This is the author's version of the work. The definitive work was published in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL), Munich, Germany, September 2-4, 2014.*

An Image Processing Library for C-based High-Level Synthesis

Moritz Schmid, Nicolas Apelt, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany

***Abstract*—We introduce a library for the productive development of image processing accelerators using C-based high-level synthesis. The key concept of our approach is to provide a set of generic building blocks that is applicable to a multitude of image processing applications. An efficient memory architecture that facilitates easy integration of point and local image processing operators is the centerpiece of the library. The generic building blocks are kept very compact and can be tailored to support sophisticated processing techniques. The representation enables the designer to comply with specific design requirements, such as stringent timing constraints or limited resource budgets. Results show a significant gain in productivity compared to hand coded implementation while delivering comparable performance and resource requirements.**

I. Introduction

Due to their high computational power in combination with excellent energy efficiency, Field Programmable Gate Arrays (FPGAs) have not only recently been a target for the implementation of signal processing systems. As designing for FPGA hardware imposes the often proclaimed productivity gap, a move to higher abstraction levels has been a lively research area for the past decades. High-Level Synthesis (HLS) frameworks have meanwhile matured to be a considerable alternative to hand-coded Register Transfer Level (RTL) designs, often providing significant productivity gains at an equal quality of results. However, the available tools are still very complex and cannot be seen as a substitute for detailed knowledge of the underlying hardware.

In this work, we present a lightweight library for C-based HLS to aid designers in the development of hardware accelerators for image processing. The library is specified around an efficient streaming data memory architecture and is thus ideally suited to support point and local image processing operators. A fundamental concept is to keep the library as compact as possible. We therefore only aim at providing the essential features to expedite the development of image filtering implementations, which, in consequence enables easy customization and adaptation to a plethora of different applications. Although the presented library is foremost targeted at the Vivado HLS framework from Xilinx [1], the presented work is applicable to C-based high-level synthesis in general.

The rest of this work is organized as follows. Section II reviews related work. The hardware architecture for stream-based image processing on FPGAs of the proposed approach is introduced in Section III using a two-dimensional Gaussian convolution as a motivational example. Section IV presents how point and local operators can be specified for image processing. Experimental results are discussed in Section V.

II. Related Work

High-level synthesis for reconfigurable logic has received a lot of attention over the past decades. Several excellent surveys exist to provide an overview of the developments [2] [3] [4]. Despite this progress, familiarity with the target architecture is paramount to increase productivity and obtain good results. For image and signal processing, it is often a challenge to bring together different areas of expertise, for example algorithm design and hardware implementation. Commercial HLS frameworks often include specific libraries to provide elementary architecture constructs and digital signal processing implementations. Special image processing libraries are also available, for example, for Calypto's CatapultC [5] or the Impulse CoDeveloper C-to-FPGA Tools from Impulse Accelerated Technologies [6]. The concept here lies mostly in providing developers with a set of custom functions and their efficient implementation.

Another approach is, for example, the partial port of the OpenCV library [7] for Vivado HLS from Xilinx [1]. OpenCV provides the designer with a memory infrastructure and specific algorithm implementations. Extending these for problems, that are not taken into account in the library directly might become very challenging. Moreover, there exist specific frameworks to create hardware implementations from signal and image processing designs developed in the Matlab/Simulink, such as HDL-coder [8] and Synphony [9]. An overview and evaluation of the different methods is presented by Zoss et al. in [10].

In contrast to the numerous closed-source commercial projects, there also exist specific academic approaches. PARO [11] [12], for example, is targeted at nested loop programs and therefore especially well suitable for accelerating dxigital signal processing applications. To aid designers in the image processing domain, PARO also features domain-specific augmentations to provide efficient implementations on a high abstraction level [13]. A widely recognized approach is SPIRAL [14], which is a framework for the generation of hard- and software implementations of digital signal processing algorithms (linear transformations) using a domain-specific language. Another considerable approach was proposed by George et al. [15].

