Generating FPGA-based Image Processing Accelerators with Hipacc

Conference Paper · November 2017

DOI: 10.1109/ICCAD.2017.8203894

5 authors, including:

Oliver Reiche
Siemens Healthineers
23 PUBLICATIONS **319** CITATIONS
SEE PROFILE

Akif Özkan
Friedrich-Alexander-University of Erlangen-Nürnberg
17 PUBLICATIONS **82** CITATIONS
SEE PROFILE

Richard Membarth
Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany
43 PUBLICATIONS **413** CITATIONS
SEE PROFILE

Frank Hannig
Friedrich-Alexander-University of Erlangen-Nürnberg
216 PUBLICATIONS **1,831** CITATIONS
SEE PROFILE

This is the author's version of the work. The definitive work was published in *Proceedings of the International Conference On Computer Aided Design (ICCAD), Irvine, CA, USA, November 13–16, 2017.*

Generating FPGA-based Image Processing Accelerators with Hipacc

(Invited Paper)

Oliver Reiche‡, M. Akif Özkan‡, Richard Membarth†, Jürgen Teich‡, and Frank Hannig‡
‡Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (FAU)
†German Research Center for Artificial Intelligence, Saarland University
{oliver.reiche, akif.oezkan, teich, hannig}@cs.fau.de, richard.membarth@dfki.de

***Abstract*—Domain-Specific Languages (DSLs) provide a high-level and domain-specific abstraction to describe algorithms within a certain domain concisely. Since a DSL separates the algorithm description from the actual target implementation, it offers a high flexibility among heterogeneous hardware targets, such as CPUs and GPUs. With the recent uprise of promising High-Level Synthesis (HLS) tools, like Vivado HLS and Altera OpenCL, FPGAs are becoming another attractive target architecture. Particularly in the domain of image processing, applications often come with stringent requirements regarding performance, energy efficiency, and power, for which FPGA have been proven to be among the most suitable architectures.**

In this work, we present the Hipacc framework, a DSL and source-to-source compiler for image processing. We show that domain knowledge can be captured to generate tailored implementations for C-based HLS from a common high-level DSL description targeting FPGAs. Our approach includes FPGA-specific memory architectures for handling point and local operators, as well as several high-level transformations. We evaluate our approach by comparing the resulting hardware accelerators to GPU implementations, generated from exactly the same DSL source code.

I. Introduction

In many of today's application domains, image processing, and computer vision play a significant role. Domains range from medical imaging up to advanced driver assistance systems, or even autonomous driving. In particular, in the automotive field, the question arises where to compute those tasks. Depending on the application's complexity, possible targets are an already existing electronic control unit, a dedicated microcontroller, a Central Processing Unit (CPU) or Graphics Processing Unit (GPU) embedded in an SoC, or even a Field Programmable Gate Array (FPGA). The right choice depends on many factors, such as development effort, power efficiency, area constraints, and real-time capabilities, which are most important for safety-critical systems.

It is well-known that application-specific hardware tends to give the highest performance and most efficient resource utilization. On the contrary, application-specific development is a time consuming and error prone task. One step towards mastering this challenge are HLS tools, such as Vivado HLS from Xilinx and Altera OpenCL. However, these tools can be considered as general purpose frameworks and do not make use of domain-specific knowledge for image processing.

Another approach is to use Domain-Specific Languages (DSLs) to concisely describe algorithms without any deeper understanding of the target architecture, i. e., to consider FPGA programming from the perspective of a software engineer [7]. Aside from Hipacc, the DSL we present in this work, other DSLs for generating hardware accelerators are Darkroom [4] and its successor Rigel [5]. They propose functional programming to describe local image operators, which are translated into streaming pipelines. While Darkroom is too simple for advanced pipelines like stereo vision and Lucas Kanade, Rigel adds support for pyramids, sparse computations, and space-time implementation tradeoffs. Another DSL that can target FPGAs is PolyMage [2], which employs a graph representation for describing image algorithms followed by optimization and scheduling techniques in the polyhedral model.

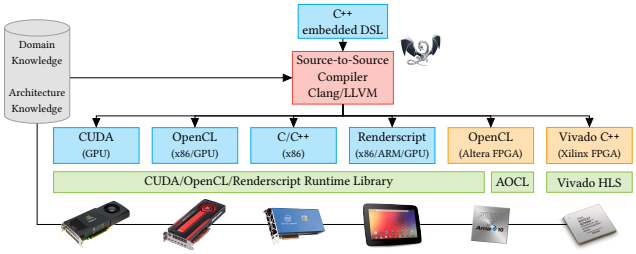


Figure 1. Overview of the Hipacc framework and its target architectures. FPGA targets highlighted in orange.

