This is the author's version of the work. The definitive work was published in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), New Dehli, India, October 12-17, 2014.*

Code Generation from a Domain-specific Language for C-based HLS of Hardware Accelerators

Oliver Reiche‡, Moritz Schmid‡, Frank Hannig‡, Richard Membarth†, and Jürgen Teich‡
‡Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
†Intel Visual Computing Institute, Saarland University, Germany
{oliver.reiche, moritz.schmid, hannig}@cs.fau.de, richard.membarth@uni-saarland.de, teich@cs.fau.de

ABSTRACT

As today's computer architectures are becoming more and more heterogeneous, a plethora of options including CPUs, GPUs, DSPs, reconfigurable logic (FPGAs), and other application-specific processors come into consideration for close-to-sensor processing. Especially, in the domain of image processing on mobile devices, among numerous design challenges, a very stringent energy budget is of utmost importance, making embedded GPUs and FPGAs ideal targets for implementation.

Recently, the HIPA^{cc} framework was proposed as a means for automatic code generation of image processing algorithms for embedded GPUs, based on a Domain-Specific Language (DSL). Despite of huge advancements in High-Level Synthesis (HLS) for FPGAs, designers are still required to have detailed knowledge about coding techniques and the targeted architecture to achieve efficient solutions. As a remedy, in this work, we propose code generation techniques 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, numerous high-level transformations, and automatic test bench generation. We evaluate our approach by comparing the resulting hardware accelerators to existing frameworks in terms of performance and resource requirements. Moreover, we assess the achieved energy efficiency in contrast to software implementations, generated by HIPA^{cc} from the same code base, executed on GPUs.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Optimization, Retargetable compilers*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Languages*

General Terms

Design, Languages, Performance

Keywords

Code generation, domain-specific language, image processing, high-level synthesis, hardware accelerator, FPGA, GPU

ESWEEK'14, October 12–17 2014, New Delhi, India

http://dx.doi.org/10.1145/2656075.2656081

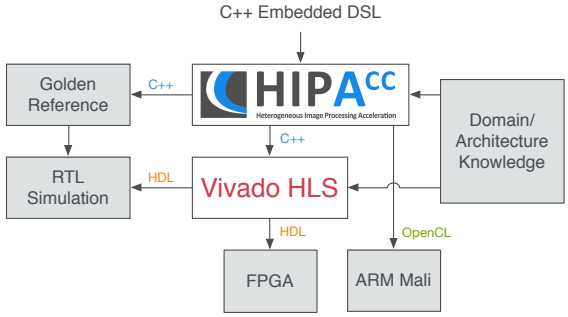

Figure 1: Design flow of the proposed combination of HIPA^{cc} and Vivado HLS.

1. INTRODUCTION

Algorithms for close-to-sensor processing, such as required by advanced driver assistance systems and mobile scanners, are becoming more and more complex and must deliver enough performance to process vast amounts of data on embedded devices with stringent resource and energy budgets. Due to these constraints, ideal targets for the implementation are hardware accelerators, such as embedded Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Efficient implementations for these, however, require a deep understanding of the algorithmic details and the hardware architecture. To ease the burden on developers, DSLs aim at combining architecture- and domain-specific knowledge, thereby delivering performance, productivity, and portability. So far, DSLs have been researched for a long time for Central Processing Units (CPUs) as well as GPUs, and recently have also targeted hardware design [8], which has mostly been the prime domain for HLS. Over the past decades, C-based HLS focusing on FPGAs has become very sophisticated, producing designs that can rival hand-coded Register-Transfer Level (RTL). A drawback is that these frameworks must be very flexible and although being able to create an efficient hardware design from a C-based language can significantly shorten the development time, architectural knowledge and specific coding techniques are still a must. A remedy to this situation is to increase the level of abstraction even further and use a domain-specific framework to generate code for FPGA HLS. HIPA^{cc} is a publicly available framework for the automatic code generation of image processing algorithms on GPU accelerators. Starting from a C++ embedded DSL HIPA^{cc} delivers tailored code variants for different target architectures, significantly improving the programmer's productivity. In this work, we use HIPA^{cc} as foundation and extend it to be able to generate C++ code specific to the C-based HLS framework Vivado HLS from Xilinx. The proposed design flow is depicted in Figure 1.

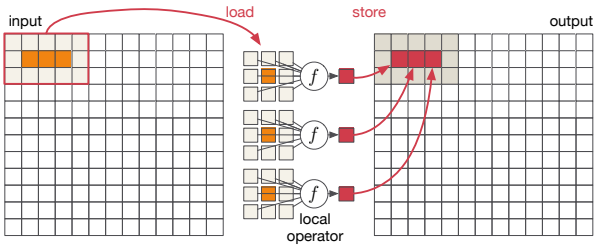

Figure 2: For local operators on GPUs, memory transfers are grouped for consecutive pixels, simultaneously serving several parallel threads, each computing a single output pixel.

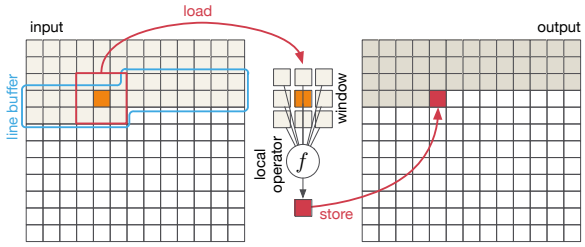

Figure 3: A combination of line buffers and memory windows is typically used to process local operators on streaming data.

The contributions of this work can be summarized as follows.

- We introduce FPGAs as a novel target to HIPA^{cc} in order to generate code for C-based HLS from a DSL.
- In detail, we discuss how the framework must be modified to cope with the design challenges of the FPGA target.
- We evaluate our approach by assessing the performance and energy requirements of the generated FPGA designs in contrast to other hardware targets, supported by HIPA^{cc}.

The generated target code is derived from a high-level description for image processing algorithms. Therefore, this work uses the high-level description presented in [6].

2. BACKGROUND

Image processing on embedded devices is mainly impelled by reduced energy consumption. As a consequence, certain imaging operations are often mapped to Application-Specific Integrated Circuit (ASIC) pipelines in packaged camera products.

A simple example are edge and feature detectors in image processing pipelines. One of the well-known edge detectors is the Laplacian operator, where multiple neighboring pixels are accessed within a given window. We denote such an algorithm as a *local operator*.

In the following, we will briefly discuss special considerations and design challenges in order to produce efficient implementations of such operators for both design targets: Embedded General Purpose GPUs (GPGPUs) and FPGAs.

2.1 Embedded GPU Solution

Embedded GPGPUs such as the ARM Mali-T600 or the Qualcomm Adreno 300 series are programmable using Open Computing Language (OpenCL) and offer a software-based solution for energy-efficient image processing.

Those architectures apply a high degree of parallelism in hardware by utilizing many compute cores simultaneously, as depicted in Figure 2. Besides having many cores available, massively exploiting Simultaneously Multithreading (SMT), where multiple threads are mapped to a single core exchanging on stalls, further increases the achievable extent of parallelism.

