Author manuscript: Journal of Signal Processing Systems (JSPS), 90(1), pp. 3–27, 2018.
The final publication is available at Springer via 10.1007/s11265-017-1229-7.

Loop Parallelization Techniques for FPGA Accelerator Synthesis

Oliver Reiche · M. Akif Özkan · Frank Hannig · Jürgen Teich · Moritz Schmid

Received: May 15, 2016 / Accepted: January 31, 2017

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. 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 processing multiple pixels in parallel, whereby only the kernel operator is replicated within a single accelerator. We present concrete implementations of tiling and coarsening for Vivado HLS and Altera OpenCL. Furthermore, we present a comparison of our implementations to the keyword-driven parallelization support provided by the Altera Offline Compiler. We augment the FPGA back end of the heterogeneous Domain-Specific Language (DSL) framework Hipacc to generate loop coarsening implementations for Vivado HLS and Altera OpenCL. Moreover, we compare the resulting FPGA accelerators to highly optimized software implementations for Graphics Processing Units (GPUs), all generated from exactly the same code base.

Oliver Reiche · M. Akif Özkan · Frank Hannig · Jürgen Teich
Hardware/Software Co-Design, Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
E-mail: {oliver.reiche, akif.oezkan, hannig, teich}@fau.de

Moritz Schmid
Siemens Healthcare GmbH, Germany
E-mail: moritz.schmid@siemens.com

Keywords Altera OpenCL · Vivado HLS · Vectorization · Loop coarsening · Loop tiling

1 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 [35]. Despite all of the benefits of the platform, including deterministic throughput and latency, highly flexible support of interconnect technologies, and a high potential for acceleration, the tedious programming effort at the Register Transfer Level (RTL) dissuades 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 with 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 [37].

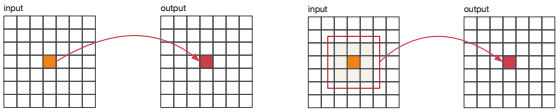
In previous work, we have suggested using DSLs and generating highly optimized code for synthesis through general purpose HLS frameworks [27]. Code generation offers true program and performance portability across different platforms. Moreover, it delivers increased pro-

ductivity, as developers do not need to be concerned about implementation details, but can focus on functionality. In combination with HLS frameworks, the support for a wide range of applications can be efficiently accomplished.

In our previous work [28], we investigated methods that exploit DLP for image processing algorithms. A well-established approach is *loop tiling* in which the input image of an algorithm is sliced into separate regions. These regions are then processed by dedicated accelerators in parallel. In addition to that, we proposed an alternative approach, called *loop coarsening*. Hereby, pixels of input images are aggregated into groups, 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 operator. Therefore, this approach does not depend on additional modules for data distribution. We showed that the throughput of image processing algorithms can be significantly increased via the discussed methods. On the other hand, manually parallelizing hardware designs of algorithms demands considerable development effort even if HLS is employed instead of a Hardware Description Language (HDL). Although a severe increase in throughput can be achieved, automatic parallelization is rarely supported in most current HLS tools. Yet, a few tools, mostly based on data-parallel programming languages such as OpenCL, provide support for automatic parallelization. However, results achieved with automatic approaches fall well short of expectations, compared to those achieved by manual parallelization.

In this work, we use the open source DSL framework Hipacc to leverage the algorithm specification to a higher level. We first analyze an algorithm, then exploit DLP through loop coarsening and finally generate highly efficient HLS code targeting both, Vivado HLS and Altera SDK for OpenCL (AOCL). Supporting both HLS tools enables us to investigate the loop parallelization techniques for devices of both main FPGA vendors, Xilinx and Altera. In the light of these, our work makes the following contributions:

- An abstract description of loop tiling and coarsening as well as a proposal for concrete implementations using Vivado HLS and AOCL.
- A comparison of loop tiling and coarsening, in terms of hardware utilization and achieved throughput, for both HLS tools with varying parallelization factors.
- A comparison between AOCL's automatic parallelization support, on a coarse- and fine-grained level, and the manually applied parallelization through loop tiling and coarsening.



(a) Point operator. (b) Local operator.

Fig. 1 Memory access patterns for point and local operators in image processing.

- A new compiler back end for Hipacc that supports AOCL and automatically applies loop coarsening during code generation.
- An evaluation of the throughput of Hipacc-generated accelerators for Vivado HLS and AOCL over an extensive application set. Furthermore, we compare those results to the throughput we were able to achieve with embedded and server-grade GPU implementations, stemming from exactly the same DSL source codes.

2 Preliminaries

Image processing can generally be classified in a three-level hierarchy [26] according to the computational complexity and data elements used, where algorithms at the lowest and intermediate level mostly involve pixel-based operations. In [4], 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 this 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 image's pixels, 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 [36]. We restrict the analysis to *rectangular* iteration spaces, consisting of the pixels within an image. For rectangular iteration spaces, 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*.

In this work, we focus on point and local operators. Point operators process only a single input pixel for every single output pixel they produce, as illustrated by Figure 1a. Typical applications for point operators are color space conversions. On the other hand, local operators process neighboring pixels within a static region, the local window, in order to produce a single output

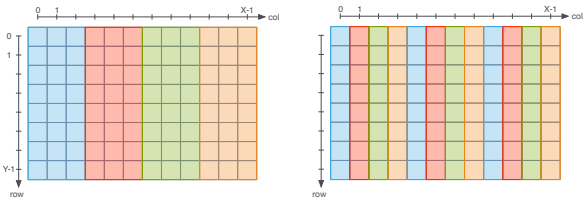
pixel. Pixels within that region are usually weighted by mask coefficients, such as those of the discrete Laplacian filter, before aggregating them to a single output pixel. The operation performed by local operators is depicted in Figure 1b.

3 Parallelism with FPGAs

Exploiting *parallelism* of loops in a program means the distribution and concurrent execution of loop iterations over different processing units. In principle, it is possible to implement any algorithm in dedicated hardware. 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 was first formulated by Amdahl in [3] 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* [12].

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 well-known methodology, called *pipelining* [13], exploits temporal locality to overlap the execution of instructions for successive loop iterations. Allocating dedicated accelerators for each step of an imaging algorithm and interconnecting them in the form of a streaming pipeline is a commonly practiced approach.

Moreover, 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 implementation of an algorithm can achieve only a fraction of the communication interface's data rate. A widely used method for increasing throughput by exploiting spatial parallelism, is replicating the accelerator, and thus increasing the number of pixels processed in each cycle, harnessing more DLP. As a result, the upper bound of the throughput is limited only by the data rate of the IO interface and the available resources on the FPGA.



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

Fig. 2 Schemes for exploiting DLP.

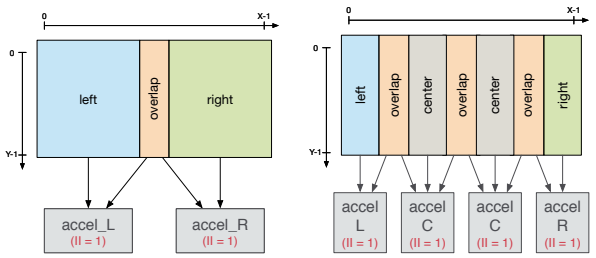
DLP can be exploited by either *block-wise* or *cyclic* data distribution, as shown in Figure 2. The chosen method also determines the architecture of the accelerator for processing purposes. The block-wise distribution scheme requires the replication of the whole accelerator, as well as additional components for distributing and reassembling data. The methodology is well established and is commonly referred to as *loop tiling*, see e. g., [36]. Cyclic distribution can, in contrast, be implemented by only replicating the kernel operator of the accelerator and processing consecutive loop iterations in parallel. Applying this 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 3.2.

3.1 Loop Tiling

Loop tiling is likely one of the most widely applied parallelization techniques for exploiting spatial parallelism on the operation level. Similar to the well-known divide-and-conquer methodology, the image is split into multiple parts perpendicular to the scan line, 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 accelerators have received their part, the distribution restarts with the next row of the image.

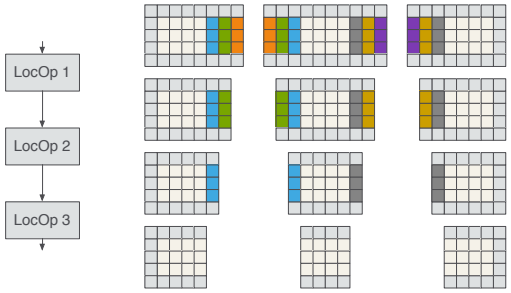
3.1.1 Overlap Region for Local Operators

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 with *border treatment*. Applying this technique to the splitting border is not advisable, since it can leave visible traces in the output image. A better approach is to define *overlap regions*, which are assigned to both accelerators. A typical sce-



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

Fig. 3 Loop tiling for local operators.



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

Fig. 4 Illustration of the differently sized iteration spaces when applying loop tiling to streaming pipelines.

nario for splitting an image into two symmetric halves, which are then processed by two accelerators, is shown in Figure 3a. 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 3b shows an example of four regions. Accelerators operating on one of the central regions must be assigned the overlap to the left as well as to the right.

3.1.2 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 First In First Out (FIFO) buffers and logic resources. An alternative approach of the implementation of such pipelines is to widen the overlap area, so that the preceding local operator in the pipeline computes the data that is necessary for the overlap of the following local operators in advance. Figure 4a shows an example pipeline of three subsequent local operators, all of size 3×3. The increase in the iteration space for the left, center, and right accelerators is shown in Figures 4b, 4c, and 4d, respectively.

The size of the overlap area depends on the operator position within the pipeline. The additional data for reading in each line for the i-th operator within a pipeline of n operators, each with a local window size of $w_{x_j} \times w_{y_j}$, is represented by

$$\text{overlap}(i) = \sum_{j=i}^{n} \left\lfloor \frac{w_{x_j}}{2} \right\rfloor \cdot c \qquad \forall\, i \in [1, n] \tag{1}$$

$$c = \begin{cases} 1, \text{ if left or right accelerator} \\ 2, \text{ else.} \end{cases}$$

The size of overlap area to write exactly corresponds to the overlap area to read by the successive $(i+1)$-th operator.

