This is the author's version of the work. The definitive work is to appear *In Proceedings of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), Toronto, Canada, Jul. 27-29, 2015.*

Loop Coarsening in C-based High-Level Synthesis

Moritz Schmid, Oliver Reiche, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design, Department of Computer Science,
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
Email: {moritz.schmid, oliver.reiche, hannig, teich}@cs.fau.de

***Abstract*—Current tools for High-Level Synthesis (HLS) excel at exploiting Instruction-Level Parallelism (ILP), the support for Data-Level Parallelism (DLP), one of the key advantages of Field Programmable Gate Arrays (FPGAs), is in contrast very limited. This work examines the exploitation of DLP on FPGAs using code generation for C-based HLS of image filters and streaming pipelines, consisting of point and local operators. In addition to well known loop tiling techniques, we propose loop coarsening, which delivers superior performance and scalability. Loop tiling corresponds to splitting an image into separate regions, which are then processed in parallel by replicated accelerators. For data streaming, this also requires the generation of glue logic for the distribution of image data. Conversely, loop coarsening allows to process multiple pixels in parallel, whereby only the kernel operator is replicated within a single accelerator. We augment the FPGA back end of the heterogeneous Domain-Specific Language (DSL) framework HIPA^{cc} by loop coarsening and compare the resulting FPGA accelerators to highly optimized software implementations for Graphics Processing Units (GPUs), all generated from the exact same code base. Moreover, we demonstrate the advantages of code generation for algorithm development by outlining how design space exploration enabled by HIPA^{cc} can yield a more efficient implementation than hand-coded VHDL.**

I. Introduction

Co-processors for multimedia applications provide energy-efficient high-performance solutions for compute-intensive applications. Alongside Application Specific Integrated Circuits (ASICs) and software-based accelerators, such as embedded GPUs (eGPUs), FPGAs are becoming significant for server applications and are, moreover, being considered for mobile computing [1]. Despite all of the benefits of the platform, including deterministic throughput and latency, highly flexible support of interconnect technologies, and high potential for acceleration, the tedious programming paradigm on the Register Transfer Level (RTL) eludes many developers from targeting FPGAs. The situation is mitigated by HLS, which allows software and algorithm engineers to specify FPGA designs using high-level programming languages. Proposed approaches are manifold, but can be broadly classified into general purpose and specialized frameworks. Specialized frameworks offer high performance at limited user interaction for specific application areas. In contrast, general purpose frameworks can produce RTL architectures for a wide range of applications, but require user interaction for synthesis and in-depth knowledge of the hardware platform. Several solutions to this drawback have been proposed, for example, specific libraries provide code templates for recurring problems [2].

In previous work, we have suggested to use DSLs and generate highly optimized code for synthesis by general purpose HLS frameworks [3]. Code generation offers true portability of programs and performance across different platforms. Moreover, it delivers increased productivity, as developers need not be concerned about implementation details, but can focus on functionality. In combination with HLS frameworks, support of a wide range of applications can be accomplished efficiently.

Our work utilizes the open source DSL framework HIPA^{cc} to generate highly optimized C++ code for Vivado HLS from Xilinx. HIPA^{cc} is specifically targeted at accelerating image processing and can generate code for heterogeneous hardware targets, covering GPUs and FPGAs. So far, the generated FPGA implementations can (a) only process one pixel per clock cycle and (b) do not make use of DLP, supported by the high data rates on FPGAs.

A well established method to increase the throughput is *loop tiling*, which exploits DLP by splitting the image into separate regions, which are then processed in parallel by dedicated accelerators. An example is depicted in Figure 1.

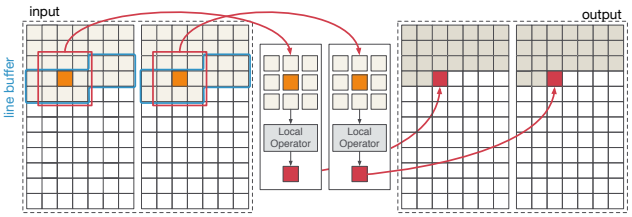


Figure 1. Loop tiling for local operators by splitting the input image into multiple parts and processing it by dedicated accelerators concurrently.

In this work, we propose an alternative methodology for local operators, we call *loop coarsening*. Input pixel data is aggregated into *superpixels*, which are then processed by the accelerator in parallel. In contrast to loop tiling, it uses a single, more complex control structure and only replicates the kernel operators. Therefore, it does not depend on additional modules for data distribution. The approach is illustrated in Figure 2 .

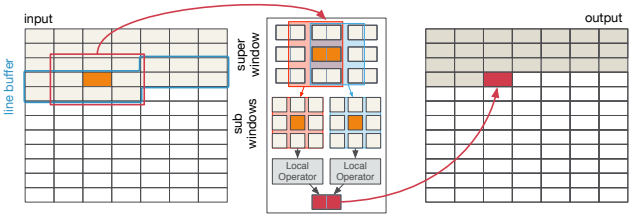


Figure 2. Loop coarsening for parallel processing of superpixels with local operators by replicating only the kernel operator.

Both approaches serve to increase the throughput, but current HLS tools do not offer any support to generate appropriate accelerators automatically and require manually created glue logic for processing local operators in the context of data streaming. Here, this work makes the following contributions:

- In addition to loop tiling, we propose the exploitation of DLP on FPGAs using a new technique for local operators in data streaming, called *loop coarsening*.
- We augment the FPGA back end of the HIPA^{cc} compiler with the ability to apply loop coarsening during code generation.
- We quantify how this new capability delivers increased performance and simplifies design space exploration.