For a Laplacian operator with a 3×3 window, this would result in 10 memory transfers and 9 Multiply-Accumulate (MAC) operations for each thread. Fulfilling a memory transaction request demands orders of magnitude more cycles than performing a compute operation. Therefore, local operators on GPUs are memory-bound and the execution time depends, above all, on the operator's window size.

For memory-bound kernels, the amount of SMT running on each core saturates at a certain level, depending on the specific architecture at hand. However, the pipeline stages of the GPU's load and store units might still be underemployed, which leaves space to further raise the achievable degree of parallelism. Therefore, it can be beneficial to increase the number of load and store operations per thread, by merging the execution of multiple threads into a single one [14] (thread-coarsening). As a downside, this technique significantly increases register usage, which might reduce the overall number of threads the GPU can stem.

A common technique to further reduce the impact of memory reads, is to use local on-chip[1] and texture memory. Texture memory offers improved cache locality for two and three dimensional data access patterns. In this way, especially for local operators, the cache hit rate may be notably increased. However, on embedded GPGPUs, this feature is usually restricted to certain formats, such as *RGBA* forcing the use of 4-channel vector types. Moreover, compute cores of embedded GPGPUs are almost solely based on Very Long Instruction Word (VLIW) or Single Instruction Multiple Data (SIMD) units. Therefore, if one cannot rely on the compiler's ability for implicit vectorization, it is crucial to provide explicitly vectorized code for achieving high performance on embedded GPGPUs.

2.2 FPGA

Due to their high computational power in combination with excellent energy efficiency, FPGAs have not only recently been a target for the implementation of signal processing systems [12]. In order to keep up with the high processing rates of GPUs, it is vital to exploit the massive parallelism of the platform.

In case of image processing, many software developers are used to having the whole image available in memory to perform calculations. The situation is completely different for FPGA-based image processing, where, foremost due to limited memory resources, having the complete image in memory for processing is unrealistic. With the advance of high-speed serial transceivers, which are used to interconnect FPGAs with communication and memory controllers, it has become a common approach to stream data onto the FPGA, provide buffering for the calculations, and immediately deliver the processed results to the off-chip sink as it becomes available. As a consequence, in order to obtain an efficient implementation of an image processing algorithm for an FPGA platform, it must be seamlessly integrated with the processing methodology.

Local operators, such as the Laplacian operator, 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. Memory resources on modern FPGAs can be broadly categorized into Block RAMs (BRAMs) and Flip-Flops (FFs). An efficient memory architecture for streaming data uses a combination of *line buffers* for storage of complete

[1]Local on-chip memory is not available on current embedded GPGPU architectures.

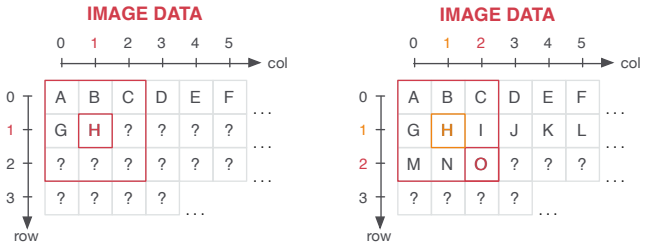

Figure 4: Data causality when processing window-based operators for streaming data.

image lines (BRAMs) and *memory windows* for the actual processing of local neighborhoods (implemented using FFs), as illustrated in Figure 3.

Another challenge for stream-based image processing using local operators is depicted in Figure 4. When computing the window around some center pixel (here at index (1,1)), almost half of the data in the window are not available, yet. To ensure that all data are available, it is necessary to enforce a delay, we refer to as the *group delay*, before processing the window. For a window of size $w \times w$, the group delay t can be calculated as $t = \lfloor w/2 \rfloor$.

One of the most important aspects for hardware acceleration is to exploit the massive parallelism of the architecture and *pipeline* as many of the operations as possible. Here, *throughput* (how much data per time unit) and *local latency* (how many clock cycles to complete one iteration) are contrasting design goals that also affect design speed as well as required area and energy. For streaming pipelines in image processing, a high throughput is often considered more important, since the local latency often becomes negligible in comparison to the amount of pixels to process.

3. PROGRAMMING MODEL

3.1 Vivado HLS

Recently, Xilinx included the high-level synthesis tool *AutoESL* (formerly known as *AutoPilot* [15]) into the Vivado Design Suite, renamed as Vivado HLS. Vivado HLS allows design entry in C/C++ or SystemC and delivers an HDL code (VHDL, Verilog, and SystemC) for a synthesizable IP core. Vivado HLS is specifically targeted at Xilinx FPGAs and fully integrates with Xilinx IDEs. A broad range of synthesis directives can be used for architecture-driven parallelization and optimization in order to transform the sequential algorithm specification into an efficient parallel hardware design. From SystemC, the concept of data streams was adopted which enables the designer to interconnect different modules and implement these in a streaming pipeline. In contrast to several other commercial HLS tools, Vivado HLS provides human readable HDL code, which allows for manual alterations, which may come especially handy for simulation and synthesis optimization. Although being able to specify complex algorithms in a C-based language might severely increase productivity, the implementation of image processing system requires sophisticated architectural knowledge in order to obtain high quality results that can keep up with hand-coded implementations at RTL.

3.2 HIPA^{cc} Framework

The Heterogeneous Image Processing Acceleration (HIPA^{cc}) framework [6] consists of a DSL that is embedded into C++ and a source-to-source compiler. Exploiting the compiler, image filter descriptions written in DSL code can be translated into multiple target languages such as Compute Unified Device Architecture (CUDA), OpenCL, or Renderscript as used in Android [7]. In this work, our approach is to generate code for C-based HLS from a high-level description. This framework will serve as a basis for generating appropriate C++ code that can be further processed by Vivado HLS.

```
1 // input image
2 const int width = 512, height = 512;
3 uchar4 *image = (uchar4*)read_image(width, height, "input.pgm");
4
5 // coefficients for Laplacian operator
6 const int coef[3][3] = { {  0,  1,  0 },
7                          {  1, -4,  1 },
8                          {  0,  1,  0 } };
9
10 Mask<int> mask(coef);
11 Image<uchar4> in(width, height);
12 Image<uchar4> out(width, height);
13
14 // load image data
15 in = image;
16
17 // reading from in with mirroring as boundary condition
18 BoundaryCondition<uchar4> bound(in, mask, BOUNDARY_MIRROR);
19 Accessor<uchar4> acc(bound);
20
21 // output image
22 IterationSpace<uchar4> iter(out);
23
24 // define kernel
25 Laplacian filter(iter, acc, mask);
26
27 // execute kernel
28 filter.execute();
```

Listing 1: Example code for the Laplacian operator.

In the following sections, we will use the Laplacian operator as a simple example for describing image filters and briefly describe which source code transformations are applied and how code generation is accomplished.

3.2.1 Domain-Specific Language

Embedded DSL code is written by using C++ template classes provided by the HIPA^{cc} framework. Therefore, DSL code is written in a similar manner as using a framework API. In fact, those compiler-known classes are fully functional and can be compiled by a normal C++ compiler, serving as a reference CPU implementation. However, with the HIPA^{cc} source-to-source compiler, code generation is involved that targets mainly GPU accelerators.

