High Pass Spatial Filter

EEL 4720

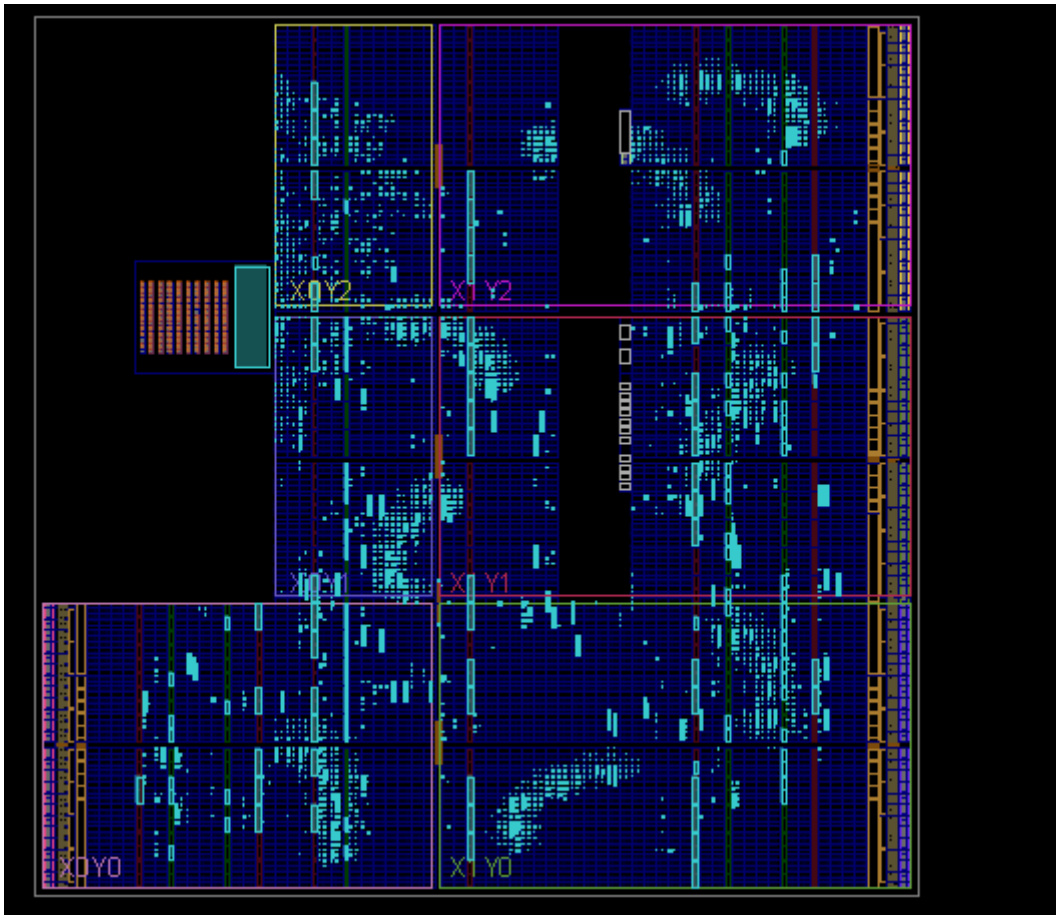
Professor:

Herman Lam

Members:

Jonathan Ganyer

Jason Lewis



Introduction

In this project we take a simple image processing application accelerate it using Xilinx Evaluation Zedboard-7020. We parallelize the application concept into one applicable on hardware in a pipeline fashion.

The high pass spatial filter is the image processing application we choose to demonstrate. This filter will basically attenuate the high frequencies in the image and output the image in the lower frequency. If we were to work with a 64 by 64 pixel image, the traditional application would take in nine of these pixels, three from row zero, three from the row one, and three from the row two. Imagine this is a 3 by 3 matrix (figure 1), the center pixel will be multiplied by nine and the remaining will be multiplied by negative one. From here it then sums these products and then divides it by nine. If this output is greater than zero, it will be stored, while if it is less than, zero will be stored. The storage location will correspond to the center pixel in the 3 by 3 matrix. This will be repeated 4096 times and each time will take multiple clocks on a traditional processor.

