133 
134                         // sends the output from  multipiler or adder to the
                            ↪  pipeline register
135 
136                         for (i = 0; i < 9; i++) begin // multipiler
137                                 mult_reg[i] <= mult[i];
138                         end
139 
140                         for (i = 0; i < 4; i++) begin // adder 1
141                                 add_1_reg[i] <= add_1[i];
142                         end
143                         add_wait[0] <= mult_reg[0];
144 
145                         for (i = 0; i < 2; i++) begin // adder 2
146                                 add_2_reg[i] <= add_2[i];
147                         end
148                         add_wait[1] <= add_wait[0];
149 
150                         add_3_reg <= add_3; // adder 3
151                         add_wait[2] <= add_wait[1];
152 
153 
154                         // Capture the final output to 8 bits
155                         if(add_final > 20'sd255) begin
156                                 cap_final <= 8'd255;
157                         end else if(add_final < 20'sd0)begin
158                                 cap_final <= 8'd0;
```

```
159                         end else begin
160                                 cap_final <= add_final[7:0];
161                         end
162
163                         // send the valid signal using pipline register
164                         valid_stage[1] <= valid_stage[0];
165                         valid_stage[2] <= valid_stage[1];
166                         valid_stage[3] <= valid_stage[2];
167                         valid_stage[4] <= valid_stage[3];
168                 end
169
170                 // buffer for convolution
171                 if(pixel_counter >= BUFFER_SIZE)begin
172                         // addition output caused by two padding when
                            ↪  transfer
173                         // rows stall valid signal for two cycles
174                         if(row_counter == 'd1) begin
175                                 valid_stage[0] = 1'b0;
176                         end else if(row_counter == 'd2)begin
177                                 valid_stage[0] = 1'b0;
178                         end else begin
179                                 valid_stage[0] <= 1'b1;
180                         end
181                 end
182
183                 if(row_counter == IMG_SIZE - 1)begin
184                         // Reset row counter for new rows
185                         row_counter <= 'd0;
186                 end
187
188         end
189
190 end
191
192
193 mult8 mult_layer0_8 (.i_a(i_buffer[0]), .i_b(r_f[2][2]), .i_o(mult[8]));
194 mult8 mult_layer0_7 (.i_a(i_buffer[1]), .i_b(r_f[2][1]), .i_o(mult[7]));
195 mult8 mult_layer0_6 (.i_a(i_buffer[2]), .i_b(r_f[2][0]), .i_o(mult[6]));
196 mult8 mult_layer0_5 (.i_a(i_buffer[IMG_SIZE]), .i_b(r_f[1][2]),
    ↪  .i_o(mult[5]));
197 mult8 mult_layer0_4 (.i_a(i_buffer[IMG_SIZE + 1]), .i_b(r_f[1][1]),
    ↪  .i_o(mult[4]));
```

```
198 mult8 mult_layer0_3 (.i_a(i_buffer[IMG_SIZE + 2]), .i_b(r_f[1][0]),
    ↪  .i_o(mult[3]));
199 mult8 mult_layer0_2 (.i_a(i_buffer[IMG_SIZE*2]), .i_b(r_f[0][2]),
    ↪  .i_o(mult[2]));
200 mult8 mult_layer0_1 (.i_a(i_buffer[IMG_SIZE*2 + 1]), .i_b(r_f[0][1]),
    ↪  .i_o(mult[1]));
201 mult8 mult_layer0_0 (.i_a(i_buffer[IMG_SIZE*2 + 2]), .i_b(r_f[0][0]),
    ↪  .i_o(mult[0]));
202
203 add19 add_layer1_3 (.i_a(mult_reg[8]), .i_b(mult_reg[7]), .i_o(add_1[3]));
204 add19 add_layer1_2 (.i_a(mult_reg[6]), .i_b(mult_reg[5]), .i_o(add_1[2]));
205 add19 add_layer1_1 (.i_a(mult_reg[4]), .i_b(mult_reg[3]), .i_o(add_1[1]));
206 add19 add_layer1_0 (.i_a(mult_reg[2]), .i_b(mult_reg[1]), .i_o(add_1[0]));
207
208 add19 add_layer2_1 (.i_a(add_1_reg[3]), .i_b(add_1_reg[2]), .i_o(add_2[1]));
209 add19 add_layer2_0 (.i_a(add_1_reg[1]), .i_b(add_1_reg[0]), .i_o(add_2[0]));
210
211 add19 add_layer3_0 (.i_a(add_2_reg[1]), .i_b(add_2_reg[0]), .i_o(add_3));
212
213 add19 add_layer4_0 (.i_a(add_3_reg), .i_b(add_wait[2]), .i_o(add_final));
214
215 assign o_y = cap_final;
216 assign o_valid = valid_stage[4];
217 assign o_ready = i_ready;
218 // End of your code
219
220 endmodule
221
222 module mult8 (
223         input unsigned [7:0] i_a,
224         input signed [7:0] i_b,
225         output signed[15:0] i_o
226 );
227         assign i_o = $signed({1'b0, i_a}) * i_b;
228 endmodule
229
230 module add19(
231         input  signed [18:0] i_a,
232         input  signed [18:0] i_b,
233         output signed [19:0] i_o
234 );
235         assign i_o = i_a + i_b;
```

```
endmodule
```