II. Preliminaries

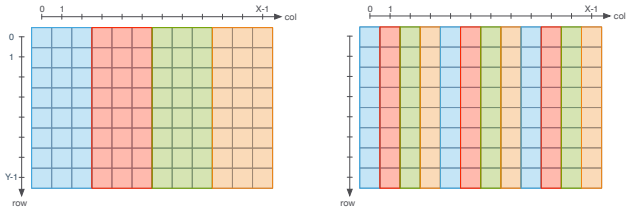
Image processing can generally be classified according to the computational complexity and data elements used in a three-level hierarchy [4], where algorithms at the lowest and intermediate level mostly involve pixel-based operations. In [5], Bailey introduces the classification into application-level algorithms, which describe the application of a sequence of image processing steps, and operation-level algorithms, which represent such individual steps. For our analysis, we consider image processing at the low and intermediate levels of the abstraction hierarchy. The algorithms at operation-level, specified in a language such as C or C++, use nested iterative loops to define when to read the pixels of the image, when to perform computations, and when to write the results of the computations. Loop computations are defined over a so-called *iteration space*, comprising a finite discrete Cartesian space with dimensionality equal to the loop nest level [6]. We restrict the analysis to *rectangular* iteration spaces, which is the case if the limits of the inner loops are static and do not depend on the values of outer loops. The semantics of the outer loop-level define the order in which pixels arrive in data streaming, also known as the *scan line*.

III. Parallelism for FPGAs

The parallelization of loop programs by exploiting *spatial parallelism* means the distribution and concurrent execution of loop iterations over different processing units. In principle, it is possible to implement and execute every step of any algorithm using a dedicated resource. An algorithm will, however, only benefit from such a solution if a substantial part of its computations can be executed in parallel. In contrast, if most of the computations are intended for sequential execution or have complex data dependencies, the effect of parallelization will only be marginal. This observation has first been formulated by Amdahl in [7], commonly known as Amdahl's law and often used to estimate the theoretical maximum speedup to be expected from parallelization. Fortunately, image processing algorithms at the low and intermediate levels possess a very high degree of parallelism that can be categorized into *temporal* and *spatial parallelism* [8].

On the operation level, we can make use of spatial parallelism by unrolling the innermost loops and balancing the expressions, thereby executing as many instructions in parallel as possible. Excessive use of hardware resources can be avoided on FPGAs, by using the concept of *data streaming*, which turns spatial parallelism into temporal parallelism. A common form to exploit this is by overlapping the execution of instructions for successive loop iterations. The methodology is called *pipelining* [9]. Allocating dedicated accelerators for each step of the imaging algorithm and interconnecting these in the form of a streaming pipeline additionally allows the exploitation of temporal parallelism contained in the algorithm at application level.

High-speed serial transceiver technology on FPGAs enables the communication interfaces to operate at high data rates and deliver multiple pixels per clock cycle in an aggregated bit vector, which we refer to as a *data beat*. In many applications, incoming data beats are separated into individual pixels outside of the accelerator by external logic. Using such an approach, the throughput of the algorithm implementation can only achieve a fraction of the data rate of the communication interface. A widely recognized method to increase the throughput exploits spatial parallelism by replicating the accelerator and DLP by letting each of the duplicates process pixels in parallel. The performance gain is thus limited only by the data rate of the Input/Output (I/O) protocol and the available resources on the FPGA. DLP can be exploited by either *block-wise* or *cyclic*



(a) Block-wise data distribution. (b) Cyclic data distribution.

Figure 3. Schemes for exploiting DLP.

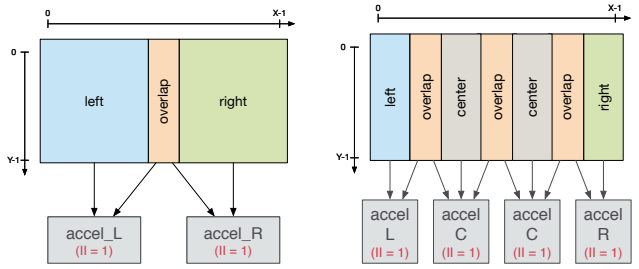
data distribution, as shown in Figure 3. The chosen method also determines the architecture of the accelerator to process it. The block-wise distribution scheme requires the replication of the whole accelerator, as well as additional components for distributing and reassembling data. The methodology is well established and is commonly referred to as *loop tiling*, see e. g., [6]. Cyclic distribution can in contrast be implemented by only replicating the kernel operator of the accelerator and processing consecutive loop iterations in parallel. Applying the technique to point operators is trivial. In the context of local operators for data streaming, however, it requires a complex control structure. We refer to this as *loop coarsening* and detail its implementation in Section V.

IV. Loop Tiling

Loop tiling is likely one of the most widely applied parallelization techniques to exploit spatial parallelism on the operation level. Similar to the well known divide and conquer methodology, the image is split orthogonal to the scan line into multiple parts, which are then processed by multiple dedicated accelerators in parallel. Every accelerator receives a continuous part of the image row in a round robin fashion. After all have received their part, the distribution restarts with the next row in the image.

A. Overlap Region for Local Operators

No data exchange is necessary for processing point operators, care must, however, be taken for local operators, since the kernel windows may overlap into the area outside of the image. At the actual borders of an image, this problem can be solved efficiently by border treatment. Applying this technique to the splitting border is not advisable, since it might leave visible traces in the output image. A better approach is to define *overlap regions*, which are assigned to both accelerators. A



(a) Splitting into two halves. (b) Splitting into four regions.

Figure 4. Loop tiling for local operators.

typical scenario for splitting an image into two symmetric halves, which are then processed by two accelerators is shown in Figure 4(a). The overlap between the neighboring image regions must be distributed to both accelerators to enable processing by local operators. Moreover, if the image is split into more than two regions, Figure 4(b) shows an example for four regions, accelerators operating on one of the central regions must be assigned the overlap to the left, as well as to the right.