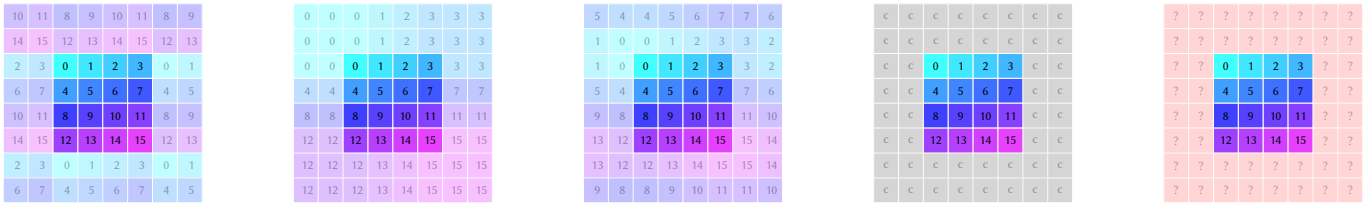
II. The Hipacc Framework

The Heterogeneous Image Processing Acceleration (Hipacc) framework [9] consists of an open source[1] image processing DSL, embedded into C++ and a source-to-source compiler. Initially developed to target GPUs from Nvidia and AMD only, Hipacc was subject to multiple extensions over the years. These extensions involve code generation for other accelerators, such as embedded GPUs [8], vector units of CPUs [12], and FPGA targets [13, 10] through high-level synthesis for Xilinx and Altera FPGAs. Figure 1 provides a visual overview of the framework and its target architectures.

A. Domain Analysis

In literature, image processing algorithms are mainly classified based on two distinct methods. The first method [15] inquires *why* an image algorithm is applied. This includes

[1] http://hipacc-lang.org



(a) Repeat (b) Clamp (c) Mirror (d) Constant (e) Undefined

Figure 2. Border handling modes for image processing. By default, the behavior is undefined when the image is accessed out of bounds (e). The framework allows to specify different border handling modes like repeating the image (a), clamping to the last valid pixel (b), mirroring the image at the image border (c), and returning a constant value when accessed out of bounds (d).

correcting image defects caused by sensor imperfection or optics' limitations, *image enhancement in the spatial domain* for improving specific image features, or *processing images in the frequency space* simply because of the inherent computational advantages. The second method, introduced by Bankman [1], classifies image algorithms on *what information* is used to map one image to another. Here, the three basic classes are *pixel operators*, computing an output pixel based on a single input pixel, *local operators*, additionally incorporating the local neighborhood within the input image, and *operators with multiple images*, where several input images are used. We decided to follow the latter approach and classify the algorithms we want to capture as *point* and *local* operators. Contrarily to [6, 1], we do not make any distinction between algorithms based on the number of input images.

B. Language Components

Image processing algorithms can be implemented as DSL code by using specific C++ template classes. Their use is detected by the Hipacc compiler, necessary to capture the computational algorithm operations. In the following, the most important classes provided by the Hipacc framework are presented.

1) Images in the DSL: An *Image* object represents a two-dimensional data structure for storing pixels of a digital image. Here, the actual data type of a single pixel can be of any supported primitive data type (`int`, `float`) or special types introduced by Hipacc, such as the `uchar4` vector type:

```
Image<pixel_t>(size_t width, size_t height, pixel_t *img)
```

Its actual data layout is hidden from the programmer as it might vary across different target platforms.

2) Accessing Images: We differentiate between reads and writes to an image in the DSL. In order to access an image from a *kernel*, it needs to be assigned to a wrapper data structure handling the access mode. For reading, an image needs to be bound to an *accessor*. Thereby, a rectangular Region Of Interest (ROI) can be optionally defined:

```
Accessor<pixel_t>(Image<pixel_t> img,
    size_t width, size_t height,
    int offset_x, int offset_y)
```

For writing an image from a *kernel*, it needs to be wrapped in an *iteration space*. To ensure that only a specific portion of the output image is written, an optional ROI can be defined here as well:

```
IterationSpace<pixel_t>(Image<pixel_t> img,
    size_t width, size_t height,
    int offset_x, int offset_y)
```

In case the input and output image sizes do not match, no one-to-one mapping from input to output can be achieved. As a solution, *interpolation* needs to be applied. Predefined interpolation modes for *nearest neighbor*, *bilinear filtering*, *bicubic filtering*, and *Lanczos resampling* can be enabled via an argument to the accessor's constructor.

These are all DSL constructs required to describe point operators. Next, we will introduce our support for filter masks, which are required for local operators.

3) Sliding Windows: Local operators employ a sliding window to iterate over neighboring pixels. Hipacc provides two constructs to describe sliding windows. The *domain*, which solely defines the iteration space of a window, and the *mask*, which additionally provides filter coefficients for that window:

```
Domain(size_t size_x, size_t size_y)
Domain(uchar domain[size_y][size_x])
Domain(Mask<pixel_t> mask)
Mask<pixel_t>(size_t size_x, size_t size_y)
Mask<pixel_t>(pixel_t data[size_y][size_x])
```

A prominent example is the following discretized Gaussian kernel of size 3×3 for image smoothing with each entry of the mask being one filter coefficient:

$$G_3 = \begin{bmatrix} 0.057118 & 0.124758 & 0.057118 \\ 0.124758 & 0.272496 & 0.124758 \\ 0.057118 & 0.124758 & 0.057118 \end{bmatrix} \quad (1)$$

4) Border Handling: Accessing neighboring pixels at the image border will inevitably lead to out-of-bounds accesses. For those particular cases, the accessor should automatically apply proper border treatment. The programmer only needs to set up a *boundary condition* defining the window size and the desired border handling mode:

```
BoundaryCondition<pixel_t>(Image<pixel_t> img,
    Domain dom, enum Boundary mode)
```

