CE 435

Embedded Systems

Lab 5 Report

SoC design and optimization

Νικόλας Ξηρομερίτης 1813
Νικόλας Κοξένογλου 1711

Outline:

Step 1 – Sobel in Software
Optimization Steps
Summary/Graphs

Step 2 – Sobel in Hardware
Implementation

Step 3 –Hardware Optimizations
Optimization Steps
Summary/Graphs

Step 1 - Sobel in Software

Opt Stage0: Getting a software baseline on ARM (1.628727 sec)

- PSNR = inf
- Run 10 samples
- Run sobel filter with the default configuration
- No code optimizations
- No compiler flags or optimizations

Opt Stage1: Enabling -O3 compiler optimizations (0.088205 sec)

- PSNR = inf
- Let the compiler apply its own set of optimizations

Opt Stage2: Introduced ARM specific compiler flags(0.086893 sec)

- PSNR = inf
- -std=c99 -mcpu=cortex-a9 -mfpu=neon -ftree-vectorize -mvectorize-with-neon-quad -mfloat-abi=hard -ffast-math
- Specify to the compiler that the CPU is a cortex A9
- Instruct the compiler to use the the NEON unit on the ARM CPU
- Instruct the compiler to vectorise loops using quad words (need conf)
- Instruct the compiler to use fast maths

Opt Stage3: Performed loop interchange (0.0774239 sec)

- PSNR = inf
- Loop interchange improved memory locality

Opt Stage4: Performed loop unroll (0.0865806 sec)

- PSNR = inf
- Unrolled the inner loop in the function convolution2D
- Since the unroll produced longer execution time it wasn't included in the next optimization step

Opt Stage5: Replaced function calling and eliminated loop invariant code (0.0774261 sec)

- PSNR = inf
- Call the convolution function and store the result in a temp value
- Perform Euclidian distance calculation using the temp value instead of calling the convolution twice to perform the calculation

Opt Stage6: Replaced Euclidian distance with Manhattan (0.0468376 sec)

- PSNR = 28.6625
- Euclidian distance calculation needs to perform a multiplication and a square root
- Manhattan requires only an addition and an absolute value
- Arbitrary precision gives more performance at the expense of calculation accuracy.
- Despite the reduction in PSNR the output image is virtually the same

Opt Stage7: Loop unrolling and code elimination(0.044295 sec)

- PSNR = 28.6625
- Program counted the total amount of ones and twos in the image
- Removed the ones and twos counter
- Eliminated both , inner and outer, loops by manually unrolling then in convolution2D function .

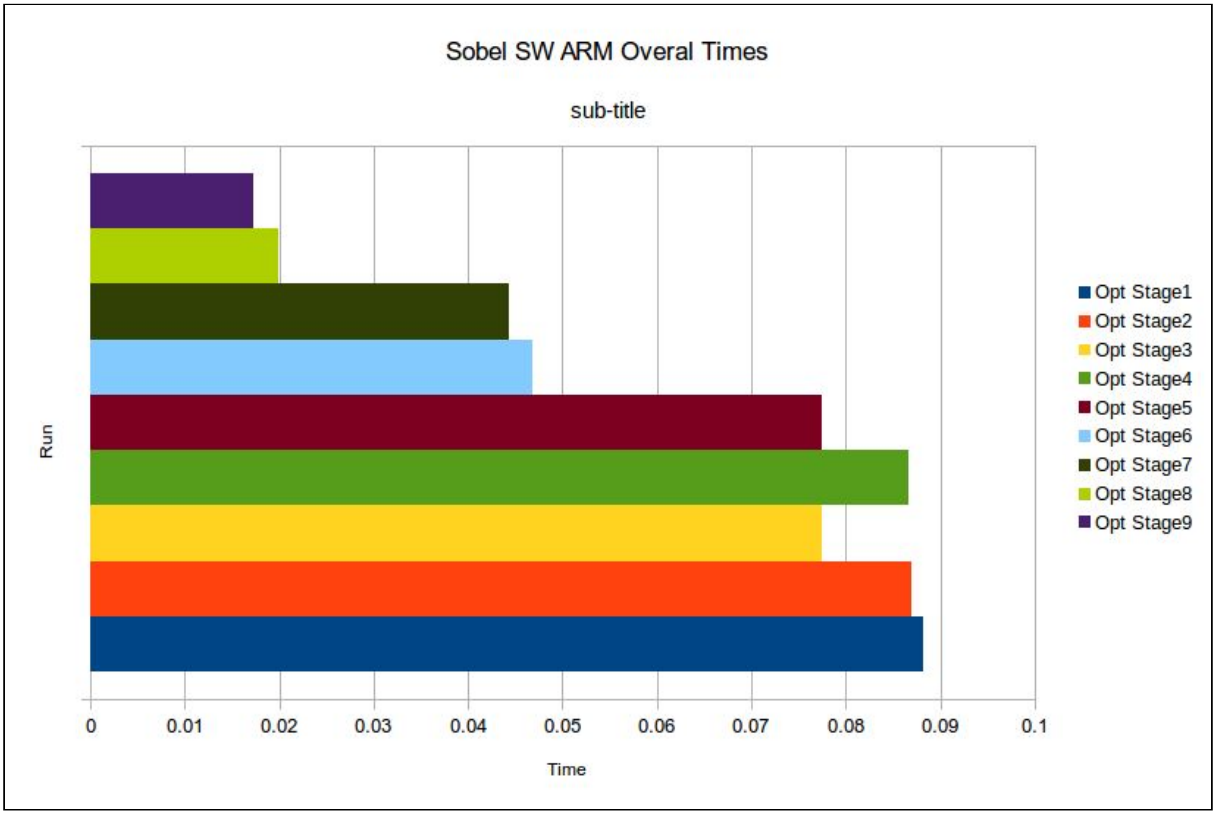
Opt Stage8: Replaced operation arrays with scalar values (0.019856 sec)

