Gaussian filtering for FPGA based image processing with High-Level Synthesis tools

O.S. Shipitko[1], A.S. Grigoryev[1]

[1]Institute for Information Transmission Problems (Kharkevich Institute) – IITP RAS, Bol'shoy Karetnyy per. 19, Moscow, Russia, 127051

Abstract. With the gradual improvement and uprising interest from the industry to High-Level Synthesis tools, like Vivado HLS form Xilinx, Field Programmable Gate Arrays are becoming an attractive option for accelerator architecture in image processing domain. However, an efficient high-level design still requires knowledge of hardware specifics. A great amount of image processing operations falls into a group of convolution-based operators - operators which result depends only on a particular pixel and its neighborhood and obtained by performing a convolution between a kernel and a part of an image. This paper investigates the impact of factors, such as kernel size, target frequency, convolution implementation specifics, floating-point vs. fixed-point filter kernel, on resulting register-transfer level design of convolution-based operators and FPGA resources utilization. The Gaussian filter was analyzed as an example of a convolution-based operator. It is shown experimentally that floating-point operators require a noticeably larger amount of resources, rather fixed-point once. Resulting clock frequency independence from kernel size is demonstrated as well as the number of used flip-flops grows with the increasing target clock frequency is investigated in this work.

Keywords: Keywords: FPGA, High-level synthesis, Image processing, Gaussian filter.

1. Introduction

Field Programmable Gate Arrays (FPGAs) gain an increasing popularity as an accelerator platform in various fields [1]. Image processing is not an exception. FPGA proved to be a suitable option for many image processing applications where severe performance, energy efficiency, and power requirements are imposed [2, 3, 4]. However, the efficient FPGA design is a non-trivial task which requires a long development period and, in case of image processing, deep understanding of both software algorithms development and hardware design [5, 6]. Introduction of High-Level Synthesis (HLS) made possible to synthesize register-transfer level (RTL) hardware design for FPGA directly from the high-level software description, reducing development time and required hardware knowledge [7, 8]. The fundamental challenges existing HLS compilers still have to overcome are discussed in [9]. Even though for particularly structured code, modern HLS tools can generate designs that comparable in performance with hand-coded RTL in terms of both resources and processing speed [7, 10], the software still has to be written in a particular way to enable automatic control and data flow identification and parallelizing [11]. For instance, it is not obvious how different software implementation-specific factors affect the efficiency of the hardware design. Thus, the goal of this work is to conduct a study aimed at evaluation of implementation specifics (kernel size, its type etc.) impact on a resulting RTL design.

The rest of the paper is structured as follows: section 2 presents the sliding window architecture used for efficient implementation of image processing pipeline. Section 3 describes conducted experiments and discusses obtained results.

2. Sliding window architecture

In order to develop an efficient image processing algorithm for FPGA stream-based processing has to be exploited [5]. Typical streaming architecture is built using a combination of row buffers (line buffers) – elements able to store one row of image and window buffers – 2D arrays able to store a local area of an image required for computation of current output pixel. The combination of these two data structures allows reading input image/video pixel by pixel, optimizing the usage of FPGAs limited memory. As a result, minimal amount of pixels required for computation of the filtered value of the current pixel is stored at the time. The minimal required amount of line buffers may be estimated as $H - 1$, where H denotes a vertical size of a kernel of a local operator being used. Window buffer has to have the same size as a kernel. Thus, for instance, for Gaussian filter with the kernel 3x3, the optimal implementation will require 2 line buffers and 3x3 window buffer. The overall architecture is shown in figure 1.