We can accelerate this by modeling each phase of the filter in a datapath (figure 2). The multiplication phase can be done in one clock cycle if we implement individual multipliers that take in each pixels. From here, simple eight bit adders can be used to sum the outputs of these multipliers. After a latency of four, we would this summation. After this, a divider with a one clock latency can be included that takes this summation and divides it by nine. Then a comparator with a one clock latency can be included that takes this number and outputs it if it is greater than zero or output zero if not.

This datapath, in total, has a latency of seven clock cycles to output one processed pixel, but image data can be fed into this datapath every clock cycle, so after the initial latency we

would see a processed pixel every clock cycle until all 4096 pixels are processed. Taking a total of 4125 clock cycles (pixel count plus initial latency) to process one 64 by 64 image.

Now imaging implementing this datapath eight time. All eight would take in a total of thirty pixels each clock cycle (9 pixels for one iteration + 3*(number of extra datapaths)) and output eight processed pixels every clock cycle. This would take a total of 512 plus the initial latency to finish processing a 64 by 64 image.

Algorithm

Most instances of image processing involves using a filter window to calculate the processed pixel. As seen in Fig. 1, the current pixel being processed is magnified by a scaling factor of 9, while the neighboring pixel are inverted with a scaling factor of -1. This creates an output of $a(x,y) = (\frac{\sum_{i=1}^{9} s_i * p_i}{9})$ where s_i is the scaling factor of the current pixel and p_i is the grey scale decimal value of the corresponding pixel. This result is store at the pixel location (x,y).

The high pass spatial filter breaks down into:

$$a(x,y) = \frac{\begin{matrix} 9 * (x,y) - 1 * \big((x-1),y\big) - 1 * \big((x+1),y\big) - 1\big((x-1),(y-1)\big) - 1 * \big(x,(y-1)\big) - 1 * \big((x+1),(y-1)\big) \\ -1 * \big((x-1),(y+1)\big) - 1 * \big(x,(y+1)\big) - 1 * \big((x+1),(y+1)\big) \end{matrix}}{9}$$

Our implementation will find the average of these values and then use zero as a threshold value, with any values passing into the negative range being mapped to zero. Since the final summation is divided by 9, it never exceeds 255 and therefore doesn't need an upper threshold. Pixels with near equal values will make the summation close to zero, basically attenuating these values away as can be seen in Fig. 3. Likewise, highly varying values results in the scaling factor making the summation much greater than the surrounding pixels. This results in darker tones of grey closer to 255 as can be seen in the second image of Fig. 3.

Window Algorithm		
-1	-1	-1
-1	9	-1
-1	-1	-1

Figure 1 *High Pass Spatial Filter Windowing Mask*

(0,0)	(0,1)	(0,2)		(0,4)	(0,5)	...
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	...
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	...
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	...
	...	...	...		...	...

Figure 2 *Filter window is currently processing pixel at location (2, 2).*

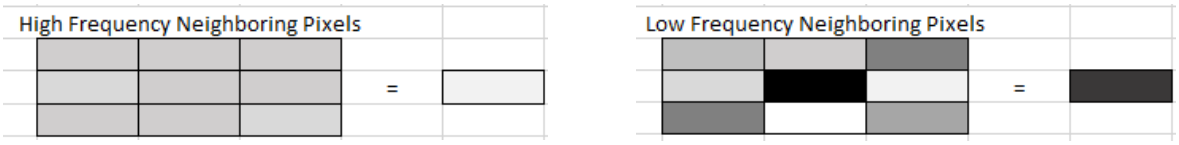


Figure 3 *Areas of similar adjacent values are attenuated. Varying values are accentuated.*

Hardware Implementation

The datapath is only one piece of the hardware that's needed to successfully instantiate the high pass spatial filter onto an FPGA. It will also need input and output block rams, address generators to drive the ram, buffers to feed the datapaths, and a controller to drive all of the above.

Block Diagram

The functional model of the High Pass Spatial Filter can be found in Fig. 4. The main components consists of three input BRAMs with 64-bit words and 2^{13} word depth where we store our original grey scale image. The image initialized in block RAM 1, 2 and 3 contain slightly offset row configurations to handle end of row transitions. An external program connects to the slave registers on the FPGA fabric, sending a go signal to the controller which stays stalled until this signal is received. Smart buffers are then enabled during the latency phase and are filled before beginning the computation.