- PSNR = 28.6625
- Removed the need for operator matrices in order to apply the calculation
- Use fully unrolled loop to insert the values in the operations
- Eliminate unnecessary operations from the filter like value * 1 or value *0 with their predicted output

Opt Stage9: Replaced multiplications with shifts (0.01721 sec)

- PSNR = 28.6625
- Instead of multiplying by 2 we used left shift by 1 to perform strength reduction

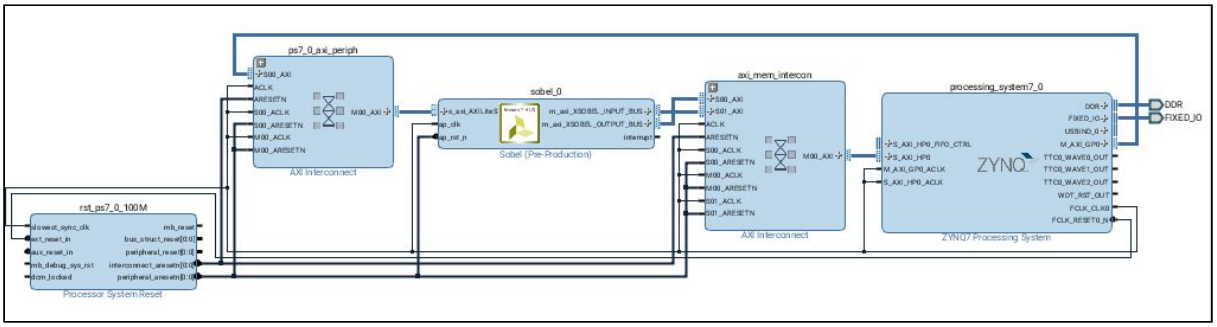
Summary/Graphs



Step 2 - Vivado HLS initial implementation

Due to neither code nor directive optimizations the initial performance of the hardware implementation was **4.35 seconds**. The HLS initial code is similar to given sobel implementation.

The communication between the processor and our peripheral was established using GP and HP registers.



Finally, we had to write a driver in order to properly configure and initialize our peripheral and the sobel input and output address arguments.

```
printf("Initializing sobel peripheral...\n");
cfgPtr = XSobel_LookupConfig(XPAR_SOBEL_0_DEVICE_ID);
if (!cfgPtr) {
    printf("ERROR: Lookup of accelerator configuration failed.\n");
    return XST_FAILURE;
}
status = XSobel_CfgInitialize(&SobelInstancePtr, cfgPtr);
if (status != XST_SUCCESS) {
    printf("HLS peripheral setup failed\n\r");
    exit(-1);
}


// initialize input and output pointers
printf("Setting input and output pointers...\n");
XSobel_Set_input_r(&SobelInstancePtr, (u32)input);
XSobel_Set_output_r(&SobelInstancePtr, (u32)output);
printf("Input at %x\n", (unsigned int)XSobel_Get_input_r(&SobelInstancePtr));
printf("Output at %x\n", (unsigned int)XSobel_Get_output_r(&SobelInstancePtr));
// check if sobel peripheral is ready
printf("Checking if sobel peripheral is ready...\n");
if (XSobel_IsReady(&SobelInstancePtr)) {
    printf("HLS peripheral is ready. Starting... \n");
}
else {
    printf("!!! HLS peripheral is not ready! Exiting...\n\r");
    exit(-1);
}

// start sobel
printf("Starting sobel...\n");
XSobel_Start(&SobelInstancePtr);
while(!XSobel_IsDone(&SobelInstancePtr)) {}
```

