1.0 Introduction

The following document serves as the lab report for assignment 2 of ECE 1756 (Fall 2021). This report is written by Sheng Zhao (1003273913).

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

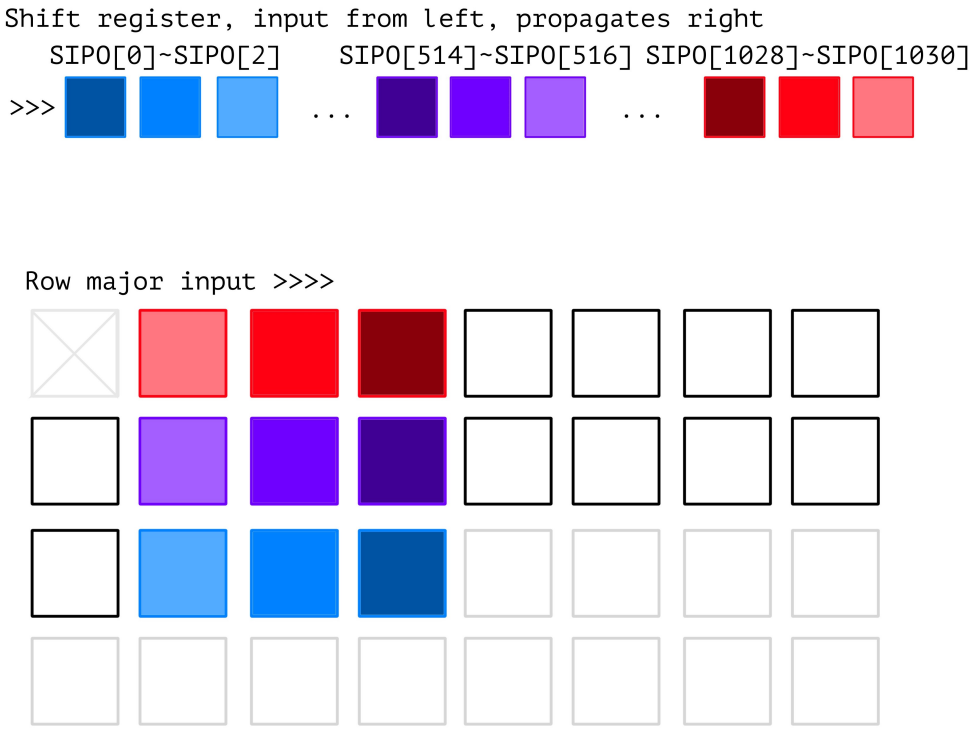


Figure 1: Data input and shift register

The goal of the convolution engine implementation on the FPGA is to compute the convolutions on a 512 pixel wide, any height image, along with a provided 3x3 filter. As shown in the bottom half of Fig. 1, image data is given in a row major format, sending in pixel by pixel starting with the top left. Convolution is then to be performed on every 3x3 set of pixels.

Given the row major input format, in order to compute convolutions on pixels that span across multiple rows, pixel data must be held for a period of time. To achieve this, a Serial-in, Parallel-out (SIPO) shift register is implemented. Given that the length of the image is fixed at 514 pixels (512 wide image with 0 padding), the minimum required length of the SIPO is 1031. Input pixels given in row major format will enter from the left (top half of Fig. 1), with the data propagating to the next register every clock cycle.

To better illustrate the pixels involved the convolution, consider an example as shown in Fig. 1. All the pixels required for the computation is highlighted in color, each with a different shade to better illustrate their position in the SIPO as the order is important in the correct calculation of the convolution.

2.2 FPGA Datapath

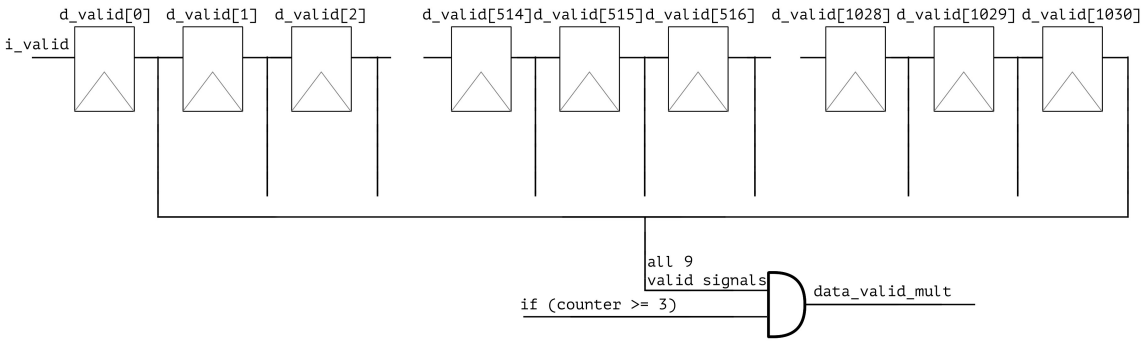


Figure 2: Data valid datapath

With the insight from Section 2.1, the datapaths for the FPGA implementation is created. A streaming dataflow model is chosen for the implementation and computations are pipelined as much as possible to enable the highest operating frequency possible.

The datapath for the computation is presented in Fig. 3. The datapath has 9 multipliers performing the 9 multiplications in parallel while the outputs are summed in an adder tree. Data is pipelined in between each stage of multiplier and adders.

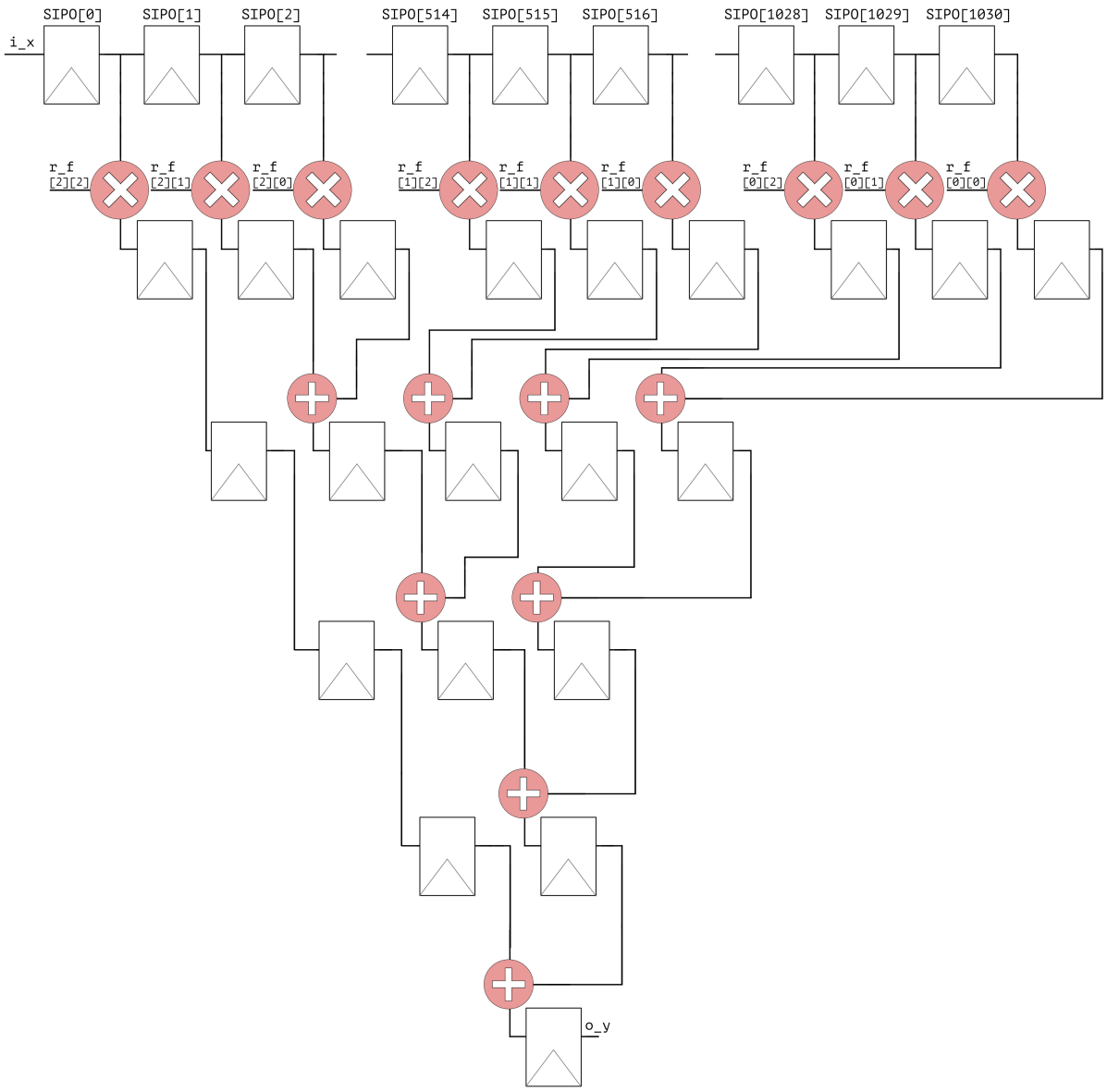


Figure 3: Data datapath

In a similar fashion, Fig. 2 shows a SIPO shift register for the data valid bit. From this, 9 bits corresponding to the 9 pixels involved in the convolution are connected to an AND gate along with a counter that keeps track of the current input's column position. While not shown explicitly, the data_valid_mult output signal from the AND gate is propagated along with the data from multiplier and adder stages in sync.

2.3 Operation

During operation, input data and its valid bit is fed into the SIPO shift register, beginning with the 1st position (SIPO[0], data_valid[0]), as long as it is valid and the downstream circuit is ready (i.e. i_valid and i_ready both high). With each clock cycle, data in the shift register propagates down and the same happens for the valid bits that correspond to each pixel (i.e. SIPO[i] = SIPO[i-1], data_valid[i] = data_valid[i-1]).

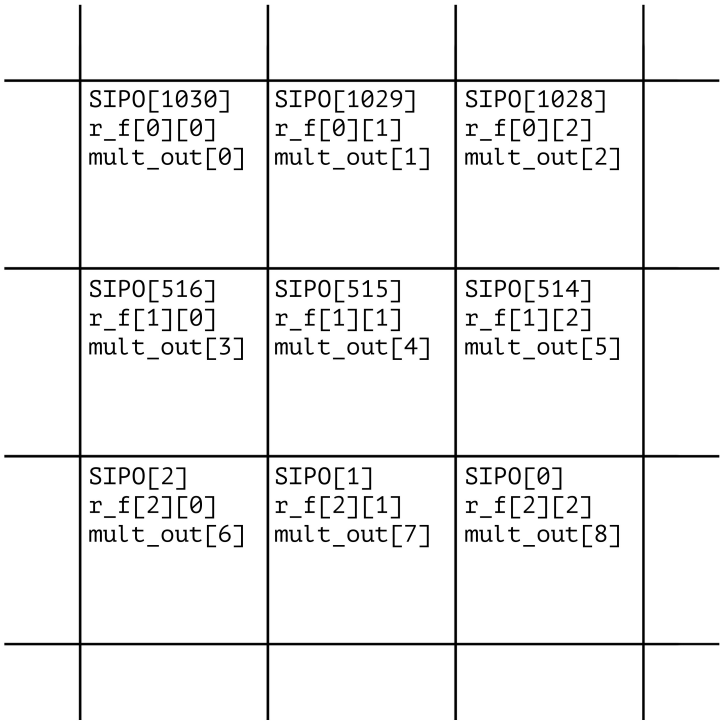


Figure 4: Convolution filter and pixel mapping

The multiply and accumulate portion of the circuit samples the 9 required pixels and the respective data valid bits from the SIPO shift registers according to Fig. 4 and performs the computations continuously regardless of the validity of the pixels sampled. Instead, the valid bits are checked separately to ensure that all 9 pixels are valid. In addition, an additional check is performed on a counter, one that keeps track of the column position of the current input. This is to ensure that convolution is not being done when the first two pixels of each row is inserted, since that would be invalid (i.e. instead of performing a 3x3 convolution, a 3x2 and 3x1 or 3x1 and 3x2 set of pixels spanning 4 rows will be used instead). Once the AND gate from Fig. 2 returns true, the output of the multipliers are now valid and this knowledge/status is passed along with the computed result via a separate set of valid bit registers that accompanies the multiplier and adder stages in sync with the computed values.