These buffers can handle up to 128-bits of data, cycling in 64-bits during each clock cycle. To avoid data loss between cycling data, the smart buffer outputs 80-bits at a time to the datapath. The input address generator is then notified to begin incrementing the block RAM addresses until it reaches its last word. For the 256x256 grey scale image with 64-bit words we ended up using 8128 addresses to complete the go state. Our datapath consists of adders, multipliers, registers, dividers and comparators. These deal with decoding valid data in addition to computing the High Pass Spatial Filter.

Once the datapath indicates data is valid, the output BRAM will begin filling up with data as the output address generator increments concurrently. Once the data has been converted, a done signal is asserted and ready to be read back from the slave register.

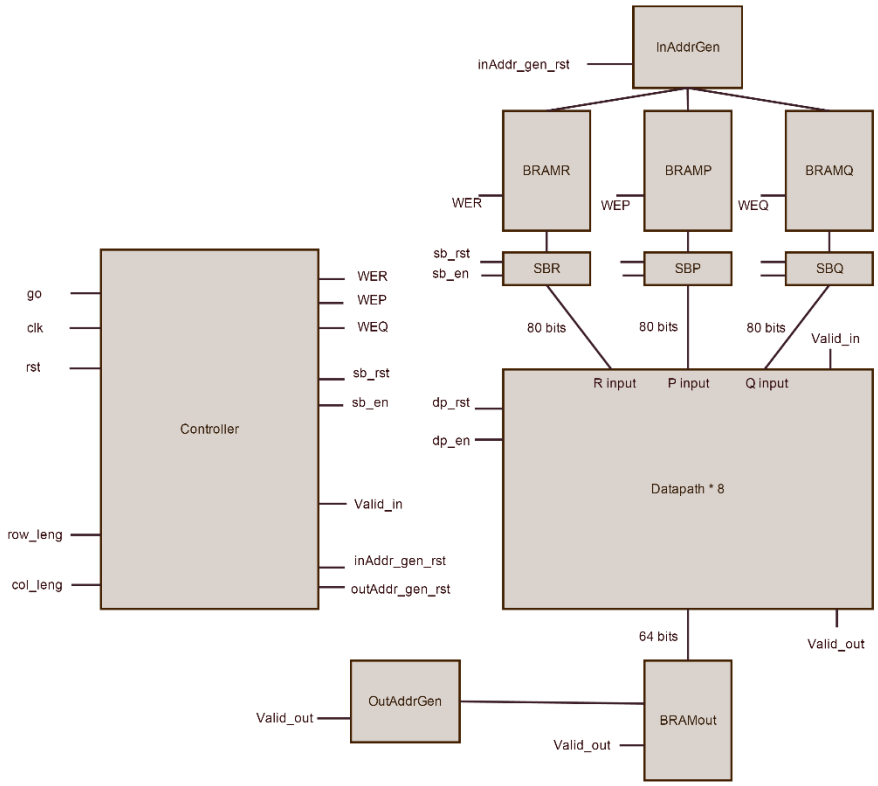


Figure 4 *Logic component of High Pass Spatial Filter*

Address Generator

The address generator references the current block RAM word being read by the smart buffers. It works just like a counter, waiting for an enable signal to begin counting. After each subsequent clock cycle, the counter will increment by one. This advances the block RAM address being referenced and feeds the next piece of data in to be processed. The output address generator enable signal is dependent on the valid out signal that comes through the datapath when data is ready to be stored.

Smart Buffer

The smart buffer works in synchronous with the address generator. When the address generator is enabled, along with the read enables for the input block RAMs, the three smart buffers take in the first word from their respective memories, each of which is a different row on the pixel map of the image. The smart buffers themselves output two words at a time so on the first clock cycle the smart buffer stores this word into the lower 64 bits of its output. On the next clock cycle these lower bits are then shifted to the higher 64 bits and a new word replaces the lower. It is not until the outputs of the smart buffers have two consecutive word in their lower and higher orders that the datapaths are then enabled to retrieve these outputs.