3.2 Loop Coarsening

The previous section discussed how to exploit the degree of parallelism in image processing using loop tiling. In this section we present *loop coarsening* as an alternative approach to parallel processing, similar to implicit vectorization. The method aggregates input pixel data in groups of a specific size, so-called *vectors*. By storing only an integral number of pixels in a vector, it is always ensured that whole pixels can be extracted and processed by multiple operator kernels in parallel. In contrast to loop tiling, loop coarsening uses a cyclic distribution of the pixels contained in such a vector and only replicates the loop kernel instead of the whole accelerator. Thus, this approach does not require any additional modules for special data distribution. Although the methodology is similar to explicit vectorization for Single Instruction Multiple Data (SIMD) architectures, there are distinct differences.

3.2.1 Differences to Explicit 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*. *Explicit 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. The vectorization of local operators is quite similar, as long as only a single pixel is stored in each vector. A common example for this is Red Green Blue Alpha (RGBA) data, where every pixel contains four different *color channel* values, which can be represented as a vector with four elements. In this

case, element positions are exactly the same in each vector. Hereby, it is ensured that vector operations do not mix values of the different color channels when accessing neighboring vectors and operating on them. However, if not exactly a single pixel is stored in each vector, vectorizing a local operator is not straightforward and requires a much more complex approach. This is due to the fact that the neighboring vectors do not contain pixel data at the correct element positions. Therefore, it might be necessary to extract elements from neighboring vectors and assemble them to new vectors in order to perform vector operations. For that reason, explicit vectorization is not performed automatically and has to be manually applied by the developer on operator level.

4 Vendor-Specific HLS Approaches

In this work, we focus on C-based HLS. Among the biggest two FPGA vendors, two HLS approaches have been particularly popular: *Xilinx Vivado HLS*, a C++-based HLS tool, and the *AOCL*. Xilinx itself also provides an Open Computing Language (OpenCL)-based tool called *SDAccel*. However, due to the fact that we already consider C++ for Vivado HLS, which is the subsequent code representation internally used by SDAccel, we do not take Xilinx' OpenCL-based HLS into account.

4.1 Xilinx Vivado HLS

Vivado HLS originates from the high-level synthesis tool AutoESL (formerly known as AutoPilot [38]) and has become an integral part of the Vivado Design Suite. Employing Vivado HLS, C/C++ or SystemC codes can be used for design entry and transformed into HDL code (VHDL, Verilog, SystemC), resulting in synthesizable IP cores. Vivado HLS itself is entirely integrated into Xilinx' Integrated Development Environments (IDEs), which explicitly target Xilinx FPGAs. In order to guide the transformation of a sequential algorithm's behavioral description into a structural hardware implementation, a large variety of synthesis directives can be applied. Setting these correctly enables the generation of efficient hardware implementations that are well-parallelized and optimized. Moreover, streaming pipelines can be implemented by employing data steaming objects, a concept inspired by SystemC, to interconnect different hardware modules. The HDL code generated by Vivado HLS is still human-readable and, therefore, subject to modifications that can be applied manually. This might be particularly useful for optimizing simulation and synthesis. Even though very complex and sophisticated algorithms can be specified with less effort by utilizing a C-based language, the efficient implementation of an image processing system still demands profound knowledge of an architecture expert in order to obtain results comparable to those of hand-coded implementations.

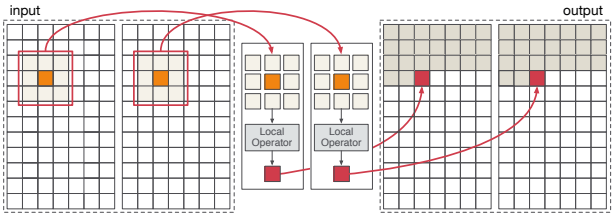


Fig. 5 Automatic instantiation of multiple compute units by specifying `num_compute_units(`n`)` for OpenCL NDRange kernels.

4.2 Altera SDK for OpenCL

The Altera SDK for OpenCL (AOCL) promises three fundamental advantages over conventional hardware design: (a) Simple structuring of Single Program Multiple Data (SPMD) applications; (b) portability to other architectures; and (c) a high level of abstraction for describing hardware.

The first claim can be confirmed without hesitation, since OpenCL follows a data-parallel programming paradigm. Thereby, source codes, so-called kernels, usually only describe data-independent computations that are processed by a single thread. The runtime system automatically addresses parallelizing and executing a single kernel representation across multiple independent data elements. As OpenCL is a standardized language, the second claim can generally be considered correct. However, one has to differentiate between functional portability, where solely functional correctness is ensured, and performance portability, where the efficient mapping to entirely different architectures is guaranteed as well. AOCL provides the former rather than the latter and requires developers to choose between an SPMD implementation, which is more suitable for many-core architectures, and a pipelined implementation, which is very different from GPU-like implementations. Regarding the third claim, AOCL does not only provide a C-based language for HLS but also goes far beyond standalone hardware design by strictly distinguishing between the host, the system's CPU, and the device, the accelerator implemented in the FPGA fabric, governing a complete heterogeneous system.

To emphasize this distinction, the Altera Offline Compiler (AOC) enforces the mapping of OpenCL's abstract memory model to FPGAs as follows: **global** off-chip DRAM, either shared with the host's CPU or

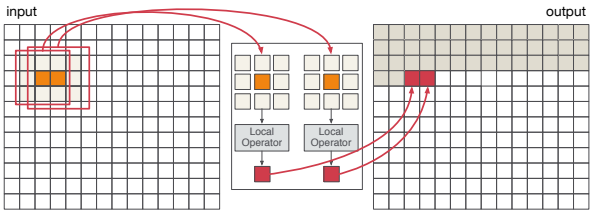


Fig. 6 Automatic consolidation of work-items within a work-group into multiple SIMD vector lanes by specifying `num_simd_work_items(`n`)` for OpenCL NDRange kernels.

dedicated memory for the FPGA but still accessible from the host side; **local** on-chip memory, not accessible from the CPU; **constant** read-only, on-chip memory with optimized cache hit performance; and **private** registers or on-chip memory, depending on compiler decisions. Aside from keywords for assigning variables to specific memory types, AOCL offers the *unroll* pragma to further guide synthesis. Thereby, for-loops, without loop-carried data dependencies, can be unrolled either completely or by a specific factor.

Furthermore, to implement a streaming pipeline, AOCL provides an extension, *channels*, not only for passing data between concurrent kernels but also synchronizing them. Channels are basically FIFO buffers with an additional feature for stalling the kernel that attempts to read and write data when empty and full, respectively. The data written to a channel is preserved as long as its program remains loaded on the device. The concurrent execution of different kernels communicating by channels is facilitated by separated command queues on the host side.

4.2.1 NDRange Kernels

As previously mentioned, OpenCL's runtime system manages the creation of threads and the parallel execution of kernel code across multiple independent data elements. How many threads are spawned depends on the range (1D, 2D, or 3D), i.e., the iteration space specified by the developer. Such kernels that are supposed to perform across a specified range are called *NDRange kernels*.

For NDRange kernels, Altera provides keywords to increase overall throughput by applying automatic replication very similar to loop tiling and loop coarsening. The first keyword controls the replication of the entire accelerator by instantiating multiple compute units for a kernel, as shown in Figure 5. The second keyword only replicates the innermost kernel computation by consolidating the work-items within a work-group into multiple SIMD vector lanes, as depicted in Figure 6. Furthermore, a combination of both techniques can also be applied.

NDRange kernels are highly portable, as they are written in a style very similar to GPU programming, and therefore are likely to perform rather well on many-core architectures. However, for synthesis, Altera recommends employing *single work-item kernels* in combination with line buffering instead of NDRange kernels.

4.2.2 Single Work-Item Kernels

Single work-item kernels are not automatically executed across a specified range by the OpenCL runtime. Instead, the developer must explicitly describe the range and the order, in which the range is processed, in the kernel's source code. Consequently, the number of threads that are spawned by the OpenCL runtime is exactly one. Unfortunately, performance portability to massively parallel many-core architectures, such as GPUs, is utterly impossible. However, single work-item kernels open up the possibility for further FPGA-specific optimizations, as data locality can be exploited.

Line buffering is a well-known design methods that is particularly beneficial for local operators in image processing, such as convolution, consisting of local operators. A local operator of size $w_x \times w_y$ might require the pixel at $(x - w_x/2, y - w_y/2)$ to calculate the output for (x, y). However, images are mostly captured and stored along a scan line. In addition, reading data from DRAM memory is far faster for consecutive access. In order to process one pixel for each read without compromising memory access speed, the minimum number of pixels satisfying the dependencies of the first pixel is stored in local memory through $w_y - 1$ FIFO buffers for reuse by later iterations. A $w_x \times w_y$ sliding window traverses over the image along the scan line by reading one pixel from the off-chip memory, while $w_y - 1$ pixels are loaded from row buffers.

Utilizing a single work-item kernel in combination with line buffering may yield more efficient hardware implementations than NDRange kernels. However, the automatic replication of entire accelerators or innermost kernel computations through Altera-specific keywords is only supported for NDRanage kernels.

5 Implementation Structure of Loop Tiling and Coarsening

We investigated the implementation of loop tiling and loop coarsening through a standalone hardware design via Vivado HLS and a heterogeneous system design based on the well-known OpenCL standard with AOCL.

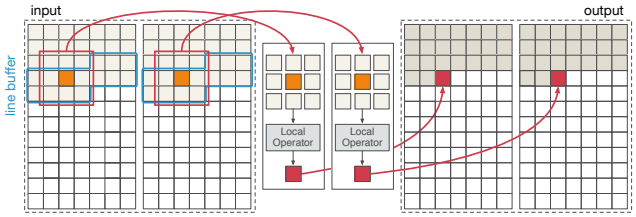


Fig. 7 Loop tiling for local operators by splitting the input image into multiple parts and concurrently processing it with dedicated accelerators.

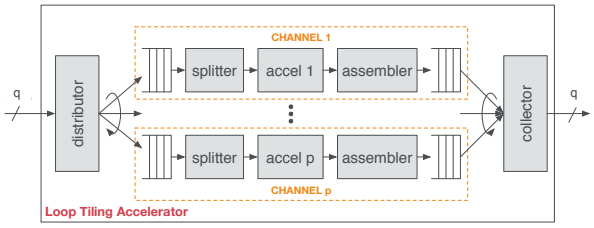


Fig. 8 *Loop tiling* architecture for Vivado HLS with p accelerators.