The most essential C++ template classes for writing 2D image processing DSL codes are: (a) an `Image`, which represents the data storage for pixel values; (b) an `IterationSpace` defining the Region Of Interest (ROI) for operating on the output image; (c) an `Accessor` defining the ROI of the input image and enabling filtering modes (e. g., nearest neighbor, bilinear interpolation, etc.) on mismatch of input and output region sizes; (d) a `Kernel` specifying the compute function executed by multiple threads, each spawned for a single iteration space point; (e) a `Domain`, which defines the iteration space of a sliding window within each kernel; and (f) a `Mask`, which is a more specific version of the `Domain`, additionally providing filter coefficients for that window. Image accesses within the kernel description are accomplished by providing relative coordinates. To avoid out-of-bound accesses, kernels can further be instructed to implement a certain boundary handling (e. g., clamp, mirror, repeat) by specifying an instance of class `BoundaryCondition`. Template parameters of all mentioned classes are used to define the type of pixel data that is processed (e. g., represented as integer or floating point), enforcing kernels to be consistent and sound in terms of type conversion within the host language's type system.

To describe a Laplacian operator, we need to define a `Mask` and load the appropriate filter coefficients, defined as constants, see Listing 1 (lines 6–10). It is further necessary to create an

```
1 class Laplacian : public Kernel<uchar4> {
2   private:
3     Accessor<uchar4> &input;
4     Mask<int> &mask;
5
6   public:
7     Laplacian(IterationSpace<uchar4> &iter,
8               Accessor<uchar4> &input, Mask<int> &mask)
9         : Kernel(iter), input(input), mask(mask) {
10      addAccessor(&input);
11    }
12
13    void kernel() {
14      int4 sum = { 0, 0, 0, 0 };
15      for (int y = -1; y <= 1; ++y)
16        for (int x = -1; x <= 1; ++x)
17          sum += mask(x, y) * convert_int4(input(x, y));
18      sum = max(sum, 0);
19      sum = min(sum, 255);
20      output() = convert_uchar4(sum);
21    }
22 };
```

Listing 2: Kernel for the Laplacian operator.

```
1 int4 sum = convolve(mask, HipaccSUM, [&] () -> int4 {
2              return mask() * convert_int4(input(mask));
3            });
4 sum = max(sum, 0);
5 sum = min(sum, 255);
6 output() = convert_uchar4(sum);
```

Listing 3: Alternative convolution kernel.

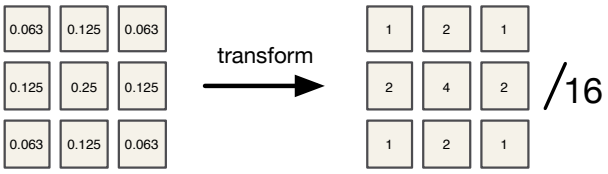

Figure 5: Transformation of a Gaussian convolution mask from floating point to integer coefficients.

input and output image for storing pixel data and load initial image data into the input image (lines 11–15). The input image is bound to an `Accessor` with enabled boundary handling mode *mirroring* (lines 18–19). After defining the iteration space, the kernel can be instantiated (line 25) and executed (line 28).

Kernels are implemented by deriving from the framework's provided `Kernel` base class, inheriting a constructor for binding the iteration space and a `kernel()` method. Within that method the actual kernel code is provided, see Listing 2 (lines 14–20). Mask and input image are accessed using relative coordinates (line 17). Thereby, it is ensured that out-of-bound accesses are caught and handled appropriately according to the specified boundary handling mode. To write the output pixel value to the iteration space, the convolution result must be assigned to the `output()` method (line 20).

Because the Laplacian operator is a local operator performing standard convolution, it is also possible to describe the kernel using the `convolve()` method shown in Listing 3. This method takes three arguments: (a) the mask defining window size and filter coefficients; (b) the reduction mode used to accumulate the results of multiple iterations; and (c) a C++ lambda function describing the computational steps that should be applied in each iteration.

Additional HIPA^{cc} language constructs for local operators also provide more general operations for iteration and reduction steps over local window regions. Those regions do not have to be rectangular. Besides local operators, global operators are also supported for describing reductions (*min*, *max*, or *sum*) of images or an image region.

3.2.2 Code Generation

The HIPA^{cc} compiler is based on the Clang/LLVM 3.4 compiler infrastructure[2]. Utilizing the Clang front end, HIPA^{cc} parses C/C++ code and generates an internal Abstract Syntax Tree (AST) representation. Operating on this representation, HIPA^{cc} will generate two kinds of code: Host code for managing kernel launches and memory transfers, and device code containing the actual kernel description in the specified target language (e. g., CUDA, OpenCL, Rendescript).

3.2.3 Generating Code for Vivado HLS

Considering Vivado HLS as a target for code generation involves numerous challenges to overcome. Convolution masks provided in DSL code must be translated in a more suitable version for FPGAs and hardware accelerators. The same applies to DSL vector types that need to be wrapped into integer streaming buffers for pipelining. Such a pipelined structural description has to be inferred from the linear execution order of kernels. Hereafter, kernel implementations need appropriate placement of Vivado HLS pragmas depending on the desired target optimization.

[2]http://clang.llvm.org

4. TRANSFORMING MASKS

Mask coefficients of a convolution f, where its result remains in the same range as the input, can be described in two ways: First by multiplying with integer values followed by a division step for normalizing to the original range, or second by providing floating point coefficients, which in total yield exactly 1. On GPUs, the latter is the preferred choice, as issuing integers or floating points instructions makes no difference in terms of latency. Since the final normalization step at the end is avoided, it is even more beneficial to use floating point coefficients. Therefore, the DSL framework follows this approach.

On FPGAs however, the multiplication with integer coefficients and the normalization step has a reduced impact on resource requirements. Ideally the whole convolution can be covered by simple shift operations.

As the HIPA^{cc} framework does not support the first approach, it is favorable to transform given floating point coefficients to integer values. An example for transforming a simple Gaussian mask is provided in Figure 5. For the transformation, certain constraints need to be met. The mask size must be constant and the coefficients must be known beforehand at compile time. Mask transformation can only be applied if the condition in Equation (1) holds, which states that every coefficient c_i scaled by the normalization factor N results in a natural number (with respect to a small error ε, which can be defined at compile time),

$$\forall c_i \in C, \exists n \in \mathbb{N}_0 : N \cdot c_i \pm \varepsilon = n \tag{1}$$

where N is defined as the inverse of the smallest coefficient $N = {}^1/_{\min_{c_i \in C}\{c_i\}}$.

The transformation of every floating point coefficient c_i to its integer representative x_i is done by scaling each coefficient by the normalization factor and correct rounding, defined as $x_i = \lfloor N \cdot c_i + 0.5 \rfloor$.