Datapath

The datapath works exactly as discussed above. After the two clock cycle latency of the smart buffers, the datapath starts receiving the first window of pixels. Each pixel is 8 bits, and a datapath needs nine pixels to output a valid result, three from each smart buffer. The smart buffers have been tailored to maximize the amount of pixels we can produce on the ZedBoard, due to hardware constraints. So we need eight datapaths in total to match the smart buffer's bandwidth. Each consecutive datapath will take in two out of the three pixels that were inputted into the previous datapath and one new pixel. This will continue until the eighth datapath has the pixels it needs. In total, all eight datapaths will take in ten pixels, or 80 bits, from each of the three smart buffers each clock cycle and outputting eight processed pixels ever clock after the initial seven clock cycle delay.

Controller

Used to control the enabling of components at certain times, the controller is a state machine that deals with initial stalling, latency, parallel computation, and data storing. When the go signal is received, the controller goes into an initial latency step of 7 cycles where it enables the smart buffers and begins filling the datapath with input pixels until it gets its first valid output. While in the go state, the datapath and output block RAM write enable will be asserted to continue storing. The done signal will not be asserted until after an internal counter reaches the max pixel value, after which point the output latency must first finish storing the remaining pixels to the output block RAM.

Memory Formatting

Memory instantiation involves taking a 256x256 image and mirroring it into three separate block RAMs, with each being 256x254. Each address in our block RAM contains 64-bit words, totaling in at 8128 addresses. The difference between the three models is the reference by which each one begins and ends at. While block RAM R begins at the first row and ends two rows early, P begins on row two and ends on the next to last row. Block RAM Q begins on the third row and ends on the last row. This type of mirroring allows data to be decoded much easier and avoids problems that arise from custom resolution sizes. Memory is initialized using .coe files generated from MATLAB. The function we used took our input image, converted it to grayscale, downscaled it to 256x256, and then made the appropriate offset memory blocks for all three block RAMs.

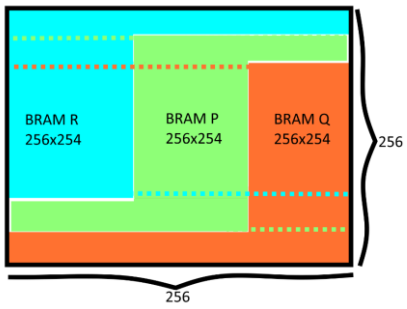


Figure 5 *Block RAMs are offset by one row each to accommodate windowing method.*

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=

32836f52756e4d73
465a74725c7c6865
b9af915d87543135
6c6a64766e767380
150e62517c6e4968
625038261c17110a
```

Figure 6 *MATLAB function was created to generate the appropriate .coe memory files.*

Software Implementation

Matlab

Function called “brom_gen_all.m” shown in Fig. 7, was written to generate the appropriate *.coe and *.h files for each of our block RAMs. The *.coe files were used by the Xilinx ISE for memory initialization in the block RAM generator IP core. The *.h files were written to create a simple array of values for each of the block RAMs so that we could easily run the filter algorithm on the software end just by including a header file in our SDK. Another function called “brom_read.m” shown in Fig. 8, was written to take the output block RAM data and convert it into a grayscale 2-D array that can output to the computer screen.

```
function [rows, cols] = brom_gen_all(infile, coeRh, coePh, coeQh, coeR, coeP, coeQ, numrows, numcols)
% BRAM .COE and .H File Generator
% Written by: Jonathan Ganver
a = zeros(65538,1);
b = zeros(65538,1);
cc = zeros(65538,1);
img = rgb2gray(imread(infile));


imgresized = imresize(img, [numrows numcols]);

[rows, cols, rgb] = size(imgresized);

imgscaled = imgresized/16 - 1;
imshow(imgscaled*16);
```

Figure 7 *MATLAB makes it easy to generate memory initialization files.*