Due to interfacing variations, implementation differences for loop tiling are observed.

5.1 Loop Tiling

In order to exploit data parallelism via loop tiling, an algorithm's iteration space must first be split into subspaces, and then, reserved to the replicates of the hardware accelerator, as illustrated with a local operator in Figure 7. Tiling through replication of an existing accelerator requires minor modifications in the data paths of replicates and extra logic for the slicing iteration space to evenly sized parts. Implementation variations of the replicates come from irregularities and the corresponding border treatment of the iteration subspaces, which are different for the far left, far right, and the center of the image.

The iteration space should be sliced perpendicular to the scan line for the local operator implementations so that the size of the line buffers for each replicate is also divided by the same ratio. This causes the memory reserved for line buffers in total to remain the same, if the augmentation for boundary handling is neglected. Horizontal slicing demands more complex addressing than loop coarsening. Therefore, the implementation of additional logic highly depends on the system level architecture, e. g., whether the data feed is read from memory or streaming via high speed transceivers.

5.1.1 Loop Tiling for Vivado HLS

An architecture for the implementation of loop tiling by replicating an accelerator is shown in Figure 8. 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 strips without being interrupted. Therefore, the data rate at the input and output should ideally match the combined processing rate of the accelerators. The presented architecture, on the other hand, handles the difference in data rate between loading the input and reading the output, which are assumed to be streams. Every accelerator has a local input FIFO buffer that can be filled with the higher IO data rate, but is emptied only with the lower processing rate. To achieve this, several pixels are combined into data beats for input and output, where the amount of pixels per data beat is denoted as the *vector length* q. If we assume p accelerator replications to concurrently process data at a rate of q pixels per clock cycle, to avoid stalls q must be $\geq p$ but ideally q and p should match.

Algorithm 1: Loop tiling for Vivado HLS

```
   input  : dataIn, q, p, w, h, w_x, w_y
   output : dataOut
1  overlap ← ⌊w_x/2⌋
2  for y ← 0 to h − 1 do                                  // distributor
3      for x ← 0 to w/q − 1 do
4          temp ← Read(dataIn)
5          for t ← 0 to p − 1 do
6              left ← t · w/qp − ⌈overlap/q⌉ + 1
7              right ← (t + 1) · w/qp + ⌈overlap/q⌉
8              if (x > left) ∧ (x < right) then
                   Write(fifoIn[t],temp)
9          end
10     end
11 end
12 for t ← 0 to p − 1 do                                  // channels
13     Split(accelIn[t], fifoIn[t], t, q, w/qp, h, w_x)
14     Accel(accelOut[t], accelIn[t], t, p, w/p, h, w_x, w_y)
15     Assemble(fifoOut[t], accelOut[t], t, q, w/p, h, w_x)
16 end
17 for y ← 0 to h − 1 do                                  // collector
18     for x ← 0 to w/q − 1 do
19         t ← ⌊xqp/w⌋
20         Write(dataOut,fifoOut[t])
21     end
22 end
```

The pseudocode in Algorithm 1 represents our loop tiling implementation for Vivado HLS. Inputs are the input FIFO queue dataIn, the parameters q and p, the image size $w \times h$, the and operator window size $w_x \times w_y$. For simplicity it is assumed that only a single local operator is defined and not multiple of them, which would require a stacked overlap as described in Eq. (1). The

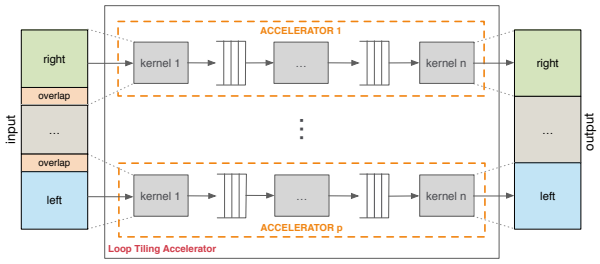


Fig. 9 *Loop tiling* architecture for single work-item OpenCL kernels with p accelerators.

output dataOut is a FIFO queue as well. The first stage of the presented implementation is a *distributor* that assigns incoming data beats to buffer queues (lines 2–11). Therefore, the distributor might also need to assign the data from the overlap areas to multiple tiles. Due to constants known beforehand, the loop in line 5 can be unrolled and its complexity can be significantly reduced. In practice, a single data beat at the overlap area usually needs to be assigned to a maximum of only two different tiles. Within each *channel* (lines 12–16), first the *splitter* consumes a data beat from fifoIn, splits it into up-to q data elements and feeds these sequentially into the accelIn queue. 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. After processing, the output of the accelerators is collected into data beats by the *assembler*, which basically performs the reverse operation of the splitter. The last step (lines 17–22) is gathering the assembled data beats by a *collector* in the same order as supplied by the distributor. Note that all line buffer structures are separately handled for each tile, and therefore are covered within the `Accel` call.

5.1.2 Loop Tiling for AOCL Single Work-Item Kernels

The proposed architecture to implement loop tiling for line-buffered single work-item kernels is given in Figure 9. An accelerator consists of multiple kernels, one for each operator, coupled with channels, in a similar fashion to streams of Vivado HLS. Likewise, the replication of the accelerators alone resembles the presented architecture for Vivado HLS. Boundary handling varies for different replicates and causes minor changes in the data path. Whereas, the far left and right accelerators utilize the necessary boundary handling only for the left and right edges, respectively. Center slices apply no horizontal

Algorithm 2: Loop tiling for Altera OpenCL

```
   input  : bufIn, p, w, h, w_x, w_y
   output : bufOut
1  overlap ← ⌊w_x/2⌋
2  for t ← 0 to p − 1 do                                  // accelerators
3      if t = 0 then                                              // left
4          offset ← 0
5          stride ← w − (w/p + overlap)
6      else if t = p − 1 then                                    // right
7          offset ← t · w/p − overlap
8          stride ← w − (w/p + overlap)
9      else                                                     // center
10         offset ← t · w/p − overlap
11         stride ← w − (w/p + 2 · overlap)
12     end
13     Accel(bufOut, bufIn, offset, stride, t, p, w, h, w_x, w_y)
14 end
```

boundary handling and instead have larger iteration spaces, being augmented with overlap areas.

In AOCL, kernels can be dynamically activated or stalled and their input can be changed from the host. This is achieved by utilizing interfacing logic according to the OpenCL standard, which introduces a noticeable overhead. Therefore, minimizing the number of kernels is crucial for optimizing the circuit. Thus, as indicated by Algorithm 2, the use of data beats, the *distributor*, and the *collector* were removed in the OpenCL implementation. In contrast, the corresponding strips of the input/output buffers are directly accessed from the first and last kernel within each accelerator, which allows us to eliminate redundant memory allocation for overlapping data. Therefore, the region to access needs to be determined (lines 3–12) before initiating each accelerator. Kernels within an accelerator that are not directly accessing input/output buffers solely communicate through streaming buffers, similar to the Vivado implementation.

5.2 Loop Coarsening

An alternative approach to exploit parallelism on FPGAs only replicates the kernel instead of complete accelerators. An illustration of this concept, which we call loop coarsening, is given in Figure 10 for a local operator. Loop coarsening binds local operators of an algorithm to one data path that executes on the same sliding window, which helps hardware synthesis tool heuristics to better exploit the locality in computations. In contrast to loop tiling, it does not require extra logic for data distribution and allows us to maximize resource sharing while not requiring any boundary handling overhead. In addition, loop coarsening might lead to more opportunities for the compiler to apply low level code optimizations and may

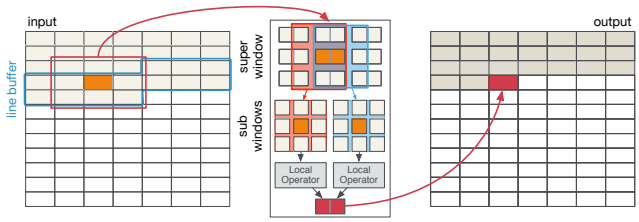


Fig. 10 Loop coarsening for parallel processing of superpixels with local operators by replicating only the kernel operator.

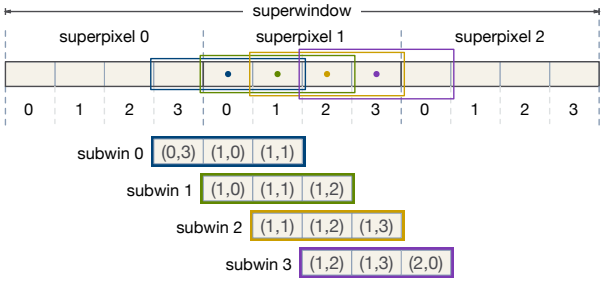


Fig. 11 Example for separating the data from the superwindow into subwindows. Here, $p = 4$ and $W_x = w_x = 3$.

thus reduce the number of arithmetic operations in a kernel. Moreover, the implementation of loop coarsening is identical for both, Vivado HLS and AOCL.

5.2.1 Superpixel and Superwindow Concept

As local operators process the pixels of a local neighborhood, an important aspect is to reduce recurring memory access. When data streaming single pixels as data units, a *line buffer* is used to retain data for recurring 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 line buffers is restricted to two accesses per clock cycle alone. 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 line buffer and stores them for recurring access. From the superwindow, the pixels are extracted and assigned to *subwindows*, which are then used by the replicated kernel operators for processing (see Figure 10). Note that the proposed superwindow implementation still is a shift register that shifts superpixels in each cycle. On the other hand, the wiring that is necessary to assign a superwindow to the inputs of subwindows changes depending on the radius of the local operator and the size of the superpixels. A formal solution to this issue is given in Section 5.2.2.

5.2.2 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 p and kernel sizes, the size of the superwindow ($W_x \times W_y$) may not match that of the subwindows ($w_x \times w_y$). For example, using a window of 3×3 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 $p = 2$, only requires a superwindow of size 5×9. 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 $S_{sp} = \{0, \ldots, p-1\}$ 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 data beats as

$$o = p - (r \bmod p)$$

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

$$f : s_y \times s_x \times P \to S_y \times S_x \times S_{sp}, \text{ where}$$
$$f(i, j, k) = (i,\ (j + o + k) \bmod p,\ \lfloor (j + o + k)/p \rfloor).$$

An example of one line using $p = 4$ and $W_x = w_x = 3$ is shown in Figure 11. At the top and bottom of the image, 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 the border treatment with the data separation.

