200 times speedup on a case of Image filters optimization using High Level Synthesis

Abstract—**This paper describes two design techniques applicable to image filters IP Cores that enable designers to improve performance and reduce area, by processing multiple pixels per cycle and sharing resources between filters, when designing using Vivado High Level Synthesis.**

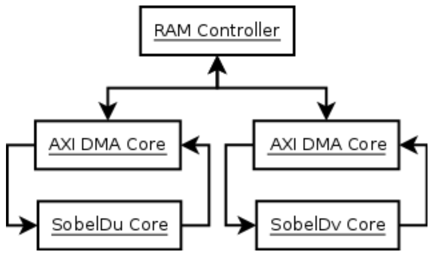
Vivado HLS optimization directives are analyzed in the context of image filters IP Cores, before introducing the two techniques that require hand made modifications to the code.

Detailed performance and area results are presented, as well as comparisons between the different optimizations and the filter algorithm implementation on software.

I. Introduction

While working on my undergraduate thesis, where we have implemented a stereo visual odometry solution in Zynq-7000, following the methodology described in our previous paper [1], we accelerated the images features detection and extraction section of the algorithm, implementing various image filters in programmable logic using Vivado High Level Synthesis. Two of these filters, SobelDu and SobelDv, were implemented for fixed size, 8 bits, gray scale images as independent IPCores that approximate the horizontal and vertical derivatives respectively.

In our implementation we use AXI DMA cores to feed Sobel IP Cores with images from external RAM, converting the source image to an AXI stream of pixels and storing the resulting image pixel stream on RAM. This way the Sobel IPCores are independent from the image source.



Conceptual interconnections diagram

In this paper we explore the optimization techniques we have applied during the implementation of the Sobel filters in programmable logic as IPCores. These techniques allow to achieve a 200 times speedup, compared to the filters implementation in software running on one of the ARM cores inside the Zynq-7000. Combining multiple filters inside the same IPCore and sharing resources enable to save area.

The next section of this paper describes the implementation, using Vivado HLS, of a simple filter. SobelDu computes a horizontal derivative approximation of every pixel in an image. We used Vivado directives [3] to define the IP interface and improve performance.

The following two sections (III and IV) introduce the optimization techniques ”processing multiple pixels per cycle” and ”merging filters”. These optimizations require changes to the source code and aren’t achievable using only the directives provided by Vivado HLS.

Finally sections V and VI present results and conclusions.

II. Initial SobelDu High Level Synthesis Implementation and Optimizations

We designed an IP Core, named SobelDu, that computes the horizontal derivative approximation for every pixel in an 8 bit image producing another 8 bit image. Details about the memory structures used can be found in ”Implementing Memory Structures for Video Processing in the Vivado HLS Tool” [2].

Image source and destination are streams, so we mapped them to AXI Stream ports using Vivado HLS directives:

```
void sobel_du(ap_axiu<8,1,1,1> *src,
ap_axiu<8,1,1,1> *dest) {
#pragma HLS INTERFACE axis port=dest
#pragma HLS INTERFACE axis port=src
#pragma HLS INTERFACE s_axilite
port=return
```

Since our filter uses a 5x5 kernel, it needs a 5x5 pixels window around each pixel to calculate its value in the output image. To be able to access all the pixels in the window on the same clock cycle, and thus, calculate the output for each pixel in one cycle, we mapped every window cell to an independent register, using the **ARRAY_PARTITION** directive, as shown below.

```
pixel_t window[5][5];
#pragma HLS ARRAY_PARTITION
   variable=window complete dim=0
partition
```

Similarly, the filter needs to store 4 complete image lines in order to feed the window with previous lines pixels. For every input pixel one read and one write must be done from every line buffer to feed the window and shift the line buffers. Using the ARRAY_PARTITION directive, as shown below, we mapped each line buffer to a different Block RAM, since Block RAMs are dual port, all the required read and write operations can be done on the same clock cycle.

```
pixel_t line_buffers[4][IMAGE_WIDTH];
```

```
#pragma HLS ARRAY_PARTITION
   variable=line_buffers complete dim=1
partition
```

Finally, we optimized the inner loop of the algorithm applying the **PIPELINE** directive to exploit parallelism, as shown below:

```
for(row=0;row<IMAGE_HEIGHT;row++)
{
  for(col=0;row<IMAGE_WIDTH;col++)
  {
    #pragma HLS PIPELINE
    pixel_t new_pixel = src->data;
    // Shift window, Feed window from
    // line buffers and Shift line buffer
    .
    .
    .
    if(row >= 4 && col >= 4)
    {  // Valid data to output
       dest->data = calc_sobel_du(window);
       .
       .
       .
       dest++;
    }
  }
}
```

Using ARRAY_PARTITION and PIPELINE directives we achieved a theoretical latency of ∼1 $\frac{pixel}{cycle}$, compared to ∼0.01 $\frac{pixel}{cycle}$ without those directives, a 100 times speedup. Resources usage is almost the same with and without applying the directives.

Experimentation showed that SobelDu IP Core is 29.5 times faster than the software version of the algorithm, running on one of the ARM cores inside the same Zynq-7000. Similar results were obtained implementing the Sobel vertical derivative approximation in another IP Core, SobelDv.