B. Implementation Structure

The highest yield can be obtained in parallel processing if data is distributed in such a way that the involved accelerators can compute their individual image stripes without being interrupted. Therefore, the data rate at the input and output should ideally match the combined processing rate of the accelerators. For the implementation, every accelerator has a local input First In First Out (FIFO) buffer that can be filled with the higher I/O data rate, but is emptied only with the lower processing rate. To achieve this, we combine several pixels into data beats for input and output, where the amount of pixels per data beat is denoted as the *vector length* VL. If we assume p accelerator replications to concurrently process data at a rate of q pixels per clock cycle, the most reasonable choice is to set $VL = q \cdot p$. An architecture to achieve this using only HLS is shown in Figure 5.

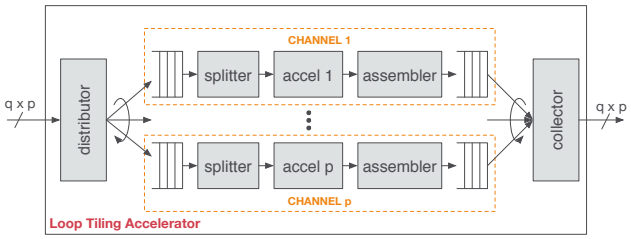
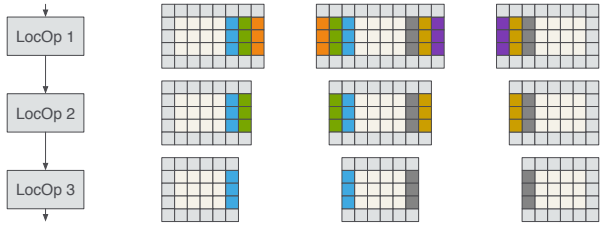


Figure 5. *Loop tiling* architecture with p accelerators.

The first stage of the implementation is a *distributor* that assigns incoming data beats to buffer queues. The distributor must also assign data from the overlap areas correctly. From the buffer queues, data is consumed by *splitters*, which extract the pixels from the data beats and forward them to the *accelerators*. Often, only a subset of the pixels aggregated in a data beat at the beginning or the end of the overlap belong to the actual iteration space of the accelerator. The accelerators, however, should only receive pixels that belong to their iteration space. Thus, the splitters must extract only those pixels that contribute to the computation and omit excess pixels. For the accelerators, we must provide different implementations for the far left, far right, and the center of the image. The reason for this are different iteration spaces and irregularities, e. g., where to read pixels and where to apply border treatment. After processing, the output of the accelerators is collected into data beats by the *assemblers* and gathered by a *collector* in the same order as supplied by the distributor.

C. Loop Tiling for Streaming Pipelines

Special considerations must be taken for applying loop tiling to streaming pipelines consisting of multiple local operators, as a certain part of the image region computed by one local operator becomes the new overlap for the next local operator. Including an explicit data interchange after every local operator would involve additional FIFO buffers and logic resources. An alternative approach for the implementation of such pipelines is to widen the overlap area, so that the preceding local operator in the pipeline already computes the data that is necessary for the overlap of the following local operators. Figure 6(a) shows



(a) Pipeline. (b) Left. (c) Center. (d) Right.

Figure 6. Illustration of the differently sized iteration spaces when applying loop tiling to streaming pipelines.

an example pipeline of three subsequent local operators. The increase in the iteration space for the left, center, and right accelerators is shown in Figures 6(b), 6(c), and 6(d), respectively.

V. Loop Coarsening

The previous section discussed how to raise the degree of parallelism in image processing using loop tiling. In this section we present *loop coarsening* as an alternative approach for parallel processing. The method aggregates input pixel data in so-called *superpixels*, where the amount of pixels contained in a superpixel is denoted by the superpixel length (SpL). In contrast to loop tiling, loop coarsening uses a cyclic distribution of the pixels contained in a superpixel and only replicates the loop kernel instead of the whole accelerator. Thus, the approach does not require any additional modules for special data distribution.

Although the methodology is similar vector to operations in Single Instruction Multiple Data (SIMD) architectures, there are distinct differences.

A. Differences to Vectorization

Vector operations are often a part of SIMD architectures and multimedia extensions. In contrast to scalar arithmetic operations, they carry out the operation on aggregated data types, called *vectors*. *Vectorization* of a point operator is similar to loop unrolling, where the kernel is not replicated but the scalar operations are replaced with vector operations. Vectorization of local operators is only possible if the elements of the vector data do not represent a continuous context. An example for this are Red Green Blue Alpha (RGBA) encoded color images, where each pixel is actually a combination of the color values. Together with the values from neighboring pixels, each value defines its own context, also referred to as a *color channel*. For such data types, local operators can be vectorized by replacing the scalar operations by vector operations. As the superpixels in our concept represent a continuous pixel space within the same context, parallelizing a local operator requires a much more complex methodology.

B. Superpixel and Superwindow Concept

As local operators process the pixels of a local neighborhood, an important aspect is to reduce repeated memory access. For data streaming of single pixels as data units, a *linebuffer* is used to retain data for repeated access and a register-based *memory window* provides the pixels for the local operator. An *Initiation Interval* (II) of one clock cycle for local operators can only be achieved, if the access to each of the Dual-Port-RAM (DPRAM)-based linebuffers is restricted to only two accesses per clock cycle. We apply the same concept for loop coarsening but introduce a hierarchy for the memory window, where a *superwindow* is used for gathering superpixels from the linebuffer and retain them for repeated access. From the superwindow, the pixels are extracted and assigned to *subwindows*, which are then used by the replicated kernel operators for processing (refer to Figure 2).