Our implementation of loop coarsening is represented by the pseudocode in Algorithm 3. In general, superpixels are read from the input queue exactly $w/p \times h$ times (lines 3–4). Afterwards, the superwindow and line

Algorithm 3: Loop coarsening for Vivado HLS and AOCL

```
   input  : dataIn, p, w, h, W_x, W_y, w_x, w_y
   output : dataOut
1  for y ← 0 to h + ⌊W_y/2⌋ − 1 do
2      for x ← 0 to w/p + ⌊W_x/2⌋ − 1 do
           // read new superpixel
3          if (x < w/p) ∧ (y < h) then
4              supPxl ← Read(dataIn)
5          end
6          if (x < w/p) ∧ (y < h + ⌊W_y/2⌋) then
               // shift superwindow
7              for i ← 0 to W_y − 1 do
8                  for j ← 0 to W_x − 2 do
9                      supWnd[i, j] ← supWnd[i, j + 1]
10                 end
11             end
               // shift line buffers
12             for i ← 0 to W_y − 2 do
13                 supWnd[i, W_x − 1] ← lineBuf[i, x]
14                 if i > 0 then
15                     lineBuf[i − 1, x] ← lineBuf[i, x]
16                 end
17             end
18             lineBuf[W_y − 2, x] ← supPxl
19             supWnd[W_y − 1, W_x − 1] ← supPxl
20         end
21         if (x ≥ ⌊W_x/2⌋) ∧ (y ≥ ⌊W_y/2⌋) then
22             for k ← 0 to p − 1 do
                   // data separation to subwindows
23                 for i ← 0 to w_y − 1 do
24                     for j ← 0 to w_x − 1 do
25                         subWnd[i, j] ←
                               Split(supWnd, f(i, j, k))
26                     end
27                 end
                   // accelerators and memory packing
28                 subPxl ← Accel(subWnd)
29                 Assemble(supPxl, subPxl, k)
30             end
               // write the output superpixel
31             Write(dataOut, supPxl)
32         end
33     end
34 end
```

buffer are maintained accordingly (lines 6–20). After an initial latency the line buffer is entirely filled and the actual processing can be performed (lines 21–32). Here, the subpixels from the superwindow are selected and assigned to subwindows by applying function f. The accelerators can operate on every subwindow in parallel and their results are assembled to a new superpixel before writing it to the output queue.

Due to constants r, p known before hand, the data separation function f has a fixed mapping, meaning that data separation and memory packing can be implemented by wiring alone (lines 23–29).

As the experimental results show (see Section 7), loop coarsening delivers far superior results considering performance, resource utilization, and scalability. We therefore only consider loop coarsening for implementation in Hipacc.

6 Code Generation for Loop Coarsening

Hipacc [17] 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), OpenCL, Renderscript [16] as used on Android, and highly optimized C++ code tailored to be further processed by Vivado HLS and Altera OpenCL. The numerous challenges to overcome for targeting HLSs have been dealt with in previous work [27,21]. However, applying loop coarsening by automatically replicating the loop kernel introduces additional demands to the implementation of the aforementioned concept of super- and subwindows.

6.1 Merging Subwindows

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

$$g : m_y \times m_x \to s_y \times s_x \times P, \text{ where}$$

$$g(i,j) = (i,\ \delta \cdot j + (1-\delta)\,(w_x - 1),\ (1-\delta)\,(j - w_x + 1))$$

$$\delta = \begin{cases} 1, & \lfloor j/w_x \rfloor = 0 \\ 0, & \lfloor j/w_x \rfloor \neq 0. \end{cases}$$

Now, instead of carrying out the filter kernel on every single subwindow, we clip the corresponding region within the merged window to apply 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 facilitates resource usage.

6.2 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 entire image. For loop coarsening, multiple iterations are issued simultaneously to process whole beats at once, instead of only single pixels. Therefore, handing over data between kernels requires specific code generation techniques, which implement wider data types than the actual pixel data type used. Regarding code generation for Vivado HLS, freely choosing the wider data types is accomplished by utilizing the Vivado-specific type `ap_uint<`*bit width*`>`, which can be employed to implement *explicit* and *implicit vectorization*. For AOCL, we can directly exploit built-in OpenCL vector types, without the need to declare any new data types.

6.2.1 Explicit Vectorization

This technique is only used if the elements of a vector represent a pixel containing multiple values, such as images in RGBA color space, where computations and memory accesses should not be mixed across color channels. Such a behavior is attained 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, Reduce::SUM, [&] () -> 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 a local operator for RGBA data using explicit vector types of length $v_e = 4$.

Within the `kernel()` method the actual local operator code is provided. The domain `dom` represents the window size of the local operator. In lines 2–4, a reduction over the specified domain is initiated with `SUM` as the reduction method and a C++ lambda function describing the computational steps applied in each iteration. The mask and input image can be accessed using relative coordinates or the domain. 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 reduction result must be assigned to the `output()` method (line 7).

6.2.2 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 5.2.1) and the merged window approach (Section 6.1) have been adapted. However, special considerations need to be taken into account for mixing both types of vectorization.

6.2.3 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 a factor of 32 for the Laplacian operator (Listing 1), which loads and stores pixels of type `uchar4`, would result in beats of type `ap_uint<1024>` for Vivado HLS and `uchar4[32]` for AOCL. 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 then filled into the merged window. On the other hand, explicit vector elements are extracted at the kernel level, as explicitly described by the programmer.

7 Experimental Results

To assess the performance characteristics of the presented approach, we have evaluated several typical algorithms from image processing:

LP3HV The *Laplacian* filter is based on a local operator. Depending on the mask variant used, different types of edges can be detected. This variant, detects horizontal and vertical edges by applying a local operator of size 3×3.

LP3D In addition to horizontal and vertical edges, this variant also detects diagonal edges.

LP5 Based on a larger window of size 5×5, this variant can detect horizontal, vertical, and diagonal edges.

HC3 The *Harris corner* detector employs single-precision floating-point arithmetic and consists of a complex image pipeline comprising 4 point and 5 local operators. After constructing 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 exceedances.

OF15 The *optical flow* issues a Gaussian blur and computes signatures for two input images using the census transform. A third kernel performs a block compare between these signatures using a 15×15 window in order to extract vectors describing the optical flow.

BM60 From the area of stereo vision, we have also evaluated a *block matching* algorithm that is similar to the optical flow. Correspondences are found using census transformed signatures, which are compared along an epipolar line of 60 pixels.

GB3 The *Gaussian blur* embodies a filter mask of size 3×3 with unsigned integer coefficients. The final step, after applying the convolution, is a division by 16.

SB3 The *Sobel* operator is a 3×3 local operator also used for edge detection by computing the derivative of an image. In our implementation, we compute vertical and horizontal derivatives and combine them using the Euclidean distance in a third kernel, involving a square root function.

BL3 The bilateral filter [31] is a 3×3 local operator used for reducing noise while preserving edges. It embodies an exponential function and employs single-precision floating-point arithmetic.

The implementation results for Vivado HLS are obtained for a Xilinx Kintex 7 XC7K325TFFG900 FPGA as follows: first, HLS sources are generated from the DSL specification of an algorithm via Hipacc, then HDL codes are acquired through Vivado HLS and finally Post Place and Route (PPnR) characteristics for an evaluation case are captured using version 2014.4 of the Xilinx Vivado Design Suite. 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). In addition, the results also specify the maximum achievable clock frequency (F [MHz]), and throughput (TP [fps]).

For the OpenCL results, we employed an Altera Stratix VFPGA (5SGXEA7), and used version 14.1 of AOCL, which is integrated into Quartus II. Similar to Vivado HLS, the evaluation resultsinclude II, DSPs, and the maximum achievable clock frequency (F). Due to technology differences, we additionally provide results for ALUTs and Registers, as well as Logic utilization, which indicates how many half-Adaptive Logic Modules (half-ALMs) are used in the implementation. However, AOC does not report overall latency.

Note that for all results, we were evaluating the achievable throughput independently of the available memory bandwidth.

7.1 Loop Tiling and Loop Coarsening

In the first assessment, we compare parallelizing the different versions of the Laplacian filter using loop tiling and loop coarsening. We use RGBA color encoded input images of size 1024×1024, where each pixel requires 32 bits. The local operator in a loop of an accelerator is implemented using *explicit vectorization* with $v_e = 4$.

7.1.1 Vivado HLS

The Vivado HLS implementation results for loop tiling and loop coarsening are listed in Table 1, where R indicates the *replication factor*. Figure 12 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. It can be observed that not only a significantly higher throughput can be achieved but also considerably less resources are required when loop coarsening is used instead of loop tiling. This is because only the local operator is replicated, whereas entire accelerators, including the line buffer, must be replicated and additional logic, handling data distribution, must be included to facilitate loop tiling as detailed in Section 5. In fact, for the Kintex-7 FPGA, Laplace implementations could be parallelized up to 16 replications through loop tiling whereas even higher parallelization is possible with loop coarsening.

Additionally, we have analyzed the speedup that can be achieved by both approaches. Table 1 lists the theoretical speedup (ThSU), which is defined as the quotient of the latency for 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. Latency for tiling is always higher than coarsening. This is caused by two reasons: First, as data beats are read sequentially in row-major order, full parallel processing can only be initiated as soon as enough data has been consumed to satisfy the line buffer of the right most tile. Therefore, the initial latency is higher in contrast to coarsening, where only a single line buffer needs to be satisfied. Second, processing tiles requires a latency of $w/p + \text{overlap}(i)$, while coarsening only needs w/p cycles.

Figure 13 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 factor of two and four, both methods can achieve near-optimal speedup and are very close to each other. Yet, for the cases in which the parallelization factor is higher than four, the throughput

Table 1 PPnR results for parallelization of the Laplacian operator using loop tiling and loop coarsening for Vivado HLS.