Hipacc supports *repeat*, *clamp*, *mirror*, *constant*, and *undefined* as border handling modes, visualized in Figure 2. As the input image is already encapsulated, it is sufficient to assign only the boundary condition to the accessor, instead of assigning the image.

```
1 // filter mask for Gaussian blur filter
2 const float filter_mask[3][3] = {
3   { 0.057118f, 0.124758f, 0.057118f },
4   { 0.124758f, 0.272496f, 0.124758f },
5   { 0.057118f, 0.124758f, 0.057118f }
6 };
7 Mask<float> mask(filter_mask);
8
9 // input image
10 size_t width, height;
11 uchar *image = read_image(&width, &height, "input.pgm");
12 Image<uchar> in(width, height, image);
13
14 // reading from in with clamping as boundary condition
15 BoundaryCondition<uchar> cond(in, mask, Boundary::CLAMP);
16 Accessor<uchar> acc(cond);
17
18 // output image
19 Image<uchar> out(width, height);
20 IterationSpace<uchar> iter(out);
21
22 // instantiate and launch the Gaussian blur filter
23 LinearFilter Gaussian(iter, acc, mask, 3);
24 Gaussian.execute();
```

Listing 1. Instantiation of an operator for the Gaussian blur filter in the Hipacc DSL.

Example: Gaussian Blur Filter: To show how the DSL components interact, we consider the Gaussian blur filter as an example application in Listing 1. There, a mask is defined with the coefficients from Eq. (1) (lines 2–7). The input image is loaded from disk and assigned to a DSL image object (lines 11–12). Afterwards *clamping* is specified as the border handling mode and the accessor is created (lines 15–16). Before executing the actual operator kernel, the output image is allocated and assigned to an iteration space without any ROI restrictions (lines 19–20). The actual operator kernel is defined in `LinearFilter` and will be introduced in the next section.

C. Defining Operator Kernels

Operator kernels are defined in a Single Program Multiple Data (SPMD) context, similar to kernels in CUDA, Open Computing Language (OpenCL), and Intel's Threading Building Blocks (TBB) [14]. Thereby, a single kernel describes all the necessary operations to compute each output pixel in the iteration space.

Hipacc does not differentiate between point and local operators. Both can be specified by implementing a user-defined class, which needs to derive from the framework's **`Kernel`** class and override the **`kernel()`** method. An example how to implement the Gaussian blur from the previous section is provided in Listing 2. Here, the iteration over the sliding window is performed by the doubly nested loop (lines 7–9). Reading neighboring pixels from the accessor can be done without considering out-of-bound accesses, as they are implicitly handled by the framework. The resulting pixel value is assigned to the output image using the **`output()`** method.

Hipacc provides built-in language support for common image processing tasks. For instance, convolutions can be concisely described via the **`convolve()`** method taking (a) the filter mask, (b) the aggregation mode, and (c) the computation instructions

```
1 class LinearFilter : public Kernel<uchar> {
2   // ...
3   void kernel() {
4     int range = size/2;
5
6     float sum = 0;
7     for (int yf = -range; yf <= range; ++yf)
8       for (int xf = -range; xf <= range; ++xf)
9         sum += mask(xf, yf) * acc(xf, yf);
10
11    output() = (uchar) sum;
12  }
13 };
```

Listing 2. Kernel description for the Gaussian blur filter.

for a single filter mask component with the corresponding image pixel described as a C++ lambda function:

```
void kernel() {
  output() = (uchar)
      convolve(mask, Reduce::SUM, [&] () -> float {
        return mask() * input(mask);
      });
}
```

Supported aggregation modes are *min*, *max*, *sum*, and *prod*.

More complex algorithms requiring local operations beyond a simple convolution can be implemented using the **`iterate()`** method. Consider the bilateral filter [18], which basically performs two convolutions simultaneously, one for computing the spatial closeness component `c` and the other for the intensity similarity component `s`. Using **`iterate()`**, both can be combined into a single sliding window defined by the domain `dom`:

```
void kernel() {
  float d = 0, p = 0;

  iterate(dom, [&] () -> void {
    float diff = in(dom) - in();
    float c = mask(dom);
    float s = expf(-c_r * diff*diff);
    d += c*s;
    p += c*s * in(dom);
  });

  output() = p/d;
}
```

Here, the aggregation is performed manually for `d` and `p` and the spatial closeness component is provided via a precomputed mask.

D. Bit-Width Annotations

To enable the annotation of customized bit widths, as common in hardware design, we introduce DSL-specific pragmas in Hipacc. Pragmas that are not known to the compiler are entirely ignored and are therefore a perfect fit for defining target-specific annotations. In our implementation, the desired bit width must be specified as well as the name of the variable in the code line just before the variable declaration:

```
#pragma hipacc bw(x,3)
uint x;
```

Specifying this pragma will instruct Hipacc to internally mark variable `x` as having a reduced width of 3 bits.

III. High-Level Synthesis Tools

In this work, we focus on C-based HLS. Among the biggest two FPGA vendors, two HLS approaches have been particularly popular: *Xilinx Vivado HLS*, a C++-based HLS tool, and the *Altera SDK for OpenCL (AOCL)*.