C. Data Separation and Border Treatment

A challenge for hardware generation using the proposed window hierarchy is to determine the static wiring for separating and assigning the elements of the superwindow to the appropriate locations of the subwindows. For different superpixel lengths SpL and kernel sizes, the size of the superwindow ($W_y \times W_x$) may not match that of the subwindows ($w_y \times w_x$). For example, using a window of 3 pixels (radius $r = 1$) always requires a superwindow of the same size, regardless of the vector length. However, processing the image data with a window size of 9×9 pixels (radius $r = 4$) with $SpL = 2$, only needs a superwindow of size 9×5. Separating the data from the superwindow into the subwindows involves determining the correspondence between the elements of the windows. Both can be represented by triplets. To describe an element of the superwindow, we denote the point $(i, j, k) \in S_y \times S_x \times S_{sp}$, where $S_y = \{0, \ldots, W_y - 1\}$ and $S_x = \{0, \ldots, W_x - 1\}$ describe the position of the data beat in the superwindow and $S_{sp} = \{0, \ldots, SpL - 1\}$ describes the position of a pixel in the superpixel. The corresponding point in a specific subwindow is represented by $(i, j, k) \in s_y \times s_x \times P$, where $s_y = \{0, \ldots, w_y - 1\}$ and $s_x = \{0, \ldots, w_x - 1\}$ describe the window coordinates, and $P = \{0, \ldots, p - 1\}$ specifies the distinct subwindow. For calculating the correspondence between the windows, we must moreover determine the offset o of the first window in the continuous context of the data beats as

$$o = \begin{cases} SpL - r & \text{if } SpL \geq r \\ r \bmod SpL & \text{else.} \end{cases}$$

For our implementation, we consider an element of a subwindow and seek the corresponding element in the superwindow using the mapping

$$\begin{aligned} f &: s_y \times s_x \times P \rightarrow S_y \times S_x \times S_{sp}, \text{ where} \\ f(i,j,k) &= (i,\ (j+o+k) \bmod SpL,\ \lfloor (j+o+k)/SpL \rfloor). \end{aligned}$$

An example for one line using $SpL = 4$ and $W_x = w_x = 3$ is shown in Figure 7. At the top and at the bottom of the image,

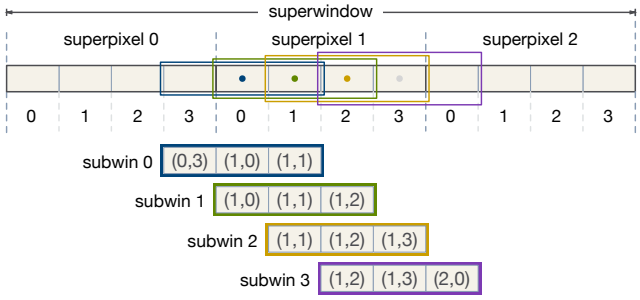


Figure 7. Example for separating the data from the superwindow into subwindows. Here, $SpL = 4$ and $W_x = w_x = 3$.

border treatment can be performed using the data beats. At the left and right border, however, we need to consider the individual pixels. Instead of reordering the superpixel before storing it in the line buffer, we combine border treatment with the data separation. As the experimental results will show (refer to Section VII), loop coarsening delivers far superior results considering performance, resource utilization, and scalability. We therefore only consider loop coarsening for implementation in HIPA^{cc}.

VI. Code Generation for Loop Coarsening

HIPA^{cc} consists of a DSL for image processing 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), Open Computing Language (OpenCL), Renderscript as used on Android, and highly optimized C++ code tailored to be further processed by Vivado HLS. The numerous challenges to overcome for targeting Vivado HLS have been dealt with in previous work [3]. However, applying loop coarsening by automatically replicating the loop kernel introduces additional demands to the implementation of the abovementioned concept of super- and subwindows.

A. Merging Subwindows

As large parts of the subwindows contain overlapping data, we merge those into a single *merged window* containing already

extracted pixel data. For describing points in the merged window, we denote $(i, j) \in m_y \times m_x$, where $m_y = \{0, \ldots, w_y - 1\}$ and $m_x = \{0, \ldots, w_x + p - 2\}$. To determine the correct values, including appropriate data separation and border treatment, we map points within the merged window into the coordinate space of the subwindows according to

$$\begin{aligned} g &: m_y \times m_x \to s_y \times s_x \times P, \text{ where} \\ g(i,j) &= (i,\ \delta \cdot j + (1-\delta)(w_x - 1), \\ &\quad (1-\delta)(j - w_x + 1)) \\ \delta &= \begin{cases} 1, & \lfloor j/w_x \rfloor = 0 \\ 0, & \lfloor j/w_x \rfloor \neq 0 \end{cases} \end{aligned}$$

Now, instead of carrying out the filter kernel on each single subwindow, we clip the corresponding region within the merged window for applying the kernel. Hereby, the creation of subwindows can be entirely skipped by applying $f(g(i, j))$, leaving only the merged window. Therefore, the memory footprint is reduced to $w_y(w_x + p - 1)$ for the merged window in comparison to all subwindows' sizes, which would be equal to $w_y \cdot w_x \cdot p$. Thus, the number of data redundancy is greatly reduced, which further affects resource usage in a positive manner.

B. Vectorization Data Types

DSL code describes image filters by using a data-parallel paradigm, similar to CUDA/OpenCL, where only the computation of a single output pixel is defined that will be applied to all pixels by iterating over the whole image. For loop coarsening, multiple of these iterations are issued simultaneously to process whole beats at once, instead of only single pixels. Therefore, handing over data between kernels requires specific techniques of code generation, which implements wider data types than the actual pixel data type used. Regarding code generation for Vivado HLS, the free choice of wider data types is accomplished by utilizing the Vivado-specific type `ap_uint<`*bit width*`>`, which can be employed to realize *explicit* and *implicit vectorization*.