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	526860	792	1680	2676	4	2	358.1	679.7	1.9	1.9	525835	241	436	1003	2	0	374.4	712.0	2.0	2.0
4	264460	1757	4225	6632	4	6	355.3	1343.7	3.7	3.9	263435	378	747	1807	4	0	374.4	1421.2	4.0	4.0
8	133260	5610	12541	18504	8	14	292.6	2196.1	6.1	7.8	132235	739	1370	3422	8	0	358.6	2711.8	7.6	7.9
16	67660	16999	38673	58104	16	30	227.3	3360.5	9.4	15.5	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	526862	984	2252	3861	4	2	353.1	670.1	1.8	1.9	525836	415	868	1645	2	0	364.6	693.3	2.0	2.0
4	264462	2251	5444	8541	4	6	350.5	1325.3	3.7	3.9	263440	695	1704	3367	4	0	342.9	1301.8	3.7	4.0
8	133262	5356	14392	21880	8	14	303.7	2279.4	6.4	7.8	132240	1184	3092	5737	8	0	347.7	2629.3	7.4	7.9
16	67662	18095	42629	64382	16	30	220.6	3260.3	9.2	15.5	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	4	0	354.4	336.6	1.0	1.0	1052691	993	2333	2015	4	0	354.4	336.6	1.0	1.0
2	529431	2346	5937	10130	8	2	343.7	649.2	1.9	1.9	526353	1630	3984	7538	4	0	343.6	652.9	1.9	2.0
4	266775	5138	12258	19344	8	6	342.7	1284.5	3.8	3.9	263700	2791	6824	12194	8	0	342.6	1299.1	3.9	4.0
8	135447	11064	26966	41520	16	14	312.6	2308.5	6.8	7.7	132371	3851	10352	17605	16	0	325.3	2457.6	7.3	8.0
16	69783	28337	65609	101696	32	30	212.7	3048.9	9.0	15.0	66707	7055	19651	30473	30	0	315.9	4735.0	14.1	15.8

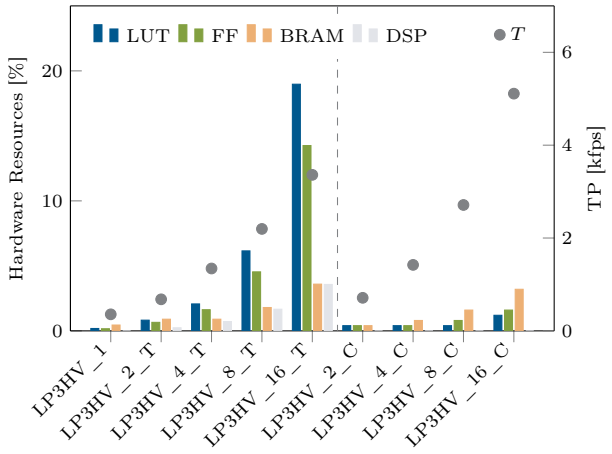


Fig. 12 Comparison of PPnR characteristics for parallelizing LP3HV using loop tiling (T) and loop coarsening (C) for Vivado HLS.

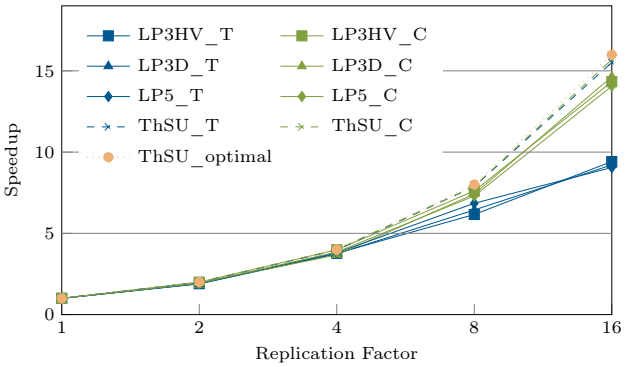


Fig. 13 Comparison of the actual and theoretical speedup (ThSU) achieved for parallelizing the implementations of the Laplacian operator using loop tiling (T) and loop coarsening (C) for Vivado HLS.

of loop tiling negatively diverges, whereas the coarsening still remains close to the theoretical speedup. Note that the main reason of the divergence comes from the clock frequency, which deteriorates with increasing resource utilization, causing loop tiling to have significantly slower logic speed. In addition, overlap regions augmented to image slices in case of tiling must be stored and shifted in the line buffers although data beats of the overlaps become redundant and must be flushed from the sliding window right after being used, causes the streaming pipeline to stall the execution and not generate valid output. For this reason, even the theoretical speedup of the throughput in tiling, ThSU_T, negatively diverges from the optimal theoretical speedup ThSU_optimal.

7.1.2 AOCL NDRange Kernels

AOC embodies a vectorization engine that automatically parallelizes a kernel according to user defined directives, which determine the number of SIMD lanes and compute units, as already explained in Section 4.2.1. On the downside, however, a line-buffered kernel that harnesses the benefits of pipelining cannot be vectorized with these directives. Therefore, in this section, we investigate the vectorization of NDRange kernels as an alternative to Hipacc's code generation. Since replicating compute units and SIMD lanes is very similar to loop tiling (T) and loop coarsening (C), we chose to use the same notation to present the AOC's implementation results.

The design space exploration for variants of the Laplacian filter by applying Altera-specific keywords to NDRange kernels is given in Table 2. In addition, resource utilization and maximum achievable throughput is illustrated in Figure 14. Applying replication through SIMD lanes instead of compute units mostly does not

Table 2 PPnR results for parallelization of the Laplacian operator using multiple compute units and SIMD lanes for OpenCL NDRange kernels.

Laplace 3 × 3 HV (Compute Units)									Laplace 3 × 3 HV (SIMD)							
R	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU
1	45064	68444	19.76	421	12	274.1	261.4	1.0	45064	68444	19.76	421	12	274.1	261.4	1.0
2	52334	80941	21.92	498	24	280.6	535.2	2.0	49008	75829	21.03	498	16	301.3	574.7	2.2
4	66150	105551	26.54	652	48	263.5	1005.2	3.8	57214	90670	23.69	650	24	273.5	1043.4	4.0
8	95471	155033	36.24	960	96	242.6	1850.9	7.1	71924	119922	29.29	954	40	229.7	1752.6	6.7
16	145540	251859	53.59	1864	192	162.8	2484.6	9.5	101162	177970	40.06	1577	72	177.7	2712.1	10.4
Laplace 3 × 3 D (Compute Units)									Laplace 3 × 3 D (SIMD)							
R	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU
1	46654	72121	20.32	481	12	287.9	274.6	1.0	46654	72121	20.32	481	12	287.9	274.6	1.0
2	55481	88633	23.28	618	24	276.2	526.7	1.9	51720	82623	22.07	618	16	286.9	547.3	2.0
4	72492	121106	29.47	892	48	257.8	983.4	3.6	62478	104576	26.28	890	24	265.2	1011.6	3.7
8	108383	185943	42.12	1440	96	176.0	1342.7	4.9	83261	147989	34.45	1434	40	207.1	1580.2	5.8
16	—	—	—	—	—	—	—	—	123452	234201	50.57	2537	72	146.1	2229.5	8.1
Laplace 5 × 5 (Compute Units)									Laplace 5 × 5 (SIMD)							
R	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU
1	55265	91019	23.65	721	16	251.2	239.6	1.0	55265	91019	23.65	721	16	251.2	239.6	1.0
2	72369	125839	30.20	1098	32	246.8	470.7	2.0	68340	118938	28.89	1098	20	244.8	467.0	1.9
4	108146	195586	43.52	1852	64	237.8	907.2	3.8	94241	173707	39.17	1850	28	190.8	727.7	3.0
8	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

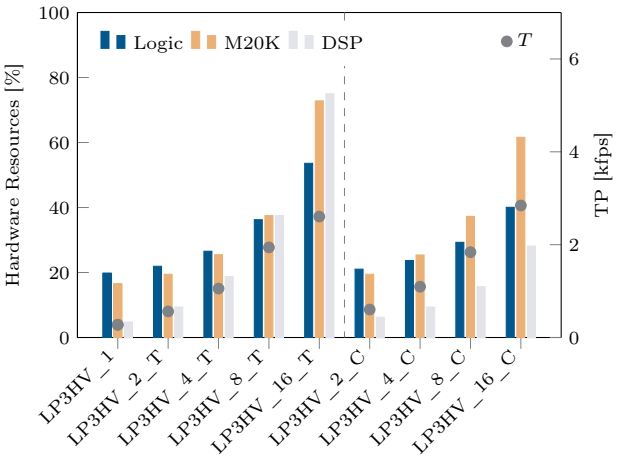


Fig. 14 Comparison of PPnR characteristics for parallelizing LP3HV using multiple compute units (T) and SIMD lanes (C) for OpenCL NDRange kernels.

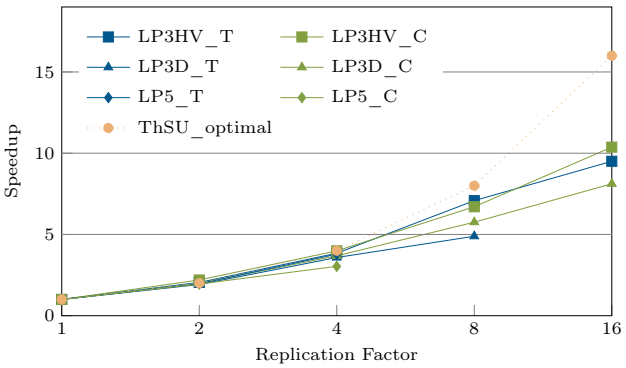


Fig. 15 Comparison of the actual speedup achieved for parallelizing the implementations of the Laplacian operator using multiple compute units (T) and SIMD lanes (C) for OpenCL NDRange kernels.

result in a noticeable increase in throughput, which is considerably below the expected linear speedup in both cases. Although, considering resource usage alone, replicating SIMD lanes clearly renders preferable over replicating compute units, as replicating compute units shows significantly higher resource usage, particularly regarding logic utilization and DSPs. For both approaches, however, the utilization of M20K on-chip memory grows linearly with the replication factor, showing the lack of resource sharing. This leads to the fact that AOC cannot fit LP3D with more than 8 compute units to the specified FPGA and that LP5 can only be successfully implemented with a maximum replication factor of 4 for both approaches.

