Hybrid Code Description for Developing Fast and Resource Efficient Image Processing Architectures

Konrad Häublein*, Marc Reichenbach*, Oliver Reiche†, M. Akif Özkan†,
Dietmar Fey*, Frank Hannig†, and Jürgen Teich†
*Computer Architecture, Martensstraße 3, 91058 Erlangen
†Hardware-Software-Co-Design, Cauerstraße 11, 91058 Erlangen
Department of Computer Science, Friedrich-Alexander University of Erlangen-Nürnberg (FAU), Germany
{konrad.haeublein, marc.reichenbach, oliver.reiche, akif.oezkan, dietmar.fey, hannig, teich}@cs.fau.de

Abstract—**Image processing algorithms applied on programmable embedded systems very often do not meet the given constraints in terms of real time capability. Mapping these algorithms to reconfigurable hardware solves this issue, but demands further specific knowledge in hardware development. The design process can be accelerated by code generation through high-level synthesis, which is very flexible. However, the quality of synthesis depends on many factors like the provided constraints, code description, and algorithmic complexity. Hence, optimizing these parameters may improve the generated results in terms of logic and memory utilization, as well as data throughput and synthesis duration. In this work, we aim to exploit domain-specific knowledge for a hybrid code description in order to benefit from rapid development through high-level synthesis in combination with throughput optimized generic hardware descriptions. By utilizing code generation techniques, the entire design flow gets accelerated. Our synthesis results show a similar resource utilization and achievable throughput to a pure HDL described hardware.**

Keywords: Image Processing, High-Level Synthesis, Code Transformation, Hybrid Code Description

I. Introduction

In the industry and medical domain image processing algorithms, which have become more and more complex, are covering many fields of applications. However, in many scenarios certain constraints like real time capability must be respected. Especially in the embedded field, this is a crucial issue. In this case, parallel processing through an accelerator must be employed. For many embedded applications reconfigurable hardware, like FPGAs, proved to be an efficient platform. The algorithm can be implemented directly in hardware, is able to run on low frequency, and therefore, requires less power than standard CPU solutions. A huge disadvantage of FPGAs is the enormous effort in development. Regular knowledge in programming languages like C or C++ is by far not enough to map an application to these complex devices, since these languages are not made for parallel description on such a low hardware level.

In the last decade, hardware description languages, like *VHDL* or *Verilog*, have been mainly used for hardware synthesis. The current FPGA design tools optimized their synthesis process to these HDLs. But with rising algorithmic complexity, e.g. pipelines in a data flow description, the design effort has tremendously increased. Additionally, the design approach still requires many experience in digital design as well as specific knowledge about the employed FPGA architecture.

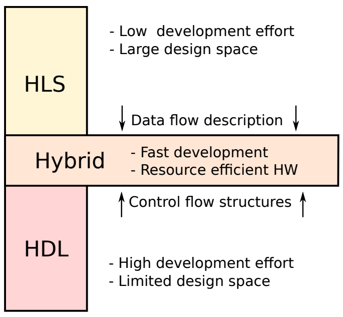


Fig. 1. Through combining HLS and HDL descriptions to a hybrid design flow, the developer benefits from a fast development in a domain-specific environment and is able to generate resource efficient hardware.

In order to simplify and accelerate the FPGA design flow *High-Level Synthesis* (HLS) has gained more and more attention in recent years. Through an HLS tool it is possible to synthesize hardware from a higher abstraction layer in form of an object oriented programming language like C++. Working with this design flow requires further constraints for providing the synthesis tool with information about how the described algorithm should be parallelized and clocked on the reconfigurable architecture. Meeting the desired constraints depends on many factors. Fixing unreached constraints is quite hard, since the resulting generated HDL code is difficult to read, especially for a software developer. These issues lead to the question if the automatically generated hardware shows a comparable quality, in form of throughput and resource utilization to hand-designed HDL code.

Nevertheless, the mapping process of an HLS tool could be simplified by including domain-specific knowledge into the design flow. Control flow descriptions like special memory buffers can be reused for many operators, and therefore do not need to be regenerated through the HLS process, instead they can exist in form of a generic HDL IP block. A data flow description, on the other hand, must be redesigned for each applied image processing algorithm. Utilizing HLS simplifies the code description process, compared to an HDL. Hence, by utilizing a hybrid design flow approach, within a domain-specific environment, the developer benefits from both forms of hardware descriptions, as illustrated in Figure 1. In this work we present a new hybrid design flow for mapping image processing operators to FPGAs. In order to demonstrate how our solution benefits from domain-specific knowledge and a hybrid description, we compare our generated hardware with the results from existing frameworks and libraries.

The paper is structured as follows: In Section II alternative approaches, frameworks, and tools are discussed in the field of accelerating the hardware design flow. Section III focuses directly on characteristics of image processing operators and their mapping on FPGAs. The idea of a hybrid code description is introduced in Section IV. How the code is described and generated is explained in Section V, while in Section VI the employed operator are described and the corresponding synthesis results are analyzed and compared with other considered approaches. Section VII finishes with a brief conclusion and outlook for future work.