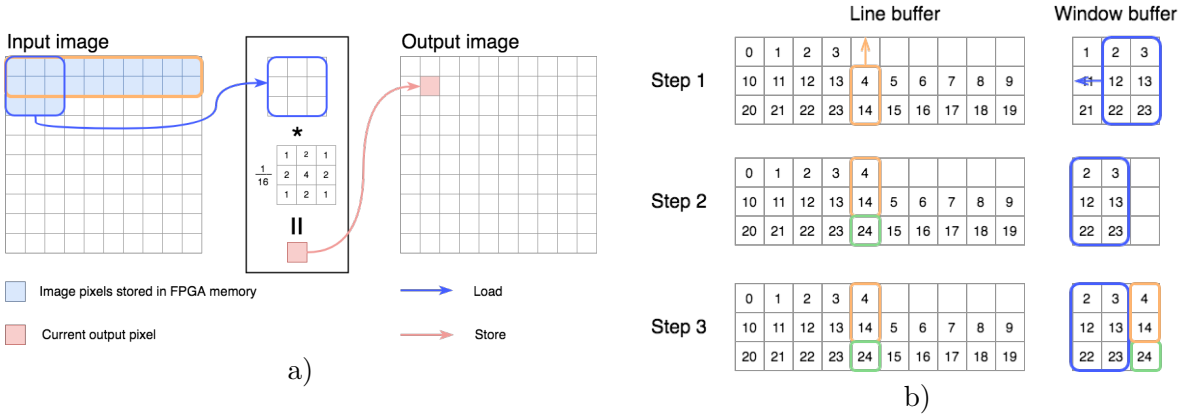

Figure 1. Sliding window architecture (a) and step-by-step line buffer and sliding window filling (b).

3. Results and discussion

This section presents a comparison of several Gaussian filter implementations, their efficiency, and corresponding FPGA resources utilization. All synthesis results, presented in this section, were obtained with single color 8-bit per pixel images. The image size has been set to 1080 x 1920 pixels. As a target FPGA the ZC702 Evaluation board based on the XC7Z020 CLG484-1 All Programmable SoC device was used. All presented in this paper experiments were conducted with Vivado Design Suite HLx Editions of version 2016.2.

Four different types of filters were implemented for further analysis: (a) Implementation using sliding window architecture presented in figure 1 and implemented with C-language arrays; (b) Implementation using HIPA^{cc} framework [12, 13]; (c) Implementation similar to (a), but using data structures (window buffer, line buffer) provided by Vivado Video Library instead of standard C arrays.

HIPA^{cc} – the Heterogeneous Image Processing Acceleration Framework developed by Richard Membarth and Oliver Reiche, allows to design image processing kernels and algorithms in a domain-specific language (DSL). It was used in this work in order to verify developed Gaussian filter.

3.1. Kernel size

For each of the implementations described above, different kernels were tested. Kernels are presented in figure 2. Resource usage comparison for different kernels is presented in tables 1, 2 and 3.

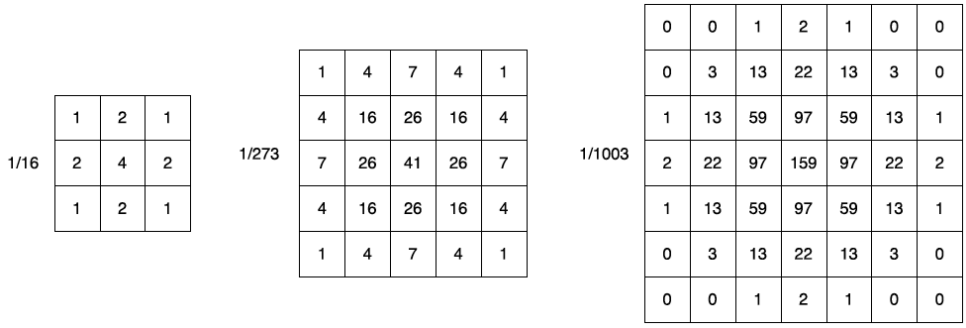

Figure 2. Discrete approximation of the Gaussian kernels 3x3, 5x5, 7x7.

Table 1. Comparison of different kernel sizes for implementation (a)

Kernel size	LUT	FF	DSP	BRAM	Latency (cycles)	Freq [MHz]
3X3	309(0%)	478(0%)	1(0%)	2(0%)	2073606	178
5X5	724(1%)	1200(1%)	7(3%)	4(1%)	2073615	178
7X7	838(1%)	1923(1%)	24(10%)	6(2%)	2073619	178

Table 2. Comparison of different kernel sizes for HIPA^{cc} implementation (b)

Kernel size	LUT	FF	DSP	BRAM	Latency (cycles)	Freq [MHz]
3X3	1409(2%)	1563(1%)	0(0%)	2(0%)	2076640	157
5X5	1372(2%)	1736(1%)	0(0%)	4(1%)	2079620	268

Table 3. Comparison of different kernel sizes for implementation (c)

Kernel size	LUT	FF	DSP	BRAM	Latency (cycles)	Freq [MHz]
3X3	319(0%)	543(0%)	1(0%)	2(0%)	2073606	178
5X5	726(1%)	1320(1%)	7(3%)	4(1%)	2073615	178

As it might be seen from the results, the system with a bigger kernel uses much more resources but performs the operation at approximately the same time as the system with a smaller kernel. However, it is not the case for HIPA^{cc} implementation, since it performs operations with bigger kernels faster than with smaller once. It is also notable that, the developed architecture (a) outperforms HIPA^{cc} implementation with kernel size 3x3, but loses tremendously in clock frequency with bigger kernel sizes. Usage of data structures provided by Vivado Video Library doesnt bring any improvements in performance, however, it simplifies and speeds up the development process, providing convenient templates of line and window buffers and functions to work with.