A. Xilinx Vivado HLS

Vivado HLS originates from the high-level synthesis tool AutoESL (formerly known as AutoPilot [19]) and has become an integral part of the Vivado Design Suite. Employing Vivado HLS, C/C++ or SystemC codes can be used for design entry and transformed into HDL code (VHDL, Verilog, SystemC), resulting in synthesizable IP cores. In order to guide the transformation of a sequential algorithm's behavioral description into a structural hardware implementation, a large variety of synthesis directives can be applied. Moreover, streaming pipelines can be implemented by employing data streaming objects, *streams*, to interconnect different hardware modules. Even though very complex and sophisticated algorithms can be specified with less effort by utilizing a C-based language, the efficient implementation of an image processing system still demands the profound knowledge of a hardware design expert to obtain results comparable to those of hand-written implementations.

B. Altera SDK for OpenCL

The Altera SDK for OpenCL (AOCL) promises three fundamental advantages over conventional hardware design: (a) Simple structuring of SPMD applications, since OpenCL follows a data-parallel programming paradigm; (b) portability to other architectures, as OpenCL is a standardized language; and (c) a high level of abstraction for describing hardware using a C-based language that can be used to govern a complete heterogeneous system, including CPU and FPGA fabric. Furthermore, to implement a streaming pipeline, AOCL provides an extension, *channels*, serving as data streaming objects for connecting different kernels. Moreover, OpenCL's programming paradigm distinguishes two different types of kernels for implementation.

1) NDRange Kernels: OpenCL's runtime system automatically spawns as many threads as defined by the range (1D, 2D, or 3D), specified by the developer. Such kernels that are supposed to perform across a specified range are called *NDRange kernels*. NDRange kernels are highly portable, as they are written in a style very similar to GPU programming, and therefore are likely to perform rather well on many-core architectures.

2) Single Work-Item Kernels: Kernels that spawn only a single thread without a range are called *single work-item kernels*. Thereby, the developer must explicitly describe the range and the order, in which the range is processed, in the kernel's source code. Unfortunately, single work-item kernels are not portable [10], regarding performance on many-core architectures but open up the possibility for further FPGA-specific optimizations, as data locality can be exploited.

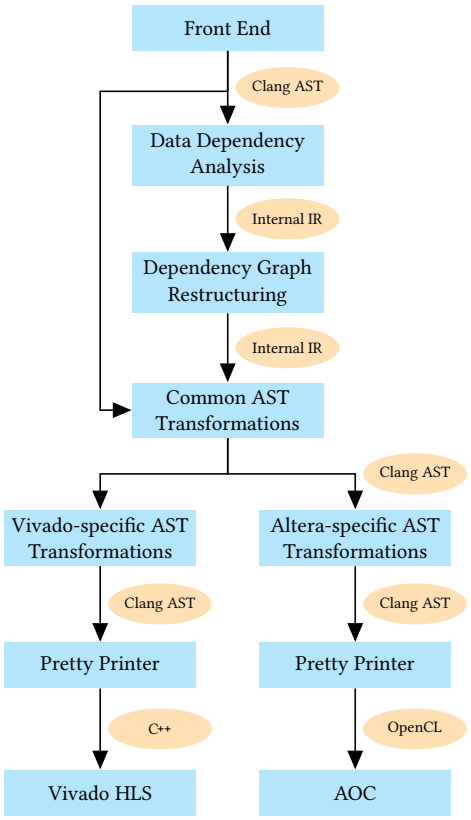


Figure 3. Hipacc workflow for generating HLS code.

IV. Generating Hardware Accelerators from a DSL

Originally designed for targeting GPUs, high-level programs given in Hipacc DSL code process image filters in a buffer-wise manner. Each kernel reads from an entire buffer and exclusively writes to a whole buffer. The previous kernel always runs to completion before the successive kernel starts its execution. Thereby, expensive synchronization is entirely circumvented, as buffers serve as synchronization points (so called host barriers). Buffers can be read and written, copied, reused, or allocated only for the purpose of storing intermediate data.

This buffered concept is fundamentally different from streaming data through kernels. In such a streaming pipeline, the next computational step is performed as soon as all input dependencies are met. Kernels are therefore interconnected with each other on a much finer granularity using streaming objects implementing First In First Out (FIFO) semantics. A streaming pipeline requires a structural description, resolving direct data dependencies unconstrained from the exact sequential order of kernel executions.

The workflow for establishing such a streaming pipeline and generating code for HLS is depicted in Figure 3. First, the *front end* reads in the DSL code that is subsequently transformed into an Abstract Syntax Tree (AST) representation with the aid of the Clang/LLVM compiler infrastructure. Based on that, we can perform a *data dependency analysis*, inferring our own internal Intermediate Representation (IR) to *restructure* the graph suitable for streaming pipelines. Based on that IR,

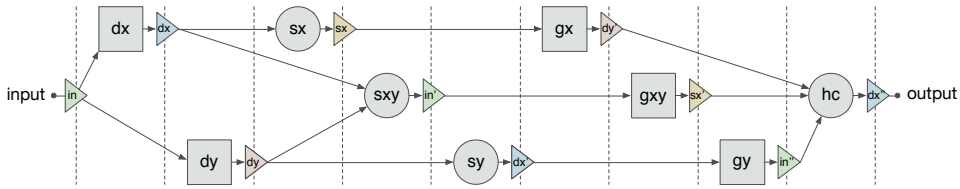


Figure 4. Sequential execution of Hipacc kernels for the Harris corner detector implemented as a pipeline of point operators (circles) and local operators (squares). Triangles mark buffers and dashed lines represent host barriers between kernel executions.