Considering speedup, Table 2 does not include theoretical speedup (ThSU) since AOC does not report the overall latency. Therefore, we only compare the actual speedup (SU) based on throughput (TP) incorporating the achieved clock frequency. Figure 15 compares the achieved speedup with the optimal theoretical speedup as an upper bound. From investigating these results alone, it is barely possible to determine which approach is superior to the other in terms of which approach is more likely to achieve a higher speedup. In all cases, the actual speedup stays sublinear due to a severe drop in clock frequency for higher replication factors. The main reason for this is that the variation in the maximum

Table 3 PPnR results for parallelization of the Laplacian operator using loop tiling and loop coarsening for line-buffered OpenCL single work-item kernels.

Laplace 3 × 3 HV (Loop Tiling)									Laplace 3 × 3 HV (Loop Coarsening)							
R	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU
1	44118	68481	19.70	367	2	303.6	289.5	1.0	44118	68481	19.70	367	2	303.6	289.5	1.0
2	47999	70531	20.35	453	4	263.5	502.6	1.7	44128	70151	19.82	368	4	304.5	580.8	2.0
4	58505	87774	23.47	551	16	278.9	1064.1	3.7	45021	72270	20.04	365	8	304.5	1161.6	4.0
8	79603	119710	29.89	755	32	201.6	1538.5	5.3	46188	76026	20.51	371	16	304.6	2323.8	8.0
16	122828	184820	43.07	1163	64	254.9	3889.5	13.4	48191	86073	21.85	392	32	303.6	4632.3	16.0
Laplace 3 × 3 D (Loop Tiling)									Laplace 3 × 3 D (Loop Coarsening)							
R	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU
1	44645	68479	19.79	363	2	273.4	260.7	1.0	44645	68479	19.79	363	2	273.4	260.7	1.0
2	48979	71684	20.57	453	4	269.7	514.5	2.0	44797	70420	19.88	363	4	305.5	582.8	2.2
4	60173	89671	23.82	551	16	291.6	1112.5	4.3	46712	74028	20.31	365	8	305.4	1165.1	4.5
8	82944	123241	30.59	755	32	265.2	2023.7	7.8	48243	78408	20.99	371	16	278.4	2123.9	8.1
16	128797	190936	44.40	1163	64	247.7	3779.6	14.5	54053	89754	22.99	392	32	278.5	4250.3	16.3
Laplace 5 × 5 (Loop Tiling)									Laplace 5 × 5 (Loop Coarsening)							
R	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU	ALUT	Register	Logic (%)	M20K	DSP	F [MHz]	TP [fps]	SU
1	48790	73232	21.07	374	4	303.6	289.5	1.0	48790	73232	21.07	374	4	303.6	289.5	1.0
2	57418	80588	23.11	460	9	279.9	534.0	1.8	52220	77640	22.21	374	8	305.5	582.8	2.0
4	77347	106810	28.84	558	24	292.0	1113.8	3.8	59528	86785	24.54	376	16	284.2	1084.0	3.7
8	118609	158332	40.50	770	48	262.8	2005.1	6.9	74056	104114	29.25	388	32	303.9	2319.0	8.0
16	188418	259174	61.37	1515	96	244.8	3735.2	12.9	107310	142069	39.70	419	64	270.6	4129.5	14.3

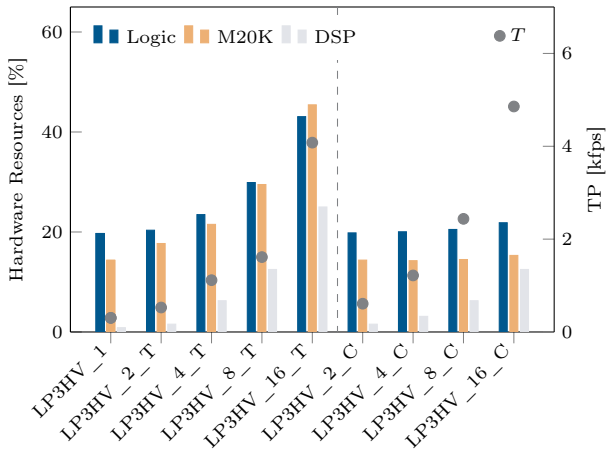


Fig. 16 Comparison of PPnR characteristics for parallelizing LP3HV using loop tiling (T) and loop coarsening (C) for line-buffered OpenCL single work-item kernels.

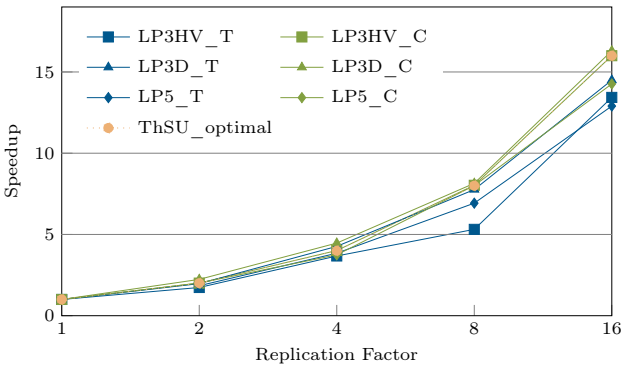


Fig. 17 Comparison of the actual speedup achieved for parallelizing the implementations of the Laplacian operator using loop tiling (T) and loop coarsening (C) for line-buffered OpenCL single work-item kernels.

achievable clock frequency of the synthesized hardware linearly affects the final throughput. We attribute the high variation in the clock frequency for the implementations of the same kernel under different directives to the heuristics internally used in AOC.

7.1.3 AOCL Single Work-Item Kernels

Results for line-buffered single work-item kernels, comparing implementations using loop tiling (T) and loop coarsening (C), generated with Hipacc, are given in Table 3. Figure 16 depicts the resource usage and overall throughput of both approaches. As line buffering is applied, the usage of M20K on-chip memory severely grows with increasing replication factors for loop tiling. The same applies to Logic, whereas both resource primitives increase only slightly for loop coarsening. On the contrary, the usage of DSPs increases roughly linear with the replication factor for both approaches but is still considerably less for loop coarsening, as only the operator kernel is replicated and the compiler can eliminate redundant arithmetic operations. In contrast to NDRange kernels, the AOC did not have any trouble to fit the generated designs for single work-item kernels onto the FPGA.

Regarding the speedup, Figure 17 clearly shows that both approaches can achieve results close to the optimal speedup. For the LP3D, loop coarsening even achieves a superlinear speedup, and therefore better than op-

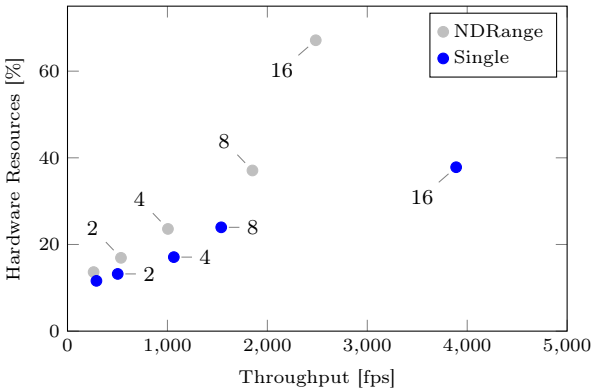


Fig. 18 Design points for parallelizing LP3HV through multiple compute units for NDRange kernels and loop tiling for line-buffered single work-item kernels.

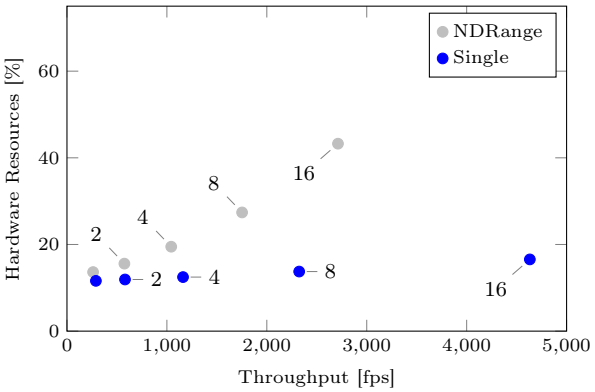


Fig. 19 Design points for parallelizing LP3HV through multiple SIMD lanes for NDRange kernels and loop coarsening for line-buffered single work-item kernels.

timal. This is due to the fact that the unparallelized implementation, which serves as the denominator for building the quotient, did result in a rather low clock frequency. Again, we attribute this to AOC's internal heuristics. For loop tiling on the other hand, synthesis still manages to maintain a rather high clock frequency for large replication factors, despite the overlapping data beats. For that reason, the overall throughput still resides quite close to the optimal speedup. However, also for line-buffered single work-item kernels, loop coarsening remains preferable to loop tiling, as it achieves slightly better results for all versions of the Laplacian operator.

7.2 NDRange vs. Single Work-Item

As Altera provides special keywords to automatically apply accelerator replication for NDRange kernels without any code modifications, the question arises whether to prefer NDRange kernels over single work-item kernels. NDRange kernels have the advantage that they are highly portable and also perform well on other architectures, such as GPUs. On the other hand, single work-item kernels represent very target-specific implementations, tailored to FPGA architectures by employing line buffering and temporal locality. Parallelizing such kernels requires severe code modifications, as either multiple line buffers need to be allocated or the superwindow concept needs to be applied.

Aside from effort for implementation, achievable throughput and resource usage are the metrics of most importance. Figure 18 shows the design points we were able to obtain for NDRange kernels and line-buffered single work-item kernels by applying replication through compute units and loop tiling, respectively. The annotated numbers represent the applied replication factor. It can be seen that single work-item kernels remain superior regarding resource usage. However, in terms of throughput, the corresponding NDRange design point of the same replication factor is not always dominated by its single work-item counterpart. Nonetheless, single work-item kernels are still preferable, as the next higher replication factor consumes roughly the same amount of resources while providing a severe increase in throughput.