3.2. Clock frequency vs. flip-flops number

It is known that the number of used flip-flops increases with the increasing target frequency [7]. The maximum frequency is limited by the longest path data has to take in one clock cycle. To shorten the longest path Vivado HLS compiler inserts additional flip-flops, trying to reach target frequency specified by a programmer. Figure 3 shows the dependency of used flip-flops vs. target clock frequency.

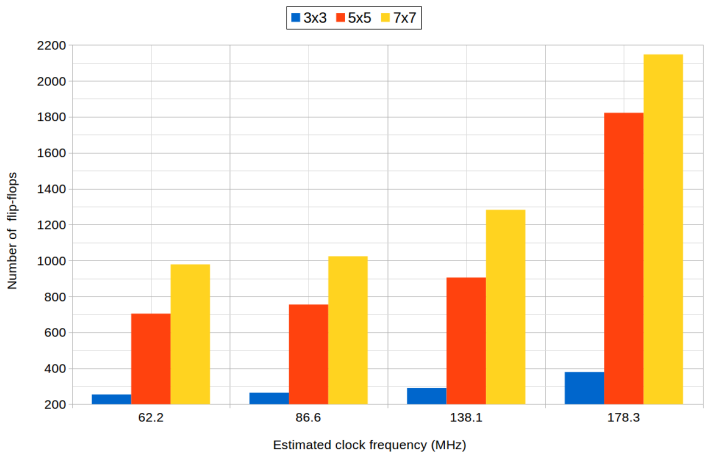

Figure 3. Clock frequency vs. flip-flop number for fixed-point filter implementation.

3.3. Floating-point vs. fixed-point implementation

Two 3x3 kernels presented on figure 4 were compared. The results of the comparison are shown in table 4.

0.075	0.124	0.075
0.124	0.204	0.124
0.075	0.124	0.075

1/16

1	2	1
2	4	2
1	2	1

Figure 4. Floating-point Gaussian kernel (on the left) and its discrete approximation (on the right).

Table 4. Floating-point vs. fixed-point implementation

Kernel type	LUT	FF	DSP	BRAM	Latency (cycles)	Freq [MHz]
Fixed	309(0%)	478(0%)	1(0%)	2(0%)	2073606	178
Floating	20656(38%)	17724(16%)	46(20%)	2(0%)	2073818	157

Fixed-point implementation uses much less resources and performs filtering faster than floating-point implementation. However, it should be noted that discrete approximation of the filter's coefficients implies accuracy losses, which might be crucial for some applications.

3.4. Different ways to perform convolution

Three different convolution implementations, presented in listing 1, listing 2, and listing 3, were tested in a combination with two different kernel sizes (3x3, 5x5). These implementations

hereinafter referred as *Impl1*, *Impl2* and *Impl3* accordingly. The goal of the experiment is to force HLS compiler to optimize a convolution function to produce a balanced adder tree which is known to be an optimal structure for operations requiring summation with accumulation [11]. Balanced adder tree reduces the area of a design by minimizing the latency, which in turn reduces the number of registers used. The result of the comparison is presented in the tables 5 and 6.

Listing 1: *Impl1.*

```
for (int i=0; i < WINDOW_SIZE; i++) {
  for (int j=0; j < WINDOW_SIZE; j++) {
    new_pix += window[i*WINDOW_SIZE+j] * gauss_kernel[WINDOW_SIZE-i][WINDOW_SIZE-j];
  }
}
```

Listing 2: *Impl2.*

```
  new_pix += window[0];
  new_pix += window[1] * 2;
  new_pix += window[2];
  new_pix += window[3] * 2;
  new_pix += window[4] * 4;
  new_pix += window[5] * 2;
  new_pix += window[6];
  new_pix += window[7] * 2;
  new_pix += window[8];
```

Listing 3: *Impl3.*

```
new_pix = window[0] + window[1]*2 + window[2] + window[3]*2 + window[4]*4 +
          window[5]*2 + window[6] + window[7]*2 + window[8];
```