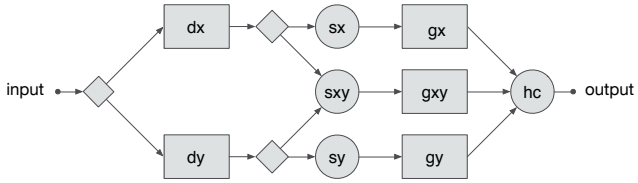


Figure 5. Streaming pipeline of kernels for the Harris corner detector. Diamond shapes represent kernels for splitting data of a single streaming object into multiple streaming objects.

common transformations on the previous AST can be applied for optimizations suitable for image filters in HLS. Afterwards, *vendor-specific transformations* are applied, before utilizing Clang's *pretty printer* to generate the actual HLS code.

In the following, we demonstrate the above design flow using a Harris corner detector [3], as depicted in Figure 4. This filter consists of a pipeline of kernels, implemented by point and local operators. Due to buffer reuse, only four buffers are necessary, as indicated by buffer colors.

A. Data Dependency Analysis

After the DSL code has been translated into an AST representation, it will be traversed by Hipacc. During this traversal process, we track the use of buffer allocations, memory transfers, and kernel executions by detecting compiler-known classes, such as `Image` and `Kernel`. For each kernel, the direct buffer dependencies are analyzed and fed into a dependency graph.

Given this graph, we can build up our internal representation, a simplified AST-like structure based on a bipartite graph consisting of two vertex types: *space* representing buffers and *process* marking kernel executions. By traversing the kernel executions in the sequential order, in which they are specified, writes to buffers are transferred to the internal representation in Static Single Assignment (SSA) manner. Hereby, also every reused buffer (indicated by the same color in Figure 4) will form a new *space* vertex in the graph.

B. Dependency Graph Restructuring

In particular, when multiple kernels read from the same buffer, and therefore depend on the same temporal instance of intermediate data (e. g., dx in Figure 4), it is required to replace this dependency by inserting a *copy process* for splitting data. Afterward, multiple additional output *spaces* need to be added as well, one for each kernel reading from the original buffer. This way, it is guaranteed that streaming data will be replicated before handing it over to the next computation step. Thereby, we can infer the structural description of a streaming pipeline, as shown in Figure 5.

In a final step, the restructured graph is traversed backwards in Depth-First Search (DFS) order, originating from the output *spaces* to prune parts that are not contributing to the output.

C. Common AST Transformations

With the aid of our restructured IR, we can transform the original Clang AST from a buffer-wise execution model into a streaming pipeline. Each *process* vertex is translated to a kernel execution. In particular for copy processes, appropriate copy kernels need to be generated and their instantiation needs to be added to the existing AST. The resulting HLS code finally embodies the structural description of the filter.

Common AST modifications include adding new nodes to fulfill the requirements of the target language such as applying index calculations and ROI index shifts. Depending on the optimizations specified via compiler arguments, vast portions of the extracted AST are modified and replicated. For instance, for constant propagation, which we enforce for FPGA targets, memory accesses are replaced by numeric literals on AST level. Further FPGA-specific optimizations are explained in detail in the following paragraphs.

1) Specialization of Streaming Objects: For every *space* vertex in our IR, a unique HLS streaming object needs to be inserted to interconnect the generated kernels. Specialization of streaming objects is one of the last steps performed in this stage. It decides whether a Vivado HLS *stream* or an AOCL *channel* is generated, depending on the specified target language.

2) Memory Allocation: Aside from only accessing off-chip DRAM, on-chip Block Random Access Memory (BRAM) and Flipflops (FFs) can be used to store data for reuse in later iterations. A widely adapted approach, employing this technique, is full line buffering. Line buffering is particularly beneficial for local operators in image processing, as each pixel is accessed more than once. To efficiently cache those pixels, we generate two types of memory architectures: *line buffers* for storage of complete image lines (BRAMs) and *memory windows* for the actual processing of local neighborhoods (implemented using FFs), as illustrated in Figure 6.

Since images are read in scan line order from DRAM memory, we can exploit data locality and fill the line buffer with one pixel per cycle. From the line buffer, we can always satisfy the data dependencies to fill the memory window. However, a local operator of size $w_x \times w_y$ might require the pixel at $(x - {}^{w_x}/_2, y - {}^{w_x}/_2)$ to calculate the output for (x, y). To ensure that all data is

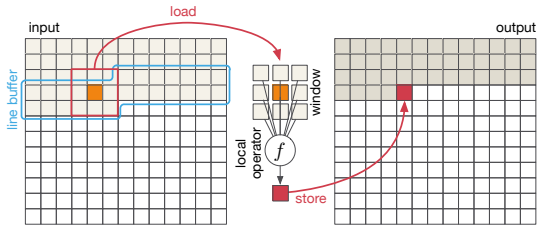


Figure 6. A combination of line buffers and memory windows is typically used to process local operators on streaming data.

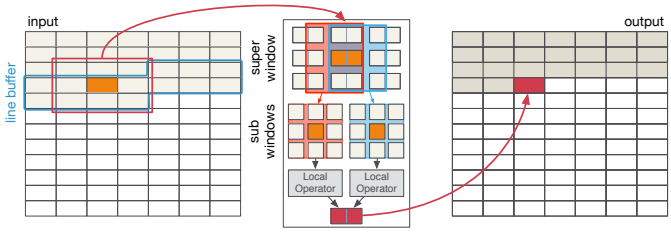


Figure 7. Loop coarsening for parallel processing of superpixels with local operators by replicating only the kernel operator. ($v = 2$)

available, it is necessary to enforce a delay. For an image of size $w \times h$, this delay can be calculated by $\lfloor {}^{w_x}/_2 \rfloor \cdot w + \lfloor {}^{w_y}/_2 \rfloor$.