It can be shown, given the constraint defined in Equation (1), that the convolution is still valid. With respect to the granted small error ε, results of both approaches are approximately equal, depicted in Equation (2).

$$\sum_{c_i \in C} c_i \cdot d_i \approx \frac{\sum_{c_i \in C} x_i \cdot d_i}{N} \tag{2}$$

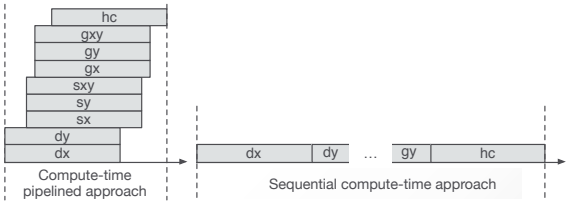

Figure 7: Pipelined and sequential execution.

As those results are not exact, they might not be suitable for every algorithm or use case. Therefore, this feature can be controlled by a compiler switch.

5. STREAMING PIPELINE

High-level programs given in HIPA^{cc} DSL code process image filters buffer-wise. Each kernel reads from and writes to buffers sequentially, running one after another with buffers serving 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.

Throughout this section, we will demonstrate our method using the Harris corner detector [4], depicted in Figure 6. This filter consists of a pipeline of kernels, implemented as point operators and local operators, which are described in great detail in Section 7.2. Although, every kernel writes to its own buffer, in total only four buffers are necessary by applying smart reuse of existing buffers containing out-dated intermediate data.

This buffered concept is fundamentally different from streaming data through kernels, processing a computational step as soon as all input dependencies are available. Kernels are therefore interconnected with each other using stream objects implementing First In First Out (FIFO) semantics. Such a streaming concept requires a structural description, resolving direct data dependencies unconstrained from the exact sequential ordering of kernel executions.

We can transform the buffer-wise execution model into a structural description suitable for streamed pipelining by analyzing the DSL host code, replacing memory transfers by stream objects, and generating appropriate kernel code. Vivado HLS can then be instructed to run all kernels in parallel, as shown in Figure 7, which can deliver a significantly shorter processing time. We first introduce how the structural description is transformed from a given host code and then give insight into device code generation.

5.1 Host Code Analysis

The host code is translated into an AST representation that is traversed by HIPA^{cc}. During this traversal process, we track the use of buffer allocations, memory transfers and kernel executions by detecting compiler-known classes. 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, reused buffers (indicated by colors in Figure 6) will form new *space* vertices in the graph. Furthermore, when the inputs of multiple kernels depend on the same buffer and the same temporal instance of intermediate data (e. g., dx in Figure 6), it is required to replace these dependencies by a *process* for splitting the data,

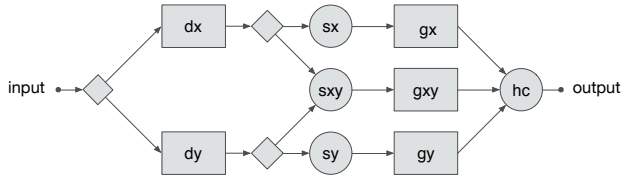

Figure 8: Streaming pipeline of Vivado kernels for the Harris corner detector. Diamond shapes represent kernels for splitting data of a single stream object into multiple stream objects.

followed by multiple *spaces*, one for each kernel. This way, it is guaranteed that streaming data later on will be copied before handing it over to the next computation steps.

From our internal representation, we can infer the structural description for the streaming pipeline shown in Figure 8. The graph is linearized for code generation by traversing backwards through the graph in Depth-First Search (DFS), originating from the output *spaces*. Herby, parts of the graph that are not contributing to the output will be pruned. For code generation, every *process* vertex is translated to a kernel execution and every *space* vertex marks the insertion of a unique Vivado HLS stream object. The resulting code embodies the structural description of the filter, which is written to a file serving as *entry function*.

5.2 Device Code Generation

For generating device code, an AST is extracted and extended from the DSL's kernel description. Additional nodes are added to fulfill the requirements of the target language such as applying index calculations and ROI index shifts. Depending on optimizations specified by compiler options, vast portions of the extracted AST are modified and replicated. For example when enabling the use of texture or local on-chip memory on GPUs, the appropriate language constructs must be inserted. Further optimizations like thread-coarsening (similar to partial loop unrolling), where multiple executions of the same kernel are merged into a single one, are applied by cloning the extracted AST and inserting it multiple times with slightly adjusted memory access indices. The resulting AST is transformed back to source code by utilizing Clang's *pretty printer* and written to separate files for each kernel. These files will be included (CUDA) or loaded (OpenCL) by the rewritten host code file.

For Vivado HLS, we needed to introduce some additional constraints. As described earlier, masks must be constant, not only for mask transformation, but also for constant propagation[3], which we enforce for FPGA targets. Further constraints are that image dimensions must be known beforehand at compile time, in order to process it by Vivado HLS synthesis right away. The resulting C++ code is based on a highly optimized library for image processing, as proposed in [9], and is still human-readable. Globally affecting parameters (such as image dimensions) are defined at a single central spot, so that those parameters can be conveniently altered for further syntheses without rerunning the HIPA^{cc} compiler. Similar to HIPA^{cc}'s other targets, separate files are created for each kernel. These files will be included by the *entry function*, which already embeds all executions in a structural description, described in the previous section. Kernel optimizations applied for device code are explained in the next section.

[3]Replacing memory loads of constant values by inserting the numerical literals.

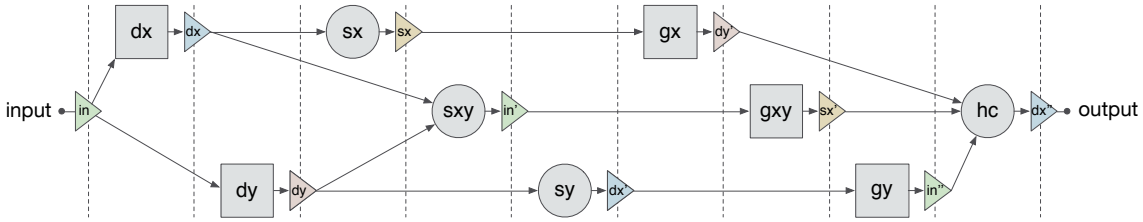

Figure 6: Sequential execution of HIPA^{cc} kernels for the Harris corner detector implemented as a pipeline of point operators (circles) and local operators (squares). Triangles mark buffers and dashed lines represent host barriers between kernel executions.

5.3 Parallelization and Design Optimization

Although the possibility to use a C-based language for design entry lowers the hurdle for acceptance of a HLS framework, algorithms stated in such a language are inherently sequential and must be parallelized and optimized in order to efficiently use the FPGAs resources. As opposed to the world of High-Performance Computing (HPC), where the fastest processing speed is paramount, developing hardware accelerators may be subject to several contrasting design goals. Of course, high throughput and short clock periods are important achievements, but often it is also necessary to comply with a certain resource budget. A central element of Vivado HLS for this task are the synthesis directives, which allow to specify how the input design is to be parallelized and optimized.