II. Related Work

Mapping image processing algorithms to reconfigurable hardware through an additional layer of abstraction has been addressed in several research groups. In [1], a video processing platform using the high-level synthesis tool *Synphony C* for various operators is demonstrated. The platform offers several connections to internal and external bus or memory interfaces, which are connected to HLS synthesized hardware. But they do not make statements about specialized memory structures and achieved throughputs. Writing the entire image to an external DRAM before processing, introduces unnecessary latency. Another interesting work has been shown by the team of [2]. They are able to map several image processing operators to various hardware platforms including programmable devices from their own lambda like programming language. But, they do not put their synthesized results in contrast to other frameworks or HDL descriptions. The company *National Instruments* provides with *Vision Development Module* [3] another solution describing image processing operations in hardware. It combines schematic descriptions with HLS utilizing Vivado HLS. Unfortunately, the tool is embedded into the *Labview* environment, and therefore limited to proprietary FPGA boards provided from NI.

That pure HLS solutions very often do not supply satisfactory results as shown in [4]. The group aimed to accelerate common image processing operators on hardware. They designed a programmable vector processor for image processing operations. Utilizing the HLS framework *Vivado HLS* [5] from *Xilinx*, which consists of a library for certain operators [6], showed exceeding resource utilization for their desired operators with an FPGA of their choice. The work of [7] demonstrates that more complex image processing pipelines produce high resource utilization, when they are described through an HLS tool. In that paper, a stereo matching technique was described through the *Hipacc* framework and mapped to hardware through Vivado HLS. The synthesis results were compared to a generic HDL architecture. Even for low resolutions, the HDL solution results in much less logic and memory on an FPGA.

In our work, we aim to demonstrate how our approach performs in contrast to conventional HDL designs and HLS frameworks, by comparing several design approaches in terms of resource utilization, processing speed and design effort. As design approaches, we consider the framework Hipacc, the video library from Vivado HLS and a library with generic components described through an HDL.

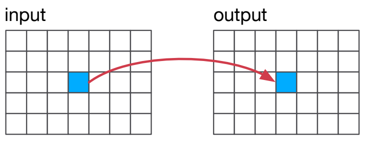


Fig. 2. Point operator: Processing is independent from other pixels in the input image.

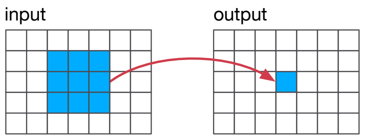


Fig. 3. Local operator: Processing depends on a local array of neighborhood pixel in the input image.

III. Image Processing on FPGAs

Any complex image processing algorithm can be described as an *Image Processing Pipeline*, which consists of several operators concatenated in a pipeline structure. Depending on the memory access scheme, we classify a segment into *Point*, *Local* and *Global* operators. While a single point operator can be mapped quite easy to hardware by processing the pixel in a pipeline scheme, local operators require a static region of neighborhood pixels. The different access patterns of point and local operators can be observed in Figure 2 and 3. In global operators every input pixel can contribute to any output pixel. Their implementation renders to be quite complex, and therefore has been left aside in this work for the time being. For implementing local operators on FPGAs, the full buffering scheme is introduced in [8]. This structure can be easily integrated into a pixel stream since in each clock cycle an entire local neighborhood can be processed. In this Section, we discuss two basic approaches how the mentioned processing and memory structures are mapped to reconfigurable hardware.

A. HDL Solutions

As mentioned in Section I, describing complex image processing algorithms takes a huge effort in development and hardware experience for generating hardware with high throughput. Such design processes can be simplified by utilizing IP Core libraries e.g. for certain memory structures or filter operations. The FPGA manufacturers *Xilinx* and *Altera* provide several IP cores [9] and [10] for image processing applications. Unfortunately, these are often limited to basic image processing operators like convolution operations with fixed window sizes. A library based on generic components, as proposed in [11], is more flexible. This library consists of a generic full buffering structure and components for various filter descriptions, like MAC (multiply and accumulate) or sort operations, which cover a wide range of local filter operators. Still, describing custom and more complex filter kernels requires additional design effort and of course hardware experience.

B. High Level Synthesis Approaches

The big FPGA vendors, Xilinx and Altera, both provide sophisticated HLS tools. Both have in common to support a C/C++-like language for describing algorithms, which are compiled down to HDL for synthesis. Although those vendors claim that unmodified algorithm code can be used to generate hardware, severe changes are necessary in order to obtain efficient implementations with reasonable resource utilization and overall latency.

To overcome this issue, the FPGA vendors deliver libraries to ease the development of software which can be synthesized easily and provide good results. For example Xilinx offers a computer vision library optimized for Vivado HLS that is heavily inspired by OpenCV's API. In fact, many of its implementations, such as the Harris corner detector or 2D convolutions (`2D_Filter`) can be directly used with Vivado HLS without applying severe modifications to C++ source codes. However, those provided implementations are predefined, cannot be altered, and previous publications [12] have shown that they are anything but efficient in terms of resource usage and throughput. Xilinx' OpenCV implementations can serve as a good starting point for fast prototyping, but it is advisable to manually develop separate implementations if efficiency is of utmost importance.