1) Explicit Vectorization: This technique is only used if the elements of a vector represent a non-continuous context, such as images in RGBA color space, where computations and memory accesses should not be mixed across color channels. Such a behavior is accomplished by using explicit vector types of a certain length v_e in DSL code, as shown in Listing 1.

```
1 void kernel() {
2   int4 sum = reduce(dom, HipaccSUM, [&] () -> int4 {
3     return mask(dom) * convert_int4(input(dom));
4   });
5   sum = min(sum, 255);
6   sum = max(sum, 0);
7   output() = convert_uchar4(sum);
8 }
```

Listing 1. Kernel description for the convolution of the Laplacian operator for RGBA data using explicit vector types of length $v_e = 4$.

2) Implicit Vectorization: Implicit vectorization is automatically applied within code generation by solely defining the desired implicit vectorization factor v_i as a compiler switch. For this purpose, the superwindow concept (Section V-B) and the merged window approach (Section VI-A) have been adapted. However, special considerations need to be taken into account for mixing both types of vectorization.

3) Supporting Mixed Vectorization Types: Mixing both vectorization types enforces the necessary bit width for beats to be $v_i \cdot v_e \cdot bw$, where bw is the bit width of the data type of the explicit vector's elements. For instance, the implicit vectorization by factor 32 of the Laplacian operator (Listing 1), which is loading and storing pixels of type `uchar4`, would result in beats of type `ap_uint<1024>`. To ensure correctness, it is of utmost importance to handle both vectorization types separately during code generation. Implicit vector elements, in this case every 32 bits, are extracted on the outer loop level, handled according to the specified boundary treatment, and are filled into the merged window. On the other hand, explicit vector elements will be extracted on kernel level, as explicitly described by the programmer.

VII. Experimental Results

To assess the performance characteristics of the presented approach, we have evaluated several typical algorithms from image processing: A) The *Laplacian* (LP) filter is based on a local operator. 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 Laplace 3×3 HV (LP3HV) and the second as Laplace 3×3 D (LP3D). Additionally, we have evaluated a third variant, Laplace 5×5 (LP5), based on a larger window size. B) The *Harris corner* detector (HC3) consists of a complex image pipeline comprising local and point operators. After building up the horizontal and vertical derivatives, the results are squared, multiplied, and processed by Gaussian blurs. The last step computes the determinant and trace, which is used to detect threshold exceedences. C) The *Optical Flow* (OF) issues a Gaussian blur and computes signatures for two input images using the census transform. A third kernel performs a block compare of these signatures using a 15×15 window in order to extract vectors describing the optical flow. D) Finally, we have also evaluated a *Block Matching* (BM) algorithm from the area of stereo vision. Similar to the Optical Flow, correspondences are found using census transformed signatures, which are compared along an epipolar line of 60 pixels.

The FPGA results were obtained using the Xilinx Vivado Design Suite in version 2014.4 for a Xilinx Kintex 7 XC7K325TFFG900 FPGA. The code for the algorithm specifications was synthesized using Vivado HLS and Post Place and Route (PPnR) characteristics were obtained using the Vivado Design Suite. Power requirements were assessed by performing a timing simulation on the post-route simulation netlists and evaluation of the switching activity in Vivado. The evaluation results include achieved minimum initiation interval (II) and latency (LAT) (total number of clock cycles), required slices (SLICE), lookup tables (LUT), flip-flops (FF), 32Kb block RAMs (BRAM), and DSP slices (DSP). Moreover, the results also specify the maximum achievable clock frequency (F [MHz]), throughput (TP [fps]), and simulated power requirements (P [mW]).

A. Loop Tiling and Loop Coarsening

In the first assessment, we compare parallelizing the different versions of the Laplacian filter using loop tiling and loop

Table I
PPnR RESULTS FOR PARALLELIZATION OF THE LAPLACIAN OPERATOR USING LOOP TILING AND LOOP COARSENING.

	Laplace 3×3 HV (Loop Tiling)										Laplace 3×3 HV (Loop Coarsening)									
R	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU
1	1050634	180	376	668	2	0	374.5	356.5	1.0	1.0	1050634	180	376	668	2	0	374.5	356.5	1.0	1.0
2	528397	1023	2275	3020	4	2	302.1	571.8	1.6	2.0	525835	241	436	1003	2	0	374.4	712.0	2.0	2.0
4	272393	2193	5424	7520	20	6	279.1	1024.6	2.9	3.9	263435	378	747	1807	4	0	374.4	1421.2	4.0	4.0
8	149513	6036	14755	21199	72	14	253.0	1692.4	4.8	7.0	132235	739	1370	3422	8	0	358.6	2711.8	7.6	7.9
16	100361	25170	46106	66695	256	30	169.2	1686.3	4.7	10.5	66635	1381	2623	6658	15	0	340.5	5109.9	14.3	15.8
	Laplace 3×3 D (Loop Tiling)										Laplace 3×3 D (Loop Coarsening)									
R	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU
1	1050368	346	795	1337	2	0	371.9	354.1	1.0	1.0	1050368	346	795	1337	2	0	371.9	354.1	1.0	1.0
2	528389	1295	2973	4136	4	2	304.2	575.8	1.6	2.0	525836	415	868	1645	2	0	364.6	693.3	2.0	2.0
4	270393	2791	6622	9438	20	6	275.0	1016.9	2.9	3.9	263440	695	1704	3367	4	0	342.9	1301.8	3.7	4.0
8	149513	7170	17119	25091	76	14	263.6	1762.9	5.0	7.0	132240	1184	3092	5737	8	0	347.7	2629.3	7.4	7.9
16	100361	27049	50080	73007	256	30	185.2	1844.9	5.2	10.5	66640	2313	5724	10484	15	0	345.5	5185.2	14.6	15.8
	Laplace 5×5 (Loop Tiling)										Laplace 5×5 (Loop Coarsening)									
R	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU
1	1052691	993	2333	2015	2	0	354.4	336.6	1.0	1.0	1052691	993	2333	2015	4	0	354.4	336.6	1.0	1.0
2	529434	3291	7928	12858	12	2	290.7	549.1	1.6	2.0	526353	1630	3984	7538	4	0	343.6	652.9	1.9	2.0
4	270341	6077	16281	22711	24	6	269.4	996.5	2.9	3.9	263700	2791	6824	12194	8	0	342.6	1299.1	3.9	4.0
8	149513	12672	33540	46562	80	14	258.9	1731.4	5.1	7.0	132371	3851	10352	17605	16	0	325.3	2457.6	7.3	8.0
16	100361	34003	81727	112648	272	30	164.0	1633.7	4.9	10.5	66707	7055	19651	30473	30	0	315.9	4735.0	14.1	15.8

