Yinhao Qian (YIQ25)

ECE 1195 – Advanced Digital Design

Lab 5 – 2D Convolution Test Report

Step 1 – Functionalities Implementations using C/C++

First of all, the implementation of sliding windows and convolutions can be done in C++. Figure 1.A showed how Sliding Windows was done, and Figure 1.B showed how Convolutions was done. HLS UNROLL directives are used to pre-compile the nested loops. It increases calculation speeds by shifting all pixels at the same time. The optimal unroll factor will be determined automatically at compile time.

```
//Sliding Windows
for (int index_height = 0; index_height < K; index_height++)
{
# pragma HLS UNROLL
for (int index_width = 0; index_width < IMG_WIDTH;
index_width++) {
# pragma HLS UNROLL
if (index_width == K - 1 && index_height == K - 1 &&
i < STREAM_LENGTH) {
beat_t beat_buffer;
sin >> beat_buffer;
kbuf[index_height][index_width] = beat_buffer.data(7, 0);
} else if (index_width < K - 1) {
kbuf[index_height][index_width] =
kbuf[index_height][index_width + 1];
} else if (index_width == K - 1 &&
index_height < K - 1) {
kbuf[index_height][index_width] =
lbuf[index_height][index_width + 1 - K];
} else if (index_width < IMG_WIDTH - 1 &&
index_height < K - 1) {
lbuf[index_height][index_width - K] =
lbuf[index_height][index_width + 1 - K];
} else if (index_width == IMG_WIDTH - 1 &&
index_height < K - 1) {
lbuf[index_height][index_width - K] = kbuf[index_height +
1][0];
}
}
}
//End of Sliding Windows
```

Figure 1.A – Implementation of Sliding Windows in C++

```
//Convolution
```

```
result = 0;
int index_kernal = 0;
for (int index_height = 0; index_height < K; index_height++)
{
# pragma HLS UNROLL
for (int index_width = 0; index_width < K; index_width++) {
# pragma HLS UNROLL
result += kbuf[index_height][index_width] *
kern[index_kernal];
index_kernal++;
}
}
result >>= shift_div;
if (result > 0xFF) {
result = 0xFF;
} else if (result < 0) {
result = 0;
}
//End of Convolution
```

Figure 1.B – Implementation of Convolutions in C++

To attain full functionalities, insert 2 code snippets into convolution.cpp, and now we are ready to test if the above code works.

Step 2 – Bench-testing using GNU Compiler

From HLS Debug Window, the above code from Step 1 is compiled without any errors. If the code was working as intended, two PGM files have now been generated. By comparing them, we can clearly see two pictures are identical, except for few pixels on the edge. Those margins of error are accepted when algorithms from the software side is different from that from the hardware side,

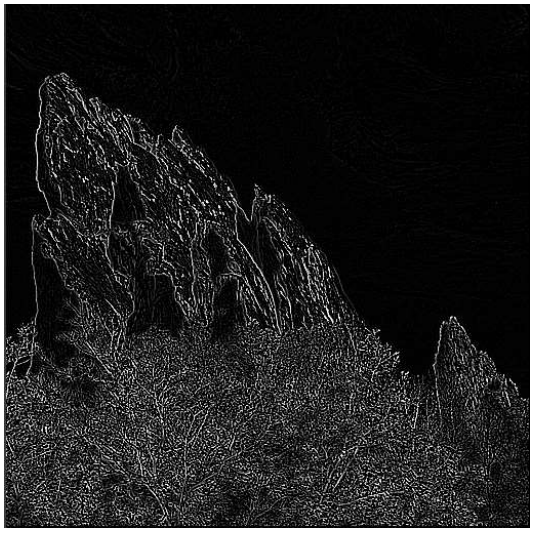

Figure 3.A – Simulated Output from Software Entry

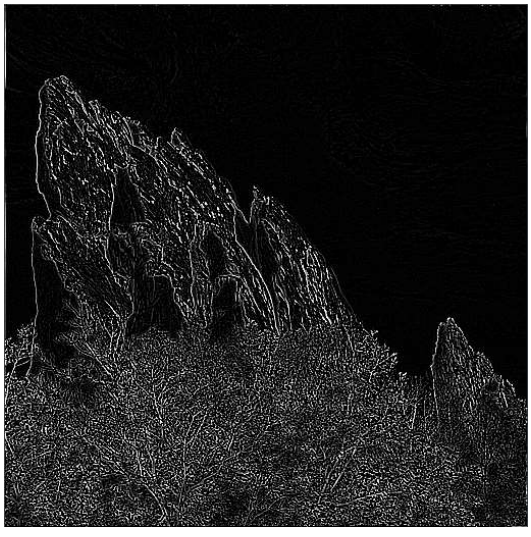

Figure 3.B – Simulated Output from Hardware Entry

However, the hardware approach is still simulated through a software-based compiler. To further validify our design, we will now attempt to convert our C/C++ source code into VHDL description code.