Another interesting work from academia for mapping image processing operators directly to hardware was presented with the *Heterogeneous Image Processing Acceleration* (Hipacc) Framework. It consists of a Domain-specific Language (DSL) and a source-to-source compiler. The DSL is embedded into C++ and represented by specific template classes. Using those classes to describe an algorithm resembles similar to using an image library, but the source-to-source compiler replaces those code fragments by generating target-specific code. Suitable target languages are CUDA, OpenCL, Renderscript, and Vivado HLS-specific C++, although Hipacc was originally developed to support GPUs only. Supporting FPGAs through HLS, leverages the algorithm description to an even higher level. Thereby, algorithms can be prototyped and tested swiftly on a GPU before switching to FPGAs and sacrificing a substantial amount of time on synthesis. The general design flow for FPGAs is shown in Figure 4.

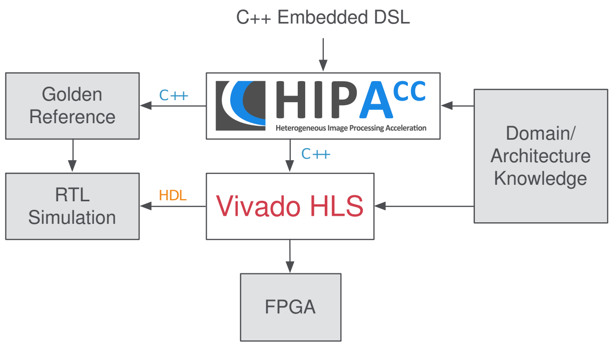


Fig. 4. Design flow of Hipacc in combination with Vivado HLS.

IV. Hybrid Description of an Image Processing Pipeline

In the previous chapter we discussed the two major hardware description techniques, which are utilized in image processing, today. HDL implementation is well-suited for reusable artifacts, like memory structures and state machine, since they need to be designed only once as generic hardware and later used with the desired configuration. An implementation of the generic full buffer is a good example for such an artifact, because it is required for every local operator in the image processing pipeline. Hence, it is not necessary to generate this structure for every local operator in a pipeline. In case this structure is already available in an HDL description, the HLS mapping process is unloaded, and therefore is able to meet the required constraints with less effort.

On the other hand, mapping data flow descriptions utilizing an HDL may take much more development effort. Since one pixel in a stream must be processed in a single clock cycle, the developer needs to take care of parallelizing and pipelining the design. We expect HLS to be able to fulfill this issue quite satisfactory. The possible design space of a pure data flow description is rather low. Additionally, HLS tools are designed to implement pipelining automatically, and therefore the designer does not need to include register structures.

Taking a look to the advantages and disadvantages of these two descriptions raises the question if it is possible to combine these approaches in order to save FPGA resources and development effort. Originating from the characteristics of these description, leads to the conclusion that data flow and control flow should be described separately from each other. Hence, all structural description should be mapped through a configurable HDL description and the data flow definition could be handled through HLS. From the perspective of an image processing pipeline, this means memory structures like a full buffer can be implemented through a full generic buffer, written in plain HDL like VHDL and the filter or kernel description realized through an HLS framework. Such a generic template for local operations has been designed in [13]. In this research, we employ this template in our approach. In Figure 5, the separation of both description forms within a module, can be observed.

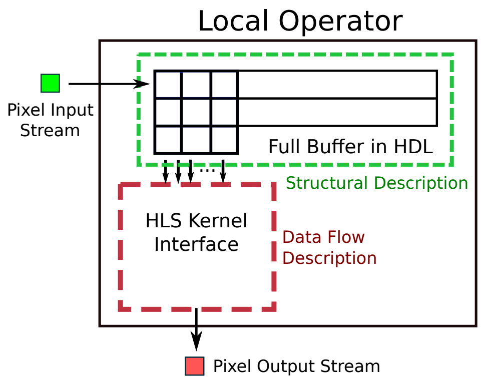


Fig. 5. Overall hybrid description of a local operator. Structural and data flow descriptions are separated within the module.

Configuring this VHDL based template still requires knowledge of HDL languages and hardware design. In Section V-A we are addressing exactly this issue, through utilizing code transformation out of a structural description. As HLS environment we choose Vivado HLS, which is a widespread tool in this field. How we employ Vivado HLS for kernel definitions is discussed in Section V-B.

V. Code Generation and Transformation

A. Structural Description

In our approach, we are integrating predefined HDL descriptions, like the mentioned full buffering template. Since we aim to simplify the configuration and usage of the required structural description, the hardware layer must be hidden from the developer. To build an arbitrary structure of an image processing pipeline, several parameters must be manually configured. Most important parameters are image size, window size, data width, number and type of operators, and structure of operator connection. VHDL offers several constructs like *Constants*, *Generics*, and *For Generate*, which are mainly used e.g. to configure the structure and size (like window size, or data width) of a single buffer. For other structural parameters, like number or type of operators, these constructs do not suffice. Additionally, type definitions of signals can only be done statically, which can be critical in case an arbitrary number of operators and connections must be instantiated. In order to overcome these issues, code generation techniques can be employed. Parameters must be defined in a separate format. The language *IPOL* (Image Processing Operator Language), defined by the authors of [14], fits perfectly to our purposes. IPOL is an XML based meta language, which is human and machine readable and is specifically designed for the structural description of an image processing pipeline in a heterogeneous processing system. The transformation process, from IPOL to the resulting HDL files, is done by XSLT (Extensible Style Sheet Language Transformation). An XSL file is usually used to transform XML documents to other XML or HTML documents, e.g. for web browsing. However, XSLT may also be used to generate plain text of other formats and since it is a Turing complete language it is suitable for VHDL file generation. For the transformation process, an XML file and an XSLT style sheet description is required. The style sheet definition is written in XSL 2.0 and the transformation tool *Saxon 9* [15], developed by *Saxonica*, has been utilized. The transformation process combined with the kernel generation is illustrated in Figure 6.