The next two sections describe hand made optimizations applied to the code to achieve an even better latency and reduce resources consumption when implementing SobelDu and SobelDv together.

III. Performance Improvement: Processing Multiple Pixels per Cycle

While it's more intuitive to think the filters input and output as streams of 8 bit pixels, that kind of designs doesn't take full advantage of wider memory buses, that allow IPCores to read and write N pixels on the same cycle.

Since Zynq-7000 provides 64 bit wide buses, we created SobelDu64 that uses 64 bits streams, reading and writing blocks of 8 pixels. To achieve that, we modified SobelDu interface, redefining the streams width as follows:

```
void sobel_du64(ap_axiu<64,1,1,1> *src,
ap_axiu<64,1,1,1> *dest) {
#pragma HLS INTERFACE axis port=dest
#pragma HLS INTERFACE axis port=src
#pragma HLS INTERFACE s_axilite
port=return
```

To be able to process 8 pixels at the same time the window had to be extended to contain a 5x5 window around each of them. It would be enough to use a 5x12 window, but we have chosen to use a 5x16 one to simplify the task of moving data from the line buffers to the window.

```
pixel_t window[5][16];
#pragma HLS ARRAY_PARTITION
   variable=line_buffers complete dim=1
partition
```

The line buffers definition was also changed to work with blocks of 8 pixels, as shown below:

```
u64 line_buffers[4][IMAGE_WIDTH / 8];
#pragma HLS ARRAY_PARTITION
   variable=line_buffers complete dim=1
partition
```

Finally the inner loop limits and body were changed, according to the size of the input data, reading and producing more pixels on each iteration. The function calc_sobel_du was extended to receive the window and the central pixel offset inside the window.

```
for(row=0; row<IMAGE_HEIGHT; row++) {
  for(col=0; col<IMAGE_WIDTH/8; col++) {
      #pragma HLS PIPELINE
      u64 new_data = src->data;
      // Shift window 8 pixels
      // Feed window 8 new pixels from
      // line buffers
      // Shift line buffer one 8 pixel
block
      .
      .
      .
      if(row >= 4 && col >= 1)
      {  // Valid data to output
         u64 tmp = 0;
         for(u8 i=0; i < 8; i++)
           tmp |= (calc_sobel_du(window,
i) << (8*i));
         dest->data = tmp;
         .
         .
         .
         dest++;
      }
  }
}
```

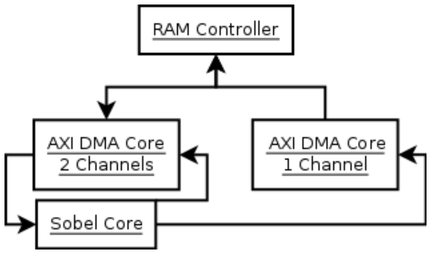
This modifications achieved a theoretical latency of ∼8 $\frac{pixel}{cycle}$. Area consumption approximately doubled compared to the original IP version.

Practical experimentation shows that the resulting IP Core is ∼200 times faster than the software version of the algorithm.

IV. Resources consumption reduction: Merging Filters

We found that when implementing multiple filters for the same image it's worth to merge them creating one IPCore for both functionalities, contrary to the divide & conquer approach. This way the input stream, line buffers, windows, control logic and, possibly, some calculations can be shared, thus reducing area usage. Sharing the same input stream also reduces the number of DMA channels required. In turn, having less DMA channels and only one IP to control, the controlling software complexity is reduced.

We merged SobelDu and SobelDv IP Cores in one IPCore.



Conceptual interconnections diagram, one DMA channel is eliminated

We defined the interface, as follows, to have two output streams.

```
void sobel(ap_axiu<8,1,1,1>
*src, ap_axiu<8,1,1,1> *dest_du,
ap_axiu<8,1,1,1> *dest_dv) {
#pragma HLS INTERFACE axis port=dest_du
#pragma HLS INTERFACE axis port=dest_dv
#pragma HLS INTERFACE axis port=src
#pragma HLS INTERFACE s_axilite
port=return
```

The rest of the code is almost the same as the one from SobelDu, shown in previous sections, but with the following modification to produce two output streams:

```
...
if(row >= 4 && col >= 4)
{  // Valid data to output
   dest_du->data = calc_sobel_du(window);
   dest_du++;
   dest_dv->data = calc_sobel_dv(window);
   dest_dv++;
}
...
```

Applying this technique to merge SobelDu and SobelDv in one new IP Core, named Sobel, a reduction of ∼50% in area was obtained with no loss in throughput.

V. Results

This sections shows the results that were obtained using the techniques described, comparing throughput and resources consumption. All the measures were made using a Digilent Zybo development board, with a Zynq-7000. Programmable Logic clock speed was setup at 150Mhz and CPUs clock at 650Mhz. To avoid memory bottlenecks each DMA channel was connected to independents memory ports.

The time measured, used to calculate the number of input pixels processed per clock cycle, include the time needed to setup DMA transactions. Images from the Karlsruhe Dataset [4] with a resolution of 1344x372 pixels were used on the tests.