Due to the fact that filter values are signed and pixel values are not. Pixel values are padded

```
// Pipeline reg for adder stage 4 out, final result, clipping performed here
if (addr_stage_4_out > 32'sd255) begin
   addr_stage_4_out_pipe <= 8'd255;
end else if (addr_stage_4_out < 32'sd0) begin
   addr_stage_4_out_pipe <= 8'd0;
end else begin
   addr_stage_4_out_pipe <= addr_stage_4_out[7:0];
end
d
```

Figure 5: Systemverilog clipping

with a 0 in the MSB position to convert it to signed and ensure that the multiplication performed is signed as well. Results are also stored in a sufficiently wide register to ensure no overflow occurs throughout the multiplier and adder stages. Once the final addition is performed, the values are clipped to be between 0 and 255 as specified by the assignment (Fig. 5).

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 Design Space Exploration

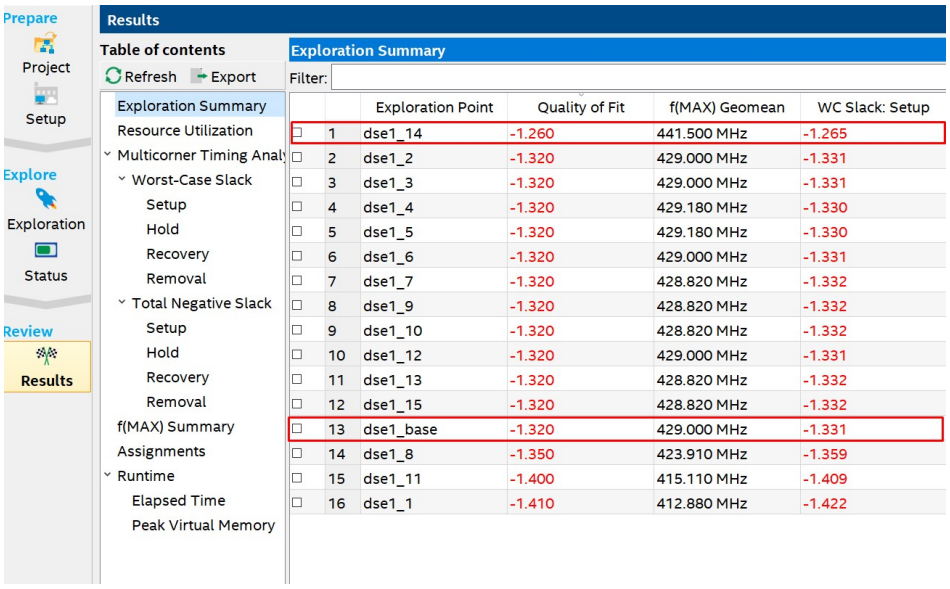


Figure 6: Design space exploration optimization

The design is further optimized using Intel's Design Space Explorer. A total of 15 different

seeds were explored in search of a better performance. The final design chosen is des1_14 as shown in Fig. , which has a faster maximum operating frequency than the base compile.

2.6 Performance Metrics

	Result (Appendix B)
ALM Utilization	2497
DSP Utilization	9
BRAM (M20K) Utilization	0
Maximum Operating Frequency (MHz)	441.5
Cycles for Test 7a (Hinton)	264216
Dynamic Power for one module @ maximum frequency (W)	111.7878×10^{-3}
Throughput of one module (GOPS)	7.4466
Throughput of a full device (GOPS)	1.2510×10^{3}
Total Power for a full device (W)	11.98

Table 2: FPGA implementation results

Details regarding resource utilization and performance for the FPGA implementation is highlighted in Table 2, calculations are included in Appendix B.

3.0 CPU and GPU Implementations

3.1 Operations

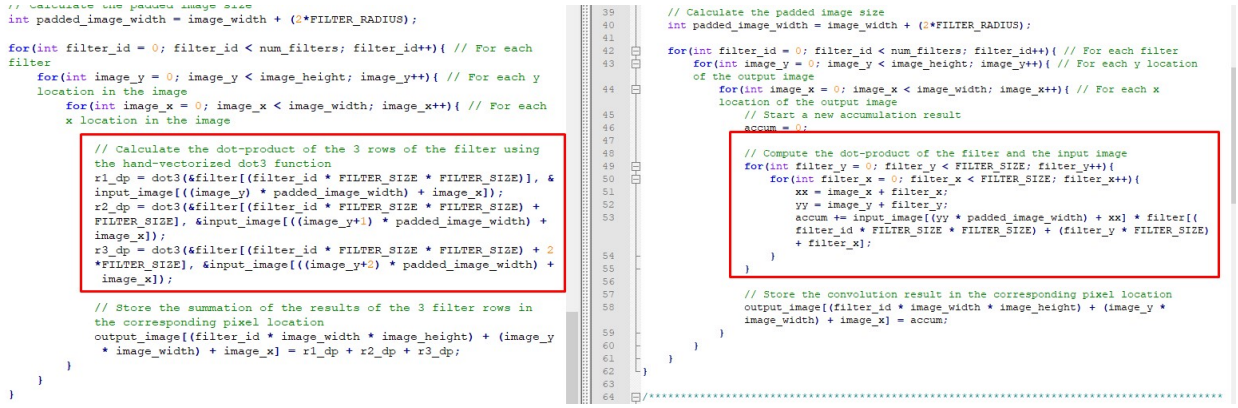
The CPU and GPU implementations are similar in function in that both computes the convolution of the image and filter given. There are minute differences in the way that the computation is implemented due to the different hardware that the code will be executed on. On the CPU, since there are no special hardware available to perform the mutliply and accumulate operations, for loops are used extensively to loop through the number of filters, rows and columns of the image and subsequently, depending on whether if the operation is vectorized, the filter as well. On the other hand, the implementation on GPU involves more optimization in terms of memory management as well as utilizing the CUDA cores effciently, leading to less nested loops.

3.2 Performance Metrics

	Runtime (ms) (Appendix C)			
No. of filters	1	4	16	64
GPU	0.0297056	0.0994058	0.372583	1.47965
CPU (basic - no opt - 1 thread)	8.82178	35.0252	139.794	558.436
CPU (vectorized - no opt - 1 thread)	5.34393	21.2444	87.1801	340.192
CPU (basic - O2 - 1 thread)	1.59544	6.37035	25.4466	101.752
CPU (vectorized - O2 - 1 thread)	1.26072	5.05124	23.0627	83.6116
CPU (basic - O3 - 1 thread)	0.679928	2.72494	10.8303	43.27
CPU (vectorized - O3 - 1 thread)	1.25945	5.03768	23.019	83.5155
CPU (basic - O3 - 4 threads)	0.728284	0.823869	2.93434	11.8149
CPU (vectorized - O3 - 4 threads)	1.19086	1.71489	7.82391	22.2321

Table 3: Runtimes of CPU and GPU convolution implementations

From Table 3, with respect to CPU runtimes, we can see that for any given test case, runtimes increases with the number of filters roughly linearly. For any given optimization level, the vectorized version of convolution computation took less time to complete compared to the basic version unless the -O3 flag is enabled. This is due to more efficient calculations being done as shown by the fewer number of nested loops in Fig. 7. With respect to compilation optimizations (i.e. -O2 and -O3 flags), a higher level of optimization leads more aggressive optimizations being done by the compiler. These include unrolling more loops, and performing more aggressive instruction optimizations late in the compiler pipeline, all leading to lower runtimes. The basic version had more loops and likely benefited from the -O3 optimzation more than the vectorized version, thus leading to the lower runtimes. Finally, when multithreading is enabled, significant performance gain is seen in runs with higher



```
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 in the image
        for(int image_x = 0; image_x < image_width; image_x++){ // For each
        x location in the image

            // Calculate the dot-product of the 3 rows of the filter using
            the hand-vectorized dot3 function
            r1_dp = dot3(&filter[(filter_id * FILTER_SIZE * FILTER_SIZE)], &
            input_image[((image_y) * padded_image_width) + image_x]);
            r2_dp = dot3(&filter[(filter_id * FILTER_SIZE * FILTER_SIZE) +
            FILTER_SIZE], &input_image[((image_y+1) * padded_image_width) +
            image_x]);
            r3_dp = dot3(&filter[(filter_id * FILTER_SIZE * FILTER_SIZE) + 2
            *FILTER_SIZE], &input_image[((image_y+2) * padded_image_width) +
             image_x]);

            // Store the summation of the results of the 3 filter rows in
            the corresponding pixel location
            output_image[(filter_id * image_width * image_height) + (image_y
             * image_width) + image_x] = r1_dp + r2_dp + r3_dp;
        }
    }
}
```

```
39  // Calculate the padded image size
40  int padded_image_width = image_width + (2*FILTER_RADIUS);
41
42  for(int filter_id = 0; filter_id < num_filters; filter_id++){ // For each filter
43      for(int image_y = 0; image_y < image_height; image_y++){ // For each y location
        of the output image
44          for(int image_x = 0; image_x < image_width; image_x++){ // For each x
            location of the output image
45              // Start a new accumulation result
46              accum = 0;
47
48              // Compute the dot-product of the filter and the input image
49              for(int filter_y = 0; filter_y < FILTER_SIZE; filter_y++){
50                  for(int filter_x = 0; filter_x < FILTER_SIZE; filter_x++){
51                      xx = image_x + filter_x;
52                      yy = image_y + filter_y;
53                      accum += input_image[(yy * padded_image_width) + xx] * filter[(
                        filter_id * FILTER_SIZE * FILTER_SIZE) + (filter_y * FILTER_SIZE)
                        + filter_x];
54                  }
55              }
56
57              // Store the convolution result in the corresponding pixel location
58              output_image[(filter_id * image_width * image_height) + (image_y *
                image_width) + image_x] = accum;
59          }
60      }
61  }
62  }
63
64  //*****************************************************************************
```

Figure 7: CPU basic (right) vs vectorized (left) implementations

number of filters, reducing the runtime by up to 4 times. With more CPU resources being used, the amount of work done by each thread is reduced, thus leading to faster completion.

Comparing between GPU and CPU runtimes, we can see that GPU outperformed CPU by a significant amount due to the GPU being much more optimized for the calculations needed in a convolution calculation. There are also many more processing units upon which work can be divided compared to the CPU (i.e. this is equivalent to running on CPU with many many more threads), thus the lower runtime.