5.3.1 Placing Vivado Synthesis Directives

Synthesis directives in Vivado HLS can either be inserted in the code directly as *pragmas*, or collected in a script file which is applied during synthesis. Here, we use both approaches. Directives, which transform the sequential specification into a typical parallel hardware structure, such as *unrolling* a for-loop to model a shift-register, or which optimize hierarchical structures, for example *inlining* calls to small functions, are rarely changed and are thus placed directly in the code as pragmas. Other directives, we frequently use for parallelization and optimization, such as *pipelining* and specifying the size of FIFOs to interconnect a streaming pipeline can be used for design space exploration and to enforce a certain optimization strategy. Searching and altering the code to adapt these directives may become inconvenient for large designs. Therefore, we provide these directives in a script file, so that they can be readily changed.

5.3.2 Optimizing Loop Counter Variables

Software developers often tend to use the convenient `int` data type to specify loop counter variables. For image processing, the image dimensions seldom require the full range of a 32-bit two's complement. Moreover, using more bits for a variable than required causes undesired excessive use of resources and may degrade the achievable maximum clock frequency, especially if the variable must often be compared to another. In Vivado HLS, appropriate bit-widths can be explicitly specified if the range of the variable is known during the code generation. Another possibility is to let the synthesis tool automatically infer the required bit-width by inserting assertions on the range of the variable, similar to the range definitions for the `integer` data type in VHDL. We apply assertions to loop counter variables in our approach, so that bit widths do not need to be explicitly adapted if image dimensions change between code generation and synthesis. In this way, we ensure that the design always uses an optimal binary representation for loop variables, reduce the amount of required resources and achieve shorter critical paths for logical operations on such variables.

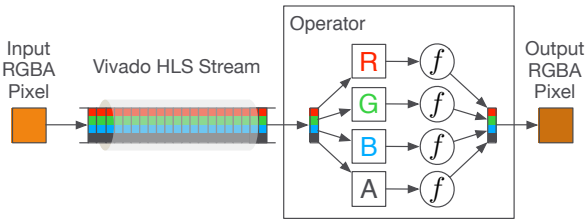

Figure 9: Packing of vector types into Vivado streams. This example shows an RGBA pixel, the channels are packed together into a single stream. Although the channels must be processed individually, a common stream and line buffer can be used.

5.3.3 Mapping Vector Types

Since HIPA^{cc} was developed for GPUs, it supports numerous vector types, which are crucial for performance on specific GPU architectures. Besides performance concerns tackled by explicit vectorization, these types are in particular well-suited for expressing computations on common formats of image processing in a natural way.

To support vector types for the Vivado HLS target, basically two approaches are available: (a) treat all vector elements separately, resulting in multiple line buffers, multiple windows, and multiple streams for each local operator; or (b) pack all vector elements into a single integer of the same bit-width as the whole vector, resulting in only a single stream for each operator, depicted in Figure 9. The first approach is realized by the Open Source Computer Vision (OpenCV) implementation provided with Vivado HLS 2014.1. The latter is followed by our code generation, because this might reduce the overall consumption of memory resources.

Given vectorized DSL code does not need to be modified. We have implemented vector types for Vivado HLS as C structures and realize computations by operator overloading. Reads and writes to DSL `Image`s, which will be mapped to stream objects, are replaced by conversion functions, either extracting the vector types from packed stream elements or packing vector types into stream elements. Inter-kernel computations, i. e., operations on local variables, are described as vector operations in DSL code already and are therefore covered by overloaded operators.

5.3.4 Delaying Point Operators

As explained earlier, obeying causality when applying local operators to streaming data causes an increased group delay. If point operators are used in streaming pipelines before a local operator, the production rate of the point operator is higher than the consumption rate of the local operator. One way to address this issue is to insert an appropriately sized stream buffer between the two operators. Alternatively, a delay can be enforced on the point operator to equalize consumption and

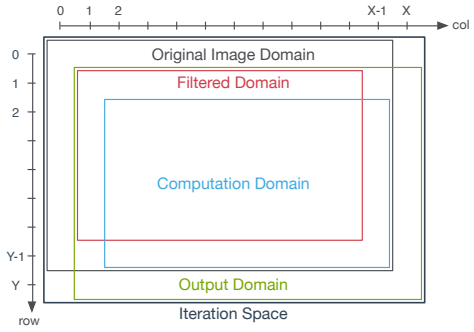

Figure 10: Iteration space for a local operator applied to streaming data.

production rates. The generated code uses the latter approach in order to save memory resources.

5.3.5 *Unified Iteration Space*

As discussed earlier, processing a local operator with window size w in a streaming data context increases the group delay. As a consequence, the iteration space of the algorithm must be enlarged and shifted by $t = \lfloor w/2 \rfloor$ in each dimension. Figure 10 illustrates an example using a local operator with a window of size 3×3. The original iteration space, ranging from $[0, X-1], [0, Y-1]$ must be enlarged by $t = \lfloor 3/2 \rfloor = 1$ to also include the output domain. In order to compute the filtered domain, the computation domain must be shifted by the same parameter $t = 1$ in each dimension. Using several local operators in a streaming pipeline may cause buffering problems if a larger window size follows a smaller window size, for example, data is first processed by a 3×3 window, then by a 5×5 window, since the smaller local operator has higher production rate than the second. The problem can be solved in two different ways by (a) introducing sufficiently sized FIFO buffers or by (b) unifying the iteration space. In this approach, we use the latter solution, since it reduces the amount of required on-chip memory resources. For this, we determine the maximum group delay $t_{\max}$ and adjust the iteration spaces of all the local and point operators in the pipeline according to the maximum delay $t_{\max}$. The iteration space of the computation domain, however, must still be defined using the local group delay t_{local}.

6. TEST BENCH GENERATION

When developing architectures and FPGA layouts, it is of great importance to cover sources of errors by testing, at best starting in early design phases. A solution to achieve effortless testing is automatic test bench generation from a high abstraction level. By applying such techniques, testing is more flexible, can be accomplished much more efficiently than on RTL, and requires less coding effort than, for example, in VHDL. Our approach allows to write DSL code from that HIPA^{cc} can derive a test bench that can be used by Vivado HLS.

In order to accomplish testing, HIPA^{cc} can utilize the provided DSL host code, which is embedded into C++ and thereby already surrounded by functioning program code. The host code's AST will be traversed by HIPA^{cc} using Clang's *Rewriter* engine. For each node that is related to the compiler-known classes, HIPA^{cc} rewrites the source code location by inserting appropriate runtime calls. On FPGAs for instance, assignments between pointers and `Image`s will be replaced by runtime calls for initially filling array data into Vivado stream objects. Similar replacements are applied to all other occurrences of compiler-known classes within the input code.

The resulting rewritten host code is still very similar[4] to the original input and the surrounding program code is left untouched. Therefore, if the designer has specified testing routines in the C++ program that verifies the results computed by DSL code, the rewritten program can serve as a test bench without further modification. The same applies to GPU targets, which further emphasizes the consistency of our approach to cover fundamentally different targets from one and the same code base.

7. EXPERIMENTAL RESULTS

We evaluate our results on several different hardware target platforms. All hardware targets are compared in terms of performance and energy efficiency. Our implementations are generated by HIPA^{cc} for each target, stemming from the same code base. Additionally, we analyze our synthesis results in contrast to an OpenCV implementation provided by Xilinx.