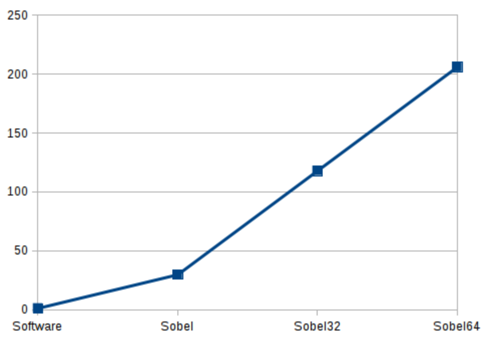
Implementation	$\frac{in\_pix}{cycle}$	BRAM	FF	LUT	DMAs
Software SobelDu	0.05	0	0	0	0
Software SobelDv	0.05	0	0	0	0
Software Sobel	0.0337	0	0	0	0
IP SobelDu	0.997	4	1004	1105	2
IP SobelDv	0.997	4	930	1125	2
IP SobelDu and IP SobelDv	0.997	8	1934	2230	4
IP Sobel	0.998	4	1098	1347	3
IP SobelDu32	3.97	4	1432	1757	2
IP SobelDv32	3.97	4	1313	1819	2
IP SobelDu32 and IP SobelDv32	3.968	8	2745	3576	4
IP Sobel32	3.97	4	1905	2675	3
IP SobelDu64	6.94	8	2162	2580	2
IP SobelDv64	6.94	8	1897	2678	2
IP SobelDu64 and IP SobelDv64	6.93	16	4059	5258	4
IP Sobel64	6.94	8	2977	4398	3

*All software versions were built with all compiler optimizations enabled. A timer at 150Mhz was used to measure the processed input pixels per cycle ratio.

- SobelDu: approximates horizontal derivative one pixel per loop cycle.
- SobelDv: approximates vertical derivative one pixel per loop cycle.
- Sobel: approximates simultaneously horizontal and vertical derivatives one input pixel per loop cycle, producing two output values.

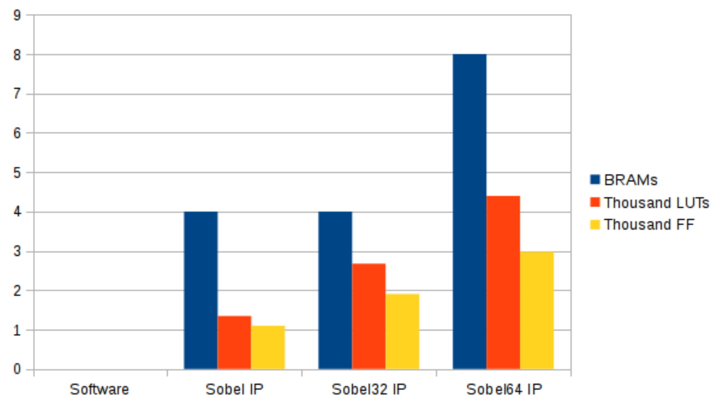
The 32 and 64 variants work processing 4 and 8 pixels per loop cycle, respectively, improving throughput at the expense of an increment in area.

The following graphic shows the ratios between the throughputs of the Sobel, Sobel32 and Sobel64 IP Cores and the algorithm implementation on software. That is $\frac{IPCore_{throughput}}{Software_{throughput}}$, where $IPCore_{throughput}$ and $Software_{throughput}$ are the throughput of each component, measured in $\frac{ProcessedInputPixels}{cycle}$.



Throughput comparison between software and the Sobel, Sobel32 and Sobel64 IP Cores

Sobel64 throughput is $\frac{6.94}{0.0377} \simeq 205$ times the software implementation throughput.



Resources usage comparison between different implementations

VI. Conclusion

Combining Vivado HLS directives and the two techniques introduced in this paper we were able to build an IP Core that is more tan 200 times faster than the software solution.

It's important to note that the techniques described are easily applicable to any image filter that process an image sequentially and uses a window around each pixel to calculate the output value. For example, in my thesis I built a single IP Core that apply 3 different convolutions to an image and outputs the results in 3 pixels streams, processing 8 input pixels on each loop iteration. The optimal number of pixels to process on each iteration must be determined finding a balance between the required latency and resources usage, accounting also for memory bottlenecks.

Acknowledgment

This work was made possible thanks to UBACYT 2014-2017 ”System on Chip (SoC) in FPGA (Field Programmable Gate Array)” in combination with the Xilinx University Program responsible of the software licenses and a reduced price for the FPGA development boards.

References

[1] M. Ángel García and P. Borensztejn, “Image processing systems in fpga: Components-connectors methodology,” in *SPL2014 - Designer Forum*, 2014.

[2] F. M. Vallina, “Implementing memory structures for video processing in the vivado hls tool,” in *XAPP793*, 2012.

[3] Xilinx, “High-level synthesis,” in *Vivado Design Suite User Guide*, 2014.

[4] A. Geiger, J. Ziegler, and C. Stiller, “Stereoscan: Dense 3d reconstruction in real-time,” in *Intelligent Vehicles Symposium (IV)*, 2011.