Step 3 – Conversions from C/C++ to VHDL

Fortunately, we are able to generate a vhd file without any errors or critical warnings. The timing and latency information is reported in Figure 4.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.223	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
262661	262661	262661	262661	none

Detail

Instance

N/A

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	262659	262659	4	1	1	262657	yes

Figure 4 – Timing and Latency Information from HDL Conversions

Step 4 – Generating Bitstream

At the same time, bitstream is generated using Vivado. Figure 5 shows the timing summary.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.244 ns	Worst Hold Slack (WHS):	0.023 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	16331	Total Number of Endpoints:	16331	Total Number of Endpoints:	5981

All user specified timing constraints are met.

Figure 5 – Timing and Latency Information from Vivado Synthetizes

Step 5 – Running on FPGA Board

After following the instructions, we are able to process the image directly from the FPGA board. The execution time is 0.002664 as indicated in Figure 6. A new PGM output file should now been saved in the TF card.

```
root@zynqpeta:~# axidma 1 /dev/axidma0 ./rock512.pgm ./output_accel.pgm 5
1
time:  0.002664
root@zynqpeta:~# mount /dev/mmcblk0p1 /mnt
root@zynqpeta:~# cp output_accel.pgm /mnt/output_accel.pgm
root@zynqpeta:~# sync
root@zynqpeta:~# umount /mnt
root@zynqpeta:~#
```

Figure 6 – Putty Command Window after Executions

Now we can compare Figure 7.A and 7.B. Figure 7.A is the same as Figure 3.B, the simulated output using the hardware entry. Figure 7.B is obtained from running the convolution directly on the FPGA board. We can tell two pictures are exactly identical with no margins of errors. With this being said, we can safely draw the conclusion that our design is implemented and synthesized correctly.

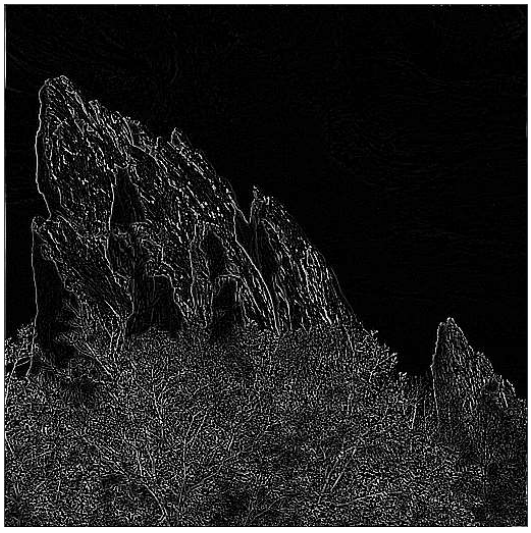

Figure 7.A – Simulated Output from Hardware Entry

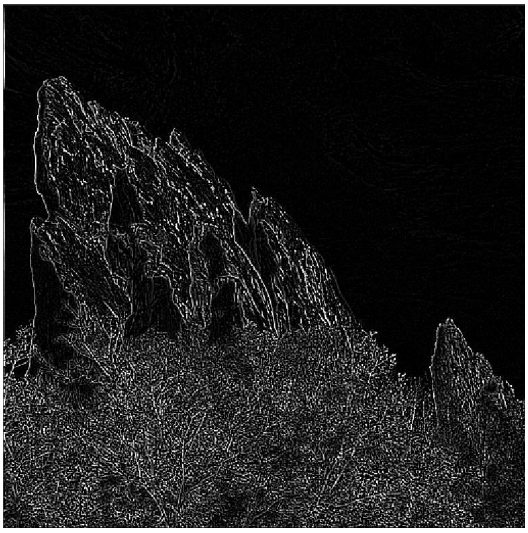

Figure 7.B – Actual-Run Output from Hardware Entry

Reflections

Although writing the C++ code was not difficult, trying to synthesize the C++ code into FPGA board can sometimes be a daunting task. When I first trying to synthesize it, a vhd file with nearly 15000 lines of codes are generated. Generating this huge vhd file took me nearly 45 minutes, and the design does not meet the timing requirements at all (a 83ns negative shift is predicted.) To eliminate the error, our TA Kevin pointed out that the rolling factor should not be specified, and array partitioning dimension should be set to 0. It made me realizes how our design can vary so much with only one or two lines of directives. It also indicated the necessity of cost/efficiency prediction prior to generating our final design.