coarsening. We use RGBA color encoded input images of size 1024×1024, where each pixel requires 32 bits. The loop kernel is implemented using *explicit vectorization* with $v_e = 4$. The results of additionally parallelizing the implementations using loop tiling or loop coarsening are listed in Table I, where R indicates the *replication factor*. Figure 8 compares the necessary resources for parallelizing LP3HV using loop tiling (T) and loop coarsening (C). Also, the achievable throughput TP [kfps] is shown in the figure. Clearly, loop coarsening can achieve a

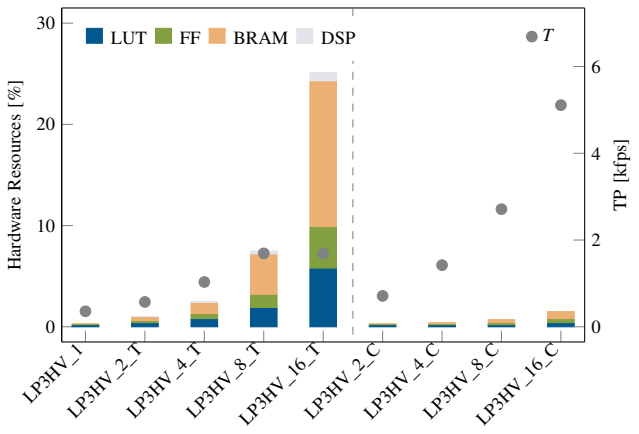


Figure 8. Comparison of PPnR characteristics for parallelizing LP3HV using loop tiling (T) and loop coarsening (C).

significantly higher throughput for each test case while using considerably less resources. As we replicate only the operator kernel, the compiler can optimize and eliminate redundant computations. Loop tiling, in contrast, must replicate entire accelerators and needs additional modules for data distribution. The very high resource utilization of loop tiling also did not allow a parallelization higher than 16 for tiling on the evaluated Kintex 7.

We have furthermore analyzed the speedup that can be achieved by both approaches. Table I lists the theoretical speedup (ThSU), which is defined as the quotient of the latency of the unparallelized version ($R = 1$) and the latency of the parallelized versions. The actual speedup (SU) is obtained using the throughput (TP), and thus reflects the influence of the clock frequency. Figure 9 shows a comparison of loop tiling (T) and loop coarsening (C). For reference, we have also depicted the optimal theoretical speedup (ThSU_optimal), which could be obtained if the amount of clock cycles would decrease with an increasing parallelization in a linear fashion. For a replication

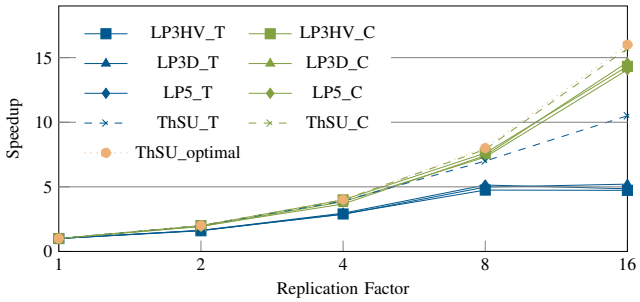


Figure 9. Comparison of the actual and theoretical speedup (ThSU) achieved for parallelizing the implementations of the Laplacian operator using loop tiling (T) and loop coarsening (C), respectively

factor of two and four, both methods can achieve near-optimal speedup and are very close together. Starting from eight, both methods diverge and coarsening stays very close to the optimal speedup. As the maximum clock frequency deteriorates with increasing resource utilization, the actual speedup is less than the theoretical speedup would suggest. The high difference between the optimal and the theoretical speedup in case of tiling can be explained if we consider that the overlap data beats also contain pixels that must not be submitted to the accelerator, which causes stall cycles to flush the pipeline.

B. Comparison of FPGA Results to Other Accelerators

As HIPA^{cc} can generate target-specific code for a wide range of accelerators from the same code base, we compare the results obtained for loop coarsening on the Kintex 7 FPGA to the Nvidia Tegra K1 SoC as a representative for eGPUs, and the Nvidia Tesla K20 server-grade GPU. Table II lists the PPnR results obtained for algorithm specifications generated by HIPA^{cc}, using the factor v_i for *implicit vectorization*. All of the algorithms use input images of size 1024×1024. Except for the variations of

the Laplacian filter, which use 32-bit RGBA encoded color input data, the algorithms receive 8-bit unsigned integer input data. Using HIPA^{cc}, we can generate CUDA code for implementation

Table II
PPnR CHARACTERISTICS OF ALGORITHM IMPLEMENTATIONS GENERATED BY HIPA^{cc} ON THE KINTEX 7.