Step 3 - Hardware Optimizations

Opt Stage1: Optimizing Convolution & if-else (0.9833661 sec)

- Split Convolution2D into two similar functions for vertical and horizontal convolution calculations
- Manually unroll convolution's "for loops"
- Store computations into variables
- Get rid of static global arrays.
- Eliminate main's if-else statement

For example: The vertical convolution function after the transformation

```
int convolution2D_vert(int posy, int posx, const unsigned char input[SIZE*SIZE]) {

    int res;
    int posy0 = (posy - 1)*SIZE;
    int posy2 = (posy + 1)*SIZE;
    int posx0 = posx - 1;
    int posx2 = posx + 1;

    res = 0;

    res += input[posy0 + posx0];                    // vert_operator[0][0]
    res += input[posy0 + posx] << 1;                // vert_operator[0][1]
    res += input[posy0 + posx2];                    // vert_operator[0][2]

    //res += input[(posy + 0)*SIZE + posx - 1] * 0; // vert_operator[1][0]
    //res += input[(posy + 0)*SIZE + posx] * 0;     // vert_operator[1][1]
    //res += input[(posy + 0)*SIZE + posx + 1] * 0; // vert_operator[1][2]

    res += input[posy2 + posx0] * (-1);             // vert_operator[2][0]
    res += input[posy2 + posx] * (-2);              // vert_operator[2][1]
    res += input[posy2 + posx2] * (-1);             // vert_operator[2][2]

    return res;
}
```

Convolution result stored and elimination of if-else statement

```
for (i=1; i<SIZE-1; i+=1) {
    for (j=1; j<SIZE-1; j+=1) {

        conv_h = convolution2D_horiz(i, j, input);
        conv_v = convolution2D_vert(i, j, input);

        p = conv_h * conv_h + \
              conv_v * conv_v;
        res = (int) sqrt((double) p);

        output[i*SIZE + j] = (res > 255)*255 + (res <= 255)*(unsigned char)res;
    }
}
```

Opt Stage2: Manhattan Distance Calculation (0.7759861 sec)

Manhattan distance is the sum of absolute values

```
for (i=1; i<SIZE-1; i+=1) {
    for (j=1; j<SIZE-1; j+=1) {
        res = abs(convolution2D_horiz(i, j, input)) + \
              abs(convolution2D_vert(i, j, input));

        output[i*SIZE + j] = (res > 255)*255 + (res <= 255)*(unsigned char)res;
    }
}
```

Opt Stage3: MasterAXI Burst Mode & Stream Logic (0.068095 sec)

Our peripheral now has three internal buffers for the storage of the three input rows that are needed to generate an output line. In the beginning of the sobel function, two extra mem-copies are necessary to store the first two input lines and then one mem-copy per outer-loop-iteration. In this way all input elements are read only once and sequentially.

```
// store the first and second input lines into the second and third buffer lines
memcpy((unsigned char *)&input_buffer[1024], &input[0], 2*SIZE*sizeof(unsigned char));

for (i=1; i<SIZE-1; i+=1) {

    // make 2nd line 1st and 3rd line 2nd
    for (k=0; k < 2*SIZE; k+=1)
        input_buffer[k] = input_buffer[k + SIZE];

    // read one more line (the one after input's i'th line)
    memcpy((unsigned char *)&input_buffer[2*SIZE], &input[SIZE*i + SIZE], SIZE*sizeof(unsigned char));

    for (j=1; j<SIZE-1; j+=1) {
        res = abs(convolution2D_horiz(j, input_buffer)) + \
              abs(convolution2D_vert(j, input_buffer));

        output[i*SIZE + j] = (res > 255)*255 + (res <= 255)*(unsigned char)res;
    }
}
```

We tried to use another internal buffer for the output but we ended up getting the same performance (relatively worse ~0.068122 sec). It is worth mentioning that writing to output is done via system's HP registers. Also, output's elements are accessed only once by default.