4.0 Evaluations

Device	Model	Process Tech.	Die Size (mm^2)	TDP (W)
CPU	Intel core i7-4790 (4 cores)	22nm	177	84
GPU	Nvidia GeForce GTX 980	28nm	396	165
FPGA	Arria 10 GX 1150	20nm	∼350	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

Based on the results in Table 5, FPGA outperforms CPU and GPU by a wide margin. This is mostly due to the fact that the calculations for throughput is computed with only the operations that directly contribute to the convolutions. As a result, the FPGA which only performs performs computation directly related to the convolution has a significant advantage (due to the streaming dataflow model it uses) over both the CPU and GPU which involves lots of other operations that handles the control flow (loops, ifs) and other setup required for the convolution computations to be carried out.

In addition, power data used for the calculation in the CPU and GPU cases are the theoretical upper limit of those devices and it is very unlikely that these devices actually consumed this much power for the operation. The power data used for FPGA however, is much more accurate as it comes from a power analysis with realistic toggle rates based off of an ModelSim simulation.

Furthermore, the area numbers also contributed to the significant advantage shown by the FPGA. Relative to the CPU and GPU, much more of the area used in the calculation for FPGA is actually involved in the convolution. This is due to the lack of overhead hardware required by the CPU and GPU to stay more generic/general purpose.

Finally, the throughput and power numbers used for FPGA may also be more optimistic than reality since they are estimates from a single module. Performance will likely not scale linearly, especially as the FPGA approaches full utilization. More power might be consumed and clock frequency will decrease due to non optimal routing/layout.

With regards to estimation for GPU at a 20nm process, it is first worth while to note that while a smaller process means more transistors can be fit within the same area, companies

rarely keep die sizes the same. Die sizes often change going from process to process as well as within each process as the technology matures and architecture design demand changes. However, assuming that we keep die sizes the same, and that the decrease in process does not lead to any additional complications with power (such as significantly increased current leakage), we can then expect the number of transistors to increase approximately linearly. Assuming proper utilization, we can expect this increase to be passed on to the throughput as well. Power, on the other hand, might scale less than ideally than linearly due to Dennard scaling breaking down beginning in 2006. With an arbitrary factor of 50% more power than linear scaling, estimates in Table 5 are calculated in Appendix D.