```
function [pic] = bramread(infile, rows, cols)

pic = zeros(rows,cols);

fid = fopen(infile);

for r = 1 : rows
   for c = 1 : (cols / 4) + 1
       tline = char(fgets(fid));
       if mod(c,64) == 0
          for p = 1 : 2
            pixel = tline((2*p-1):(2*p));
            pic(r,4*(c-1)+p) = hex2dec(pixel);
          end
       else
           for p = 1 : 4
            pixel = tline((2*p-1):(2*p));
            pic(r,4*(c-1)+p) = hex2dec(pixel);
          end
       end
   end
end
figure(3);
imshow(uint8(pic * 16));

fclose(fid);
```

Figure 8 *MATLAB is also useful for converting raw data back into an image array.*

Xilinx SDK

The SDK provided by Xilinx is where we retrieved the data from the output BRAM and calculate the high pass spatial filter algorithm using the ARM processor. Results from these hardware and software solutions are then displayed on putty. It's also where we fetch the clock count of both the FPGA and ARM processes.

For the ARM algorithm we chose not to store the contents of the input BRAMS directly from the FPGA. Instead we used separate header files for each BRAM that was created by MATLAB in the same manner we created the .coe files for BRAM instantiation. This allows for the ARM processor to start working independent of the FPGA. From here we use a for loop to simulate the algorithm in the same way we did on the FPGA and on MATLAB (figure 9) and store the results into an array.

For the FPGA portion of our C code, we simply retrieve the data made by the hardware and store each address into an array (figure 10).

After this, we output each index in the ARM output array followed by the FPGA output array. These outputs are displayed onto the putty terminal and then copied and saved into a txt file for MATLAB parsing and display.

In order to time these processes we created an independent IP core in the Xilinx XPS that's sole purpose was to count the elapsed FPGA clocks when the appropriate signals were asserted. We then took this clock count and multiplied it by the FPGA clock period to get a rough estimate of the time elapsed for ARM algorithm duration in the SDK.

```
    parseArr(bromr, bramr);
    parseArr(bromp, bramp);
    parseArr(bromq, bramq);

    PROJECT11_TIMER_mWriteSlaveReg0(TIMER, 0, 0x00000001);
    PROJECT11_TIMER_mWriteSlaveReg0(TIMER, 0, 0x00000000);

    PROJECT11_TIMER_mWriteSlaveReg1(TIMER, 0, 0x00000001);
    for (o = 0; o < 65024; o++) {
        sum = ((-1)*bramr[o]+(-1)*bramr[o+1]+(-1)*bramr[o+2]+
                (-1)*bramp[o]+(9)*bramp[o+1]+(-1)*bramp[o+2]+
                (-1)*bramq[o]+(-1)*bramq[o+1]+(-1)*bramq[o+2]);
        final = sum/9;

        if (final >= 0) {
            ARMOutput[o] = final;
        }
        else {
            ARMOutput[o] = 0;
        }
    }
    PROJECT11_TIMER_mWriteSlaveReg1(TIMER, 0, 0x00000000);
    int ARM_clocks = PROJECT11_TIMER_mReadSlaveReg4(TIMER, 0);
```

Figure 9 *C code algorithm for the ARM processor in Xilinx SDK*