Due to XSLT's web development background, the processing is defined by the transformation of exactly one XML and one XSL file to one output file, in our case VHDL. Therefore, a tool is required, which allows the transformation of multiple style sheets to multiple output files by executing the XSLT processor several times with different parameters. With *codegen*, a tool was created that embeds Saxonica in a C++ environment to allow batch processing of multiple files. For a fine grained configurability by the user, an description language, similar to traditional Makefiles, has been created to allow the generation of multiple VHDL files in an fully automated way, without knowing any details about the transformation process. The capability of this tool, could be shown first in [16] for generating complete processor fields using XSLT.

B. Kernel Definitions

As mentioned in Section III, point and local operators differ in their data access behavior. Defining an efficient implementation of a local operator in a pure Vivado HLS definition usually involves the declaration of a full buffering structure. Thereby, appropriate HLS directives must be specified for enabling data flow analysis, defining suitable data partitioning, and unrolling certain loop nests that are necessary for shifting pixels in the buffer. Those structures are highly repetitive and usually HLS does not manage to come up with an implementation that is similarly efficient as a plain HDL implementation. Separating the buffering structure significantly simplifies the HLS description. In this case the input is not the pixel stream, instead all pixels from the mask must be addressed in parallel. The kernel process can be described in form of loops, just like a sequential description. Through adding additional pragmas, the HLS receives the information how the described operation should be parallelized. An example for a convolution operation, which is required in every linear local operation, is shown in Listing 1.

```
conv: for(int i=0; i < VEC_SIZE;i++) {
    prod = coeffs[i] * din0[i];
    accu += prod;
 }
```

Listing 1. Convolution operation applied in Vivado HLS.

The array *din0* is mapped to the mask of the window. First each array element is multiplied with the dedicated constant element of the Gaussian filter mask. The variable *accu* accumulates the intermediate results. Basically a point operator is described in a similar way, but instead of an array, the input variable must be defined as a single value. Implementing this algorithm through an HDL would involve much more lines of code, since parallelization and pipelining must be done by

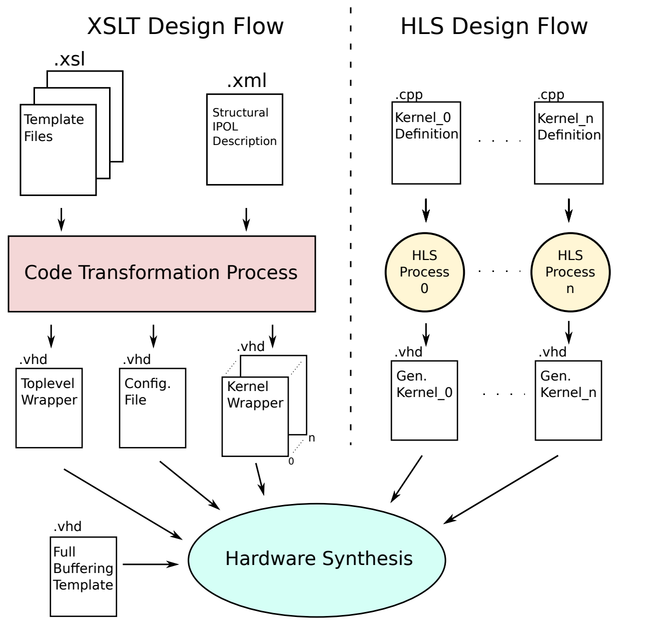


Fig. 6. Hybrid code generation process. The left side shows the XSLT code generation design flow for all structural components. On the right side the kernel code generation through an HLS framework is displayed.

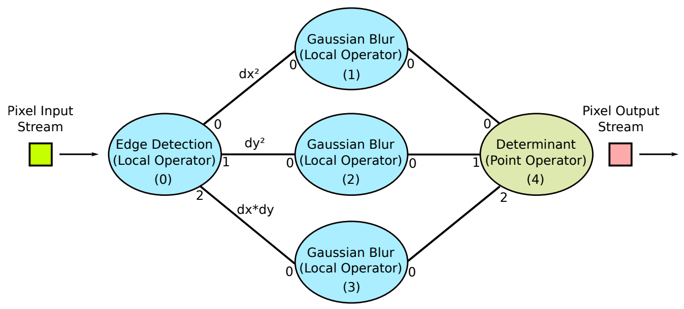

Pixel Input Stream
Edge Detection (Local Operator) (0)
dx²
dy²
dx*dy
Gaussian Blur (Local Operator) (1)
Gaussian Blur (Local Operator) (2)
Gaussian Blur (Local Operator) (3)
Determinant (Point Operator) (4)
Pixel Output Stream

Fig. 7. Image processing pipeline of the implemented Harris corner detector. The numerical labels indicate the input/output port number of the dedicated operator.

hand. In addition to that, applying changes in filter sizes or data width would involve much more overhead in a hardware description.