5.0 Appendix

Appendix A: FPGA Correctness

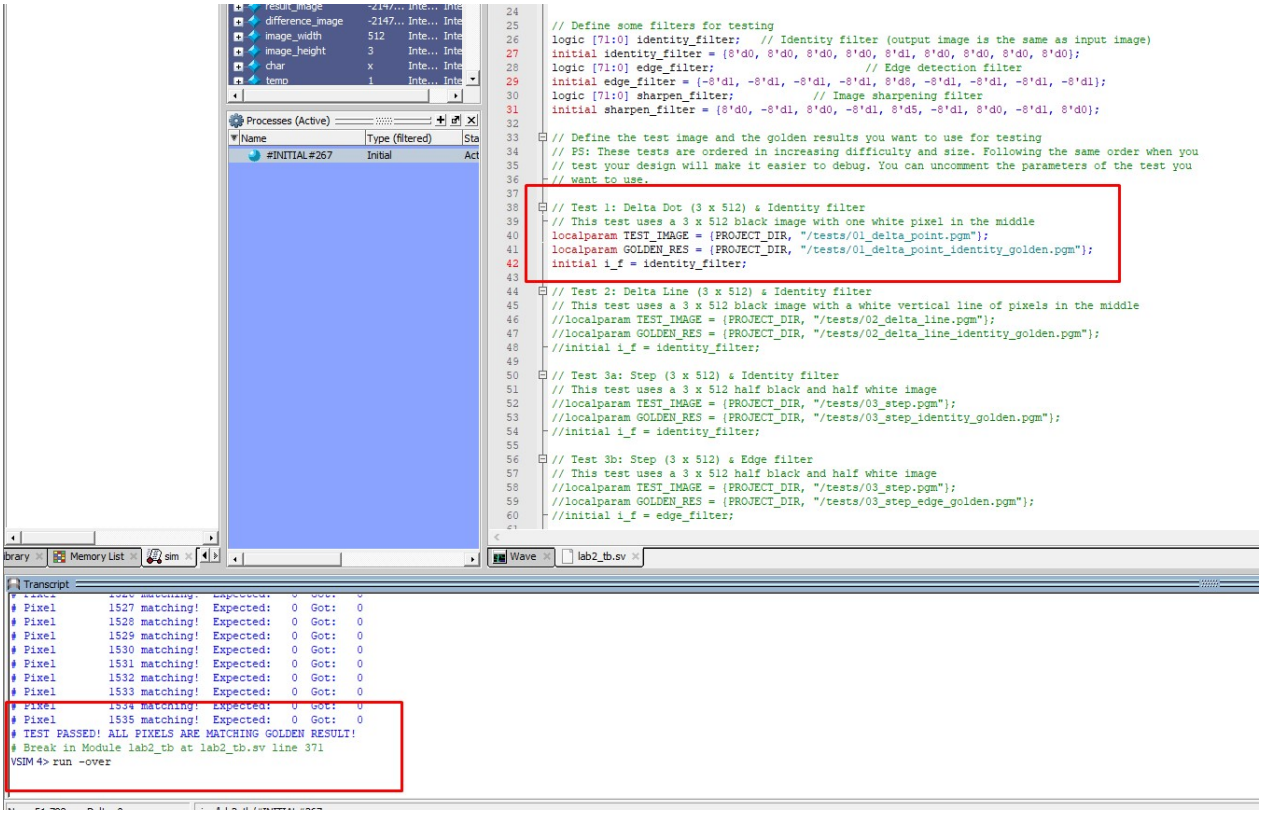


Figure 8: FPGA test 1 test bench

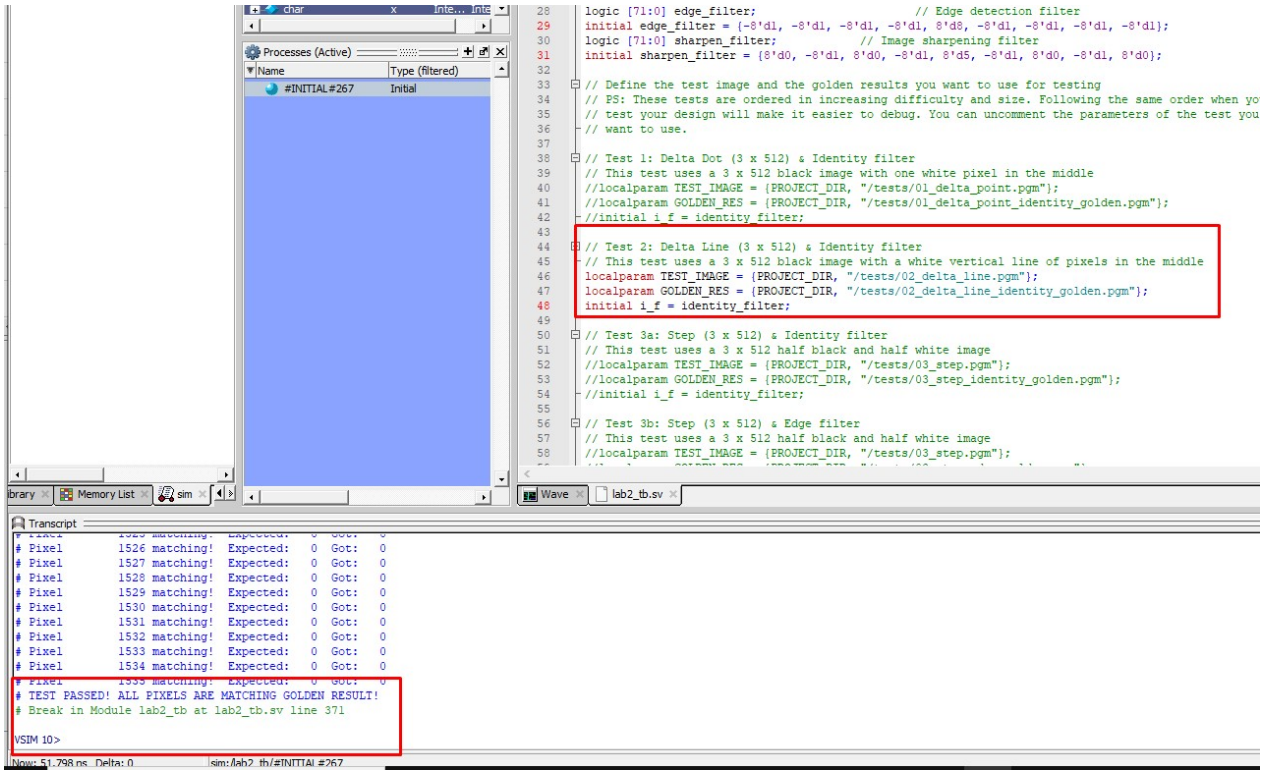


Figure 9: FPGA test 2 test bench

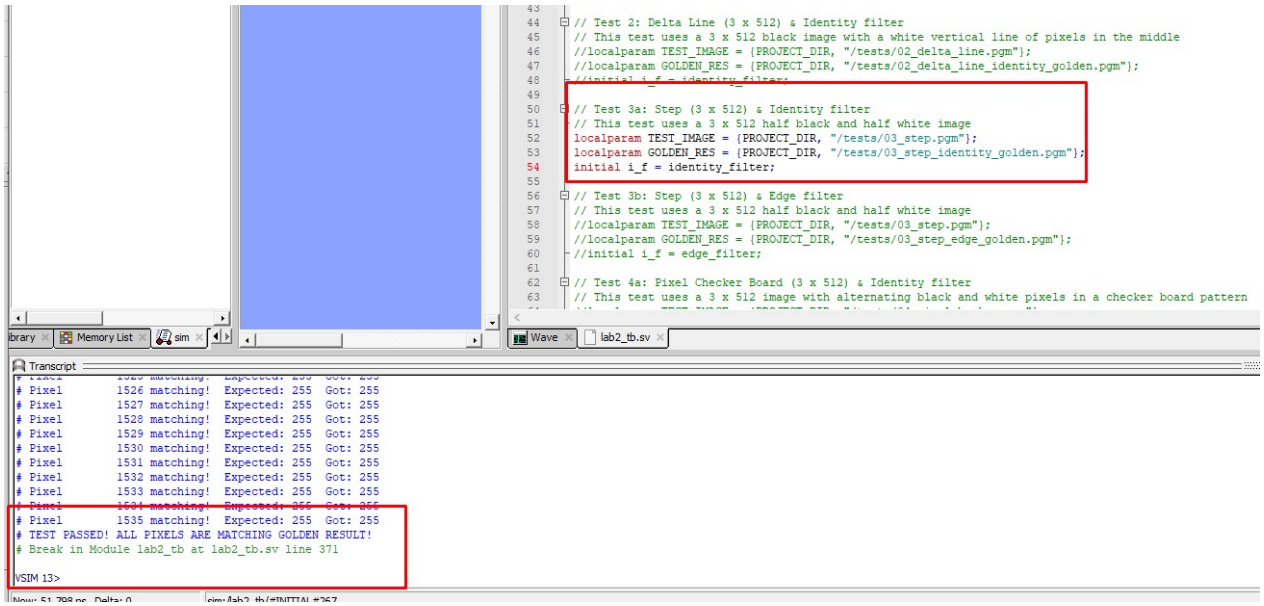


Figure 10: FPGA test 3a test bench

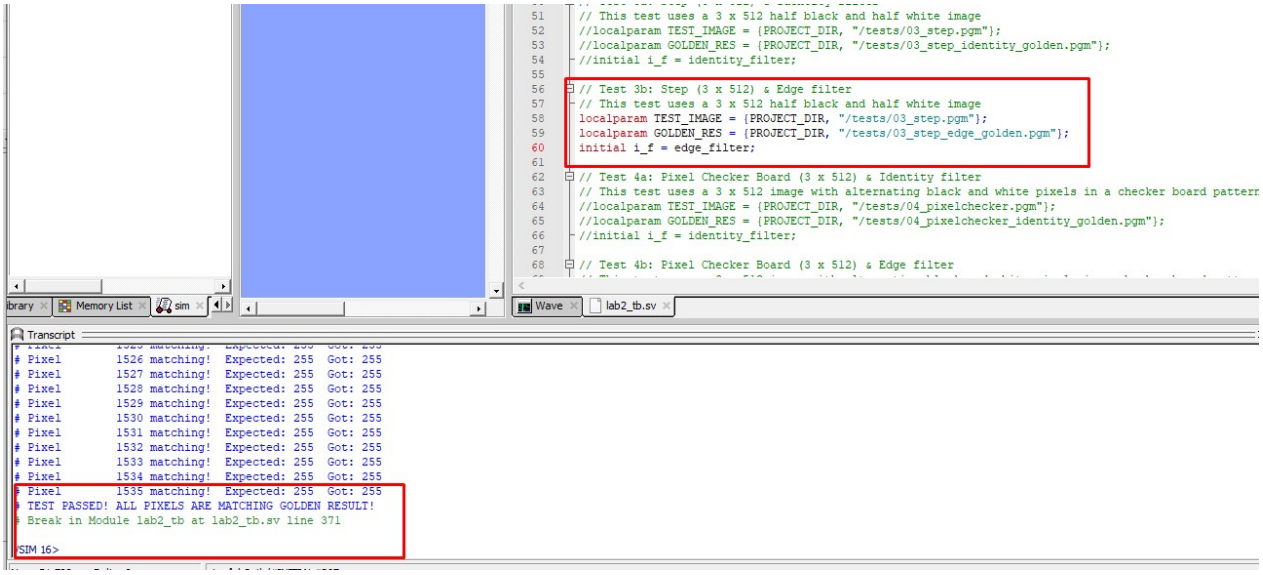


Figure 11: FPGA test 3b test bench

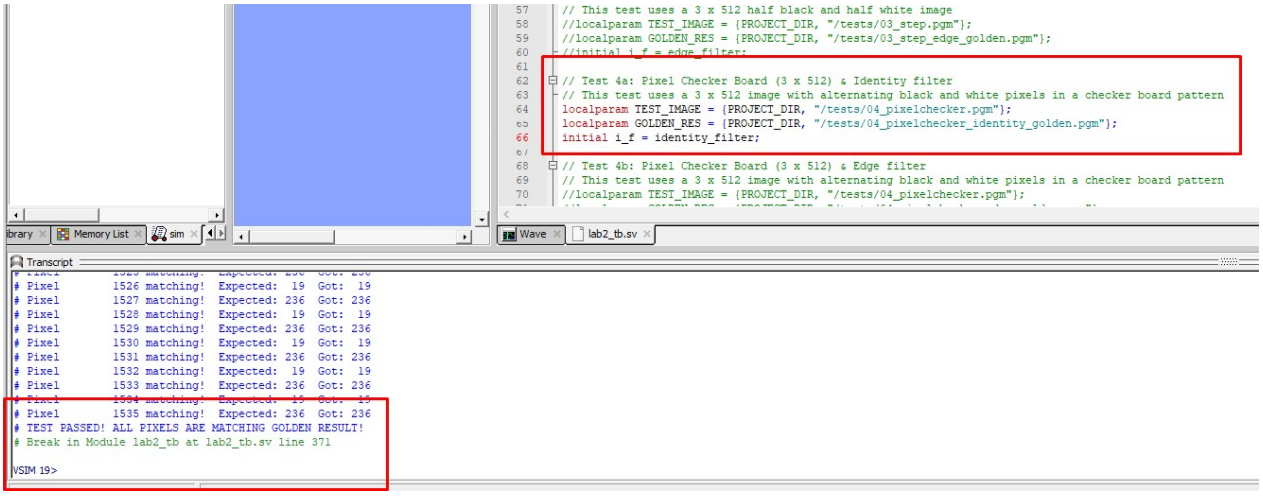


Figure 12: FPGA test 4a test bench

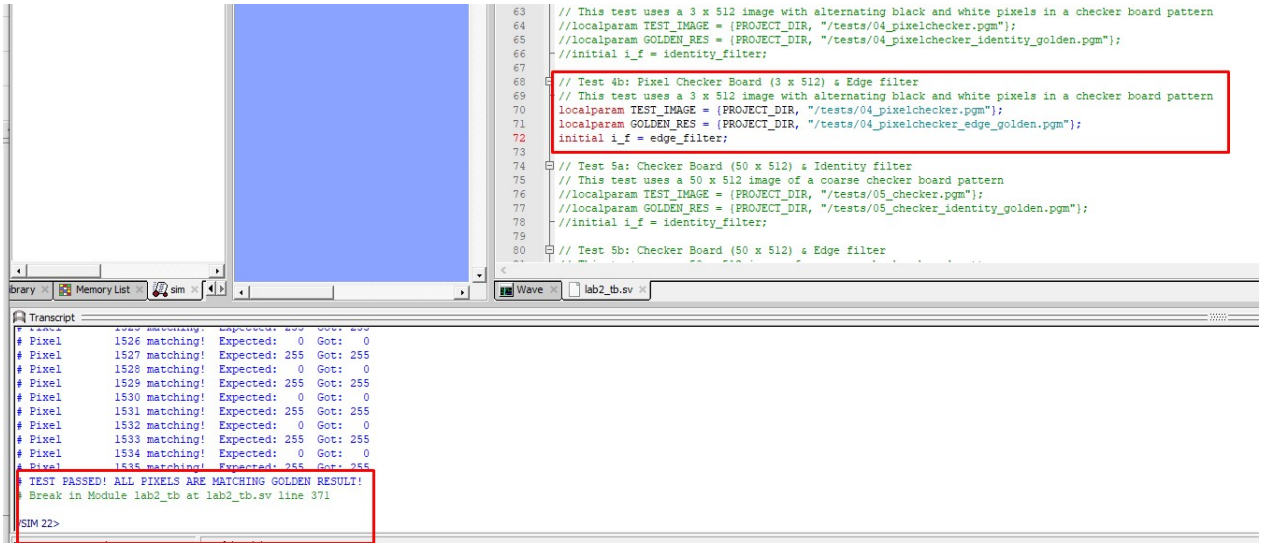