In consequence, for every local operator in the image pipeline, we allocate a line buffer and memory window, and furthermore ensure that an appropriate delay is introduced.

3) Loop Coarsening: One approach to increase the overall throughput with FPGAs is replicating the entire accelerator. The alternative approach, which we denote as *loop coarsening* [16], replicates only the innermost kernel computation. An illustration of this concept for a local operator is given in Figure 7. In contrast to replicating the entire accelerator, it does not require extra logic for data distribution and allows us to maximize resource sharing while not requiring any border handling overhead. In addition, loop coarsening might lead to more opportunities for the compiler to apply low-level code optimizations and may, thus, reduce the number of arithmetic operations in a kernel.

We implement the concept of loop coarsening by introducing a hierarchy of memory windows. A *superwindow* of size $W_x \times W_y$ is used for gathering superpixels containing v pixels from the line buffer. The superwindow[2] stores pixels for recurring access and is used to extract and assign pixels to the *subwindows* of size $w_x \times w_y$, which are then used by the replicated kernel operators for processing (see Figure 7).

Extracting and assigning pixels from the superwindow into the subwindows involves determining the correspondence between the elements of the windows. To describe the position within a subwindow, we denote an element in $Y_w \times X_w \times V$, where elements in $Y_w \times X_w$ describe subwindow coordinates and V specifies the distinct subwindow in the range $[0, v-1]$. The positions within the superwindow are represented by $Y_W \times X_W \times V$, where elements in $Y_W \times X_W$ describe the position of the superpixel within the superwindow and V describes the position of a pixel within the superpixel.

First, we must determine the offset o of the first subwindow in the first superpixel:

$$o = v - (\lfloor {}^{w_x}/_2 \rfloor \bmod v)$$

Considering a specific position within the subwindow, the corresponding position within the superwindow can be calculated using

$$\begin{aligned} f &: Y_w \times X_w \times V \to Y_W \times X_W \times V, \text{ where} \\ f(i,j,k) &= (i,\ (j+o+k) \bmod v,\ \lfloor {}^{j+o+k}/_v \rfloor). \end{aligned}$$

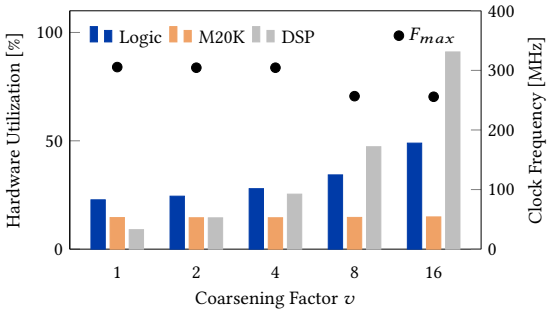


Figure 8. Hardware utilization for the loop coarsening of a 3×3 bilateral filter on an image of size 1024×1024 on a Stratix V.

Figure 8 shows the results we were able to achieve through loop coarsening on a Stratix V. The sublinear increase in resource usage proves to be exceptionally beneficial, in particular, if loop coarsening can be applied through code generation, without the necessity of modifying the algorithm's source code.

D. Vendor-Specific Transformations

1) Vector Data Types: There exist two basic types of vectorization: *implicit* and *explicit vectorization*. *Implicit vectorization* is a concept of replication that is automatically applied by the compiler. We already introduced our implementation of this concept with *loop coarsening*. The contrary concept is *explicit vectorization*, where the programmer explicitly specifies vector types such as `uchar4` to operate on multiple vector elements at once.

Handing over vector data between kernels requires specific code generation techniques, which implement wider data types than the actual pixel data type used. How those types can be represented in source code depends on the particular HLS tool the code is generated for. Regarding Vivado HLS, a flexible type width can be accomplished by utilizing the Vivado-specific type `ap_uint<`*bit width*`>`. For AOCL, we can directly exploit built-in OpenCL vector types, without the need to declare any new data types. However, mixing both vectorization types enforces the necessary bit width to be even larger. For instance, for an implicit vectorization of factor 32 for an image operator that uses pixels of type `uchar4`, our compiler would generate the

[2]Note that the our superwindow implementation still is a shift register that shifts superpixels in each cycle.