III. Stream-based Image Processing on FPGAs

One of the key concepts of the here presented library is to provide means to productively develop accelerators for image

processing using HLS and efficiently integrate these with the data streaming methodology. Vivado HLS, for example, implements a specific `stream` class to represent streaming data interfaces using a FIFO semantic. Our approach uses this streaming class as an interface between hardware accelerators and the FPGA fabric. In this way, we can integrate the library components seamlessly with other key components of the FPGA fabric, such as memory controllers and bus interfaces. Moreover, we can use the streaming interface to construct fully pipelined complex filtering systems.

A. Motivational Example

Denote an image of resolution $n \times m$ as a function $\mathbf{f} : \{1, \ldots, n\} \times \{1, \ldots, m\} \rightarrow \mathbb{R}$. Applying a Gaussian convolution as a local operator to an image $\mathbf{f}(\mathbf{x})$ to obtain the smoothed image $\tilde{\mathbf{f}}(\mathbf{x})$ can be expressed as

$$\tilde{\mathbf{f}}(\mathbf{x}) = \sum_{\xi} \mathbf{f}(\mathbf{x}) g(\mathbf{x}, \xi), \tag{1}$$

where $g(\mathbf{x}, \xi)$ measures the *geometric distance* between the center pixel $\mathbf{x}$ and a pixel located at a nearby point ξ within a neighborhood centered around $\mathbf{x}$. In terms of Gaussian filtering, it is defined as

$$g(\mathbf{x}, \xi) = \frac{1}{2\pi\sigma^2} e^{-\frac{1}{2}\left(\frac{||\xi - \mathbf{x}||}{\sigma}\right)^2}, \tag{2}$$

where σ serves as a parameter to control the impact of the low-pass filtering in the spatial domain. In the programming language C, the Gaussian convolution for a neighborhood of size $w \times w$ can be stated rather easily using a two-dimensional array of coefficients, as shown in Listing 1.

```
1 data_t coeff[w][w];
2 data_t gauss(data_t window[w][w])
3 {
4   data_t accu = 0;
5   for(int i = 0; i < w; i++)
6     for(int j = 0; j < w; j++)
7       accu += window[i][j]*coeff[i][j];
8   return accu;
9 }
```

Listing 1. 2D Gaussian convolution for a single pixel.

In general, Listing 1 can be regarded as a template for two-dimensional convolution, not only for Gaussian kernels, but also, for example, for edge detection using a Sobel kernel.

B. Memory Architecture

Local operators, such as the Gaussian convolution, operate on a local neighborhood, also referred to as *window*, and therefore often need to access a pixel more than once. Thus, handling streaming data on an FPGA requires a memory architecture to retain data for multiple accesses. An efficient memory architecture for streaming data uses a combination of *line buffers* for storage of complete image lines and *memory windows* for the actual processing of local neighborhoods. The interaction between the line buffer and the memory window is shown in Figure 1. New image data are stored in the line buffer, which is then used to update the memory window. In order to update the window, that is, to move it to the right, all of its contents are shifted to the left and the pixels of the rightmost column are obtained from the current column of the line buffer as well as the new input element.

C. Causality and Border Handling

Another challenge for stream-based image processing using local operators is that almost half of the data in the window is not available when processing the central pixel. This means the filter is acausal because it uses some samples from the *future*. In practice, only causal filters may be used, so the response of the filter must be shifted by $\tau_x = \tau_y = \frac{w-1}{2}$ steps into the past. A solution is to simply wait until all the data for the window are available, which increases the *group delay* by $\tau = (\tau_y \cdot w) + \tau_x$ pixels.

In addition to causality issues, we also have to deal with processing the image borders, where a part of the window lies outside of the actual image. Just skipping over these positions is usually not an option since it may either decrease the output image by nearly half of the window size in each direction or leave a visible frame around the processed region. Border treatment provides several techniques to extrapolate the missing information from available image data or replace it with a constant value.