Figure 13: FPGA test 4b test bench

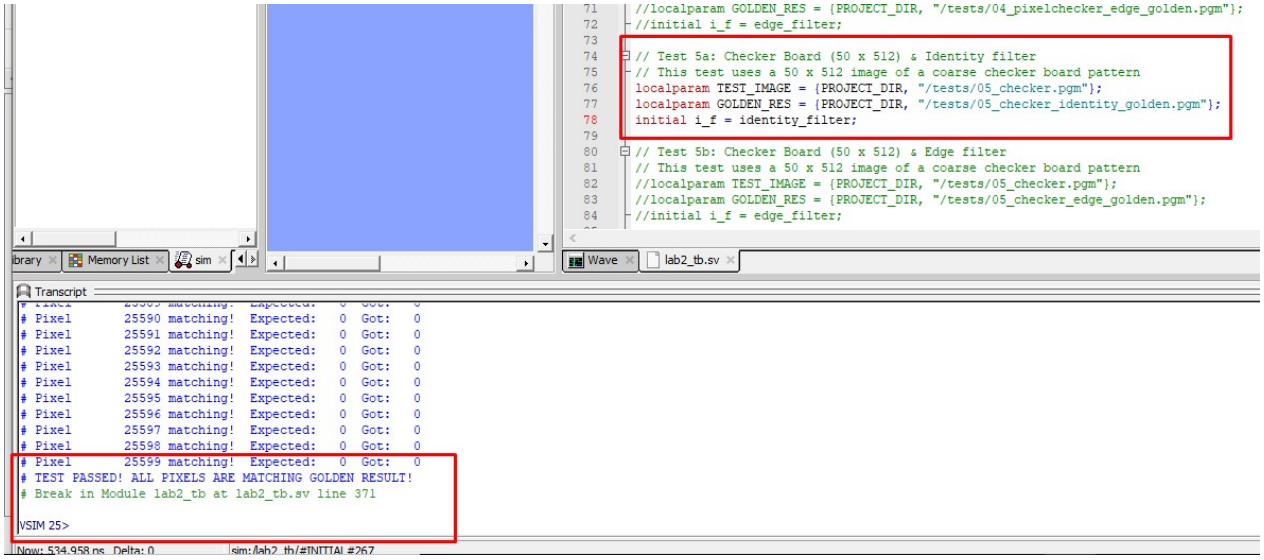


Figure 14: FPGA test 5a test bench

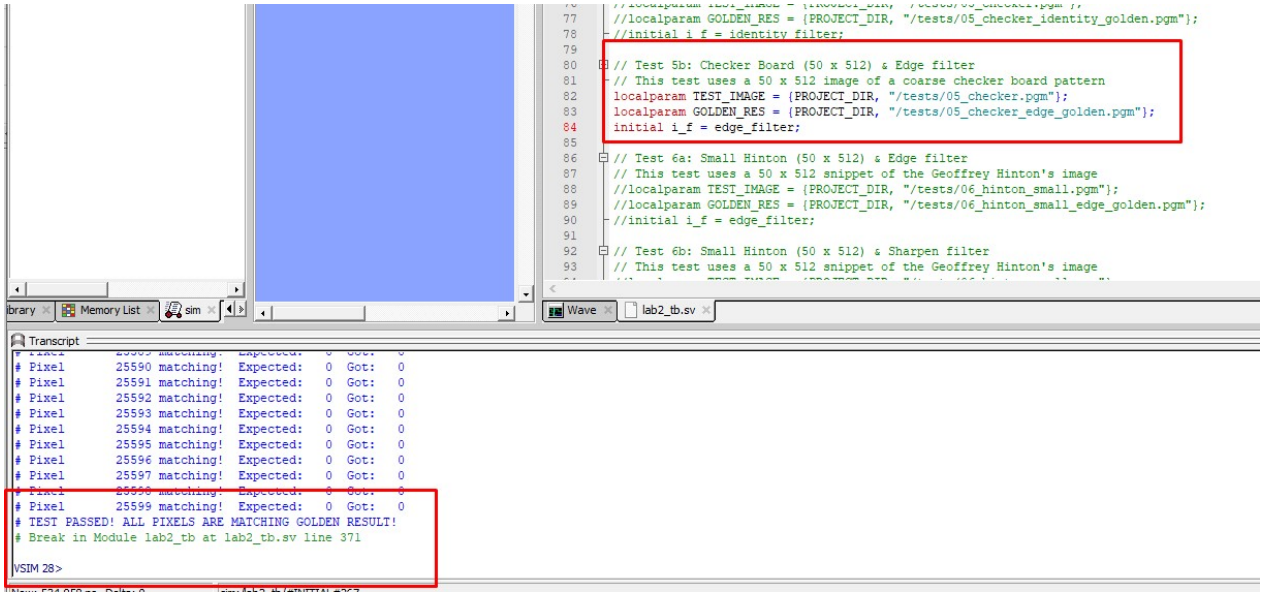


Figure 15: FPGA test 5b test bench

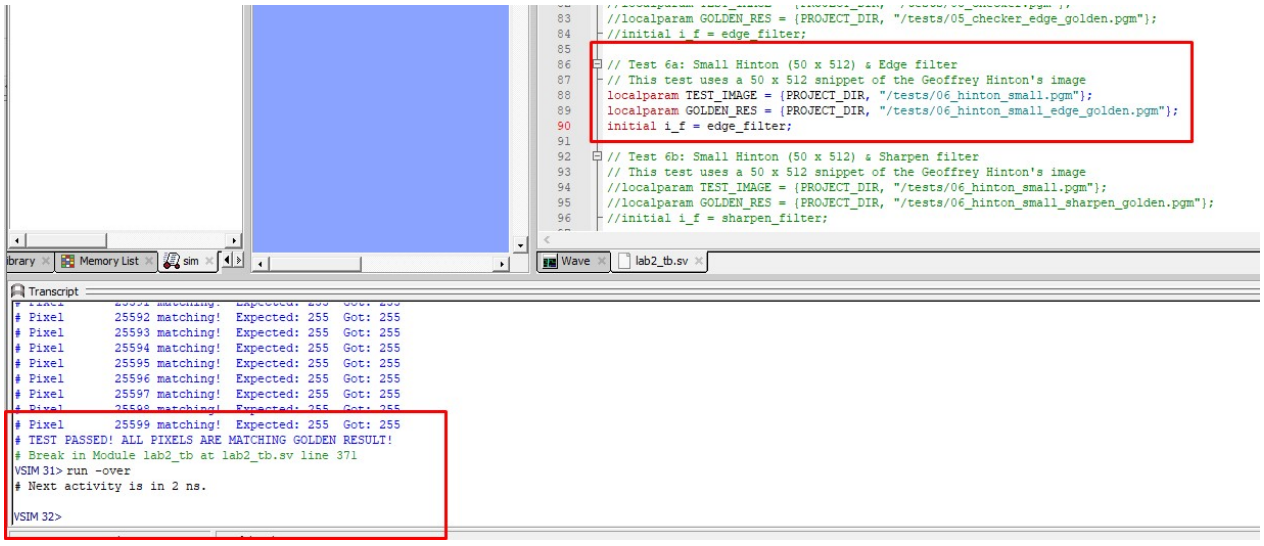


Figure 16: FPGA test 6a test bench

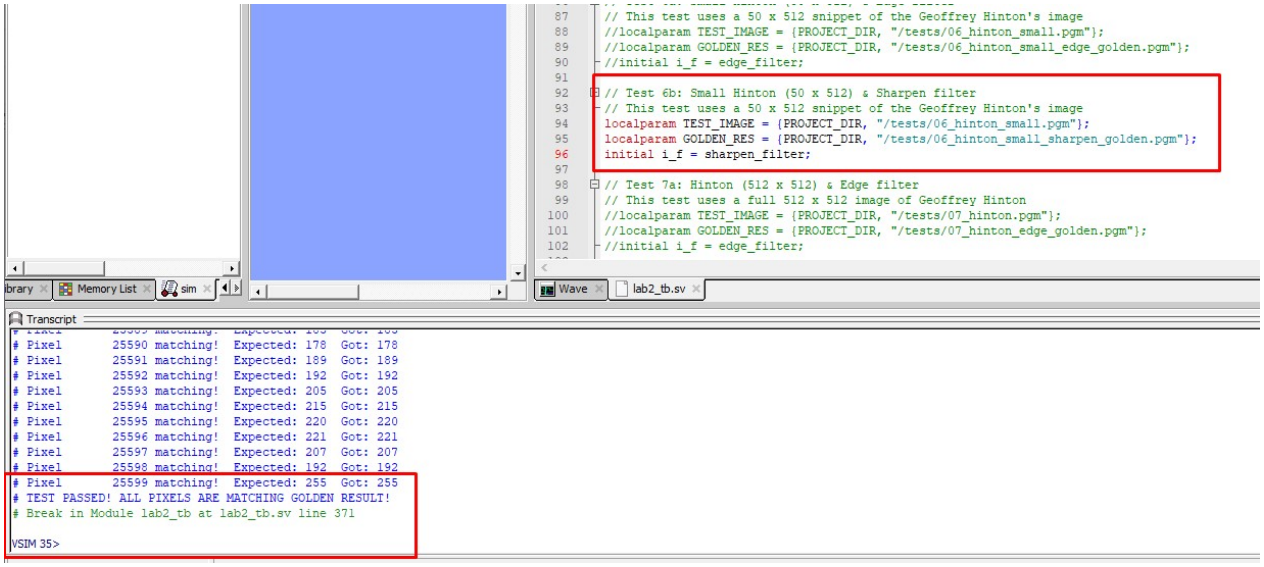


Figure 17: FPGA test 6b test bench

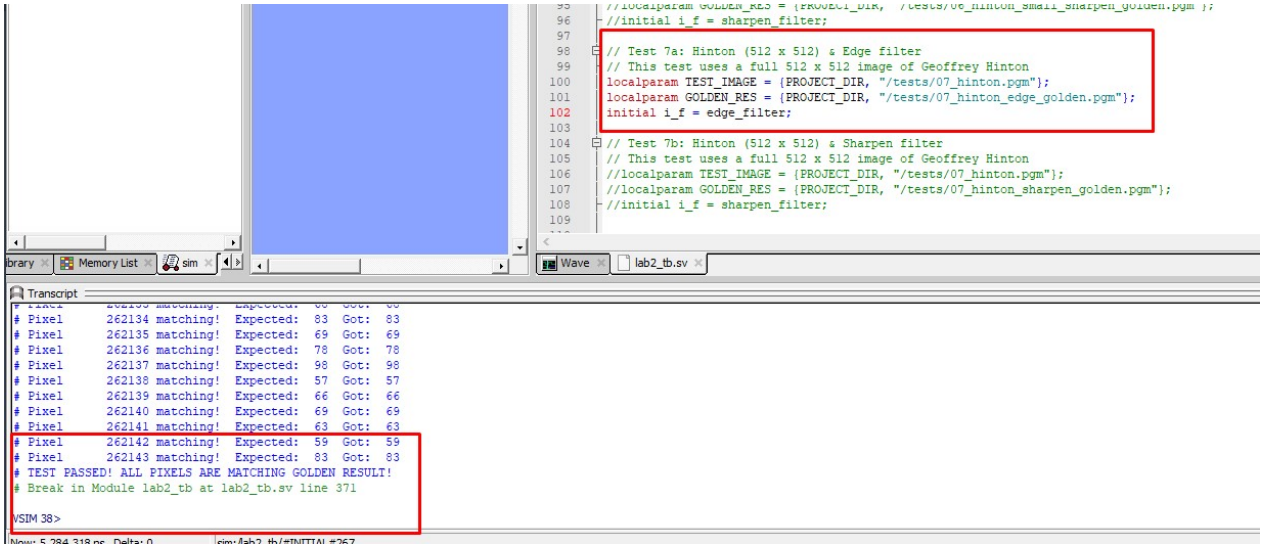


Figure 18: FPGA test 7a test bench