1 Local Operator & 1 Invocations

Hipacc:	$11 + 10$ LoC	
Tesla K20:	3356.0 FPS	
Tegra K1:	25.3 FPS	
Kintex 7:	676.3 FPS	($v = 4$)
Stratix V:	4301.7 FPS	($v = 16$)

(a) Bilateral

1 Local + 1 Point Operator & 3 Invocations

Hipacc:	$8 + 24$ LoC	
Tesla K20:	4103.0 FPS	
Tegra K1:	72.1 FPS	
Kintex 7:	4514.3 FPS	($v = 16$)
Stratix V:	4162.1 FPS	($v = 16$)

(b) Sobel

2 Local + 3 Point Operators & 9 Invocations

Hipacc:	$23 + 53$ LoC	
Tesla K20:	921.7 FPS	
Tegra K1:	33.5 FPS	
Kintex 7:	1620.2 FPS	($v = 8$)
Stratix V:	1057.8 FPS	($v = 4$)

(c) Harris Corner

2 Local + 1 Point Operator & 5 Invocations

Hipacc:	$31 + 52$ LoC	
Tesla K20:	452.5 FPS	
Tegra K1:	18.4 FPS	
Kintex 7:	483.6 FPS	($v = 4$)
Stratix V:	249.6 FPS	($v = 4$)

(d) Optical Flow

Figure 9. Algorithms in Hipacc: noise reduction (a), edge detection (b), corner detection (c), as well as optical flow (d). Listed are the number of operators in DSL code to describe the algorithms as well as the total number of operator invocations. The (X + Y) LoC denote the DSL parametrization (X) and the algorithm implementation (Y) in DSL code.

data type `ap_uint<1024>` for Vivado HLS and `uchar4[32]` for AOCL.

2) Bit-Width Reduction: Altera OpenCL provides the ability to reduce integer data types to an arbitrary bit width. The OpenCL standard, however, only contains primitive data types of fixed width, such as `char`/`short`/`int`. Therefore, to guide the Altera Offline Compiler (AOC) to reduce the bit width of a variable, a bitwise AND operation with a specific bit mask must be applied. To reduce the bit with of a variable, a masking operation must be applied to any *read* of that variable. Moreover, if such a variable is modified via an *assignment*, the entire Right Hand Side (RHS) expression must be masked as well. This particularly affects compound assignment operations (+=), as those combine *reading* and *assigning* a variable. Therefore, in the very last stage, just before pretty printing the AST to the target language's source code, the following actions are executed in this order:

1) All compound assignment and unary increment/decrement operations on a marked variable are transformed to the corresponding arithmetic operation followed by the assignment.
2) Every read operation of a marked variable is masked.
3) The entire RHS of every assignment to a masked variable is masked.

The resulting source code contains plenty of operations that decrease readability but do not affect functional correctness. For instance, by reducing the bit width of a variable `x` to 3, a simple `x += RHS` expands to:

```
x = ((x & 0x7) + RHS) & 0x7
```

V. Evaluation and Results

For evaluation, we are presenting results for a Kintex 7 (XC7K325T FFG900) and Stratix V (5SGXEA7) FPGA to results on server-grade and embedded GPUs. We used the Intel FPGA SDK for OpenCL v16.1, Xilinx Vivado HLS v2016.3 and CUDA v7.5 for the synthesis and the compilation. Moreover, memory transfer overheads are excluded from our throughput measurements, which is the common approach for GPU benchmarking. In this section, we first introduce our application set before we discuss the performance achieved for different target architectures stemming from the exact same DSL source codes.

A. Applications

We consider typical image processing algorithms for preprocessing and feature detection: a bilateral noise reduction, a Sobel edge detector, a Harris corner detector [3], and the computation of optical flow using the census transform [17]. Those algorithms form a realistic scenario and reflect possible cost-sensitive implementations in commercial products, such as medical imaging or advanced driver assistance systems. All of them are based on local and point operators or a combination of both but differ greatly in implementation detail.

Figure 9 shows the number of operators defined and how often the are invoked for these algorithms. It shows also the Lines of Code (LoC) required to describe them in Hipacc as well as the performance in Frames Per Second (FPS) of the automatically generated implementations. In the following, we summarize which operators are required for each algorithm and highlight features of the DSL to realize them:

- The bilateral filter [18] is a 3×3 local operator used for reducing noise while preserving edges. It consists of an exponential function and employs floating point arithmetic. The actual kernel implementation was already provided in Section II-C.
- The Sobel computes first vertical and horizontal derivatives with 3×3 masks, then, calculates the Euclidean distance and clamps the result with a given threshold in the third kernel to detect edges.
- The Harris corner detector embodies a combination of point and local operators that form a complex image pipeline. In total, 9 operator invocations are required to detect the corners in the input image.
- The optical flow includes the computation of a signature for each pixel of the smoothed input images. The signatures of two successive images are used to compute the optical flow:

Table I
RESOURCE UTILIZATION OF SYNTHESIZED ALGORITHMS.

Filter	Kintex 7 SLICE (%)	Kintex 7 BRAM (%)	Kintex 7 DSP (%)	Stratix V Logic (%)	Stratix V M20K (%)	Stratix V DSP (%)
Bilateral	44.68	0.45	90.48	46.91	13.16	91.02
Sobel	28.79	3.60	15.24	25.82	16.68	25.00
Harris Corner	55.75	3.60	60.00	43.16	15.19	14.06
Optical Flow	60.09	11.69	0.00	42.26	42.30	28.13

They are compared within a sliding window of size 15×15 using the `iterate()` function over a *Domain* that excludes the center.

B. Comparing FPGA and GPU Throughput

Results on throughput for the chosen FPGA and GPU targets are also presented in Figure 9. The optimizations for GPUs applied by Hipacc include constant propagation and unrolling of the convolve method, using texture memory when reading from global memory, staging the data to shared memory as well as unrolling of the global iteration space. Furthermore, the CUDA block configuration is automatically determined by Hipacc.