As replicating the inner kernel operator proved to be more promising regarding resource utilization, it is of considerable interest to compare replication through multiple SIMD lanes and loop coarsening as well. A visual representation of the resulting design points can be found in Figure 19. Here, the differences between NDRange and line-buffered single work-item kernels are even more intense, not only in terms of resource utilization but also in terms of achievable throughput. Yet, single work-item kernels with the same replication factor dominate NDRange kernels in both dimensions. With these results at hand, if an efficient implementation is the primary design goal, single work-item kernels should always be chosen over NDRange kernels. On the contrary, applying the superwindow concept for different replication factors requires severe code modifications, and thus more development effort. However, we circumvent this drawback by employing automatic code generation through the Hipacc DSL.

7.3 Comparison of FPGA Results to Other Accelerators

As Hipacc 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 and Stratix V FPGAs to those of the Hipacc-generated implementations for the Nvidia Tegra K1 SoC

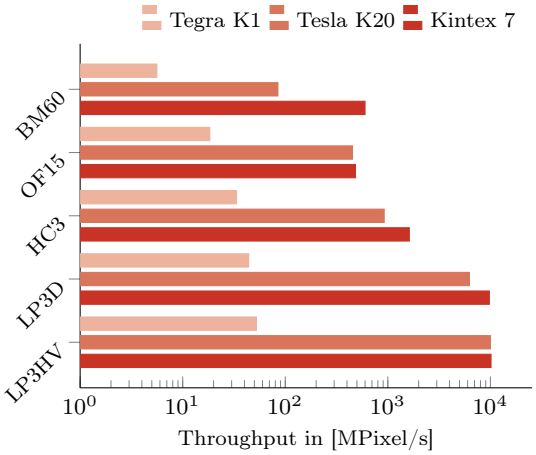


Fig. 20 Throughput comparison for the Kintex 7.

Table 4 PPnR characteristics of algorithm implementations generated by Hipacc on the Kintex 7.

Design	v_i	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]
LP3HV	32	33835	2810	5122	13136	29	0	343.3
LP3D	32	33840	4159	10933	19996	29	0	330.6
HC3	8	132268	28403	87916	104479	16	504	214.3
OF15	4	266112	30617	80480	100066	52	0	128.7
BM60	2	539728	5771	19222	23213	4	0	323.5

and Nvidia Tesla K20, as representatives for eGPUs and server-grade GPUs, respectively. Tables 4 and 5 list the PPnR results obtained for algorithm specifications generated by Hipacc for Kintex and Stratix, respectively. Note that, because of technology differences, we explicitly do not want to compare the Kintex with the Stratix, and therefore chose different algorithms. For all algorithms, an *implicit vectorization* factor v_i through code generation. Input size of the algorithms is 1024×1024 pixels. 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 Hipacc, we can generate CUDA code for implementing the algorithms on the Tegra K1 and the Tesla K20 GPUs. Note that the implementations were obtained using automatic code generation based on *exactly the same DSL source code*. To ensure a fair comparison, Hipacc's exploration feature was 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 on the GPUs.

The throughput is moreover compared in Figures 20 and 21. For increasing algorithm complexity, the potential for parallelization, and therefore throughput, of the FPGA accelerators is limited by the available logic resources. In case of the GPUs, the throughput is dramatically reduced for greater window sizes with an increased number of memory accesses, exposing the available memory bandwidth clearly as the main performance bottleneck. For the FPGAs, however, this is not an issue, as data is perfectly streamed, and thus, producing a new output pixel every clock cycle.

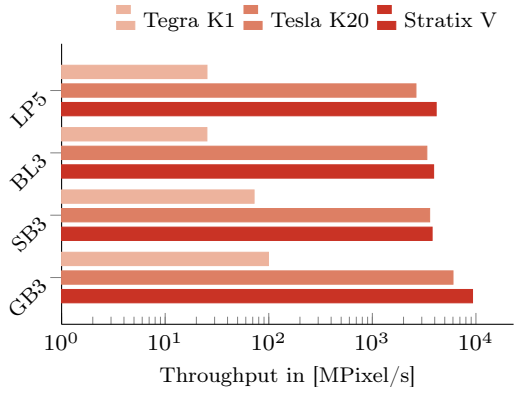


Fig. 21 Throughput comparison for the Stratix V.

Table 5 PPnR characteristics of algorithm implementations generated by Hipacc on the Stratix V.

Design	v_i	ALUTs	Registers	LU (%)	M20K	DSP	F [MHz]
GB3	32	47045	73584	20.64	363	0	303.58
SB3	16	58308	96673	24.59	497	96	247.34
BL3	16	140368	189010	48.93	381	233	255.75
LP5	16	107310	142069	39.70	419	64	270.63

Regarding energy efficiency, Table 6 compares the energy efficiency of the different accelerators in frames per Watt (E [fpW]). Power requirements were assessed by performing a timing simulation on the post-route simulation netlists and evaluation of the switching activity in Vivado. Since separating the infrastructure logic for OpenCL from the accelerator itself is not possible and no direct tool is available for AOCL we only discuss energy measurements through Vivado HLS results. The GPU's power consumption 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). It can be seen that FPGAs outperform GPUs by far in terms of frames per Watt. This is particularly the case for algorithms that are demanding higher memory bandwidth, which negatively affects GPU throughput.

8 Related Work

HLS for FPGAs has received a great deal of attention over the past years, spurring successful general purpose frameworks from major Electronic Design Automation (EDA) companies. Most prominent examples

[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/

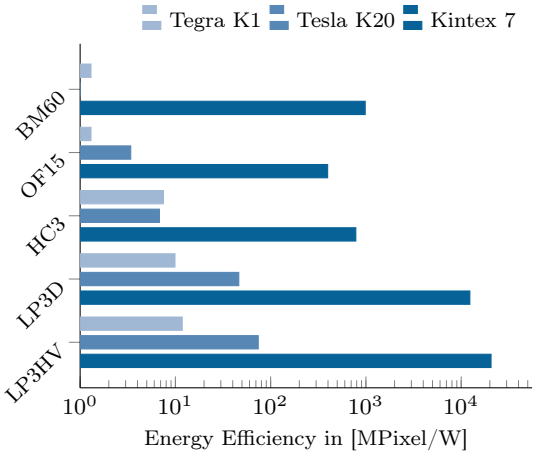


Fig. 22 Energy efficiency for the Kindex 7.

Table 6 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

include Catapult [18] from Mentor Graphics, Cynthesizer [19] from Cadence (before Forte), NEC's CyberWorkbench [34], and Synopsys' Synphony C Compiler (formerly PICO Express [1] by Synfora), but also tools from academia, such as ROCCC [33] and LegUp [6]. Recently, also the main FPGA vendors extended their product portfolio by HLS tools and seamlessly integrated them as a front end, employing source-to-source compilation, into their existing design environments for logic synthesis and technology mapping. Xilinx acquired AutoPilot [38] to transform C codes into HDL description, whereas Altera developed its own OpenCL-based HLS tool, called AOCL [30].

Languages utilizing a data-parallel programming paradigm, such as OpenCL, are inherently well-suited for exploiting DLP. Early work, proposing the compilation of a data-parallel programming model to hardware implementations, was presented with FCUDA in [22]. Here, CUDA, heavily driven by GPU industry and the main inspiration for the OpenCL standard, was adapted to serve as source language. Employing a source-to-source compiler, CUDA thread blocks are transformed into parallel C code for Autopilot [38]. Other work [20] deals with the synthesis of parallel hardware implementations starting from OpenCL kernels, similar to the Altera Offline Compiler [30]. The main difference between these approaches and our methodology is that the FPGA implementations imitate typical accelerators found in the programming model, such as GPUs or multi-core systems. We also employ certain GPU parallelism concepts, however, we use a computational model that closely resembles RTL implementations.

For programs without loop-carried data dependencies, [14] proposes parallelization through vectorization of outer loops. In case a loop program contains loop-carried data dependencies or is not defined over a rectangular iteration space, the *polyhedron model* [8] is often the method of choice for automatic parallelization. Since more than two decades, it is a powerful instrument that relies on the concept of linear algebra and linear programming. Polyhedral techniques can be widely applied for solving problems such as analysis of data dependencies, scheduling, data locality optimization (e.g., cache optimization), parallelization (e.g., loop tiling or vectorization), and communication minimization. Thus, it is not astonishing that the model has not only been applied for the generation of highly optimized parallel code [5, 32], but also in HLS. For instance, the PARO HLS tool [10] uses its own input language, which allows to specify recurrence equations defined over polyhedral iteration domains. Subsequently, a number of affine transformations, partitioning, as well as scheduling are applied, and finally, synthesizable VHDL code, describing a deeply pipelined and highly parallelized hardware accelerator, is emitted. Recently, researchers considered the polyhedral model also in combination with C-based HLS. Alias et al. presented in [2] an approach how to optimize off-chip memory traffic and on-chip data reuse in case of FPGA accelerators, and integrated their concept into the Altera C-to-Hardware (C2H) toolchain. Cong et al. [24, 15] studied the impact of several polyhedral transformations in order to improve data reuse and throughput in HLS, and prototypically implemented the transformations in PolyOpt/HLS. All of the aforementioned polyhedron-model-based HLS approaches perform aggressive pipelining and loop tiling, but none considers loop coarsening.

Using DSLs for hardware development has also been a fruitful area of research for several projects, such as SPIRAL [25], PARO [10], and Darkroom [11]. The concept of using DSLs to generate code for general purpose HLS tools was also explored by George et al. in [9], amongst others. Several approaches target established parallel computing models and use source-to-source transformations to generate code for HLS frameworks. Additionally, [7] is closely related to our work, where the authors describe an approach of how to compile programs to C code, using OpenMP and Pthreads, that can serve as input to LegUp. The support for thread-level parallelism can be compared to loop tiling, however, our

approach also supports replicating the kernel alone. In [23], Plavec et al. presented the compilation of FPGA accelerators from the streaming language Brook. The concept of stream reductions is similar to aggregating several pixels into vectors, which then represent the data for streaming. The author's work is, however, focused on another application domain and does not discuss the parallelization of local operators. At Register Transfer Level, our architecture for loop coarsening can be compared to VHDL templates for stencil computations by Schmidt and others [29]. Since our implementation is on a higher abstraction level, it also allows compiler optimizations across the replicated kernels to eliminate redundant computations, which is not possible if the kernels are distinct entities.