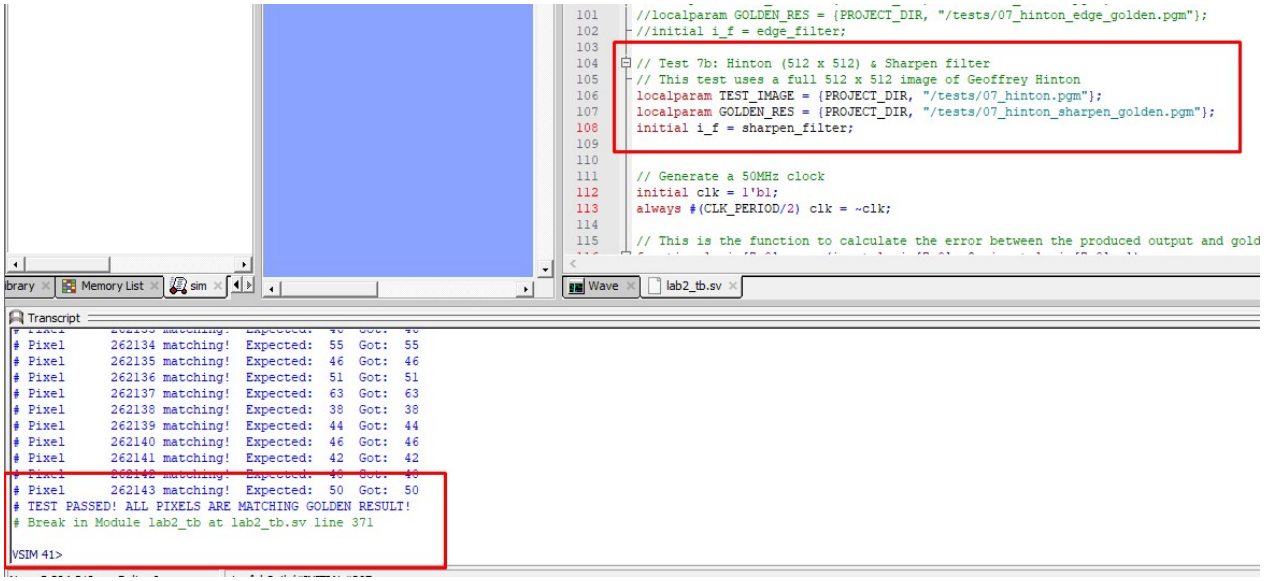


Figure 19: FPGA test 7b test bench

Appendix B: FPGA Implementation Results Calculations

ALM, DSP, BRAM Utilization

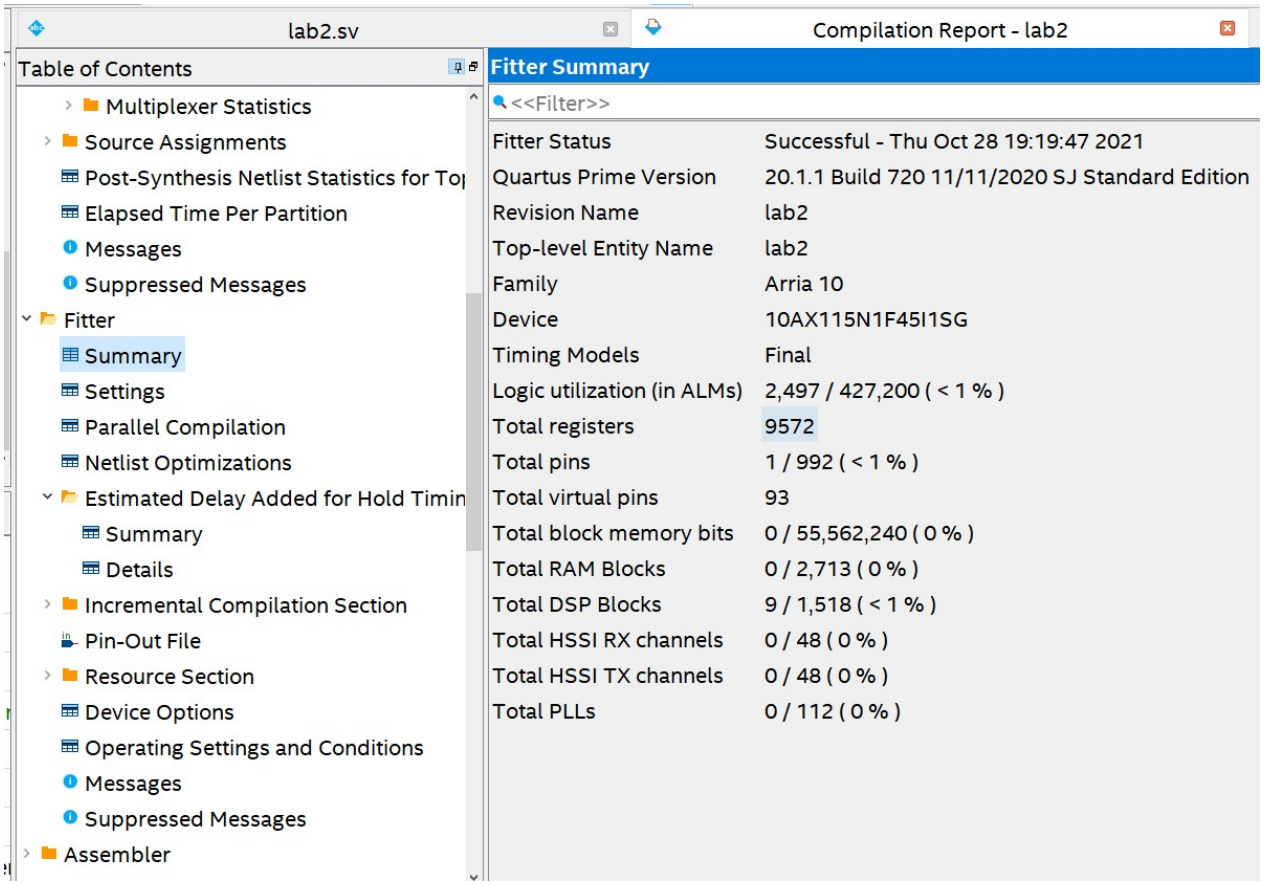


Figure 20: FPGA ALM, DSP and BRAM utilization

Maximum Operating Frequency

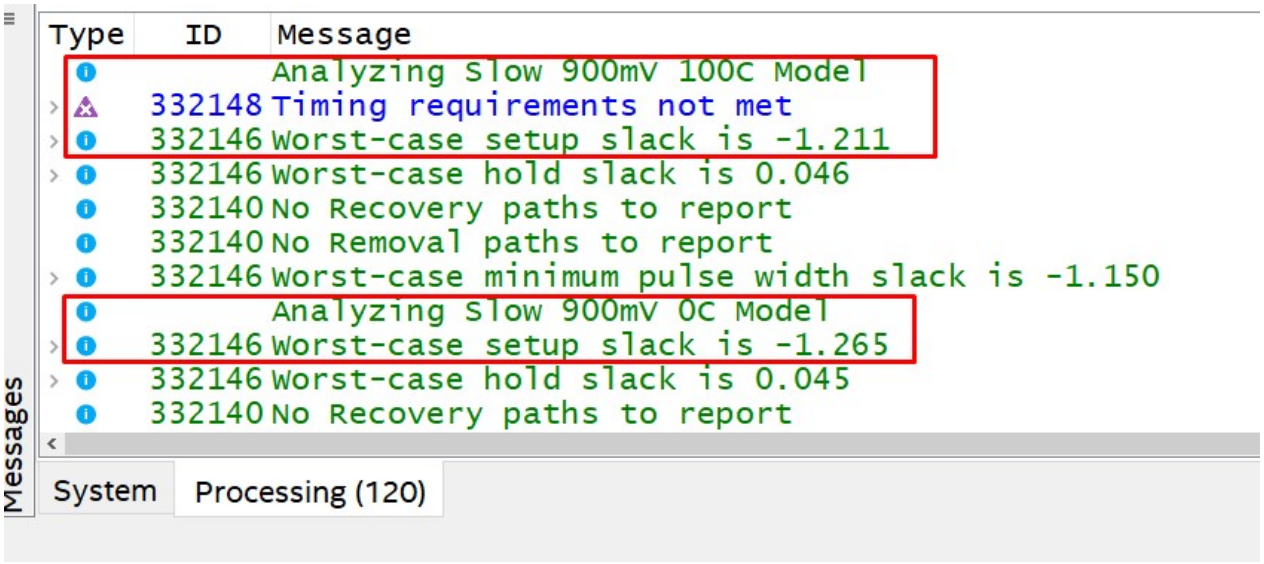


Figure 21: FPGA setup and hold times

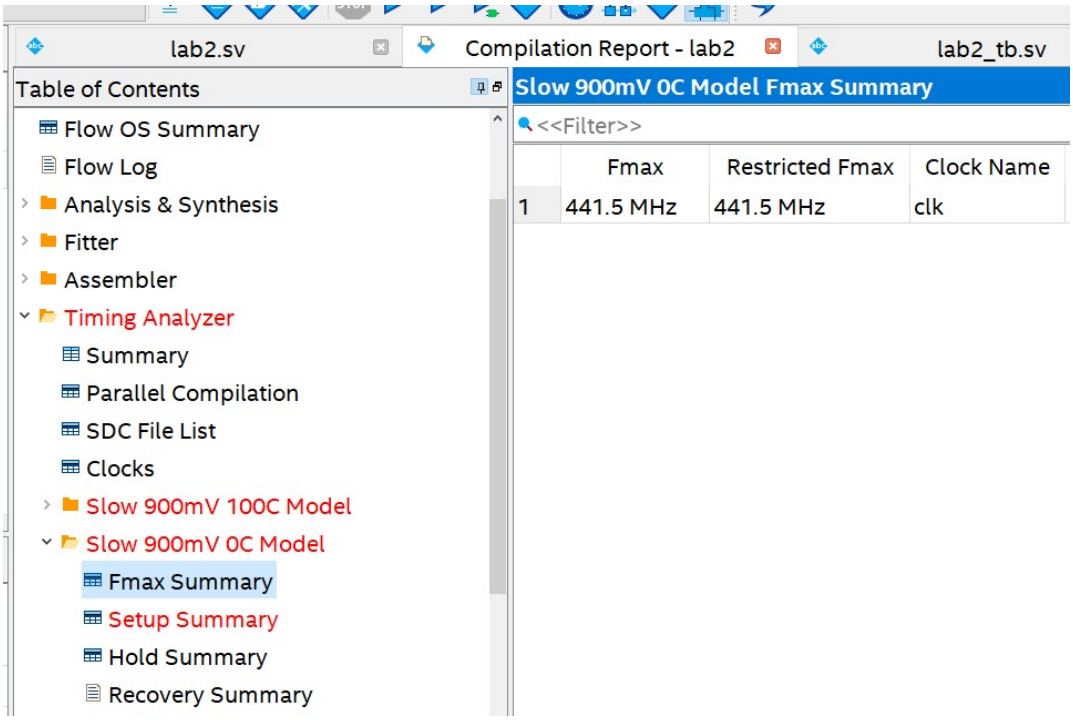


Figure 22: FPGA maximum operating frequency

Cycles for Test 7a (Hinton)