D. Filter Assembly

Under consideration of the aforementioned aspects, we can assemble the filter structure to obtain a hardware accelerator for the two-dimensional Gaussian convolution. We read the input image pixel by pixel from the input stream, store the data in the line buffer and compute the 3×3 Gaussian convolution for every pixel using the neighborhood pixels as provided by the memory window and write the processed pixel to the output stream. The assembly is encapsulated in the `process`-function in the following Listing 2, which also serves as the entry point for high-level synthesis.

```
1 #define GRP_DLY 1 // group delay
2 void process(
3   hls::stream<data_t> &img_in, // input stream
4   hls::stream<data_t> &img_out // output stream
5 {
6   data_t[2][n] lb; // line buffer
7   data_t[3][3] wnd; // memory window
8
9   data_t in_pix, out_pix;
10
11  ROW_LOOP: for(int row = 0; row < m+GRP_DLY; row++){
12    COL_LOOP: for(int col = 0; col < n+GRP_DLY; col++){
13      //READ NEW PIXEL
14      if(row < m && col < n){
15        new_pix << img_in;
16        // BUFFERING AND BORDER TREATMENT
17        ...
18      }
19      if(row >= GRP_DLY && col >= GRP_DLY){
20        // COMPUTE CONVOLUTION
21        out_pix = gauss(wnd);
22        // WRITE OUTPUT PIXEL
23        img_out << out_pix;
24 } } } }
```

Listing 2. Streaming data computation of the 2D Gaussian convolution. High-level transformations are not included.

As Listing 2 shows, the complete code can become quite lengthy. Especially if border treatment is included, the required code may easily exceed 200 lines and poses many chances for coding errors. In contrast, the next section will demonstrate how two-dimensional convolutions, such as this motivational example can be specified using the proposed approach.

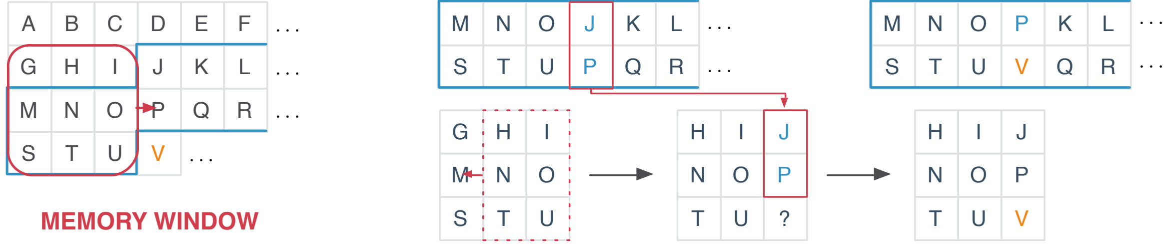

Fig. 1. An example of the interaction between the line buffer and the memory window. As the window traverses in a scanline from left to right over the image, the current window content is shifted to the left and the new column is inserted at the rightmost position.

E. Library Integration

The Gaussian convolution, as shown in Listing 2 can be constructed using the library with the help of two of its components: (a) a class for 1D and 2D convolution, called `ConstWindow`, and (b) the `proc_locOp`-function, which encapsulates the proposed solution for the memory architecture. The specification of the example discussed above is shown in Listing 3.

```
1 void gauss(hls::stream &img_in,
2   hls::stream &img_out){
3   ConstWindow<data_t> local_operator(Gauss_3x3);
4   proc_locOp(img_in, img_out,
5     BorderTreatment::BORDER_MIRROR,
6     local_operator
7   );
8 }
```

Listing 3. Specification of the Gaussian filter using the proposed library.

The `proc_locOp` function takes as arguments the data streams for input and output, the desired method for border treatment, and the local operator. The constructor takes the coefficients for a 3×3 Gaussian kernel and computes the convolution. In essence, the `proc_locOp` function can handle any local operator that is specified in the form of class in similar fashion as the `ConstWindow` class.

IV. Specification of Point and Local Operators

Image processing can be broadly classified into *point*, *local*, and *global operators*. A point operator maps each pixel of the input image to a single pixel of the output image. Local operators use a local region of pixels of the input image to derive a single pixel of the output image. Examples are window-based image filters. Global operators, in contrast, require the complete image for the computation of the output and are thus not discussed in this work.