Table 5. 3x3 kernel

Implementation	LUT	FF	DSP	BRAM	Latency (cycles)	Freq [MHz]
Impl1	309(0%)	478(0%)	1(0%)	2(0%)	2073606	178
Impl2	306(0%)	470(0%)	1(0%)	2(0%)	2073606	178
Impl3	306(0%)	470(0%)	1(0%)	2(0%)	2073606	178

Table 6. 5x5 kernel

Implementation	LUT	FF	DSP	BRAM	Latency (cycles)	Freq (MHz)
Impl1	724(1%)	1200(1%)	7(3%)	4(1%)	2073615	178
Impl2	604(1%)	1039(0%)	4(1%)	4(1%)	2073614	178
Impl3	604(1%)	1039(0%)	4(1%)	4(1%)	2073614	178

As it might be seen from the results, there is almost no difference between these three approaches. The reason is that Vivado HLS tries to build a balanced tree structure out of a number of related additions that can be scheduled in parallel if a directive telling the compiler to unroll loops is specified. Thereby, the results are proving that the Vivado HLS is able to construct a balanced adder from all three forms of convolution functions.

4. Conclusion

Although using of HLS can simplify and accelerates the development of FPGA-based applications, it is still requires careful design space exploration. It is crucial to remember that existing HLS tools do not provide full abstraction and the result of the development is not software but hardware. The efficiency of resulting FPGA solution and its resources utilization depends heavily on many factors which have to be taken into account on the programming stage. Floating-point operations implemented on FPGA are usually inefficient and consume a tremendous amount of resources, therefore should be avoided. Kernel size doesn't affect clock frequency and just increases the number of resources required for storing bigger kernel and temporary image areas. A number of used flip-flops grows rapidly with the increasing target clock frequency and generally bigger for bigger kernels. Therefore a trade-off between target speed and resources utilization should be considered by a developer. A benefit achieved with the use of vendor-provided libraries has to be noted. They provide convenient abstractions usually at no additional resources cost. Thus, for instance, window and line buffers from Vivado Video Library might be used as an alternative to hand-programmed data structures. Results obtained in this work might be extended to any convolution-based image processing operator implemented on FPGA with HLS.

5. References

[1] Berry, P.C.F. Design of an Imaging System based on FPGA Technology and CMOS Imager // IEEE Field Programmable Technology, 2004.

[2] Dias, F., Berry, F., Srot, J., Marmoiton, F. Hardware, design and implementation issues on a FPGA-based smart camera // Distributed Smart Cameras. First ACM/IEEE International Conference. - 2007. - P. 20-26.

[3] Mosqueron, R., Dubois, J., Paindavoine, M. High-speed smart camera with high resolution // EURASIP Journal on Embedded Systems. - 2007. - Vol. 1. - P. 23-23.

[4] Hu, Y., Prasanna, V. K. Energy-efficient parameterized 2-D separable convolution on FPGA // Green Computing Conference International (IGCC), 2014. - P. 1-10.

[5] Bailey, D.G. Design for embedded image processing on FPGA // John Wiley & Sons, 2011.

[6] Lim, Y.K., Kleeman, L., Drummond, T. Algorithmic methodologies for FPGA-based vision // Machine vision and applications. - 2013. - Vol. 24(6). - 1197-1211.

[7] BDTI. High-level synthesis tools for Xilinx FPGAs. Technical report, Berkley Design Technology Inc., 2010.

[8] Page, I. Closing the gap between hardware and software: hardware-software cosynthesis at Oxford, 1996.

[9] Bailey, D.G. The advantages and limitations of high level synthesis for FPGA based image processing // Proceedings of the 9th International Conference on Distributed Smart Cameras, 2015. - P. 134-139

[10] Winterstein, F., Bayliss, S., Constantinides, G. A. High-level synthesis of dynamic data structures: A case study using Vivado HLS // Field-Programmable Technology (FPT) International Conference on IEEE, 2013. - P. 362-365.

[11] Fingeroff, M. High-level synthesis: blue book. Xlibris Corporation, 2010.

[12] Membarth, R., Reiche, O., Hannig, F., Teich, J., Krner, M., Eckert, W. Hipa cc: A domain-specific language and compiler for image processing // IEEE Transactions on Parallel and Distributed Systems. - 2016. - Vol. 27(1). - P. 210-224.

[13] Reiche, O., Zkan, M. A., Membarth, R., Teich, J., Hannig, F. Generating FPGA-based Image Processing Accelerators with Hipacc.