The hardware accelerators generated by Hipacc are designed to support high-speed serial data communication. Thus, the loop coarsening factor v is not limited by the number of available parallel I/O package pins. With increasing algorithm complexity, the opportunity for further parallelizing FPGA accelerators is mostly constrained by available resources, as shown in Table I. In contrast to FPGAs, the throughput of the GPUs is dramatically reduced with greater sliding window sizes, and thus an increased number of memory accesses, exposing the available memory bandwidth clearly as the main performance bottleneck.

VI. CONCLUSION

In this work, we introduced Hipacc, a DSL for image processing and a source-to-source compiler to generate highly efficient hardware accelerators. With the aid of domain and architecture knowledge, Hipacc can produce tailored implementations, capable of competing with hand-written source codes, without the need of an hardware design expert. The abstractions of the DSL enable to map an algorithm to a wide range of target architectures and to exploit the target-specific types of parallelism.

As Hipacc also targets GPU architectures, we can moreover generate highly optimized GPU implementations from exactly the same code base. The evaluation of several typical image processing algorithms demonstrates that an even higher performance can be achieved by FPGA implementations. Although HLS still has some deficiencies in contrast to hand-written Register Transfer Level (RTL) implementations, it enables a truly rapid design space exploration [11] through which it is possible to achieve an improved ratio between resource usage, throughput, and energy efficiency.

ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG), as part of the Research Training Group 1773 “Heterogeneous Image Systems”, and as part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89). The Tesla K20 used for this research was donated by the NVIDIA Corporation.

REFERENCES

[1] I. N. Bankman. *Handbook of Medical Image Processing and Analysis*, volume 2. Academic Press, Dec. 2008. ISBN: 978-0-123-73904-9.

[2] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pages 327–338, Haifa, Israel. ACM, 2016. ISBN: 978-1-4503-4121-9. DOI: 10.1145/2967938.2967969. URL: http://doi.acm.org/10.1145/2967938.2967969.

[3] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[4] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics (TOG)*, 33(4):144:1–144:11, July 2014. DOI: 10.1145/2601097.2601174.

[5] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics (TOG)*, 35(4):85:1–85:11, July 2016. DOI: 10.1145/2897824.2925892.

[6] R. Klette and P. Zamperoni. *Handbook of Image Processing Operators*, volume 1. John Wiley & Sons, Apr. 1996. ISBN: 978-0-471-95642-6.

[7] D. Koch, F. Hannig, and D. Ziener, editors. *FPGAs for Software Programmers*. Springer, June 2016. 327 pages. ISBN: 978-3-319-26406-6. DOI: 10.1007/978-3-319-26408-0.

[8] R. Membarth, O. Reiche, F. Hannig, and J. Teich. Code generation for embedded heterogeneous architectures on Android. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. (Dresden, Germany), 86:1–86:6. European Design and Automation Association (EDAA), Mar. 24–28, 2014. DOI: 10.7873/DATE.2014.099.

[9] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. HIPAcc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, Jan. 1, 2016. DOI: 10.1109/TPDS.2015.2394802.

[10] M. Özkan, O. Reiche, F. Hannig, and J. Teich. FPGA-based accelerator design from a domain-specific language. In *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL)*. (Lausanne, Switzerland). IEEE, Aug. 29–Sept. 2, 2016. ISBN: 978-2-8399-1844-2. DOI: 10.1109/FPL.2016.7577357.

[11] O. Reiche, K. Häublein, M. Reichenbach, M. Schmid, F. Hannig, J. Teich, and D. Fey. Synthesis and optimization of image processing accelerators using domain knowledge. *Journal of Systems Architecture*, 61(10):646–658, Oct. 9, 2015. DOI: 10.1016/j.sysarc.2015.09.004.

[12] O. Reiche, C. Kobylko, F. Hannig, and J. Teich. Auto-vectorization for image processing DSLs. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded systems (LCTES)*. (Barcelona, Spain), pages 21–30. ACM, June 21–22, 2017. ISBN: 978-1-4503-5030-3. DOI: 10.1145/3078633.3081039.

[13] 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)*. (New Dehli, India), 17:1–17:10. ACM, Oct. 12–17, 2014. ISBN: 978-1-4503-3051-0. DOI: 10.1145/2656075.2656081.

[14] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, July 2007. ISBN: 978-0-596-51480-8.

[15] J. C. Russ. *The Image Processing Handbook*, volume 5. CRC Press, Dec. 2006. ISBN: 978-0-8493-7254-4.

[16] M. Schmid, O. Reiche, F. Hannig, and J. Teich. Loop coarsening in C-based high-level synthesis. In *Proceedings of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. (Toronto, Canada), pages 166–173. IEEE, July 27–29, 2015. ISBN: 978-1-4799-1925-3. DOI: 10.1109/ASAP.2015.7245730.

[17] F. Stein. Efficient computation of optical flow using the Census Transform. In C. Rasmussen, H. Bülthoff, B. Schölkopf, and M. Giese, editors, *Pattern Recognition*, Lecture Notes in Computer Science, pages 79–86. 2004.

[18] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In pages 839–846. IEEE, Jan. 1998.

[19] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. AutoPilot: A platform-based ESL synthesis system. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis: From Algorithm to Digital Circuit*, part 6, pages 99–112. Springer, 2008. ISBN: 978-1-4020-8587-1. DOI: 10.1007/978-1-4020-8588-8_6.