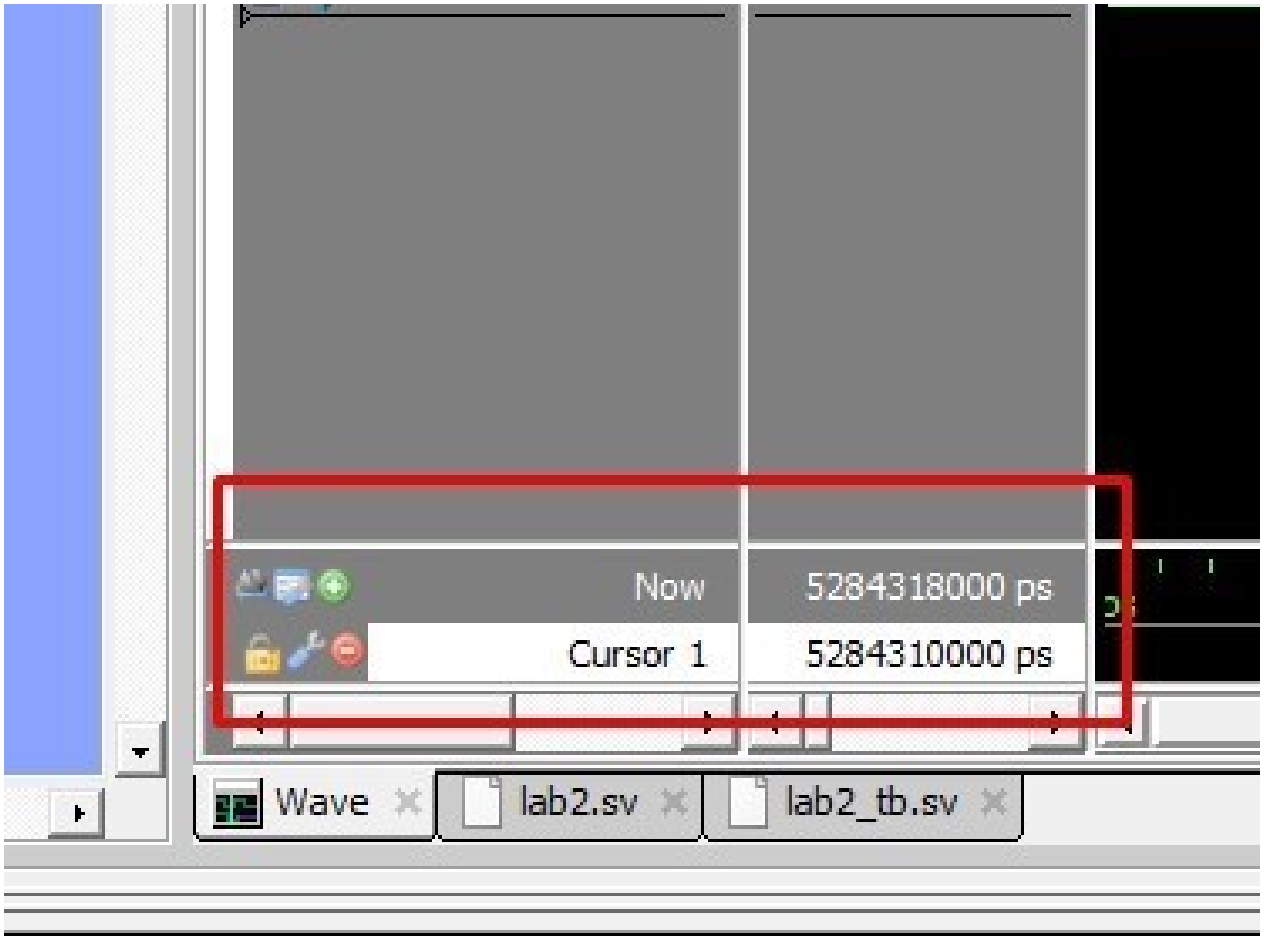


Figure 23: Test 7a total elapsed time

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

Dynamic Power for one module @ 441.5MHz

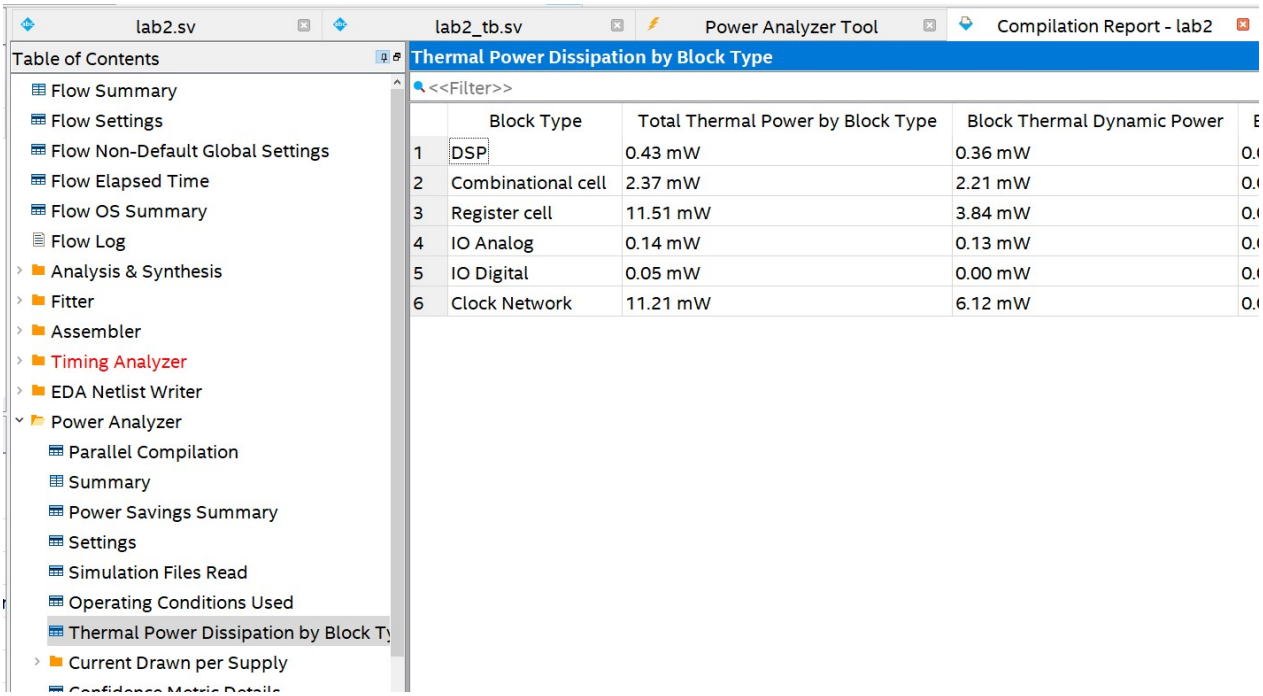


Figure 24: Power dissipation by block type

$$Total\ Dynamic\ Power\ @50MHz = 0.36 + 2.21 + 3.84 + 0.13 + 6.12 = 12.66\ mW$$

$$Total\ Dynamic\ Power\ @441.5MHz = 12.66{*}(441.5/50) = 111.7878\,mW = 111.7878{\times}10^{-3}\,W$$

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 441.5 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)

$$Max.\ copies\ (based\ on\ ALM\ count) = 427200/2497 \approx 171$$

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

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:

$$Total\ Dynamic\ Power\ @50MHz = (0.36 + 2.21 + 3.84) * 168 + 0.13 + 6.12 = 1083.13\ mW$$

$$Total\ Dynamic\ Power\ @441.5MHz = 1083.13 * (441.5/50) = 9564.0379\ mW$$

$$Total\ Power\ @441.5MHz = 9564.0379 + (0.43 + 2.37 + 11.51) * 168 + 0.14 + 0.05 + 11.21$$

$$= 11979.5179\ mW \approx 11.980\ W$$

Appendix C: Runtimes for CPU and GPU implementations

```
ug60:~/ece1756/lab2_cpu_gpu% rm conv_cpu
/bin/rm: remove regular file 'conv_cpu'? y
ug60:~/ece1756/lab2_cpu_gpu% g++ -mavx conv_cpu.cpp utils.cpp -o conv_cpu
ug60:~/ece1756/lab2_cpu_gpu% ./run_cpu.sh 1 4 16 64
Runtime of simple convolution for 1 filter(s) averaged accross 20 runs = 8.82178 ms
Runtime of hand vectorized convolution for 1 filter(s) averaged accross 20 runs = 5.34393 ms
-----------------------
Runtime of simple convolution for 4 filter(s) averaged accross 20 runs = 35.0252 ms
Runtime of hand vectorized convolution for 4 filter(s) averaged accross 20 runs = 21.2444 ms
-----------------------
Runtime of simple convolution for 16 filter(s) averaged accross 20 runs = 139.794 ms
Runtime of hand vectorized convolution for 16 filter(s) averaged accross 20 runs = 87.1801 ms
-----------------------
Runtime of simple convolution for 64 filter(s) averaged accross 20 runs = 558.436 ms
Runtime of hand vectorized convolution for 64 filter(s) averaged accross 20 runs = 340.192 ms
-----------------------
ug60:~/ece1756/lab2_cpu_gpu%
```

Figure 25: CPU no optimization runtimes

```
ug60:~/ece1756/lab2_cpu_gpu% rm conv_cpu
/bin/rm: remove regular file 'conv_cpu'? y
ug60:~/ece1756/lab2_cpu_gpu% g++ -mavx -O2 conv_cpu.cpp utils.cpp -o conv_cpu
ug60:~/ece1756/lab2_cpu_gpu% ./run_cpu.sh 1 4 16 64
Runtime of simple convolution for 1 filter(s) averaged accross 20 runs = 1.59554 ms
Runtime of hand vectorized convolution for 1 filter(s) averaged accross 20 runs = 1.26072 ms
-----------------------
Runtime of simple convolution for 4 filter(s) averaged accross 20 runs = 6.37035 ms
Runtime of hand vectorized convolution for 4 filter(s) averaged accross 20 runs = 5.05124 ms
-----------------------
Runtime of simple convolution for 16 filter(s) averaged accross 20 runs = 25.4466 ms
Runtime of hand vectorized convolution for 16 filter(s) averaged accross 20 runs = 23.0627 ms
-----------------------
Runtime of simple convolution for 64 filter(s) averaged accross 20 runs = 101.752 ms
Runtime of hand vectorized convolution for 64 filter(s) averaged accross 20 runs = 83.6116 ms
-----------------------
ug60:~/ece1756/lab2_cpu_gpu%
```

Figure 26: CPU runtimes with O2 optimization

```
ug60:~/ece1756/lab2_cpu_gpu% rm conv_cpu
/bin/rm: remove regular file 'conv_cpu'? y
ug60:~/ece1756/lab2_cpu_gpu% g++ -mavx -O3 conv_cpu.cpp utils.cpp -o conv_cpu
ug60:~/ece1756/lab2_cpu_gpu% ./run_cpu.sh 1 4 16 64
Runtime of simple convolution for 1 filter(s) averaged accross 20 runs = 0.679928 ms
Runtime of hand vectorized convolution for 1 filter(s) averaged accross 20 runs = 1.25945 ms
------------------------
Runtime of simple convolution for 4 filter(s) averaged accross 20 runs = 2.72494 ms
Runtime of hand vectorized convolution for 4 filter(s) averaged accross 20 runs = 5.03768 ms
------------------------
Runtime of simple convolution for 16 filter(s) averaged accross 20 runs = 10.8303 ms
Runtime of hand vectorized convolution for 16 filter(s) averaged accross 20 runs = 23.019 ms
------------------------
Runtime of simple convolution for 64 filter(s) averaged accross 20 runs = 43.27 ms
Runtime of hand vectorized convolution for 64 filter(s) averaged accross 20 runs = 83.5155 ms
        ------------------------
ug60:~/ece1756/lab2 cpu gpu%
```

Figure 27: CPU runtimes with O3 optimization

```
ug60:~/ece1756/lab2_cpu_gpu% g++ -mavx -fopenmp -O3 conv_cpu.cpp utils.cpp -o conv_cpu
ug60:~/ece1756/lab2_cpu_gpu% ./run_cpu.sh 1 4 16 64
Runtime of simple convolution for 1 filter(s) averaged accross 20 runs = 0.728284 ms
Runtime of hand vectorized convolution for 1 filter(s) averaged accross 20 runs = 1.19086 ms
------------------------
Runtime of simple convolution for 4 filter(s) averaged accross 20 runs = 0.823869 ms
Runtime of hand vectorized convolution for 4 filter(s) averaged accross 20 runs = 1.71489 ms
------------------------
Runtime of simple convolution for 16 filter(s) averaged accross 20 runs = 2.93434 ms
Runtime of hand vectorized convolution for 16 filter(s) averaged accross 20 runs = 7.82391 ms
------------------------
Runtime of simple convolution for 64 filter(s) averaged accross 20 runs = 11.8149 ms
Runtime of hand vectorized convolution for 64 filter(s) averaged accross 20 runs = 22.2321 ms
------------------------
ug60:~/ece1756/lab2_cpu_gpu%
```

Figure 28: CPU runtimes with O3 optimization, multithreaded

```
ug60:~/ece1756/lab2_cpu_gpu% ./run_gpu.sh 1 4 16 64
Runtime for 1 filter(s) averaged accross 20 runs = 0.0297056 ms
------------------------
Runtime for 4 filter(s) averaged accross 20 runs = 0.0994058 ms
------------------------
Runtime for 16 filter(s) averaged accross 20 runs = 0.372853 ms
------------------------
Runtime for 64 filter(s) averaged accross 20 runs = 1.47965 ms
------------------------
ug60:~/ece1756/lab2_cpu_gpu%
```

Figure 29: GPU runtimes

Appendix D: Evaluation Calculations

FPGA related

$$Energy\ Efficiency = 1.2510 \times 10^3/11.98 \approx 104.42\ GOPS/W$$
$$Area\ Efficiency = 1.2510 \times 10^3/350 \approx 3.5743\ GOPS/mm2$$

CPU related

```
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 30: CPU basic code

Based on Fig. 30 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%

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

$$Energy\ Efficiency = 163.72/84 \approx 1.9490\ GOPS/W$$
$$Area\ Efficiency = 163.72/177 \approx 0.92497\ GOPS/mm2$$

GPU related

Based on Fig. 31 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