C. Hybrid Design Flow

In order to generate hardware with a hybrid description, a synchronization of both design flow branches must be implemented. Figure 6 illustrates this process. Besides the IPOL description, the XSLT code transformation process requires XSL style sheet for every VHDL file, which depend on the set parameters. As a result, several files are generated; A *Configuration File*, consisting of all constants like buffer size, window size or data width. For each operator in the chain, a kernel wrapper needs to be created. These files consist of an interface of the entity, generated by the corresponding HLS process. Finally, a *Top level Wrapper* is generated, for connecting all operators correctly. The HLS can be executed separately. A Kernel Definition must be described for every operator. In order to eliminate errors during the synchronization of both design flow, C++ file templates with the correct port names and interfaces can be generated from the XSLT process. In that case the developer only needs to place kernel code as described Listing 1. Since each design flow process generates a VHDL file, the synchronization can be made at this level. In combination with the generic full buffering template, all inputs for executing the hardware synthesis are available.

VI. Experimental Results

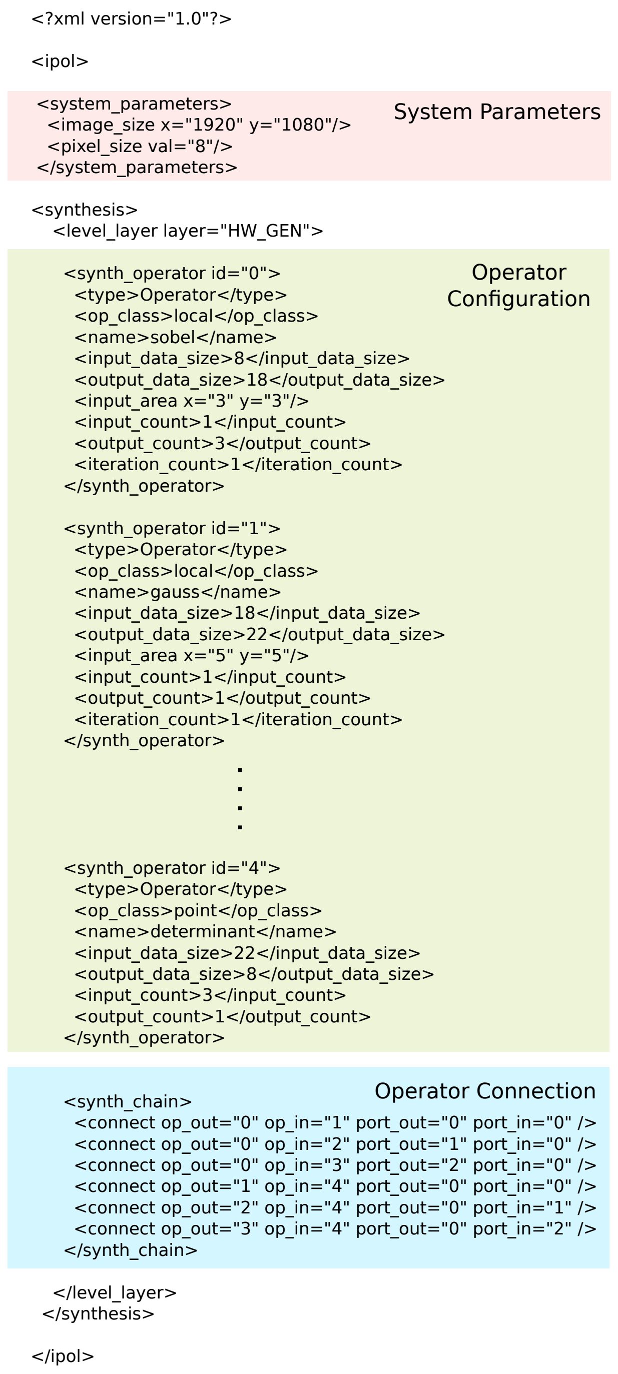
A. Applied Operators

To demonstrate our hybrid design flow, we choose common image processing applications. As basic operations, we selected the Gaussian blur filter and the Median filter, which are both local operators and are employed for noise canceling applications. Despite, these filters share their application and memory access pattern, these filters differ very much in their data processing scheme. While Gaussian blur filter is based on a convolution operation with static mask coefficients, the median kernel involves a sorting operation of pixels in the local neighborhood.

In order to perform synthesis analysis for complex operations, the Harris corner detector, firstly introduced from the authors of [17], has been selected. This operator is quite common for detecting corners, and therefore has been implemented in several image processing frameworks and libraries. The pipeline structure can be divided into several local and point operators, as demonstrated in Figure 7. In the first stage, a Sobel edge detector is applied and returns the derivations dx and dy from detected edges of the input image. The resulting derivations are multiplied in the form of dx^2, dy^2 and $dx \cdot dy$. All three results are passed to the next stage, which consists of a Gaussian blur filter for each output stream in parallel. The filtered outputs are finally used to calculate the determinant in the last stage of the pipeline. Reaching a dedicated threshold defines whether the pixel is detected as a corner or not. How the structural description of the Harris corner detector is defined in IPOL can be observed in Figure 8.

In the first section of the IPOL description, overall pipeline parameters are set (*System Parameters*). The image size mainly influences the generated full buffer structure and has been set to Full HD in our example. Setting the parameter *pixel_size* defines the data width of pixel streaming for the output of the operator pipeline.