Design	v_i	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	P [mW]
LP3HV	32	33835	2810	5122	13136	29	0	343.3	493
LP3D	32	33840	4159	10933	19996	29	0	330.6	792
LP5	16	66707	7055	19651	30473	30	0	315.9	890
HC	8	132268	28403	87916	104479	16	504	214.3	2067
OF	4	266112	30617	80480	100066	52	0	128.7	1219
BM	2	539728	5771	19222	23213	4	0	323.5	610

of the algorithms on the Tegra K1 and the Tesla K20 GPUs. Note that the implementations were obtained using code generation based on *exactly the same DSL source code*. To ensure a fair comparison, HIPA^{cc}'s exploration feature has been used to thoroughly evaluate suitable parameters for thread-coarsening, block size, and whether or not to use shared memory, in order to maximize the achievable throughput. Table III compares the throughput and energy-efficiency in frames per Watt (E [fpW]) of the different accelerators. Power consumption of the GPUs can be estimated by considering about 60 % of the reported peak power values (4.45 W for the Tegra K1[1] and about 135 W for the Tesla K20).

Table III
COMPARISON OF THE THROUGHPUT (TP) AND ENERGY CONSUMPTION (E) FOR THE TEGRA K1, TESLA K20, AND KINTEX 7.

	Tegra K1		Tesla K20		Kintex 7	
	TP [fps]	E [fpW]	TP [fps]	E [fpW]	TP [fps]	E [fpW]
LP3HV	52.4	11.8	10000.0	74.1	10146.3	20580.7
LP3D	44.0	9.9	6250.0	46.3	9769.5	12335.2
LP5	25.3	5.7	2634.6	19.5	4735.6	5320.9
HC	33.5	7.5	921.7	6.8	1620.2	783.8
OF	18.4	4.1	452.5	3.4	483.6	396.7
BM	5.6	1.3	84.8	0.6	599.4	982.6

The throughput and energy efficiency is moreover compared in Figures 10 and 11. For increasing algorithm complexity, the opportunity for parallelization of the FPGA accelerators is limited by the available logic resources. In case of the GPUs, the throughput is dramatically reduced in case of greater window sizes with an increased number of memory accesses, exposing the available memory bandwidth clearly as the main performance bottleneck.

C. Comparison between HIPA^{cc} and Hand-coded VHDL

Previous experiements, in which we have compared an FPGAimplementation for the Block Matching (BM) algorithm generated by HIPA^{cc} to a highly optimized hand-coded version in VHDL, have shown a significant advantage for design at the Register Transfer (RT) level [10]. Especially considering Flipflops (FFs), the VHDL version is decisively more resource-efficient. For this work, we have explored the design space by applying loop coarsening in addition to pipelining for the HIPA^{cc}

[1]Jetson DC power analysis of the CUDA smoke particle demo, adjusted for fan and system power consumption: http://wccftech.com/nvidia-tegra-k1-performance-power-consumption-revealed-xiaomi-mipad-ship-32bit-64bit-denver-powered-chips/

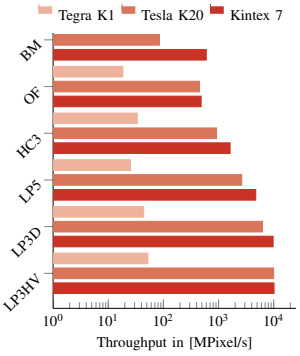


Figure 10. Throughput comparison.

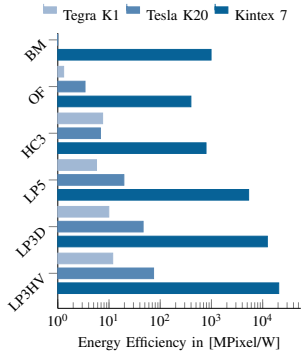


Figure 11. Energy efficiency.

solution. PPnR results for HIPA^{cc} and the hand-coded VHDL version are shown in Table IV. Evidently, loop coarsening does

Table IV
COMPARISON OF HAND-CODED AND HIPA^{cc}-GENERATED VERSIONS FOR BLOCK MATCHING WITH IMAGE SIZE 450×375.

Version	v_i	LAT	LUT (%)	FF (%)	BRAM	DSP	F [MHz]	TP [fps]
Hand-coded	1	170561	19247 (6.9)	9978 (1.8)	4	0	319.2	1871.5
HIPA^{cc}	2	90081	18867 (6.8)	23175 (4.2)	4	0	328.6	3647.8

not only increase the throughput (TP) of the HIPA^{cc} version, but in addition enables the HLS compiler to optimize the hardware implementation and considerably reduce the required resources. Undoubtedly, the same transformations can be applied to the hand-coded version. In HIPA^{cc}, however, loop coarsening can applied as a compiler switch, which does not require any changes to the code. The VHDL version would have to be altered manually, which, in turn, may take a lot of time and might introduce coding errors.