7.1 Evaluation Environment

First, we briefly describe our environment by providing detailed information about the architectures and libraries we take into account for evaluation.

ARM Mali-T604 is an embedded GPGPU integrated into the Samsung Exynos 5 Dual Multi-Processor System-on-Chip (MPSoC). The GPU is featuring four 128-bit SIMD units per core. In total, it has four cores available, resulting in 64 SIMD lanes for floating point operations, running at 533 MHz. Counting Fused Multiply-Add (FMA) as two operations, its theoretical peak performance caps at approximately 68.22 GFLOPs. Besides 2 GB of DDR3 RAM, which is shared among the CPUs and the GPU, it also contains 256 KB of L2 cache. It further supports utilizing texture memory from OpenCL restricted to the *RGBA* format, forcing the use of integer vectors containing four elements. Local on-chip memory is not available.

Nvidia Tesla K20 is a discrete GPGPU solely for the purpose of computing. It contains 2496 scalar compute cores, distributed among 13 multi-processors, each running at 705 MHz. This results in approximately 3520 GFLOPs, doubled as usual, because of FMA. Besides 5 GB of GDDR5 RAM, also includes L1 and L2 caches as well as up to 48 KB local on-chip memory.

Xilinx Zynq 7045 is a System on Chip (SoC), which tightly integrates an ARM Cortex-A9 dual core CPU and a Kintex FPGA, using an ARM AMBA interconnect. The included FPGA can be considered a mid-range device, offering 350K logic cells, comprised of 218,600 Lookup Tables (LUTs), 437,200 flip-flops, 2,180 Kb of on-chip memory, and 900 DSP slices. Furthermore, the device includes a Gen2 hardcore PCI Express block and up to 16 high-speed serial GTX transceivers, each capable of transmitting at 12.5 Gb/s.

Xilinx OpenCV is a Vivado HLS-specific video processing library, similar to the popular computer vision framework. Legacy code for OpenCV can be implemented by Vivado HLS, requiring only minor modifications. Although, the library is designed to be used exclusively with the AXI4 streaming interfaces, we have extended it to also support standard streaming interconnects. In its current state (Vivado 2014.1), only a subset of OpenCV functions have been implemented at different optimization levels. Therefore, the here presented performance applies only to this specific version of Vivado HLS and might be subject to improve in future releases.

[4] e. g., `Image` names are maintained, only given a prefix when converting to streams

Table 1: PPnR results for the Laplacian operator and Harris corner detector implementations comparing the here proposed approach to OpenCV (as of Vivado HLS 2014.1).

	HIPA^{cc}									OpenCV 2014.1								
	II	LAT	SLCE	LUT	FF	B	DSP	F[MHz]	P [mW]	II	LAT	SLCE	LUT	FF	B	DSP	F[MHz]	P [W]
LPHV 3x3	1	1050638	141	288	521	2	0	349.9	232	1	1092846	1659	3515	8772	12	75	258.3	260
LPD 3x3	1	1050641	226	398	1034	2	0	341.1	232	1	1092846	1717	3470	8768	12	75	247.2	258
LP 5x5	1	1052768	3917	4521	23795	4	200	220.1	247	1	1098218	2171	5316	9076	10	103	201.7	268
HC 3x3	1	1063233	9349	23331	31102	8	254	239.4	498	-	-	-	-	-	-	-	-	-
HC 3x3	2	2101442	6097	17129	21138	8	154	207.9	552	2	2349003	6782	20037	26259	12	84	224.1	428

7.2 Algorithms

For the evaluation, we consider three typical image processing algorithms: an edge detector based on the Laplacian operator, the Harris corner detector [4], and the computation of optical flow using the census transform [11]. Those algorithms are of high relevance for richer imaging applications, such as augmented reality or driver assistance systems in the automotive sector. They all embody a high degree of parallelism and are equally well suited for every of our considered target architectures. Although these algorithms are well known, their implementation details may differ significantly, thus we briefly clarify the algorithm specifics used for our evaluation.

Laplace The Laplacian (LP) filter is based on a local operator, as already described in Section 2. Depending on the mask variant used, either horizontal and vertical edges or both including diagonal edges can be detected. In our results, we denote the first variant as LPHV 3×3 and the second as LPD 3×3. Additionally, we evaluated a third variant LP 5×5, based on a larger window size, also detecting diagonal edges.

Harris Corner (HC) was first introduced by Harris and Stephens [4]. It consists of a complex image pipeline depicted in Figure 6. After building up the horizontal and vertical derivatives (dx, dy), the results are squared and multiplied (sx, sy, sxy), and processed by Gaussian blurs (gx, gy, gxy). The last step (hc) computes the determinant and trace, which is used to detect threshold exceedances.

Optical Flow (OF) issues a Gaussian blur and computes signatures for two input images using the census transform. Therefore, for each image two kernels need to be processed. A third kernel performs a block compare of these signatures using a 15×15 window in order to extract vectors describing the optical flow. Regarding continuing streams of images (e. g., videos), for GPU targets the first image's signatures can be reused. Hereby, it is only necessary to process the second image and to reperform the block compare, resulting in the execution of 3 kernels per iteration. Whereas on FPGAs, the signatures for both images always have to be computed, resulting in the execution of 5 kernels per iteration. For fair comparison, we were considering this fact when evaluating our throughput results.

For the evaluation, we have used an 8-bit integer data types and images of size 1024×1024. The algorithms we synthesized in Vivado HLS with *high effort* settings for scheduling and binding. Post-Place and Route (PPnR) resource requirements were obtained by implementing the generated Verilog code in Vivado 2014.1. Power requirements were assessed by performing a timing simulation on the *post-route* simulation netlists and evaluation of the switching activity in Vivado.

7.3 Comparison to OpenCV

To evaluate the proposed code generation techniques, we have generated code for the algorithms discussed in Section 7.2 in HIPA^{cc}. As of release 2014.1, Vivado HLS provides the Harris corner detector as an OpenCV implementation. Although no explicit implementation of the Laplacian operator is provided, the video processing library from Xilinx implements the `2D_Filter` class from OpenCV, which can be used for 2D convolution with predefined coefficients. The optical flow is omitted, since the Xilinx OpenCV library does not contain an implementation for it, yet. The evaluation results, including achieved minimum initiation interval (II) and latency (LAT) (total number of clock cycles), required slices (SLCE), lookup tables (LUT), flip-flops (FF), block RAMs (B), and DSP slices (DSP) are listed in Table 1. Moreover, the table also specifies the maximum achievable clock frequency (F[MHz]) and simulated power requirements (P[mW]).