A. Point Operators

Point operators can be specified as *Multiple Input, Multiple Ouput* (MIMO) systems, as, for example, the following listing shows a specific point operator to duplicate a stream.

```
1 void splitStream(hls::stream<data_t> &img_in,
2   hls::stream<data_t> &img_out_1,
3   hls::stream<data_t> &img_out_2){
4   const data_t in_pix;
5   for(int i = 0; i < width*height; ++i){
6     img_in >> in_pix;
7     img_out_1 << in_pix; img_out_2 << in_pix;
8   }
9 }
```

Listing 4. Point operator for stream duplication.

Point operators for image transformations, such as to increase the brightness of an image, can be specified as follows.

```
1 template<typename T> class PntOpBright{
2 private:
3   const T factor;
4 public:
5   PntOpBright(const T _factor) : factor(_factor) {}
6   T operator() (const T &pixel) const{
7     return pixel*factor;
8   }
9 };
```

Listing 5. Point operator for brightness increase.

The specified point operator can then be included in an image processing accelerator as follows.

```
1 PntOpBright<data_t> pnt_operator(1.1);
2 proc_pntOp(img_in,img_out, pnt_operator);
```

Listing 6. Execution of a point operator.

B. Local Operators

We provide two simple ways of implementing local operators in our library. Either by specifying the weights right ahead in the HLS code which will be synthesized and therefore cannot be modified afterwards. Or by passing them as interface parameters prior to the image processing, so the weights can be changed between the processing of two images. For convenience, we have predefined frequently used kernels (such as Gaussian, Sobel, and averaging kernels) to elude the process of repeatedly typing these in. The implementation of such a kernel (here, for example, a 5×5 Gaussian) can be kept rather short, as in Listing 7.

```
1 ConstWindow<float> local_operator(Gauss5x5);
```

Listing 7. Execution of a predefined convolution kernel.

A custom kernel only requires to add a definition of the kernel elements, as for example a two-dimensional emboss kernel:

```
1 const float emboss_krnl[] = {
2     -2.0f, -1.0f, 0.0f,
3     -1.0f,  1.0f, 1.0f,
4      0.0f,  2.0f, 2.0f
5   };
6 ConstWindow<float> emboss(emboss_krnl);
7 proc_locOp(img_in, img_out,
8     width, height,
9     BorderTreatment::BORDER_CLAMP,
10    emboss);
```

Listing 8. Specification of a custom convolution kernel.

TABLE I
COMPARISON OF GAUSSIAN AND MEDIAN FILTER IMPLEMENTATIONS USING OPENCV AND THE HERE PROPOSED APPROACH.

	Proposed Library								OpenCV 2013.4							
	II	LAT	BRAM	DSP	SLICE	FF	LUT	F [MHz]	II	LAT	BRAM	DSP	SLICE	FF	LUT	F [MHz]
Gauss 3x3	1	265362	2	45	1650	6068	4362	223.1	1	281091	3	27	3048	8851	9254	174.2
Gauss 5x5	1	268590	4	125	4270	14079	10780	220.4	1	281091	5	75	7806	22712	24448	168.2
Gauss 7x7	1	271937	6	246	8721	27696	22341	202.5	1	282115	7	147	14462	43638	45875	167.9
Sobel 3x3	1	265308	2	45	1652	6069	4380	214.1	1	281116	3	27	4311	12901	11995	180.4
Sobel 5x5	1	268527	4	125	4905	10950	14241	217.0	1	281186	5	75	10669	33015	30540	169.0
Sobel 7x7	1	271822	6	246	7821	22672	27965	194.1	1	282237	7	126	20989	62925	57981	149.9

Apart from two-dimensional convolution, the proposed approach is also suitable for other local operators, in general.

V. EXPERIMENTAL RESULTS

In order to assess the performance of the image processing accelerators developed with the proposed library, we have evaluated two-dimensional convolution with Gaussian (Gauss) and Sobel kernels. For these kernels, Vivado HLS contains an OpenCV implementation, we were able to compare our results to (refer to Table I).