```
//Read the output BRAM result.
PROJECT12_REDUCED_mWriteSlaveReg7(PROJECT, 0, 0x00000001);
int tmp = 0, tmp2 = 0;
for(i=0;i<8128;i++)
    {
        if (i%32 != 31) {
            PROJECT12_REDUCED_mWriteSlaveReg6(PROJECT,0,i);
            tmp = PROJECT12_REDUCED_mReadSlaveReg9(PROJECT,0);
            tmp2 = PROJECT12_REDUCED_mReadSlaveReg10(PROJECT,0);
            hwOutput[i] = tmp;
            hwOutput2[i] = tmp2;
        }
        else {
            PROJECT12_REDUCED_mWriteSlaveReg6(PROJECT,0,i);
            tmp = PROJECT12_REDUCED_mReadSlaveReg9(PROJECT,0);
            tmp2 = PROJECT12_REDUCED_mReadSlaveReg10(PROJECT,0);
            tmp2 = backW(tmp2);
            hwOutput2[i] = snip(tmp2);
            hwOutput[i] = tmp;
        }
    }
```

Figure 10 *C code method for FPGA output BRAM retrieval in Xilinx SDK*

Timing Results

Image processing is usually a good example of how FPGAs can be used to accelerate typically sequential algorithms into pipelining architecture. The high pass spatial filter is no different, with the 32-bit pipeline running at 84 times the throughput of our ARM processor clocking in at 162.66 µs. When upgrading to a 64-bit pipeline, we nearly double these results at 81.38 µs. This equals roughly a 167 times throughput over the ARM processor.

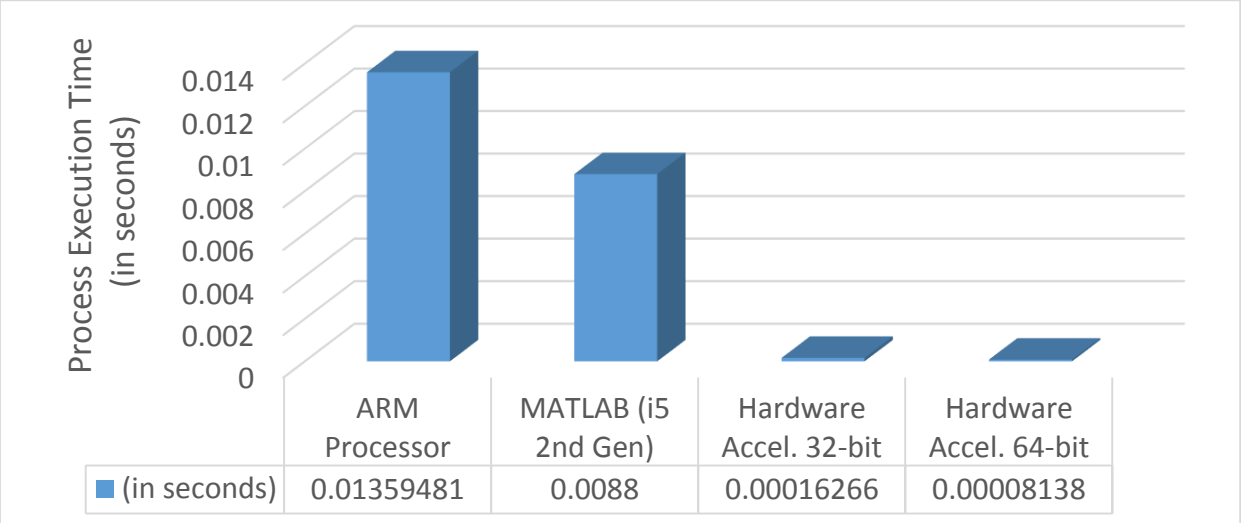


Figure 11 *Hardware acceleration resulted in massive increases in throughput.*

Conclusion

We ended up getting great increase in throughput by creating a High Pass Spatial Filter on the Zed board. The mirroring technique with the three block RAMs greatly simplified the controller and the 64-bit bandwidth allowed for some considerable pipelining. The speedup is great for an image of such small resolution and would only be more notable as the image size increased as can be seen in our timing results. As shown in Fig. 12, the results are also the same as the software solution, meaning hardware acceleration is the perfect application for this kind of image processing.

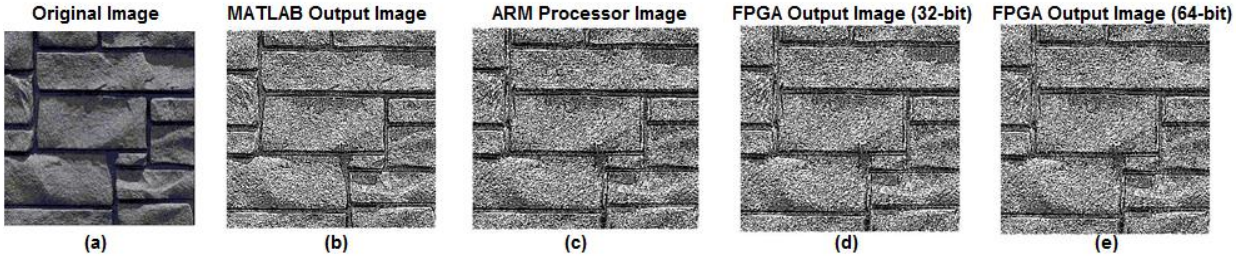


Figure 12 Original image (a) compared against many software/hardware outputs (b-e).