The middle part of the description (*Operator Configuration*) configures the structure of the image processing pipeline and the operators as well. With the attribute *layer=”HW_GEN”* of the tag *level_layer*, it is indicated that the following elements are supported to be generated through a hardware synthesis process. This separation is necessary, since IPOL is designed for describing an image processing pipeline on a heterogeneous system. Hence, chosen operators could be implemented on a programmable platform like a CPU or GPU. Each operator gets labeled with an ID, which is basically


System Parameters
Operator Configuration
Operator Connection

```
<?xml version="1.0"?>

<ipol>

 <system_parameters>
  <image_size x="1920" y="1080"/>
  <pixel_size val="8"/>
 </system_parameters>

<synthesis>
  <level_layer layer="HW_GEN">

   <synth_operator id="0">
    <type>Operator</type>
    <op_class>local</op_class>
    <name>sobel</name>
    <input_data_size>8</input_data_size>
    <output_data_size>18</output_data_size>
    <input_area x="3" y="3"/>
    <input_count>1</input_count>
    <output_count>3</output_count>
    <iteration_count>1</iteration_count>
   </synth_operator>

   <synth_operator id="1">
    <type>Operator</type>
    <op_class>local</op_class>
    <name>gauss</name>
    <input_data_size>18</input_data_size>
    <output_data_size>22</output_data_size>
    <input_area x="5" y="5"/>
    <input_count>1</input_count>
    <output_count>1</output_count>
    <iteration_count>1</iteration_count>
   </synth_operator>
                    .
                    .
                    .
                    .
   <synth_operator id="4">
    <type>Operator</type>
    <op_class>point</op_class>
    <name>determinant</name>
    <input_data_size>22</input_data_size>
    <output_data_size>8</output_data_size>
    <input_count>3</input_count>
    <output_count>1</output_count>
   </synth_operator>

   <synth_chain>
    <connect op_out="0" op_in="1" port_out="0" port_in="0" />
    <connect op_out="0" op_in="2" port_out="1" port_in="0" />
    <connect op_out="0" op_in="3" port_out="2" port_in="0" />
    <connect op_out="1" op_in="4" port_out="0" port_in="0" />
    <connect op_out="2" op_in="4" port_out="0" port_in="1" />
    <connect op_out="3" op_in="4" port_out="0" port_in="2" />
   </synth_chain>

  </level_layer>
 </synthesis>

</ipol>
```

Fig. 8. IPOL Description of the Harris Corner Detector. Within the *System Parameters* tag overall image processing parameters are set, while in *Operator Configuration* section arbitrary operators are instantiated and modified. In *Operator Connection* all operators get connected.

necessary for naming the HDL files correctly. The tag *op_class* defines each operator as a point or local operator. Setting the elements *input_data_size* and *output_data_size* correctly is the responsibility of the developer. The *input_data_size* of an operator must match the *output_data_size* of the previous connect operator in order to avoid interface mismatches during the elaboration process of the hardware synthesis. For setting the dimension of the neighborhood area of a local filter, the attributes of the tag *input_area* must be configured, as defined in Figure 8. The amount of in- and outputs of an operator can be set through the *input_count* and *output_count* set. In the definition for a local operator, the number of inputs is fixed to “1”, since the number of inputs is described as a single vector through the already set filter dimensions. For point operators, on the other hand, there are no limitations on setting these, as long as it is conform with the interface of the corresponding high-level synthesis kernel description. Finally, the *iteration_count* generates additional instances of the full buffer and kernels in order to apply an operation multiple times on an image.

How the ports of operators are connected to each other can be described through *connect* elements of the *synth_chain* (*Operator Connection*), which matches the structural description in Figure 7. The first and last listed operators are automatically connected to the main entity of the image processing pipeline.

Through these descriptions, any combination of point and local operators within an image processing pipeline can be designed. In combination with the XSLT code transformation and the HLS description of the kernels, a direct mapping can be achieved. Despite that, this form of description relieves the developer from writing VHDL. Defining an image processing pipeline with this XML structure may seem inconvenient for a description, since it requires basic knowledge in the rules of this language. Nevertheless, the IPOL definition can later be easily generated out of an schematic editor and GUI based drop down menus. Hence, the end user never gets in contact with IPOL, since it may be used as an intermediate language.

B. *Implementations*

For evaluating the generated hardware of our hybrid approach, we implemented the previous mentioned operators with different frameworks and libraries. The generic library, mentioned in III-A, serves as comparison of a HDL description. This library is very suitable for our comparisons, because the hardware can be easily adapted to different settings and configuration through altering the considered generic parameter or filter coefficients in the configuration file. The generated operators with this library can be considered as a handcrafted solution designed with an HDL, which is a perfect basis for comparisons. Implementing the functions from the Vivado HLS video library stands for a comparison of a pure HLS tool, while with the Hipacc framework a platform for an alternative domain-specific solution is given. All our generated results are placed and routed for the *Zynq Z-7020* with *Vivado 2015.3* from Xilinx. The Zynq Z-7020 is a commonly used midrange FPGA with dual core ARM A9 system and belongs