Opt Stage4: Inserting Directives (0.026673 sec)

Our code has three loops (shown in the picture below). After many experimentations we decided to pipeline the “Col” loop and unroll the “Copy” loop. It is worth mentioning that “Copy” loop has dependencies that have to do with the reads and writes of input-buffer’s middle 1024 elements. Another directive that led to even better performance was the partitioning of the input buffer (type = block). This creates smaller arrays from consecutive blocks of the original array, thus it greatly improves parallelism.

```
unsigned char input_buffer[3*SIZE];
#pragma HLS ARRAY_PARTITION variable=input_buffer block factor=64 dim=1

// store the first and second input lines into the second and third buffer lines
memcpy((unsigned char *)&input_buffer[1024], &input[0], 2*SIZE*sizeof(unsigned char));

Row: for (i=1; i<SIZE-1; i+=1) {

    // make 2nd line 1st and 3rd line 2nd
    Copy: for (k=0; k < 2*SIZE; k+=1) {
        #pragma HLS UNROLL
        input_buffer[k] = input_buffer[k + SIZE];
    }

    // read one more line (the one after input's i'th line)
    memcpy((unsigned char *)&input_buffer[2*SIZE], &input[SIZE*i + SIZE], SIZE*sizeof(unsigned char));

    Col: for (j=1; j<SIZE-1; j+=1) {
        #pragma HLS PIPELINE
        res = abs(convolution2D_horiz(j, input_buffer)) + \
              abs(convolution2D_vert(j, input_buffer));

        output[i*SIZE + j] = (res > 255)*255 + (res <= 255)*(unsigned char)res;
    }
}
```

We also tried unrolling the “Row” loop, but despite the decreased and better latency indication after the “C-synthesize”, the overall performance on the board was worse (+0.03 seconds). Generally, unrolling the “Row” loop with a factor greater than two and/or partitioning the input buffer with a very large factor leads to C-synthesize failure due to too many load/store operations.

	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Attempt 5	Attempt 6	Attempt 7	Attempt 8	Attempt 9
input_buffer	-	-	-	-	-	part. blck. fact=4	part. blck. fact=16	part. blck. fact=32	part. blck. fact=64
Row (loop)	-	-	-	-	-	-	-	-	-
Copy (loop)	unroll	pipeline	flatten	unroll	unroll	unroll	unroll	unroll	unroll
Col (loop)	-	-	-	flatten	pipeline	pipeline	pipeline	pipeline	pipeline
Latency	11506720	11509786	13601820	11506720	7332872	5922525	5464670	5366558	5317501

Opt Stage5: Even More Optimizations (0.021323 sec)

Vivado HLS automatically inlines functions. Although, we got a better latency after manually defining inline directives for our convolution functions. Additionally, performance was slightly improved after replacing “+” with a “|” in the output assignment statement.