```
// 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 31: GPU code

```
// 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 32: 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%

$$Total\ Operations = (15*((34*34)/(32*32))+(9*3*3)*512*512+9)*1.10/1.479\,65\times10^{-3}$$
$$\approx 15.786\ GOPS$$

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

$$Area\ Efficiency = 15.786/396 \approx 0.39864\ GOPS/mm2$$

GPU Estimation

$$Throughput = 15.786 * (28/20) \approx 22.1\ GOPS$$

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

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

$$Area\ Efficiency = 22.1/396 \approx 0.055808\ GOPS/mm2$$

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)

// The following code is intended to show you an example of how to use
↪  paramaters and
```

```
// for loops in SytemVerilog. It also arrages the input filter coefficients for you
// into a nicely-arranged and easy-to-use 2D array of registers. However, you can ignore
// this code and not use it if you wish to.

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

```
end

////////////////////////////////////////// Start of your code
↪  //////////////////////////////////////////

localparam PIC_WIDTH = 512+2; // How wide is our picture

logic [PIXEL_DATAW-1:0] SIPO
↪  [PIC_WIDTH*2+FILTER_SIZE];                    // SIPO, shift reg
↪  for data input
```

```
logic data_valid
↪  [PIC_WIDTH*2+FILTER_SIZE];                                                    //
↪  Data valid propagate

// Intermediate outputs
logic signed [31:0] mult_out [9];                                  // Output
↪  of multiplier
logic
↪  data_valid_mult;                                                             //
↪  Valid bit for if multiplier output is valid

logic signed [31:0] addr_stage_1_out [4];                 // Output of adders
↪  in stage 1
logic signed [31:0] addr_stage_2_out [2];                 // Output of adders
↪  in stage 2
logic signed [31:0] addr_stage_3_out;                         // Output of
↪  adder in stage 3
logic signed [31:0] addr_stage_4_out;                         // Output of
↪  adder in stage 4

// Pipeline registers
logic signed [31:0] mult_out_pipe [9];
logic data_valid_mult_pipe;

logic signed [31:0] addr_stage_1_out_pipe [5];
logic data_valid_addr_stage_1_pipe;
logic signed [31:0] addr_stage_2_out_pipe [3];
logic data_valid_addr_stage_2_pipe;
logic signed [31:0] addr_stage_3_out_pipe [2];
logic data_valid_addr_stage_3_pipe;
logic unsigned [PIXEL_DATAW-1:0] addr_stage_4_out_pipe;
```

```
logic data_valid_addr_stage_4_pipe; // Final, same level (in terms of
↪  pipelining) as o_valid

// Counter to ensure correctness, no computation is to happen for the
↪  first 2 valid inputs
// for each new row of pixels
logic [15:0] counter;

// Instantiating multipliers. All products are computed in one and the
↪  same clock cycle
mult8x8 convo_mult_0 (.i_dataa(SIPO[2+PIC_WIDTH*2]),
↪          .i_datab(r_f[0][0]), .o_res(mult_out[0]));        // Row 0, Col
↪  0
mult8x8 convo_mult_1 (.i_dataa(SIPO[1+PIC_WIDTH*2]),
↪          .i_datab(r_f[0][1]), .o_res(mult_out[1]));        // Row 0, Col
↪  1
mult8x8 convo_mult_2 (.i_dataa(SIPO[0+PIC_WIDTH*2]),
↪          .i_datab(r_f[0][2]), .o_res(mult_out[2]));        // Row 0, Col
↪  2
mult8x8 convo_mult_3 (.i_dataa(SIPO[2+PIC_WIDTH*1]),
↪          .i_datab(r_f[1][0]), .o_res(mult_out[3]));        // Row 1, Col
↪  0
mult8x8 convo_mult_4 (.i_dataa(SIPO[1+PIC_WIDTH*1]),
↪          .i_datab(r_f[1][1]), .o_res(mult_out[4]));        // Row 1, Col
↪  1
mult8x8 convo_mult_5 (.i_dataa(SIPO[0+PIC_WIDTH*1]),
↪          .i_datab(r_f[1][2]), .o_res(mult_out[5]));        // Row 1, Col
↪  2
mult8x8 convo_mult_6 (.i_dataa(SIPO[2]),
↪                                          .i_datab(r_f[2][0]),
↪  .o_res(mult_out[6]));        // Row 2, Col 0
mult8x8 convo_mult_7 (.i_dataa(SIPO[1]),
↪                                          .i_datab(r_f[2][1]),
↪  .o_res(mult_out[7]));        // Row 2, Col 1
mult8x8 convo_mult_8 (.i_dataa(SIPO[0]),
↪                                          .i_datab(r_f[2][2]),
↪  .o_res(mult_out[8]));        // Row 2, Col 2

// Instantiating adders. Each grouping of adders in one clock cycle each
addr32p32 convo_addr_stage_1_0 (.i_dataa(mult_out_pipe[0]),
↪  .i_datab(mult_out_pipe[1]), .o_res(addr_stage_1_out[0]));
```

```
addr32p32 convo_addr_stage_1_1 (.i_dataa(mult_out_pipe[2]),
↪  .i_datab(mult_out_pipe[3]), .o_res(addr_stage_1_out[1]));
addr32p32 convo_addr_stage_1_2 (.i_dataa(mult_out_pipe[4]),
↪  .i_datab(mult_out_pipe[5]), .o_res(addr_stage_1_out[2]));
addr32p32 convo_addr_stage_1_3 (.i_dataa(mult_out_pipe[6]),
↪  .i_datab(mult_out_pipe[7]), .o_res(addr_stage_1_out[3]));

addr32p32 convo_addr_stage_2_0 (.i_dataa(addr_stage_1_out_pipe[0]),
↪  .i_datab(addr_stage_1_out_pipe[1]), .o_res(addr_stage_2_out[0]));
addr32p32 convo_addr_stage_2_1 (.i_dataa(addr_stage_1_out_pipe[2]),
↪  .i_datab(addr_stage_1_out_pipe[3]), .o_res(addr_stage_2_out[1]));

addr32p32 convo_addr_stage_3_0 (.i_dataa(addr_stage_2_out_pipe[0]),
↪  .i_datab(addr_stage_2_out_pipe[1]), .o_res(addr_stage_3_out));

addr32p32 convo_addr_stage_4_0 (.i_dataa(addr_stage_3_out_pipe[0]),
↪  .i_datab(addr_stage_3_out_pipe[1]), .o_res(addr_stage_4_out));

// Valid for computed values is only true if all inputs are valid
// otherwise, similar to lab 1, these are just propagated along with
↪  computed values
always_comb begin
        // Multiplier output only valid if all 9 data bytes are valid
        data_valid_mult = data_valid[2+PIC_WIDTH*2] &
        ↪  data_valid[1+PIC_WIDTH*2] & data_valid[0+PIC_WIDTH*2] &
                                                        data_valid[2+PIC_WIDTH*1]
                                                        ↪  &
                                                        ↪  data_valid[1+PIC_WIDTH*1]
                                                        ↪  &
                                                        ↪  data_valid[0+PIC_WIDTH*1]
                                                        ↪  &
                                                        data_valid[2]
                                                        ↪  & data_valid[1]
                                                        ↪  & data_valid[0]
                                                        ↪  &
                                                        (counter >=
                                                        ↪  FILTER_SIZE); //
                                                        ↪  Need at least 3
                                                        ↪  element in the
                                                        ↪  row before
                                                        ↪  computing
end
```

```
logic enable;
always_comb begin
        enable = i_ready & i_valid;
end


int i;
always_ff @ (posedge clk) begin

        if(reset)begin
                // Reset counter
                counter <= 16'b0;

                // Reset SIPO and valid propagation
                for(i = 0; i < $size(SIPO); i = i + 1) begin
                        SIPO[i] <= 8'b0;
                        data_valid[i] <= 1'b0;
                end

        end else begin
                if(enable)begin
                        // Counter
                        // If counter at 514 means current row complete,
                        ↪  reset to 1 since it's the first entry of the
                        ↪  next row
                        if (counter == PIC_WIDTH) begin
                                counter <= 16'b1;
                        end else begin
                                counter <= counter + 1'b1;
                        end

                        SIPO[0] <= i_x;
                        ↪                                              // If
                        ↪  input is valid, load into SIPO[0]
                        data_valid[0] <= i_valid;                  //
                        ↪  Similarly with valids
                        for(i = 1; i < $size(SIPO); i = i + 1) begin
                                SIPO[i] <=
                                ↪  SIPO[i-1];                                          //
                                ↪  Propagate signal in shift reg
```

```
        data_valid[i] <= data_valid[i-1];        //
        ↪ Propagate valid signal
end

// Pipeline reg for multiplier out, adder stage 1
↪ in
mult_out_pipe <= mult_out;
data_valid_mult_pipe <= data_valid_mult;


data_valid_addr_stage_1_pipe <=
↪ data_valid_mult_pipe;
data_valid_addr_stage_2_pipe <=
↪ data_valid_addr_stage_1_pipe;
data_valid_addr_stage_3_pipe <=
↪ data_valid_addr_stage_2_pipe;
data_valid_addr_stage_4_pipe <=
↪ data_valid_addr_stage_3_pipe;

// Pipeline reg for adder stage 1 out, adder stage
↪ 2 in
addr_stage_1_out_pipe[0] <= addr_stage_1_out[0];
addr_stage_1_out_pipe[1] <= addr_stage_1_out[1];
addr_stage_1_out_pipe[2] <= addr_stage_1_out[2];
addr_stage_1_out_pipe[3] <= addr_stage_1_out[3];
addr_stage_1_out_pipe[4] <= mult_out_pipe[8];

// Pipeline reg for adder stage 2 out, adder stage
↪ 3 in
addr_stage_2_out_pipe[0] <= addr_stage_2_out[0];
addr_stage_2_out_pipe[1] <= addr_stage_2_out[1];
addr_stage_2_out_pipe[2] <=
↪ addr_stage_1_out_pipe[4];

// Pipeline reg for adder stage 3 out, adder stage
↪ 4 in
addr_stage_3_out_pipe[0] <= addr_stage_3_out;
addr_stage_3_out_pipe[1] <=
↪ addr_stage_2_out_pipe[2];

// Pipeline reg for adder stage 4 out, final
↪ result, clipping performed here
```

```
                        if (addr_stage_4_out > 32'sd255) begin
                                addr_stage_4_out_pipe <= 8'd255;
                        end else if (addr_stage_4_out < 32'sd0) begin
                                addr_stage_4_out_pipe <= 8'd0;
                        end else begin
                                addr_stage_4_out_pipe <=
                                ↪  addr_stage_4_out[7:0];
                        end
                end
        end

end

assign o_y = addr_stage_4_out_pipe;
assign o_ready = i_ready;
assign o_valid = data_valid_addr_stage_4_pipe & i_ready;

////////////////////////////////////////// End of your code
↪  //////////////////////////////////////////

endmodule

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

// Multiplier module for 8x8 multiplication
// Taken from lab 1 with modifications for signed multiplication
module mult8x8 ( /* synthesis multstyle = "dsp" */
        input unsigned [7:0] i_dataa,
        input signed [7:0] i_datab,
        output signed [31:0] o_res
);

logic signed [31:0] result;

always_comb begin
        result = i_datab * $signed({1'b0,i_dataa});
end

assign o_res = result;

endmodule
```

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

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

// Adder module for 32b+32b addition operations
// Taken from lab 1 with modifications for signed addition
module addr32p32 (
        input signed [31:0] i_dataa,
        input signed [31:0] i_datab,
        output signed [31:0] o_res
);

assign o_res = i_dataa + i_datab;

endmodule

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