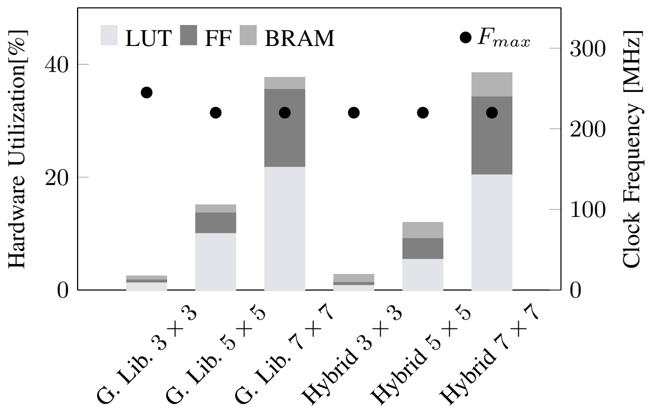


Fig. 9. Resource utilization of a Median operator for filter dimension $n = 3$, $n = 5$, and $n = 7$, implemented with a generic HDL library and our hybrid approach.

to the Vivado HLS supported platforms. For RTL synthesis of operators, generated by the generic library and our hybrid approach, *Synplify Pro* from *Synopsys* has been utilized. As image size, Full HD (1920×1080) with 8-bit gray scale pixel has been selected.

1) Median Operator: In our first experiment, we aim to investigate if our hybrid solution can compete with an HDL description in terms of throughput and resource utilization. As a test scenario, we implemented the Median operator with various filter sizes. Finding the Median value of the considered window is implemented as a pipelined sort network, which has been described in [18] and allows a pixel result with each clock cycle. In our hybrid version, the sort array is implemented through C++, while in the generic library, the sort array is designed as a generic HDL description. Figure 9 displays the results. The bar graphs show that in both approaches the resource utilization rapidly increases with rising window dimension $n \times n$, since a number of $n^2/2$ swap components, consisting of a comparator functionality, is required for an efficient mapping. It also reveals that the hybrid version exposes a similar resource utilization like HDL described hardware, for larger filters the logic utilization is even slightly lower with our hybrid approach. The maximum clock frequencies remain on a high level above 200 MHz in all designs. This results show very well, that the hybrid description is able to produce hardware with the same quality as achieved through a pure HDL solution, even though the kernel description is performed on a high abstraction layer.

2) Gaussian Blur Operator: Table I lists the results of our second experiment. The Gaussian blur operator has been implemented with all considered libraries and frameworks for the filter sizes 3×3 and 5×5. For both configurations all frameworks and libraries, except for the Vivado HLS video library, show similar results with quite low resource utilization. Besides an increased number of logic the HLS library requires additional DSPs for implementing the filters and reaches much lower clock frequencies. Unlike the other implementation it

TABLE I
COMPARISON RESULTS OF GAUSSIAN BLUR OPERATOR.

	HDL Library		Hipacc		HLS Library		Hybrid Design	
	3×3	5×5	3×3	5×5	3×3	5×5	3×3	5×5
LUT	183	592	166	365	543	1192	159	520
FF	248	734	320	705	358	751	158	568
BRAM	2	4	1	2	2	3	2	4
DSP	0	0	0	0	4	13	0	0
F[MHz]	235	235	271.518	265.182	120.2	111.3	250	235

was not possible to achieve the demanded design pragma *HLS PIPELINE II=1* with the HLS library. With an achievement of this goal, the generated hardware is able to process an element, e.g. one pixel, from the defined input data set within one clock cycle, which is mandatory for creating a streaming pipeline. Concerning logic and flip-flops, the hybrid approach utilizes less resources than the generic library and Hipacc (except for logic utilization of the 5×5 filter). This might result from selecting the correct data width for forming the pipelined adder tree used for summing up the weighted pixel values. In the HLS library the data width of the adder tree is held constant, while analyzing the generated VHDL from HLS shows, that the width is adapted in every stage, which saves logic and memory. The resulting values show that domain-specific code generation, as performed in Hipacc on the HLS level and in the hybrid version through separating the data flow and control flow makes it possible to generate resource-efficient hardware utilizing HLS. The high resource utilization from the HLS Video library demonstrates that the quality of generated hardware depends very much on the C-level based description. Concerning the clock frequencies Hipacc shows the best results.

3) Harris Corner Operator: Table II displays the result for the Harris corner detector. Again the HLS library can not keep up with the other approaches concerning resource utilization and achieved maximum clock frequency. Also for this example, the demanded constraints could not be reached, unlike the other considered solutions. Except for the number of DSPs in the HDL library shows, the Logic and flip-flop utilization for the hybrid approach and for the HDL library are lower than for the Hipacc results. This issue again could be addressed to the data width handling. Due to the fact that Hipacc is based on a domain-specific language, which is mainly used for programmable platforms, the focus is on standard data types like *Character*, *Integer*, or *Float*. Since in the Harris corner example several arbitrary bit widths are used, as described in Figure 8, implementing data types based on “power of 2”, wastes hardware resources. The exceeding number of DSPs also could resolve from non-optimal data width selection in the HDL design. Data types with arbitrary bit width and even fix point variables are supported in Vivado HLS. Another reason for the differences in the result might be because of Hipacc’s structural description on HLS level compared to RTL, which would explain the high demand on flip-flops. Describing structures on a higher level can be the reason for synthesizing redundant hardware, since the possible design space is higher than on RTL level. Nevertheless, Hipacc again outperforms the other solution concerning maximum

TABLE II
COMPARISON RESULTS OF THE HARRIS CORNER OPERATOR.