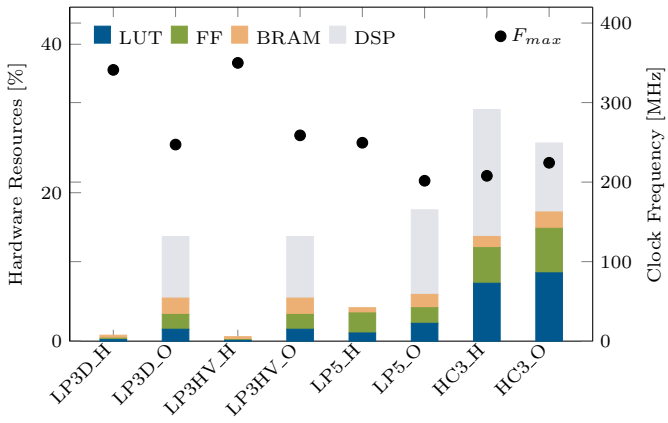

Figure 11: Amount of used FPGA resources by automatic synthesis. The suffixes H and O denote HIPA^{cc} and OpenCV, respectively.

Both OpenCV and HIPA^{cc} can deliver implementations for the Laplacian operator at Initiation Interval (II)=1. For the Harris corner detector, the synthesis of the OpenCV implementation reported loop-carried dependencies and can thus only achieve an II of 2. To be able to compare the results from HIPA^{cc}, we also enforced an II of 2 in a second evaluation. A comparison of resource requirements and clock frequencies is shown in Figure 11. As it can be seen, the highly optimized HIPA^{cc} implementations can achieve significantly higher clock frequencies, which also enables a maximum clock frequency of close to 350 MHz for the 3×3 Laplacian operators. The results of the Harris corner detector compare the II=2 implementations. Here, it can be seen that despite of achieving a lower clock frequency, the HIPA^{cc} solution uses more of the available DSP slices, which in turn causes less LUTs and FFs for the arithmetic operators. We have also compared the achievable throughput between HIPA^{cc} and OpenCV, which is shown in Figure 12. For all algorithms, HIPA^{cc} can generate solutions with a lower latency. In combination with a higher clock frequency and a single initiation interval, the throughput of the Harris corner detector can be considerably increased.

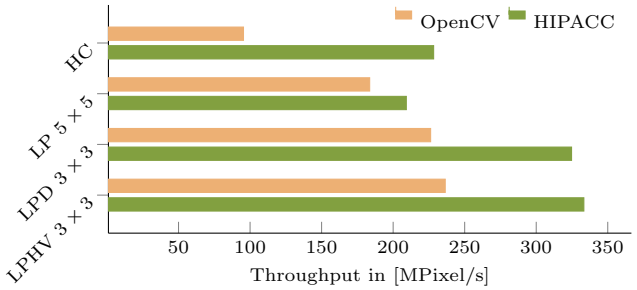

Figure 12: Throughput comparison between OpenCV and HIPA^{cc}.

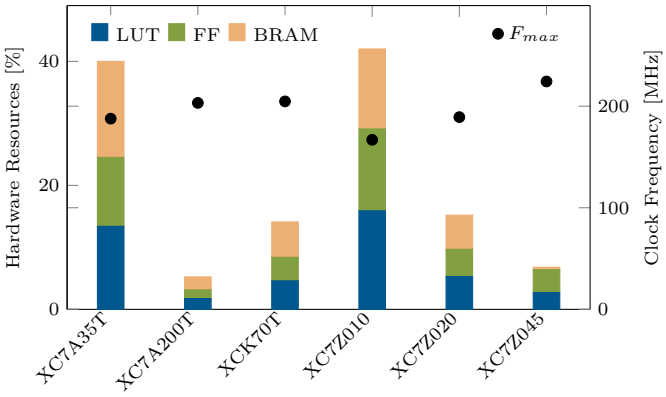

Figure 13: Amount of used resources for the optical flow on various FPGA types.

7.4 Evaluation of Different FPGA Types

Especially for close-to-sensor image preprocessing, it is important to generate efficient high-performance accelerators. To evaluate whether the solutions generated by HIPA^{cc} can also be implemented on low-end FPGAs and SoCs, we have synthesized the optical flow algorithm for various FPGA types of the 7 series family from Xilinx. The PPnR results are presented in Table 2. Figure 13 shows a graphical representation of the

Table 2: PPnR results for the optical flow implementation comparing different types of Xilinx FPGAs

FPGA	SLCE	LUT	FF	BRAM	F[MHz]	P [W]
XC7A35T	4417	11123	18540	31	187.7	0.155
XC7A200T	4351	15526	8997	31	203.2	0.213
XC7K70T	3521	12856	7677	31	204.7	0.247
XC7Z7010	4352	18607	11226	31	166.9	0.211
XC7Z7045	5669	11321	18587	31	204.7	0.470

implementation results. The Artix A35T, the Kintex K70T and the Zynq Z7010 are the smallest devices available for the three families. As the reconfigurable logic of the small Zynq is based on the Artix architecture, the resource requirements and the performance are roughly equal to that of the A35T. As well as the mid-range Zynq 7045, the K70T can easily handle the optical flow algorithm's resource requirements and achieve the requested clock performance of 200 MHz.

7.5 FPGA vs. Embedded GPU Implementation

One of the benefits of using a DSL for HLS is to have a target-independent high-level description at hand, which can

Table 3: Comparison of execution times on ARM Mali-T604, Xilinx Zynq 7045, and Nvidia Tesla K20. Time in *ms* for an image of size 1024×1024.

	Mali-T604	Zynq 7045	Tesla K20
LPHV 3×3	9.96	3.00	0.10
LPD 3×3	16.09	3.08	0.16
LP 5×5	45.47	4.78	0.38
HC	83.65	4.38	0.91
OF	2284.73	5.19	2.21

Table 4: Comparison of throughput and energy consumption for the ARM Mali-T604, Xilinx Zynq 7045, and Nvidia Tesla K20.

	Mali-T604		Zynq 7045		Tesla K20	
	TP [fps]	E [fpW]	TP [fps]	E [fpW]	TP [fps]	E [fpW]
LPHV 3x3	100.4	41.8	333.3	1423.1	10000.0	74.1
LPD 3x3	62.2	25.9	324.7	1387.6	6250.0	46.3
LP 5x5	22.0	9.2	209.2	846.9	2631.6	19.5
HC	11.9	4.9	228.3	458.5	1098.9	8.1
OF	0.4	0.2	192.5	409.6	452.5	3.4

generate code for several different hardware targets from the same code base. In this section, we will analyze the above presented algorithms for Mali-T604, Zynq 7045, and Tesla K20 in terms of performance and energy consumption. Performance results are summarized in Table 3. An efficient configuration for the GPU implementations was found using HIPA^{cc}'s exploration feature. Except for the optical flow, the Tesla K20 can exceed the performance of the embedded Mali GPU by a factor of approximately 100. The window size of 15×15 used in the optical flow leads to devastating results for Mali, presumingly because of missing L1 cache and on-chip memory. Execution times for both GPUs increase with the window size and kernel complexity, clearly decelerated by the number of memory loads. For the FPGA, the impact of larger window sizes is by far less noticeable. We further provide a comparison of the throughput in Figure 14 and energy consumption in Table 4. Power consumption of the GPUs can be estimated by considering about 60 % of the reported peak power values (2.28 W for the ARM Mali and about 135 W for the Nvidia Tesla). The FPGA provides the most energy efficient solution. Even though achievable frame rate per watt decimates notably faster than throughput, this effect is much more distinctive on GPUs, leaving FPGAs clearly as the architecture of choice in particular for more complex algorithms.