9 Conclusion

In this work, we have presented the parallelization of image processing algorithms using loop tiling and loop coarsening for different vendor-specific HLS approaches. For both approaches, 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 fewer stall cycles, and therefore yields better accelerator performance. Therefore, loop coarsening should be preferred unless the input image needs to be strided. It should be noted that a hybrid solution that strides the input stream with loop tiling and applies loop coarsening in each tile can as well be implemented. However, this will again use more resources than loop coarsening. Consequently, we have augmented the FPGA back end of the DSL framework Hipacc with transformations for implementing loop coarsening when generating code for Vivado HLS and the Altera SDK for OpenCL. As Hipacc 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 able to deliver considerably higher performance levels. Although HLS still has some deficiencies in contrast to hand-coded RTL implementations, it allows rapid design space exploration, by means of which we were able to achieve an improved ratio between resource usage and performance.

Acknowledgements 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.

Acronyms

ALUT . . . . . Adaptive Look-Up Table
AOC . . . . . Altera Offline Compiler
AOCL . . . . . Altera SDK for OpenCL
ASIC . . . . . Application-Specific Integrated Circuit

BRAM . . . . . Block Random Access Memory

CUDA . . . . . Compute Unified Device Architecture

DLP . . . . . Data-Level Parallelism
DPRAM . . . . . Dual-Port-RAM
DSL . . . . . Domain-Specific Language
DSP . . . . . Digital Signal Processor

EDA . . . . . Electronic Design Automation
eGPU . . . . . embedded GPU

F . . . . . Frequency
FF . . . . . Flipflop
FIFO . . . . . First In First Out
FPGA . . . . . Field Programmable Gate Array

GPU . . . . . Graphics Processing Unit

half-ALM . . . . . half-Adaptive Logic Module
HDL . . . . . Hardware Description Language
Hipacc . . . . . Heterogeneous Image Processing Acceleration
HLS . . . . . High-Level Synthesis

IDE . . . . . Integrated Development Environment
II . . . . . Initiation Interval
ILP . . . . . Instruction-Level Parallelism
IO . . . . . Input/Output

LAT . . . . . Latency
LU . . . . . Logic Utilization
LUT . . . . . Look-Up Table

OpenCL . . . . . Open Computing Language

PPnR . . . . . Post Place and Route

RGBA . . . . . Red Green Blue Alpha
RTL . . . . . Register Transfer Level

SIMD . . . . . Single Instruction Multiple Data
SPMD . . . . . Single Program Multiple Data
SU . . . . . Speedup

ThSU . . . . . Theoretical Speedup
TP . . . . . Throughput

References

1. Aditya, S., Kathail, V.: Algorithmic synthesis using PICO: An integrated framework for application engine synthesis and verification from high level C algorithms. In: P. Coussy, A. Morawiec (eds.) High-Level Synthesis: From Algorithm to Digital Circuit, chap. 4, pp. 53–74. Springer (2008). DOI 10.1007/978-1-4020-8588-8_4
2. Alias, C., Darte, A., Plesco, A.: Optimizing remote accesses for offloaded kernels: application to high-level synthesis for FPGA. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), pp. 575–580 (2013)
3. Amdahl, G.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference (AFIPS), pp. 483–485 (1967)
4. Bailey, D.: Design for embedded image processing on FPGAs. John Wiley & Sons, Inc. (2011)
5. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer **43**(6), 101–113 (2008)
6. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J., Brown, S., Czajkowski, T.: LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pp. 33–36 (2011)
7. Choi, J., Brown, S., Anderson, J.: From software threads to parallel hardware in high-level synthesis for FPGAs. In: Proceedings of the International Conference on Field-Programmable Technology (FPT), pp. 270–277 (2013)
8. Feautrier, P., Lengauer, C.: Polyhedron model. In: D. Padua (ed.) Encyclopedia of Parallel Computing, pp. 1581–1592. Springer (2011). DOI 10.1007/978-0-387-09766-4_502
9. George, N., Novo, D., Rompf, T., Odersky, M., Ienne, P.: Making domain-specific hardware synthesis tools cost-efficient. In: Proceedings of the International Conference on Field-Programmable Technology (FPT), pp. 120–127 (2013)
10. Hannig, F., Ruckdeschel, H., Dutta, H., Teich, J.: PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In: Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC), *Lecture Notes in Computer Science (LNCS)*, vol. 4943, pp. 287–293. Springer (2008). DOI 10.1007/978-3-540-78610-8_30
11. Hegarty, J., Brunhaver, J., DeVito, Z., Ragan-Kelley, J., Cohen, N., Bell, S., Vasilyev, A., Horowitz, M., Hanrahan, P.: Darkroom: Compiling high-level image processing code into hardware pipelines. In: Proceedings of the 41st International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 144:1–144:11 (2014)
12. Hwang, D., Cho, S., Kim, Y., Han, S.: Exploiting spatial and temporal parallelism in the multithreaded node architecture implemented on superscalar RISC processors. In: Proceedings of the International Conference on Parallel Processing (ICPP), pp. 51–54 (1993)
13. Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 318–328 (1988). DOI 10.1145/53990.54022
14. Lattuada, M., Ferrandi, F.: Exploiting outer loops vectorization in high level synthesis. In: Proceedings of the 28th International Conference on Architecture of Computing Systems (ARCS), *Lecture Notes in Computer Science (LNCS)*, vol. 9017, pp. 31–42. Springer (2015)
15. Li, P., Pouchet, L.N., Cong, J.: Throughput optimization for high-level synthesis using resource constraints. In: S. Rajopadhye, S. Verdoolaege (eds.) Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques. Vienna, Austria (2014)
16. Membarth, R., Reiche, O., Hannig, F., Teich, J.: Code Generation for Embedded Heterogeneous Architectures on Android. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE), pp. 86:1–86:6. IEEE, Dresden, Germany (2014). DOI 10.7873/DATE.2014.099
17. Membarth, R., Reiche, O., Hannig, F., Teich, J., Körner, M., Eckert, W.: HIPAcc: A domain-specific language and compiler for image processing. IEEE Transactions on Parallel and Distributed Systems **27**(1), 210–224 (2016). DOI 10.1109/TPDS.2015.2394802
18. Mentor Graphics: Catapult High-Level Synthesis (2016). URL https://www.mentor.com/hls-lp/catapult-high-level-synthesis/
19. Meredith, M.: High-level SystemC synthesis with Forte's Cynthesizer. In: P. Coussy, A. Morawiec (eds.) High-Level Synthesis: From Algorithm to Digital Circuit, chap. 5, pp. 75–97. Springer (2008). DOI 10.1007/978-1-4020-8588-8_5
20. Owaida, M., Bellas, N., Daloukas, K., Antonopoulos, C.: Synthesis of platform architectures from OpenCL programs. In: Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 186–193 (2011)
21. Özkan, M., Reiche, O., Hannig, F., Teich, J.: FPGA-based accelerator design from a domain-specific language. In: Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL). DOI 10.1109/FPL.2016.7577357
22. Papakonstantinou, A., Gururaj, K., Stratton, J., Chen, D., Cong, J., Hwu, W.M.: FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In: Proceedings of the IEEE 7th Symposium on Application Specific Processors (SASP), pp. 35–42 (2009). DOI 10.1109/SASP.2009.5226333
23. Plavec, F., Vranesic, Z., Brown, S.: Exploiting task- and data-level parallelism in streaming applications implemented in FPGAs. ACM Transactions on Reconfigurable Technology and Systems (TRETS) **6**(4), 16:1–16:37 (2013)
24. Pouchet, L.N., Zhang, P., Sadayappan, P., Cong, J.: Polyhedral-based data reuse optimization for configurable computing. In: Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, pp. 29–38. ACM (2013)
25. Püschel, M., Franchetti, F., Voronenko, Y.: Spiral. In: D. Padua (ed.) Encyclopedia of Parallel Computing, pp. 1920–1933. Springer (2011). DOI 10.1007/978-0-387-09766-4
26. Ratha, N., Jain, A.: Computer vision algorithms on reconfigurable logic arrays. IEEE Transactions on Parallel and Distributed Systems (TPDS) **10**(1), 29–43 (1999)
27. Reiche, O., Schmid, M., Hannig, F., Membarth, R., Teich, J.: Code generation from a domain-specific language for C-based HLS of hardware accelerators. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 17:1–17:10. ACM (2014). DOI 10.1145/2656075.2656081
28. Schmid, M., Reiche, O., Hannig, F., Teich, J.: Loop coarsening in C-based high-level synthesis. In: Proceedings of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 166–173. IEEE (2015)
29. Schmidt, M., Reichenbach, M., Fey, D.: A generic VHDL template for 2D stencil code applications on FPGAs. In: Proceedings of the 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time

Distributed Computing Workshops (ISORCW), pp. 180–187 (2012). DOI 10.1109/ISORCW.2012.39
30. Singh, D.: Implementing FPGA design with the OpenCL standard. Altera whitepaper (2011)
31. Tomasi, C., Manduchi, R.: Bilateral filtering for gray and color images. In: Proceedings of the Sixth International Conference on Computer Vision (ICCV), pp. 839–846. IEEE (1998)
32. Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., Rosen, I.: Polyhedral-model guided loop-nest auto-vectorization. In: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 327–337. IEEE (2009)
33. Villarreal, J., Park, A., Najjar, W., Halstead, R.: Designing modular hardware accelerators in C with ROCCC 2.0. In: Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 127–134 (2010)
34. Wakabayashi, K., Okamoto, T.: 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) **19**(12), 1507–1522 (2000)
35. Wang, C., Yuan, F.L., Yu, T.H., Markovic, D.: 27.5 a multi-granularity FPGA with hierarchical interconnects for efficient and flexible mobile computing. In: Proceedings of the IEEE International Solid-State Circuits Conference - Digest of Technical Papers, pp. 460–461 (2014)
36. Wolfe, M.: More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 655–664 (1989)
37. Xilinx Inc.: Vivado High-Level Synthesis (2016). URL http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
38. Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., Cong, J.: AutoPilot: A platform-based ESL synthesis system. In: P. Coussy, A. Morawiec (eds.) High-Level Synthesis: From Algorithm to Digital Circuit, chap. 6, pp. 99–112. Springer (2008). DOI 10.1007/978-1-4020-8588-8_6