	HDL Library	Hipacc	HLS Library	Hybrid Design
LUT	2569	3445	9289	2740
FF	4005	6310	11301	3789
BRAM	14	14	37	14
DSP	83	11	30	13
F[MHz]	150	214.64	115	179.4

clock frequency. As a final result of this experiment, it can be stated that also for more complex pipeline structures, generated hardware, synthesized through a hybrid solution, features a similar quality like a pure HDL design.

VII. CONCLUSION

In our work, we compared several design approaches for mapping image processing operators to FPGAs. All these approaches aim to accelerate the entire design flow through hiding the HDL layer from the developer and/or utilizing domain-specific knowledge in form of optimal structures to design an image processing pipeline. One of the main question in this context is: Can HLS accelerate the design flow without experiencing drawbacks in the quality of generated Hardware? We pursued this question by putting different HLS based solutions in contrast to each other and presented a new hybrid approach combined with code transformation. Our results show that when a well-defined HLS description is utilized for data flow, the generated hardware has similar quality in terms of resource usage and processing speed, compared to an HDL designed solution. Especially for more complex pipelines structures, the hybrid approach shows promising results, since the structural description is totally encapsulated, and therefore HLS processes get unloaded. These conclusions manifest that, for domain-specific accelerators, HLS in combination with domain-specific knowledge is able to significantly improve the development time without introducing drawbacks in hardware quality. Code transformation techniques also support the developer, because less knowledge in hardware development and languages is required as well as less effort in code development is necessary. Nevertheless, we think that HDLs will not be replaced with HLS in the near future. More likely, they may become a meta language for synthesis tools, at least in the field of domain-specific accelerators. In future work, we will further employ the hybrid approach in order to find optimal architectures and design flows for mapping global image processing operations. Adding additional techniques for parallel kernel execution is also an important issue we will take focus on.

ACKNOWLEDGMENT

This work was financially supported by the Research Training Group 1773 "Heterogeneous Image Systems", funded by the German Research Foundation (DFG).

REFERENCES

[1] C. Desmouliers, E. Oruklu, S. Aslan, J. Saniie, and F. Vallina, "Image and video processing platform for field programmable gate arrays using a high-level synthesis," in *IET Computers and Digital Techniques (Volume:6 , Issue: 6)*, 2012, pp. 414–425.

[2] J. Hegarty, J. Bruhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and H. P., "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2014*, vol. 33, Jul. 2014.

[3] N. Instruments, "Vision Development Module," 2016. [Online]. Available: http://www.ni.com/vision/software/vdm/

[4] G. Hedge and N. Kapre, "Energy-Efficient Acceleration of OpenCV Saliency Computation using Soft Vector Processors," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on* , 2015, pp. 76–83.

[5] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis," Oct. 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/ug902-vivado-high-level-synthesis.pdf

[6] ——, "Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries," June 2015. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf

[7] O. Reiche, K. Häublein, M. Reichenbach, F. Hannig, J. Teich, and D. Fey, "Automatic optimization of hardware accelerators for image processing," in *Proceedings of the DATE Friday Workshop on Heterogeneous Architectures and Design Methods for Embedded Image Systems (HIS)*, pp. 10–15.

[8] X. Liang, J. Jean, and K. Tomko, "Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems," *The Journal of Supercomputing*, vol. 19, no. 1, pp. 77–91, 2001.

[9] Altera, "Video and Image Processing Suite," February 2014. [Online]. Available: www.altera.co.jp/ja_JP/pdfs/literature/ug/ug_vip.pdf

[10] Xilinx, "Image Enhancement v8.0: LogiCORE IP Product Guide," Oct. 2014. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/v_enhance/v8_0/pg003_v_enhance.pdf

[11] K. Häublein, C. Hartmann, M. Reichenbach, and D. Fey, "Fast and resource aware image processing operators utilizing highly configurable ip blocks," in *Proceedings of ARC 2016 (Applied Reconfigurable Computing)*, V. Bonato, Ed., 2016, pp. 1–8.

[12] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code generation from a domain-specific language for C-based HLS of hardware accelerators," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

[13] M. Schmidt, M. Reichenbach, and D. Fey, "A Generic VHDL Template for 2D Stencil Code Applications on FPGAs," in *15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, April 2012, pp. 180–187.

[14] C. Hartman, M. Reichenbach, and D. Fey, "Ipol - a Domain Specific Language for Image Processing Applications," in *Proceedings of the International Symposium on International Conference on Systems (ICONS)*, Mar. 2015, pp. 40–43.

[15] [Online]. Available: http://saxon.sourceforge.net/

[16] M. Reichenbach, T. Lieske, S. Vaas, K. Häublein, and D. Fey, "FAUPU - A Design Framework for the Development of Programmable Image Processing Architectures," in *Proceedings of ReConFig' 15*, R. Cumplido, Ed., 2015, pp. 1–8.

[17] C. Harris and S. M., "A Combined Corner and Edge Detector," in *Alvey Vision Conference*, 1988, pp. 147–151.

[18] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*. Wiley-IEEE Press, 2011.