To give an idea of how productively the accelerators may be developed, Table II lists the required *Lines of Code* (LoC) to specify the image filters. As the Gaussian and the Sobel filter can be computed using predefined coefficient windows, the required code is quite short, if compared to a hand coded (h/c) implementation. As the table also lists the LoC for the specification of these two filters in OpenCV, it can be seen that our approach is almost equal. The algorithms were implemented

TABLE II
LINES OF CODE FOR THE EVALUATED FILTERS

	Gauss (h/c)	Gauss	Gauss/O CV	Sobel	Sobel/O CV
LoC	187	31	29	31	29

for images of dimension 512×512 and evaluated for different window sizes targeting a fixed initiation interval (II = 1). We have used the Vivado Design Suite 2013.4 for HLS and implementation on a ZYNQ 7045. The results are shown in Table I in terms of achieved latency (LAT, total number of required clock cycles), required BRAMs (BRAM), DSP blocks (DSP), flip-flops (FF), lookup tables (LUT), and required slices (SLICE). Also, the table shows the achievable maximum clock frequency (F [MHz]). The comparison of the two-dimensional convolution kernels between our approach and OpenCV shows that for the used version of Vivado HLS, our approach can produce faster accelerators at more efficient resource usage, in terms of choosing DSP slices for arithmetic operations over LUTs. Moreover, since our approach enables easy access to high-level transformations which, for example control the targeted initiation interval, the designer can easily evaluate different levels of parallelization for more complex designs.

VI. CONCLUSIONS

In this work, we have presented a lightweight library to support productive development of hardware accelerators for stream-based image processing using C-based HLS. The library supports point and local operators and seamlessly integrates with the FPGA fabric using FIFO interfaces. We have shown that the accelerators designed with the proposed library can achieve a higher quality of results at an equal coding effort than existing frameworks. Moreover, the compact representation of the library allows easy access to high-level transformations in order to control performance and resource requirements. In its current state, the proposed library can be obtained from the authors and will be made publicly available at a later development stage.

REFERENCES

[1] Xilinx Inc., "Vivado Design Suite User Guide – HLS," 2013.

[2] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 8–17, July 2009.

[3] D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *Design Test of Computers, IEEE*, vol. 11, no. 4, pp. 44–54, Winter 1994.

[4] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18–25, July 2009.

[5] Calypto Design Systems Inc., "Calypto Product Family Datasheet," 2012.

[6] Impulse Accelerated Technologies, "Impulse CoDeveloper C-to-FPGA Tools," 2013, http://www.impulseaccelerated.com/products_universal.htm, [Accessed: 20-Mar-2014].

[7] G. Bradski, "The OpenCV library," *Dr. Dobb's Journal of Software Tools*, 2000.

[8] MathWorks, "HDL Coder," 2013, http://www.mathworks.com/products/hdl-coder, [Accessed: 20-Mar-2014].

[9] Synopsis Inc., "Synphony Model Compiler," 2014, http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/Synphony-Model-Compiler.aspx, [Accessed: 20-Mar-2014].

[10] R. Zoss, A. Habegger, V. Bandi, J. Goette, and M. Jacomet, "Comparing signal processing hardware-synthesis methods based on the Matlab tool-chain," in *Electronic Design, Test and Application (DELTA), 2011 Sixth IEEE International Symposium on*, Jan 2011, pp. 281–286.

[11] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich, "PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications," in *Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC)*, ser. Lecture Notes in Computer Science (LNCS). London, United Kingdom: Springer, Mar. 2008, pp. 287–293.

[12] M. Schmid, F. Hannig, A. Tanase, and J. Teich, "High-level synthesis revised – Generation of FPGA accelerators from a domain-specific language using the polyhedron model," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing. Amsterdam, The Netherlands: IOS Press, 2014, vol. 25, pp. 497–506.

[13] M. Schmid, A. Tanase, V. Bhadouria, F. Hannig, J. Teich, and D. Ghoshal, "Domain-specific augmentations for high-level synthesis," in *Proceedings of the 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, Jun. 2014, pp. 1–5.

[14] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*. Springer, 2011, ch. Spiral.

[15] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne, "Making domain-specific hardware synthesis tools cost-efficient," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, Dec. 2013, pp. 120–127.