VIII. RELATED WORK

HLS for FPGAs has received a lot of attention over the past years, spurring successful general purpose frameworks from major Original Equipment Manufacturers (OEMs), for example, Catapult [11], and CyberWorkbench [12], but also from academia, such as ROCCC [13] and LegUp [14]. Using DSLs for hardware development is also a lively research area for several projects, such as SPIRAL [15], PARO [16] and Darkroom [17]. The concept of using DSLs to generate code for general purpose HLS tools, was also explored, for example by George in [18]. Several approaches target established parallel computing models and use source-to-source transformations to generate code for HLS frameworks. The synthesis of OpenCL kernels to parallel hardware implementations is, for example, covered in [19]. The main difference between such approaches and our methodology is that the FPGA implementations imitate the usual accelerators found in the programming model, such as GPUs or multi-core systems. We also employ some of the concepts of GPU parallelism, however, we use a computational model that closely resembles RTL implementations. Related to our work is [20], where the authors describe an approach of how to compile programs using OpenMP and Pthreads to C code that can serve as input to LegUp. The support of thread-level parallelism can be compared to loop tiling, however, our

approach also supports replicating only the kernel. On the RT level, our architecture for loop coarsening can be compared to Schmidt's VHDL templates for stencil computations [21]. Since our implementation is on a higher abstraction level, it also allows compiler optimizations across the replicated kernels, for example, to eliminate redundant computations, which is not possible if the kernels are distinct entities.

IX. Conclusion

In this work, we have presented how HLS implementations of image processing algorithms can be parallelized using loop tiling and loop coarsening. The comparison of the methodologies shows clear advantages of loop coarsening in terms of required hardware resources since loop tiling requires full accelerator replications, whereas loop coarsening only replicates the operator kernel. Loop coarsening also causes less stall cycles and therefore yields better accelerator performance. In consequence, we have augmented the FPGA back end of the DSL framework HIPA^{cc} with the ability to apply loop coarsening when generating code for Vivado HLS. As HIPA^{cc} also includes GPUs as hardware targets, we can moreover generate highly optimized GPU implementations from the exact same code base. The evaluation of several typical image processing algorithms demonstrates that the FPGA implementations are not only more energy-efficient but can also deliver a higher performance. In addition, we have compared a HIPA^{cc} implementation to highly optimized, hand-coded VHDL. Although HLS still has some deficiencies in contrast to hand-coded RTL implementations, it allows for rapid design space exploration through which we were able to achieve an improved ratio between resource usage and performance.

X. Acknowledgement

This work is partly 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] C. Wang, F.-L. Yuan, T.-H. Yu, and D. Markovic, "27.5 a multi-granularity FPGA with hierarchical interconnects for efficient and flexible mobile computing", in *Proc. of the IEEE Int. Solid-State Circuits Conference - Digest of Technical Papers*, (San Francisco, CA, USA), Feb. 9–13, 2014, pp. 460–461.

[2] Xilinx Inc. (2015). Vivado High-Level Synthesis, [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html (visited on 02/27/2015).

[3] 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 *Proc. of the Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, (New Dehli, India), Oct. 12–17, 2014.

[4] N. Ratha and A. Jain, "Computer vision algorithms on reconfigurable logic arrays", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 10, no. 1, pp. 29–43, Jan. 1999.

[5] D. Bailey, *Design for embedded image processing on FPGAs*. John Wiley & Sons, Inc., 2011.

[6] M. Wolfe, "More iteration space tiling", in *Proc. of the 1989 ACM/IEEE Conf. on Supercomputing*, (Reno, NV, USA), Jan. 1, 1989, pp. 655–664.

[7] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", in *Proc. of the Spring Joint Computer Conference (AFIPS)*, (Atlantic City, NJ, USA), Apr. 18, 1967, pp. 483–485.

[8] D. Hwang, S. Cho, Y. Kim, and S. Han, "Exploiting spatial and temporal parallelism in the multithreaded node architecture implemented on superscalar RISC processors", in *Proc. of the Int. Conf. on Parallel Processing (ICPP)*, (Syracuse, NY, USA), Aug. 16–20, 1993, pp. 51–54.

[9] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines", in *Proc. of the ACM SIGPLAN 1988 Conf. on Programming Language Design and Implementation (PLDI)*, (Atlanta, GA, USA), Jun. 1, 1988, pp. 318–328.

[10] O. Reiche, K. Häublein, M. Reichenbach, F. Hannig, J. Teich, and D. Fey, "Automatic optimization of hardware accelerators for image processing", in *Proc. of the DATE Friday Workshop on Heterogeneous Architectures and Design Methods for Embedded Image Systems (HIS)*, (Grenoble, France), Mar. 13, 2015, pp. 10–15. arXiv: 1502.07448 [cs.PL].

[11] Calypto Design. (2014). Catapult: Product Family Overview, [Online]. Available: http://calypto.com/en/products/catapult/overview (visited on 02/27/2015).

[12] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 19, no. 12, pp. 1507–1522, Dec. 2000.

[13] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0", in *Proc. of the Int. Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Charlotte, NC, USA), May 2–4, 2010, pp. 127–134.

[14] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems", in *Proc. of the Int. Symposium on Field Programmable Gate Arrays (FPGA)*, (Monterey, CA, USA), Feb. 27, 2011, pp. 33–36.

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

[16] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich, "PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications", in *Lecture Notes in Computer Science (LNCS)*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 287–293.

[17] 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. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, (Vancouver, Canada), Aug. 10–14, 2014, 144:1–144:11.

[18] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne, "Making domain-specific hardware synthesis tools cost-efficient", in *Proc. of the Int. Conf. on Field-Programmable Technology (FPT)*, (Kyoto, Japan), Dec. 9–11, 2013, pp. 120–127.

[19] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of platform architectures from OpenCL programs", in *Proc. of the Int. Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Salt Lake City, UT, USA), May 1–3, 2011, pp. 186–193.

[20] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs", in *Proc. of the Int. Conf. on Field-Programmable Technology (FPT)*, (Kyoto, Japan), Dec. 9–11, 2013, pp. 270–277.

[21] M. Schmidt, M. Reichenbach, and D. Fey, "A generic VHDL template for 2D stencil code applications on FPGAs", in *Proc. of the 15th IEEE Int. Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, (Shenzhen, Guangdong, China), Apr. 11, 2012, pp. 180–187.