We attribute the lower energy efficiency of the Mali compared to Tesla to an inefficient code mapping by the OpenCL compiler. In fact, our implementation of the Laplacian filter uses four element wide vector types with a size of 32 bit. Since Mali contains 128-bit SIMD units, in theory a speedup of 4× at unchanged energy consumption can be expected for optimal occupation by implicit vectorization. Currently, HIPA^{cc} does not support implicit vectorization. The Harris corner execution times are negatively influenced by a similar issue. However, the performance results of Tesla GPUs are not affected by this problem and therefore outperform Mali in terms of energy efficiency.

8. RELATED WORK

High-level synthesis for reconfigurable logic has received a lot of attention over the past decades. Calypto's Catapult or the Impulse CoDeveloper C-to-FPGA Tools from Impulse Accelerated Technologies, as well as ROCCC [13] can be considered as general purpose frameworks for HLS and do not offer domain-specific knowledge for image processing. The smart buffer concept of ROCCC, is similar to our buffering

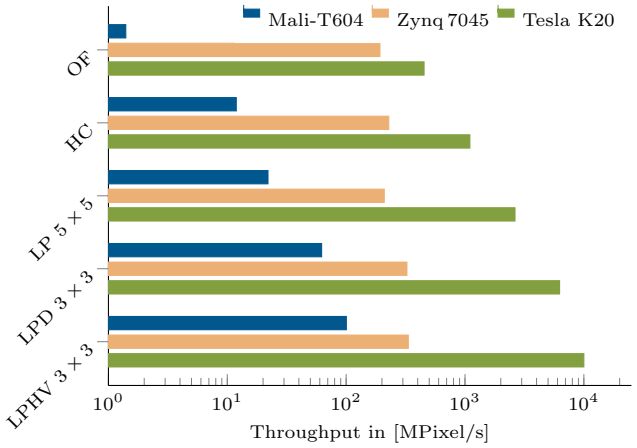

Figure 14: Comparison of throughput for the ARM Mali-T604, Xilinx Zynq 7045, and Nvidia Tesla K20.

approach, however, it does not support the construction of processing pipelines and also contains no further extensions, specific to image processing. For image and signal processing, it is often a challenge to bring together different areas of expertise, for instance algorithm design and hardware implementation. HLS frameworks sometimes include specific libraries to provide elemental architecture constructs and filtering implementations, as for example, the partial port of the OpenCV library [1] for Vivado HLS from Xilinx. MathWorks' HDL-coder and Synopsys' Synphony C Compiler are specific frameworks for generating hardware implementations from the Matlab/Simulink environment. Extending such libraries might become quite a burden and lowers portability. Furthermore, libraries are often restricted to a certain language and do not provide the same degree of freedom as a DSL. DSLs offer the advantage that domain and architecture knowledge are available for parallelization and the compilation flow. SPIRAL [8], for example, is a widely recognized framework for the generation of hard- and software implementations of digital signal processing algorithms (linear transformations, such as FIR filtering, FFT, and DCT) using a domain-specific language. Another considerable approach was proposed by George et al. [3] using *lightweight modular staging* and LegUp [2] as a back end for DSL-based generation of data paths. In the image processing domain, PARO [10] is a HLS environment that provides domain-specific augmentations, such as, border treatment and reductions (e. g., median filtering)—however, PARO generates only dedicated hardware accelerators but does not support multi- and many-core architectures. Another approach called Darkroom—was developed in parallel to ours—that can emit parallel code for multi-core systems as well as generate hardware pipelines was proposed by Hegarty et al. [5]. However, Darkroom does not offer advanced language constructs as for instance border treatment.

9. CONCLUSIONS

In this work, we have demonstrated how the DSL-based framework HIPA^{cc} can be extended to also include FPGAs as an additional hardware target by generating highly optimized code as design entry for the high-level synthesis framework Vivado HLS. In comparison to hand-coded C++ implementations for local operators in Vivado HLS, which might easily exceed 200 lines of code, specifying such an algorithm in the C++ embedded DSL only requires about a quarter, thereby performing numerous optimizations and eliminating coding errors. The proposed approach is evaluated in contrast to Xilinx OpenCV and can deliver significantly decreased resource usage and dramatically increased performance. As HIPA^{cc} also includes GPUs as hardware targets, we have compared the proposed FPGA approach to highly optimized GPU implementations, generated from the same code base. The assessment exposes the benefits of using a heterogeneous framework for algorithm development and can easily identify a suitable hardware target for efficient implementation.

Acknowledgment

This work is partly supported by the German Research Foundation (DFG), as part of the Research Training Group 1773 "Heterogeneous Image Systems". The Tesla K20 used for this research was donated by the Nvidia Corporation.

References

[1] G. Bradski. "The OpenCV library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. "LegUp: High-level synthesis for FPGA-based processor/accelerator systems". In: *Proc. of the 19th ACM/SIGDA Int. Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2011, pp. 33–36.

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

[4] C. Harris and M. Stephens. "A combined corner and edge detector". In: *Alvey Vision Conference*. 1988, pp. 147–151.

[5] 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". In: *Proc. of the 41st Int. Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. Aug. 10–14, 2014.

[6] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. "Generating device-specific GPU code for local operators in medical imaging". In: *Proc. of the 26th IEEE Int. Parallel & Distributed Processing Symposium (IPDPS)*. IEEE. May 21–25, 2012, pp. 569–581.

[7] R. Membarth, O. Reiche, F. Hannig, and J. Teich. "Code generation for embedded heterogeneous architectures on Android". In: *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*. Mar. 24–28, 2014.

[8] M. Püschel, F. Franchetti, and Y. Voronenko. "SPIRAL". In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Springer, 2011.

[9] M. Schmid, N. Apelt, F. Hannig, and J. Teich. "An image processing library for c-based high-level synthesis". In: *Proc. of the Int. Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2–4, 2014.

[10] M. Schmid, A. Tanase, F. Hannig, J. Teich, V. Bhadouria, and D. Ghoshal. "Domain-specific augmentations for high-level synthesis". In: *Proc. of the 25th IEEE Int. Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, June 18–20, 2014, pp. 173–177.

[11] F. Stein. "Efficient computation of optical flow using the census transform". In: *Pattern Recognition*. Vol. 3175. Lecture Notes in Computer Science. Springer, 2004, pp. 79–86.

[12] R. Tessier and W. Burleson. "Reconfigurable computing for digital signal processing: a survey". English. In: *Journal of VLSI signal processing systems for signal, image and video technology* 28.1-2 (2001), pp. 7–27.

[13] J. Villarreal, A. Park, W. Najjar, and R. Halstead. "Designing modular hardware accelerators in C with ROCCC 2.0". In: *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2010), pp. 127–134.

[14] V. Volkov. *Better performance at lower occupancy*. Presentation at the GPU Technology Conference (GTC). Sept. 2010. URL: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf (visited on 04/21/2014).

[15] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. "AutoPilot: A platform-based ESL synthesis system". In: *High-Level Synthesis*. Ed. by P. Coussy and A. Morawiec. Springer, 2008, pp. 99–112.