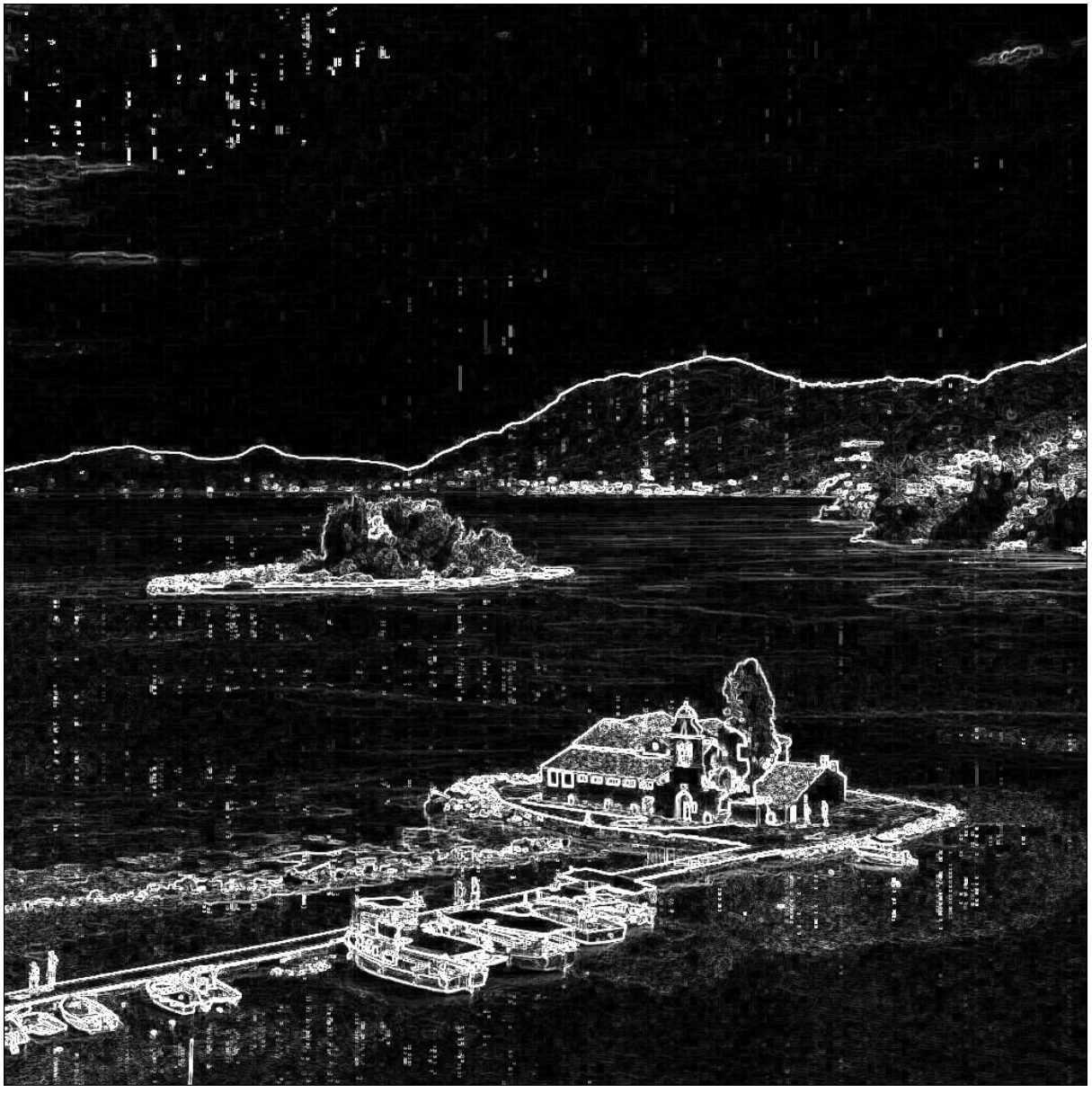
```
    unsigned char input_buffer[3*SIZE];
#pragma HLS ARRAY_PARTITION variable=input_buffer block factor=128 dim=1

    // store the first and second input lines into the second and third buffer lines
    memcpy((unsigned char *)&input_buffer[1024], &input[0], 2*SIZE*sizeof(unsigned char));

    Row: for (i=1; i<SIZE-1; i+=1) {

        // make 2nd line 1st and 3rd line 2nd
        Copy: for (k=0; k < 2*SIZE; k+=1) {
#pragma HLS UNROLL
            input_buffer[k] = input_buffer[k + SIZE];
        }

        // read one more line (the one after input's i'th line)
        memcpy((unsigned char *)&input_buffer[2*SIZE], &input[SIZE*i + SIZE], SIZE*sizeof(unsigned char));

        Col: for (j=1; j<SIZE-1; j+=1) {
#pragma HLS PIPELINE
            res = abs(convolution2D_horiz(j, input_buffer)) + \
                  abs(convolution2D_vert(j, input_buffer));

            //output[i*SIZE + j] = (res > 255)*255 + (res <= 255)*(unsigned char)res;
            output[i*SIZE + j] = (res > 255)*255 | (res <= 255)*(unsigned char)res;
        }
    }
```

Note: We also tried unrolling the “Col” loop (factor=2) without erasing the pipeline directive which led to better latency after the “C-Synthesize”. Although, the real time execution on the FPGA was significantly worse (~0.068 sec)

Opt Stage6: Overclocking Zedboard PL fabric (0.009992)

At this step we packaged our accelerator from HLS with a 4ns clock period specification. We decided to set the PL frequency to 225Mhz (through the processing system) which significantly improved our peripheral's performance without producing image artifacts and silent data corruption (0.009992 seconds).

Note: By setting a 250Mhz PL frequency, the execution time decreased to 0.008571 seconds but it also led to decreased PSNR (= 22.1034) and corrupted image pixels as seen in the picture below.



Summary/Graphs